Security of Esoteric firmware And Trusted Execution Environments

By

Zitai Chen

A thesis submitted to
the University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
College of Engineering and Physical Sciences
University of Birmingham
JANUARY 2023

UNIVERSITY OF
BIRMINGHAM

University of Birmingham Research Archive

e-theses repository

This unpublished thesis/dissertation is copyright of the author and/or third parties. The intellectual property rights of the author or third parties in respect of this work are as defined by The Copyright Designs and Patents Act 1988 or as modified by any successor legislation.

Any use made of information contained in this thesis/dissertation must be in accordance with that legislation and must be properly acknowledged. Further distribution or reproduction in any format is prohibited without the permission of the copyright holder.

ABSTRACT

Computer systems have become deeply integrated into everyday life and become more complex in order to satisfy functionality requirements. In modern vehicles, there are multiple embedded devices of different architectures, connected together to improve the driving experience. Modern computers including PC and server systems are also becoming a piece of embedded system with multiple components. Apart from the main Central Processing Unit (CPU), there are multiple microcontrollers on the motherboard. Additionally, modern architectures often provide a Trusted Execution Environment (TEE) which aims to provide a secure environment to protect the information used in it, even if the device is under the attacker's control.

These new developments in computer systems raise new challenges to security analysis. With multiple devices and microcontrollers of different architectures used in the system, firmware analyses have become a challenging task. Besides, with the complex design of x86 systems, the threat model of TEE, where the device holder can be considered malicious, has not been fully evaluated.

In this thesis, we contributed to three aspects of the security analysis of the aforementioned systems. Firstly, MetaEmu provides an architecture-agnostic emulator for re-hosting firmware of different architectures, which makes dynamic analysis possible for esoteric firmware. Furthermore, the firmware partitioning method introduced in Incision makes it easier to dissect large firmware blobs to aid manual analysis.

Secondly, with the analysis of Voltage Regulator Module (VRM) and TEE on Intel platforms, which is Intel SGX, we developed VoltPillager, a low-cost tool for injecting messages on the Serial Voltage Identification bus between the CPU and the voltage regulator on the motherboard. This allows us to precisely control the CPU core voltage. We leverage this powerful tool to mount the first hardware-based fault-injection attacks

that breach confidentiality and integrity of Intel SGX enclaves. This attack would require a rethink of the SGX adversarial model, where a cloud provider is untrusted and has physical access to the hardware.

Last, but not least, PMFault investigated the board-level security of server motherboards. With the analysis of Baseboard Management Controller (BMC) firmware and the motherboard design, we showcase new attacks which utilize the software-hardware interface exposed by microcontrollers on the motherboard. These attacks can cause serious security vulnerabilities, including leak of sensitive information in SGX and permanent damage to the CPU. This work gives a new perspective in the security evaluation of modern computer systems, which is board-level security analysis.

ACKNOWLEDGMENTS

First and foremost I am incredibly grateful to my supervisor Flavio Garcia, for his invaluable guidance and support during my PhD study. I would also like to thank David Oswald, Sam Thomas and Mihai Ordean for their immense knowledge and experience that inspired me to keep innovating and testing new ideas.

I would like to thank my examiners, Aurélien Francillon and Daniel Page for their time reviewing this thesis and my thesis group member Mark Ryan for his advice on my PhD study.

During the 4-year PhD at the University of Birmingham, I have met so many great colleagues and friends. I would like to thank them for the lunchtime chats, coffee breaks and meals cooked together, which make me feel less lonely and give me the energy to overcome the ups and downs during this changing time in my life. They are Georgios Vasilakis, Jan Van Den Herrewegen, Abdulla Aldoseri, Jing Zhang, Kit Murdock, Mo Zhang, Andreea-Ina Radu, Rajiv Ranjan Singh, Daniel Clark, Owen Pemberton, Yuhang Qu, Xiaoyu Qin, Xi Jia, Xiaoyu Su, Zishuai Cheng, Yufan Xu and Weiqing Chu.

I am especially thankful for my friends Yingxin Yu and Wei Wang, who have encouraged me to pursue my PhD. My gratitude extends to my dear mom Guiying Liu and dad Jiandong Chen for their love and years of selfless support!

This research is funded by Engineering and Physical Sciences Research Council (EPSRC)

List of Papers

This thesis is based on the following articles that were accepted for publication or currently under review. Where listed, the co-authors also advised on experiment/tool design, result analysis and paper editing. **Papers accepted for publication:**

- ***Chen, Z.****, Thomas, S. L., & Garcia, F. D.* (Accepted/In press). MetaEmu: an architecture agnostic rehosting framework for automotive firmware [1]. In CCS '22: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. Association for Computing Machinery (ACM).

- *Thomas, S. L., Van den Herrewegen, J., Vasilakis, G.,* ***Chen, Z.****, Ordean, M., & Garcia, F. D.* (2021). Cutting through the complexity of reverse engineering embedded devices [2]. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2021(3), 360–389.

- ***Chen, Z.****, Vasilakis, G., Murdock, K., Dean, E., Oswald, D., & Garcia, F. D.* (2021). VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface [3]. In 30th USENIX Security Symposium (USENIX Security 21) (pp. 699-716).

- ***Chen, Z.****, Oswald, D.,* (2023) PMFault: Faulting and bricking server CPUs through management interfaces [4]. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2023(2), 1–23.

Contents

Page

Acronyms **i**

1 General Introduction **1**

1.1 Firmware Analysis 2

1.2 Attacks against Trusted Execution Environments 3

1.2.1 Fault Injection Attack 3

1.2.2 Micro-architecture Based Attacks 4

1.3 Control Interfaces of Voltage Regulator Module 5

1.4 Inter-device Attacks 6

1.5 Contributions and Overview of the Thesis 8

2 MetaEmu: An Architecture Agnostic Rehosting Framework for Automotive Firmware **11**

2.1 Introduction 12

2.2 Background 15

2.3 System Overview 17

2.3.1 Framework Architecture 17

2.3.2 Challenges Rehosting Complex Firmware 19

2.4 Implementation of MetaEmu 22

2.4.1 Lifting & IR Generation 22

2.4.2 Firmware Simulation 25

2.4.3 Observers & Analysis Coordination 26

2.4.4 Peripheral Support 27
2.4.5 Interrupt Support 31
2.4.6 Integration with External Tools 32
2.5 Evaluation 33
2.5.1 Performance Benchmarks 34
2.5.2 Case Studies 39
2.6 Human Effort 49
2.7 Limitations 50
2.8 Discussion & Related Work 51
2.9 Conclusion 53

3 Cutting Through the Complexity of Reverse Engineering Embedded Devices 55

3.1 Introduction 56
3.1.1 Our Contribution 57
3.2 Background 59
3.2.1 Reverse Engineering Embedded Firmware 59
3.3 Overview 62
3.4 Methodology 65
3.4.1 Inputs 65
3.4.2 Region Inference 66
3.4.3 Control-Flow Extraction 68
3.4.4 Feedback-Driven Refinement 75
3.5 Evaluation 77
3.5.1 Firmware Data-Set 77
3.5.2 Correctness (C.1) 78
3.5.3 Real-World Usability (C.2.1, C.2.2) 82
3.5.4 Human Effort (C.3) 86

3.5.5 Discussion and Limitations . . . 88
3.6 Related Work . . . 89
3.7 Conclusion . . . 92

4 VoltPillager: Hardware-based Fault Injection Attacks against Intel SGX Enclaves 93

4.1 Introduction . . . 94
4.1.1 Our Contribution . . . 94
4.1.2 Responsible Disclosure . . . 95
4.1.3 Related Work . . . 96
4.1.4 Attacker Model . . . 97
4.1.5 Experimental Setup . . . 98
4.1.6 Outline . . . 99
4.2 Intel's Mitigation for Software-based Undervolting Attacks on SGX Enclaves 99
4.3 Power Management Interfaces . . . 100
4.3.1 Serial Voltage Identification . . . 100
4.3.2 System Management Bus and Power Management Bus . . . 104
4.4 VoltPillager for SVID Command Injection . . . 104
4.4.1 Implementation of VoltPillager . . . 106
4.5 Fault Injection into SGX Enclaves with Hardware-based Undervolting . . 109
4.5.1 Reproducing Plundervolt Proof-of-Concepts . . . 110
4.5.2 Comparison with Software-based Undervolting . . . 113
4.6 Delayed-Write Fault Attacks through Undervolting . . . 116
4.6.1 Initial Proof-of-Concept . . . 116
4.6.2 Practical Exploitation Scenario . . . 120
4.7 Implications and Countermeasures . . . 123
4.8 Conclusions . . . 125

5 BMC Firmware Analaysis and PMBus Based Power Attacks 127

5.1 Introduction . . . 128
5.1.1 Our Contribution . . . 129
5.1.2 Adversary Model . . . 130
5.1.3 Responsible Disclosure . . . 130
5.1.4 Related Work . . . 131
5.1.5 Paper Outline . . . 132
5.2 Analysis of Power Management Bus . . . 133
5.2.1 Experimental Setup . . . 133
5.2.2 Protocol Structure . . . 135
5.2.3 PMBus Commands . . . 136
5.2.4 Jumper Settings . . . 137
5.3 Supermicro's BMC and Server Management Interface . . . 138
5.3.1 SSH Shell . . . 139
5.3.2 BMC Firmware Analysis . . . 139
5.3.3 Firmware Upgrade . . . 140
5.3.4 IPMI I2C functionality . . . 142
5.4 Practical Experiments . . . 142
5.4.1 PMBus-based Voltage Control . . . 143
5.4.2 Enabling SSH Access and Firmware Repacking . . . 146
5.4.3 Attack Chains for PMBus Access . . . 147
5.5 Undervolting and Overvolting Attacks . . . 148
5.5.1 Undervolting Attack against Intel SGX . . . 148
5.5.2 Overvolting to Permanently Brick a CPU . . . 151
5.6 Evaluation of other Server Motherboards . . . 152
5.6.1 ASRock Power-Down Attack . . . 153
5.6.2 Other Supermicro X11 Motherboards . . . 155
5.6.3 Supermicro X12 Motherboards . . . 155
5.7 Conclusions and Countermeasures . . . 156

5.7.1 Server Platform Security and Embedded System Security 156
5.7.2 SGX Security 157
5.7.3 Countermeasures 157

6 Conclusion and Future Work **161**

A Appendix of MetaEmu **163**
A.1 Execution Trace visualised in Ghidra 163
A.2 Execution Observer Event API 164
A.3 Benchmark Programs 164
A.4 List of Automotive Firmware 165

B Appendix of Incision **167**
B.1 Huawei R216h 4G/LTE CoreSight Configuration 167
B.2 Analyzing the Huawei R216h Baseband with INCISION 169

C Appendix of Voltpillager **171**
C.1 Glitch Configuration and Results for Multiplication and CRT-RSA 171
C.2 Example Results for Faults during Memory Accesses 173
C.3 Details of SVID 174

D Appendix of PMFault **175**
D.1 `i2cdetect` Result for Supermicro X11SSL-CF 175
D.2 PMBusDetect Result for Supermicro X11SSL-CF 176

References **177**

Acronyms

AST Abstract Syntax Tree

BCM Body Control Module

CAN Controller Area Network

CBC Cipher Block Chaining

CFG Control Flow Graph

CMT Compare and Match Timer

CPU Central Processing Unit

CSTF CoreSight Trace Funnel

DFA Differential Fault Analysis

DMA Direct Memory Access

DoS Denial-of-Service

DRAM Dynamic Random-Access Memory

DRM Digital Rights Management

ECU Electronic Control Unit

ETF Embedded Trace Funnel

ETR Embedded Trace Router

FIVR Fully Integrated Voltage Regulator

FPGA Field Programmable Gate Array

GPIO General Purpose I/O

HAL Hardware Abstraction Layer

I2C Inter-Integrated Circuit

IC Instrument Cluster

IC Integrated Circuit

IL Intermediate Language

IoT Internet of Things

IRQ Interrupt Request

IR Intermediate Representation

ISA Instruction Set Architecture

ISR Interrupt Service Routine

JTAG Joint Test Action Group

LCP Longest Common Prefix

MCU Microcontroller Unit

MMIO Memory-mapped I/O

MSR Model Specific Register

µC Microcontroller

muC Microcontroller

OS Operating System

PCB Printed Circuit Board

PC Personal Computer

PMBus Power Management Bus

PoC Proof-of-Concept

PTM Program Trace Module

PWM Pulse Width Modulation

RAPL Running Average Power Limit

RTL Register Transfer Language

RTOS Real-Time Operating System

SDR Software Defined Radio

SDR Software-Defined Radio

SGX Software Guard Extensions

SMBus System Management Bus

SMT Satisfiability Modulo Theories

SPI Serial Peripheral Interface

SUT System Under Test

SVID Serial Voltage Identification

TCB Trusted Computing Base

TCU Telematics Control Unit

TEE Trusted Execution Environment

TPIU Trace Port Interface Unit

UART Universal Asynchronous Receiver Transmitter

μCode Microcode

UCODE Microcode

UDS Unified Diagnostic Services

USB Universal Serial Bus

VLE variable length encoded

VR Voltage Regulator

VXE Virtual Execution Environment

VID Voltage Identifier

OCP Over Current Protection

SEV Secure Encrypted Virtualization

SP Secure Processor (AMD)

PCI-E Peripheral Component Interconnect Express

BMC Baseboard Management Controller

I2C Inter-Integrated Circuit

VRM Voltage Regulator Module

PMBus Power Management Bus

SMBus System Management Bus

SVID Serial Voltage Identification

KCS Keyboard Controller Style

IPMI Intelligent Platform Management Interface

Intel ME Intel Management Engine

PCH Platform Controller Hub

EC Embedded Controller

GPU Graphics Processing Unit

BIOS Basic Input/Output System

UEFI Unified Extensible Firmware Interface

Chapter One

General Introduction

Microcontrollers have become more powerful and commonly used in IoT devices, cars and servers. In modern automobiles, there can be more than 70 Electronic Control Units (ECUs) that are responsible for controlling subsystems of the cars [5]. In each automotive ECU, there are multiple microcontrollers responsible for different tasks. The IoT devices that are empowered by microcontrollers are also used in daily life. In modern PCs or servers, the Central Processing Unit (CPU) is not the only component that can process data, there exist multiple other microcontrollers that run on different firmware to gather information and control the system. They can configure and monitor the on/off status of the system, voltage, power consumption etc. This makes the modern PC/server to be a type of IoT device as well. In addition, some of the microcontrollers or CPUs feature Trusted Execution Environment (TEE) that aims to provide a secure and isolated environment for confidential information to be processed without being altered by the service provider.

In order to implement these new features, computer systems have become more and more complex in both firmware and hardware design. More services are packed into the firmware which increased the software attack surface. If the system is not carefully designed, the exposure of connections between the components can also cause security vulnerabilities. Moreover, the large number of devices and hardware interfaces in the system makes security analysis a challenging task. Since the system is a mixture of soft-

ware/firmware and hardware, it requires both software/firmware and hardware analysing approaches to evaluate the security of modern computing systems.

1.1 Firmware Analysis

In firmware security research, Tereshkin and Wojtczuk [6], Goryachy and Ermolov [7], and *CVE-2017-5689* [8] showed that compromised firmware in subsystems of a PC can be used to take full control of the system and even cause firmware corruption and brick the system [9]. Similar vulnerabilities have also been found in automotive embedded systems. Tencent Keen Security Lab [10, 11] found that with a compromised ECU firmware, the attacker can remotely take control of the car. Attacks on both systems target the vulnerabilities in the firmware of the devices and show the importance of security analysis of firmware.

However, analysing embedded firmware is not an easy task. Firstly, unlike typical OS-based programs, the programs in the firmware are sometimes running directly on the hardware, which makes it bare-metal firmware. In bare-metal firmware, there is no clear division of functionality by program file, instead, all functions are packed into a single binary blob and stripped of the symbols. This makes it hard to identify the target functionalities for security reviews.

Secondly, the large amount of firmware in complex systems makes it infeasible to manually inspect every piece of firmware. To more efficiently detect bugs and security vulnerabilities, dynamic firmware analysis approaches such as fuzzing are required. A typical fuzzer will execute the program under test and feed it with different inputs. Based on the program behaviours, it will mutate inputs to either improve the fuzz coverage or provoke more crashes. However, dynamic analysis approaches require emulation support and proper execution context, does not come by default with microcontrollers and their firmware.

Microcontrollers can have different architectures and different peripherals function-

alities. Each different architecture has different program semantics and required the emulator to be implemented differently. This problem is especially prominent in automotive firmware analysis. The current state-of-the-art of firmware emulators such as QEMU [12] and Unicorn [13] support x86, ARM, and MIPS, however, there is rarely emulation support for esoteric architectures such as RH850, SH2A and other architectures which are commonly used in ECU. Besides, implementing the peripheral for every variant of device is a complicated task and requires a deep understanding of both the processor and emulator framework. Some of these peripherals are not useful for security analysis, but are still required to provide the context to execute the main functionalities of the firmware.

1.2 Attacks against Trusted Execution Environments

TEEs aim to protect the confidentiality and integrity of the code and data used inside it. It achieves this goal by providing an isolated environment for the program semantics to be executed faithfully. Different CPU manufacturers have different implementations of TEEs. Intel has Software Guard Extensions (SGX), ARM uses TrustZone, AMD has Secure Encrypted Virtualization (SEV). TEEs are commonly used in both servers and end-user devices. In servers, it aims to provide trust to the execution environment even if the service providers are malicious. In end-user devices, TEEs are often used by Digital Rights Management (DRM), which ensures videos can not be copied without permission. With the threat model of TEEs, hardware-based attacks are also in the scope of security analysis of server platforms. Fault injection and micro architecture based attacks are the most common ways that make use of vulnerability in hardware design to break TEEs.

1.2.1 Fault Injection Attack

Fault injection attacks actively change the execution of program semantics and cause errors. It was originally introduced by Boneh et al. [14]. This type of attack affects the

execution semantics by changing the operating environment of the IC. Early research on fault injection attacks are mainly targeted at microcontrollers [15, 16] and require physical access to the device. There are several approaches to conduct fault injection attacks, examples including voltage change, clock change, temperature change or exposing the IC to laser [17, 18]. However, later research showed that there are also software interfaces in microcontrollers which can be used.

With the discovery of software-based fault injection attacks against x86 systems in Rowhammer [19], the threat model of fault injection has changed. The attacker no longer needs physical access to the system to inject a fault. The software-based fault injection attack uses interfaces such as Memory-mapped I/O (MMIO) or special registers of a microcontroller to control the clock or voltage and inject a fault. This reduces the difficulty since the attacker does not need to design a complicated power supply system of a CPU to control the voltage. Plundervolt [20], V0LTpwn [21] and VoltJockey [22] all showed that by exploiting the power control interface provided by the x86 CPU itself, the attacker can inject faults and recover secrets from an Intel SGX enclave. There are also attacks that target the TEE of ARM microcontrollers – TrustZone, including CLKSCREW [23] and VoltJockey [24].

1.2.2 Micro-architecture Based Attacks

Most side-channel attacks and transient-execution attacks are pure software-based, which exploit the micro-architecture design of the CPU. Gyselinck *et al.* [25] shows that the memory access pattern can be revealed in page-level granularity by introducing a page fault. A similar attack can also be conducted with a more stealthy approach by monitoring page table access [26]. Wang *et al.* [27] introduced cross-enclave Prime+Probe and DRAMA attacks that enable fine-grained observation into the enclave. Besides, Moghimi *et al.* [28], Götzfried *et al.* [29], Moghimi *et al.* [30], and Schwarz *et al.* [31] shows that cache can also be used as a side-channel to leak information from the enclave.

In the meantime, transient-execution attacks against SGX were discovered. Fore-

shadow [32] shows that with Meltdown-type [33] attack, unprivileged users can extract secrets from the enclave. Later, ZombieLoad [34] discovered another attack that breaks SGX security guarantees even on foreshadow-resistant hardware. RIDL [35] shows that unprivileged attackers can make use of line feed buffer to extract secrets from other processes. LVI [36] attack makes the target program transiently compute on unintended data to expose secret information. All other attacks against SGX, including Koruyeh *et al.* [37] and Schaik *et al.* [38] exploit the side effect of the micro-architecture design of the CPU.

Clearly, there is a lot of research focusing on the software interface of x86 systems to either inject fault or use side channels to attack SGX. The software-based fault injection attacks against SGX show that the power delivery system on modern desktop/server motherboards can be used as a vector to attack the TEE, but the hardware design of the power control interface and its protocol has not yet been studied in depth[1]. Besides, due to the high complexity and the cost of developing tools to independently power up the CPU, hardware-based fault injection attacks to x86 systems used to be considered a nearly impossible task. However, it is still important to analyse the attack surface and possibility of such attacks in order to thoroughly understand the security guarantee provided by TEE in these systems.

1.3 Control Interfaces of Voltage Regulator Module

In modern computer systems, the voltage is controlled by an external IC called Voltage Regulator Module (VRM). The VRM is connected to the CPU to receive control messages via Serial Voltage Identification (SVID) bus. This interface can be controlled by software running in the Intel CPU by configuring a Model Specific Register (MSR). It provides a software interface to control the voltage. As with those mentioned in Sec-

[1]Inspired by the approach used in Chapter 4 VoltPillager, there is a hardware-based fault injection attack against AMD SEV [39] published in late 2021, which analysed the hardware design of the power control interface of AMD platforms.

tion 1.2.1, this interface is commonly used in software-based fault injection attacks to Intel SGX. The mitigation proposed for these vulnerabilities is a firmware update which disables this interface [40]. However, this mitigation is not enough in the threat model of TEEs, where the attacker has full control of the hardware.

Another interface that is commonly present in server VRMs is Power Management Bus (PMBus). This interface is based on System Management Bus (SMBus) protocol and uses Inter-Integrated Circuit (I2C) as physical layer for communication. With a set of semi-standardized PMBus commands, the user can control the voltage and current being supplied to the CPU. With these functionalities, PMBus can provide attackers with an interface to conduct fault injection attacks mentioned in Section 1.2.1 and side-channel attacks similar to the one discovered in Lipp *et al.* [41]. Besides, PMBus is usually connected to multiple other microcontrollers on the motherboard for better monitoring and controlling of the CPU voltage. This makes it a high-value target for conducting fault injection attacks, but there has rarely been any board-level security research of computer motherboards and security-orientated research about this interface.

1.4 Inter-device Attacks

The design of modern computers has become more and more complex. In addition to the CPU, there are multiple other microcontrollers on the motherboard that are responsible for different functionalities, which opens up a large attack surface [42]. On a desktop/server motherboard of an x86 system, there are usually Platform Controller Hub (PCH)(Chipset Controller), Embedded Controller (EC) and BMC for server management. They run their own firmware, which may have vulnerabilities inside. They are also connected and communicate with each other, which makes it possible for a single point of venerability to affect the whole system. These microcontrollers can also control other peripherals and IC in a computer system such as power supply, VRM, Graphics Processing Unit (GPU) or memory sticks via hardware interfaces. This essentially makes

modern computers an embedded system in the aspect of security analysis.

Different components in the system can affect the functionality of each other and lead to vulnerabilities. Previous attacks on automotive embedded systems [11] have shown the vulnerability of one component can be used as an entry point to get access to internal connection and control other systems via software exploit. A similar attack can also be conducted on x86 systems. Research on BMC discovered that Direct Memory Access (DMA) interface can be used to control the host CPU with shared memory page. With the DMA interface, an attacker can implement backdoor in the OS by overwriting unused kernel functions and hijacking `syscall` table. [43]. There are also security researchers who exploited this attack chain in reverse, Stewart Smith discovered the Pantsdown attack which uses DMA interface from host CPU to gain control of the BMC [44]. For other parts of the system, there are multiple vulnerabilities [45] discovered in Basic Input/Output System (BIOS) or Unified Extensible Firmware Interface (UEFI) firmware which can be used as a backdoor to access the system and change the system configurations.

Previous system-level security research on x86 platforms mostly focuses on exploiting software or firmware vulnerabilities. The strong attack model introduced by TEE is considering the system under the control of a malicious attacker. This means that the attacker can be the system manager who can conduct firmware upgrades of the BIOS and possibly remotely or locally manage the system via BMC. But research about microcontrollers on motherboards has been missed in the scope of hardware security research and attack surface of TEE in server systems. One example is that in Intel systems, the SGX attestation will verify if the Microcode and Intel Management Engine (ME) which comes with BIOS has been up-to-date to launch a secure enclave [46], but there are no mechanisms to verify the security of other components of the system such as BMC.

Besides, previous research of x86 platforms have been focusing on the security of firmware running in each individual component such as BMC [43, 47], Intel Management

Engine (Intel ME) [6–8] and the software-hardware interface provided in each component [22, 23, 48]. There is a lack of board-level security analysis that considers the system as a whole. The software-hardware interface of CPU can also be exposed by microcontrollers on the motherboard, even if the microcontroller does not have a security vulnerability on its own. Besides, the improper configuration of devices on the motherboard or improper wire layout can also cause security problems to the whole system.

1.5 Contributions and Overview of the Thesis

In this thesis, I start by introducing the background information of firmware analysis, trusted execution, fault injection and their relation in x86 server/PC system security. This thesis is composed of four published papers, in all the papers where I am the first author, I contributed to a significant amount of the work.

Chapter 2 and Chapter 3: Embedded Firmware Analysis

In the first part of the thesis, which is composed of Chapter 2 and Chapter 3, I will talk about the difficulties and proposed solutions for analysing esoteric firmware. MetaEmu [1] provides a novel universal emulation environment which is especially useful for analysing microcontrollers that are commonly used in automotive ECU. In this paper, I added architecture support for SH2A, RH850, V850 to the PCode lifter developed by my colleague. Then I tested those architectures with real-world firmware. I also extracted the firmware of RH850 from real-world ECU devices. Besides, I implemented three types of peripheral support for the MetaEmu. The first one is a peripheral solver which makes use of symbolic execution engine and bypasses the peripheral status checking. The second one is a peripheral hooking framework which can be used to implement arbitrary peripherals. Lastly, I implemented the universal peripheral backend, which allows users to easily implement common peripherals such as Compare and Match Timer (CMT) etc.

I also contributed to the case studies by implementing a peripheral solver on the V850E2 architecture; a CAN ID detector and CAN peripheral on RH850 architecture and CMT on SH2A and C166 architecture. With the fuzzing framework developed by my college, I implemented a prototype fuzzing program for the CAN protocol.

INCISION [2] introduced a method to find functions of interest in large firmware blobs. In this paper, I contributed to the development and evaluation of the region inference and grouping method proposed in the paper. METAEMU and INCISION greatly reduced the manual effort needed for analysing embedded firmware in computer systems.

Chapter 2 is based on the following publication: *METAEMU: An Architecture Agnostic Rehosting Framework for Automotive Firmware* [1] by *Zitai Chen*, Sam Thomas and Flavio D. Garcia. Chapter 3 is based on the article: *Cutting Through the Complexity of Reverse Engineering Embedded Devices* [2] by Sam L. Thomas, Jan Van den Herrewegen, Georgios Vasilakis, *Zitai Chen*, Mihai Ordean and Flavio D. Garcia.

Chapter 4. Hardware Based Fault Injection against Intel SGX

The hardware-based fault injection attack – VoltPillager [3] is introduced in Chapter 4. In this chapter, I will start the analysis of power delivery systems of Intel CPUs with SGX. I discovered and reverse engineered the SVID bus and the SMBus which is used by the VRM to control the voltage. With the protocol and the connection used to control the VRM reverse engineered, I developed the hardware-based fault injection tool with a Teensy 4.0 and injected packets to the SVID bus to control the voltage of the CPU. With two wires connected to the SVID bus, I conducted the first hardware-based fault injection attack against x86 systems. To evaluate the fault injection accuracy and discover the voltage range to inject fault to the CPU without causing crash, I also conducted more experiments with this tool. In these experiments, I also initially discovered a new delayed write fault, and my college did further work on developing Proof-of-Concept (PoC) and analysing this type of fault.

This chapter is based on the following publication: *VoltPillager: Hardware-based*

fault injection attacks against Intel SGX Enclaves [3] by *Zitai Chen*, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D. Garcia.

Chapter 5. PMBus Based Software Fault Injection with BMC

To further analyse the security of the x86 systems and the CPU power control interface, Chapter 5 dig deeper into the design of motherboards. This is where the software or especially firmware vulnerabilities meets with hardware-based voltage attacks. In this work called PMFault, I will talk about the importance of board-level security analysis of server systems. Firstly, I worked on analysing the design of the server motherboard and discovered that there is a PMBus which can be used to control the voltage of the CPU. Moreover, I also found that there is a BMC on server motherboards, which is connected to the PMBus. With deeper firmware analysing of the BMC, I discovered that with vulnerabilities found in BMC firmware, an attacker can launch a hardware-based voltage attack on the CPU via the PMBus interface of VRM. The attacker can either inject fault to the SGX enclave by undervolting or permanently brick the CPU by overvolting the CPU. These findings show that the security of the modern x86 systems would not only rely on the security of the CPU and software running in the CPU, but also rely on the design of motherboards and firmware running in the microcontrollers on these motherboards.

This chapter is based on the paper *PMFault: Faulting and Bricking Server CPUs through Management Interfaces* [4] by *Zitai Chen* and David Oswald.

In the last chapter, I conclude this thesis and suggest further research directions.

Chapter Two

MetaEmu: An Architecture Agnostic Rehosting Framework for Automotive Firmware

Zitai Chen, Sam Thomas and Flavio D. Garcia

This chapter is based on the following publication:

MetaEmu: An Architecture Agnostic Rehosting Framework for Automotive Firmware [1] which is published in Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)

2.1 Introduction

Automobiles fulfill a vital function in our societies—a means of transport—and, as with many technologies, have become increasingly connected, and at the same time more closed to scrutiny [49]. Kocher et al. [50] and Checkoway et al. [51] demonstrated that this increased connectivity introduces a significant attack surface, and, inspired by these seminal works, Miller and Valasek [52] showed that automobiles, like any other Internet-connected device can be remotely compromised. Meanwhile, Garcia et al. [53] and Verdult et al. [54, 55] showed that physical entry to a vehicle—something traditionally handled by a mechanical means—could also be defeated.

While researchers have sought to mitigate these flaws by protocol enhancements, e.g., Bulck *et al.* [56], analyzing the black-box firmware of the ECUs connected to a vehicle's internal network, has largely been left as a manual endeavor, e.g., Herrewegen and Garcia [57]. We posit that this is due to the fact that, while ECU firmware is not necessarily hard to obtain, tool support for performing anything more than manual analysis is severely lacking for the processor architectures they are based on.

Recent approaches enabling security analysis of embedded devices have primarily focused on rehosting, i.e., the process of transplanting a device's firmware to run inside a virtualized execution environment, e.g., QEMU [12] or derived tools, such as Unicorn [13], to enable specific analysis tasks. While these environments support a wide-range of commonly used embedded architectures, such as ARM and MIPS, they do not provide out-of-the-box support for arbitrary peripherals, or any support for esoteric architectures, such as those found in automotive components. At the time of writing, most published work has sought to address the former challenge: peripheral support. However, as noted by Fasano et al. [58] in their systematization of the field, for devices whose firmware is not supported by an off-the-shelf emulator, the latter challenge—obtaining a suitable execution environment—remains an open problem, hampering the analysis of a large and vital class of devices.

An orthogonal issue arising from the use of commodity emulators for rehosting is that they implicitly force a device-centric approach, where a device and its peripherals are assumed to operate with little or no constraints on the inputs they receive. For many scenarios where embedded devices are deployed, we believe that this simplification is unjustified, as evidenced in recent work by industry practitioners [10, 11]. In the automotive setting, for example, ECUs are interconnected by a Controller Area Network (CAN) bus, which they use to communicate. Many ECUs require the presence of other ECUs to successfully initialize and operate, hence, to realistically simulate a device's operating environment, we often require more than a single ECU to be rehosted.

Our contribution In this chapter, we address the challenge of analyzing devices that cannot be rehosted using commodity emulators. We present our framework, MetaEmu, that takes as input widely available processor and instruction set definitions—Ghidra's language definitions, and automatically synthesizes virtualized execution environments capable of rehosting multiple devices of different architectures simultaneously. Our synthesized environments enable deep introspection of each rehosted firmware's state and facilitate complex inter-device analyses.

We demonstrate that our approach is general (it can support rehosting firmware of many esoteric architectures, irrespective of their peculiarities), scalable (it can analyze many instances of a firmware in parallel and share analysis facts), and enables dynamic analysis of a class of devices that until now have been largely overlooked.

To evaluate MetaEmu, we benchmark its performance and provide six case studies detailing its use for different kinds of security analysis, including backdoor detection, inter-device analysis, and various automotive-focused reverse engineering tasks. For these analyses, we implement a fuzzer (by integrating MetaEmu with LibAFL [59]), a symbolic executor, which we use to solve peripheral access checks, a CAN bus reverse-engineering tool, and an inter-firmware coverage tracker. Our case studies demonstrate that MetaEmu is effective in analyzing complex, binary-only firmware, without any architecture support or performance trade-off. Our evaluation data-set consists of thir-

teen benchmarks and six device firmware: two based on open-source SDKs and four extracted from automotive ECUs: a Body Control Module (BCM), an IC, and two Telematics Control Units (TCUs). Each end-user firmware is based on a different CPU architecture (Infineon C166 and Renesas RH850, SH-2A, and V850E2M-M), and each presents a different set of challenges for emulation and analysis.

To summarize, our work contributes to the state-of-the-art in the following ways:

1. We present, to the best of our knowledge, the first generic framework for rehosting and dynamically analyzing end-user automotive firmware, and the first framework supporting device inter-dependent analysis of multiple rehosted firmware executing simultaneously.

2. We show that our framework is capable of synthesizing Virtual Execution Environments (VXEs) for multiple esoteric architectures, generically enabling analysis of a vital class of devices currently unsupported by any other dynamic analysis framework.

3. We show how our framework enables complex analyses, such as symbolic execution and fuzz testing, to be applied to binary-only automotive firmware.

4. We provide an in-depth evaluation of MetaEmu, with respect to performance, implementation flexibility, and real-world usability. It spans five different architectures, bare-metal and RTOS-based firmware, and three kinds of automotive ECU from four distinct vendors.

The latest version of our framework, processor definitions, and firmware data-set will be released as open-source under MIT license[1]. We name the core lifter and optimizer as Fugue and it is available at: https://github.com/fugue-re/fugue-core. We name the latest simulator and introspector MetaEmu and it will be released soon. An older version of MetaEmu is available at https://github.com/fugue-re/fuguex-core

[1]Please check at https://github.com/fugue-re for update

2.2 Background

In this section, we provide the necessary background to understand our contributions. We first provide an overview of the challenges involved in analyzing automotive components, and then cover the terminology used in the rest of this chapter.

A modern vehicle is composed of multiple ECUs connected by a CAN bus. Each ECU may have multiple peripherals, and often, each will be manufactured by a different vendor, and may be based on a different CPU architecture. In contrast to the majority of devices analyzed under the umbrella of "embedded device security" in the literature, ECUs are generally based on more esoteric architectures, e.g., V850/RH850, or C166, rather than ARM or MIPS. Unfortunately, these architectures are not supported at all by the de facto emulation environments for security analysis—QEMU and Unicorn. This makes it impossible to apply modern security analysis approaches such as symbolic execution to these devices. While one might assume that this challenge can be addressed by adding support for new architectures to those tools, doing so requires an unreasonably large amount of engineering effort: 1000s of lines of code and many months of development time—an unjustifiable amount of work for most applications. While Ghidra has `EmulatorHelper` which can be used to emulate some of these architectures, it does not have an easy-to-use interface for peripheral handling or adequate performance for intensive analyses.

The tangential problem of peripheral support is a significant challenge in the automotive context: on the one hand, due to the variety and number of peripherals a single firmware might interface with, and on the other, due to a lack of documentation. For some analysis tasks, peripheral interactions can be *bypassed* by providing a satisfying value which will work even with limited information about the peripheral, as is done in previous work [60–62]. However, such a bypass is often no better than (unsoundly) forcing execution of a given branch target. This is because the *real* constraints on the peripheral register might depend on another device's output, or be constrained outside

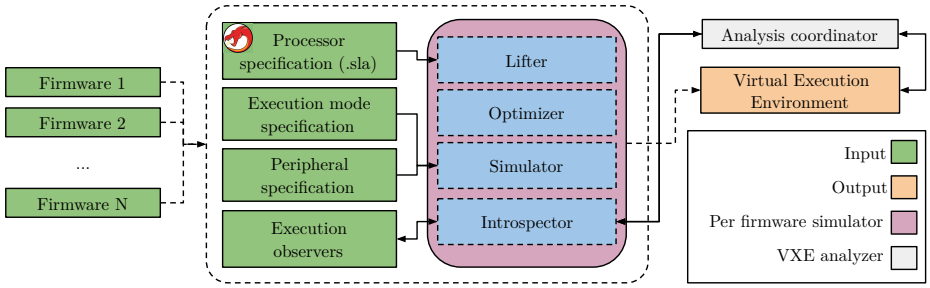


Figure 2.1: Overview of MetaEmu. Each firmware along with corresponding processor, execution mode, and peripheral specifications are used to synthesize a simulator. Each simulator (pink box) can have execution observers attached, which monitor and manipulate its execution state and interface with an analysis coordinator (gray box) to facilitate inter-firmware interaction and analyses. The resulting VXE (yellow box) is composed of many per firmware simulators operating in parallel.

the execution path being analyzed. An obvious example of such a peripheral is a CMT, which is often used by firmware to implement task scheduling. Clearly, providing *any* satisfying value for the peripheral's output will lead to undesired behavior, and is better handled by a modeling approach, such as that proposed by Gustafson et al. [63]. Thus, support for peripheral models and the ability to have multiple rehosted firmware interact and communicate is vital to ensure faithful simulation.

MetaEmu provides a means to rehost one or more device firmware inside a virtual execution environment. We use the term rehost to jointly refer to the process of transplanting a device's firmware into an emulator, and the simulation of its execution and interaction with its peripherals. In this work, we refer to the rehosting environment as a VXE, and assume that a VXE facilitates multiple firmware to coexist and interact with each other. MetaEmu synthesizes VXEs; we distinguish a synthesized VXE from a standard VXE, such as those built on top of Unicorn [13], e.g., Ruge *et al.* [64] and Maier *et al.* [65], by how they are specified: synthesized environments are specified using a declarative approach, rather than purely programmatically.

2.3 System Overview

In this section we provide an overview of MetaEmu, its inputs, and the assumptions we make about the firmware it can be used to rehost.

2.3.1 Framework Architecture

We summarize MetaEmu's architecture in Figure 2.1. MetaEmu takes as input one or more binary blob firmware and outputs a combined simulation/analysis environment capable of rehosting all the input firmware within a single VXE. In the combined VXE, each firmware has its own simulator (pink box in Figure 2.1). To synthesize these simulators, MetaEmu requires four inputs: ❶ a processor specification, ❷ an execution mode specification, ❸ a list of execution observers, and ❹ a peripheral specification. To dynamically introspect and manipulate the firmware's state, MetaEmu attaches user-defined *observers* to the simulator, which can be triggered on, e.g., register reads and memory writes. The attached observers enable us to handle complex addressing modes (e.g., that used for the C166 architecture in Section 2.5.2) and implement symbolic solvers for peripheral checks (similar to μEmu [61] and Laelaps [60]). By interfacing with the analysis coordinator (gray box), observers can communicate with other rehosted devices (e.g., allowing one device to provide input to another's peripherals) and share analysis facts, enabling inter-device analyses.

Framework Inputs

MetaEmu is capable of rehosting many different types of devices; from those with bare-metal firmware to those based on embedded RTOSes, such as ThreadX. Regardless of the type of firmware being rehosted, we make the following assumptions:

1. We do not have access to the firmware's source-code.

2. We know its basic memory map and architecture.

3. We know the memory regions it uses for memory-mapped I/O with its peripherals.

4. We have a Ghidra language definition for its architecture.

Our assumptions are consistent with existing approaches, such as P^{2}IM [62, 66] and μEmu [61, 67], which also require manual specification of the memory mappings for each target firmware/device.

In addition to the firmware to rehost, MetaEmu requires four auxiliary inputs to synthesize a VXE (shown in green in Figure 2.1).

Processor specification To disassemble and obtain an architecture independent representation of each firmware, MetaEmu leverages Ghidra's language definitions. These consist of a *pspec* which defines register name and basic memory mappings, a *cspec* which defines calling conventions and size information of primitive types, and a *slaspec* which describes how to disassemble the architecture's instructions and an encoding of their semantics. At the time of writing, there are 28 processor families supported in Ghidra master branch and many others created by the community.

Execution mode specification MetaEmu can simulate each firmware's execution using different strategies (further detailed in Section 2.4.2), including concrete execution and concolic execution. Each specification defines how MetaEmu simulates firmware at the level of individual Intermediate Language (IL) operations and how we represent the firmware's state. All mode specifications implement a common interface and each synthesized firmware simulator is parameterized by a generic type constrained by that interface, allowing it to be instantiated to operate under any execution mode.

Execution observers To extend each synthesized simulator to support architectural nuances, peripherals, interrupts, and user-defined analyses, we use so-called observers. As discussed later in Section 2.4.3, our observers intercept events during simulation specific to the simulator's execution mode, and, in response, can modify the firmware's state and coordinate with other simulators and observers through a VXE-shared *analysis coordinator*.

Peripheral specification As detailed later in Section 2.4.4, MetaEmu supports peripherals through generic and user-specified models (Section 2.4.4), and automatic model inference (Section 2.4.4). We specify all types of peripheral using execution observers, which we attach to simulators via their introspection interface. This interface is flexible and can support peripherals that receive input from outside of a VXE, enabling us to interact with rehosted firmware via SocketCAN, fuzzers such as AFL [68], and even physical devices.

2.3.2 Challenges Rehosting Complex Firmware

Fundamentally MetaEmu is a multi-target emulation framework purpose-built to facilitate complex security analysis that can be customized and extended using a specification-based approach to alter execution policies, architecture definitions, and peripheral models. We encountered several challenges during the design and implementation of MetaEmu. In this section, we provide an overview of these difficulties and how we have addressed them.

Architecture support and IR optimization To facilitate architecture agnostic analysis, MetaEmu emulates a firmware by first translating its architecture-specific instructions to an intermediate representation, and performs emulation of the resulting Intermediate Representation (IR) operations. The emitted operations are based on Ghidra's SLEIGH language definitions for the target's architecture. These definitions produce an IR representation that is not well suited for emulation, as it is overly verbose, as shown in Figure 2.2a. This is particularly problematic when using a symbolic execution policy, as a more complex IR leads to more complex symbolic formula. To improve the performance, we implement a IR optimizer in our lifter, which can produces a much more optimal IR representation, as shown in Figure 2.2b. We use IL to refer to the language of the IR and use IR to refer to the actual intermediate code generated by the MetaEmu from the binary.

While our lifting and optimization approach is mostly standard, i.e., optimizations

are applied on the SSA form representation of our IR, due to our use of SLEIGH specifications to generate our IR, we encountered a number of additional challenges in our implementation. The first being that Ghidra's language definitions allow for variables (both registers and temporaries) to overlap. SLEIGH represents each variable as a $(offset, size)$ pair that indexes into a flat address space, e.g., the register `RAX` might have offset 0 and size 8, while `EAX` might have offset 4 and size 4. This complicates SSA transformation, as it assumes that variables do not overlap. We thus, modify the standard algorithm to treat overlapping variables as equivalent. A further challenge is that SLEIGH's temporaries have a liveness that only spans a single instruction's IL operations. Hence, the same temporaries may be reused between consecutive instructions. We adapt the standard liveness data-flow analysis to account for this implicit lifetime information by providing unique labels for temporaries across instructions, thus preventing data-flow information propagating over instruction boundaries.

Missing analysis state and context When emulating embedded firmware, we often have to deal with missing or inconsistent state. For example, the values of a device's peripheral status registers, or global variables that may be have been set to particular values when the firmware was dumped, which prevent the firmware executing correctly when starting emulation from another location. Under these circumstances, the emulated firmware will usually stall in a check-and-wait loop. To solve this problem automatically, we implement a peripheral check solver, which ❶ identifies *looping* states and ❷ updates the firmware's context with suitable values to allow execution to proceed. The main difficulty is how to detect such states during emulation. A naive approach would be to track and count the number of times blocks of operations are executed and if a particular block is executed repeatedly over a threshold number of times, we assume that the emulated firmware has entered a stall-state. While such an approach works, in practice, it is both inefficient and requires a significant amount of resources. To efficiently address this in MetaEmu, we use a state machine-based approach, shown in Figure 2.4 to detect stall-states based on both code and memory access patterns, which reduces the

amount of program state to track simultaneously.

While our peripheral solver can be used to bypass checks that do not impact the behavior of the target firmware with respect to a particular analysis, when the data or functionality of the peripheral does matter for a given analysis, we cannot simply bypass a stall-state check with *any* satisfying state configuration. Examples of this include the CAN peripheral of our RH850-based firmware (Section 2.5.2) and the CMT of our C166-based firmware (Section 2.5.2). In such scenarios, the peripheral's behavior needs to be explicitly modeled. As even SoCs from the same family have slightly different implementations of such peripherals, and different kinds of context switching logic, it is both error-prone and time-consuming to implement versions of the same peripheral for different devices. To address this issue, we provide a universal peripheral backend (Section 2.4.4) and interrupt model. They abstract the peripheral behaviors and provide building blocks for implementing peripherals for different devices. Our interrupt framework enables a means to rapidly implement interrupt status tracking, context switching, and interrupt handler hooks and overrides.

Inter-device analysis As mentioned in Section 2.2, a micro-controller may rely on the data from other chips or devices to function properly. There are two sets of challenges faced when performing such inter-device analysis. Firstly, devices being rehosted in the same VXE may come from different manufactures, have different methods of handling peripherals, and may be based on different architectures. Secondly, the devices may lack a documented shared channel for communication, that would otherwise be facilitated by *black-box* hardware in a real-world setting. In MetaEmu, we overcome the first set of challenges by providing a wide-range of architecture support and three methods to implement peripheral models. We address the second challenge by providing an analysis coordinator that facilitates message passing using a common interface between both device's emulated peripherals.

2.4 Implementation of MetaEmu

```
// mov dword [RBP - 0xc], ECX
var0620 := RBP + 0xfffffffffffffff4
var1790 := ECX    // redundant variable var1790
*var0620 := var1790
CF := 0x0
OF := 0x0
// xor ECX, ECX
ECX := ECX ^ ECX           // ECX is constant value 0x0
RCX := zext(ECX, 64)       // Zext of 0 is constant 0x0
SF := ECX s< 0x0           // Constant 0x0
ZF := ECX == 0x0           // Constant 0x1
var2580 := ECX & 0xff      // Constant 0x0
var2590 := popcount(var2580)//number of 1s in 0 is 0
var25a0 := var2590 & 0x1 // Constant 0x0
PF := var25a0 == 0x0       // Constant 0x1
```

(a) IR produced by Ghidra's lifter. The highlighted lines indicate instructions that can be simplified.

```
var0620 := RBP + 0xfffffffffffffff4
*var0620 := ECX
CF := 0x0
OF := 0x0
ECX := 0x0
RCX := 0x0
SF := 0x0
ZF := 0x1
PF := 0x1
```

(b) Optimized IR produced by MetaEmu for the sequence in Figure 2.2a.

Figure 2.2: Generated IR for the x86-64 instructions: `mov dword [RBP - 0xc], ECX; xor ECX, ECX`. The optimization to "`mov dword [RBP - 0xc], ECX`" eliminates the intermediate variable `var1790`. The optimization to "`xor ECX, ECX`" uses an identity for XOR, i.e., $v \oplus v = 0$. Our optimizations are *safe* and preserve both memory and register side-effects.

In this section, we describe MetaEmu's key features in depth and detail how they can be used to analyze rehosted firmware. MetaEmu is written entirely in Rust and spans ~40kloc (excluding comments and dependencies). This includes ~24kloc for *IR lifting and optimization*, ~5kloc for *IR simulation and observers*, and ~11.5kloc for *symbolic execution and other execution policy backends*. We do not rely on Ghidra's code-base for any functionality, other than its XML-based processor specifications.

2.4.1 Lifting & IR Generation

To support emulation in an architecture agnostic way, MetaEmu operates on an IR that explicitly models all processor instructions and their side effects. We represent each architectural instruction as one or more IL operations, which we obtain by a process called lifting, i.e., the translation of bytes into IL operations. Our IR is inspired by Ghidra's [69] IR, P-Code [70]. In contrast to P-Code, our IR uses two isomorphic IL encodings: a SSA-form expression-based variant, which we use for optimization, and a

Register Transfer Language (RTL) variant, semantically equivalent to P-Code, which we use for emulation. Similar to P-Code, our IR supports arbitrary extensions via intrinsic operations, which we use to model dynamic processor state changes, such as localized address mode switches.

Our lifter takes as input unmodified Ghidra language definitions, which contain information about calling conventions and the necessary information to lift a byte stream into our IRs. This approach enables us to make use of the vast number of processor architecture definitions from Ghidra and greatly reduces the workload of adding support for new architectures to MetaEmu. Inspired by B2R2 [71], our lifter is completely parallel, and supports lifting chunks of the same firmware across many cores.

While one might assume we could have used Ghidra's lifter as a basis for MetaEmu, unfortunately, it does not perform any optimization of the IR it generates—it simply produces a literal translation of the specification for each instruction into IL operations—which we find leads to as much as 25% of the operations emitted being superfluous (Section 2.5.1). Clearly, naively interpreting IL operations emitted from Ghidra's lifter will be inefficient, and for analyses such as symbolic execution will lead to much more complex formula due to extraneous operations. Figure 2.2a provides a taste of just how inefficient the generated IR is.

IR Optimization

In MetaEmu, we address these inefficiencies in two ways. First, within our lifter, we rewrite and optimize our IR prior to it being emitted and processed by our simulator. Second, we cache optimized IR blocks, both on disk and within a cache shared among all executor instances of the same firmware, which enables us to amortize the overhead of performing our optimizations. Accordingly, MetaEmu lifter can produce a much more optimal IR representation, as shown in Figure 2.2b.

We optimize IR blocks by rewriting and simplification. To do so, we leverage e-

graphs[2] [72] to perform equality saturation [73] on the SSA-form statements of each IR block after applying rewrite rules. This enables us to obtain the most optimal representation of each statement with respect to our rewrite rules. For e-graph construction and manipulation, we use `egg` [74].

We use our SSA-form expression-based IR, to perform our optimizations, rather than our RTL-based IR, to ensure the correct application of our rewrite rules in the presence of instruction-local control-flow. Such control-flow occurs when the IR corresponding to an architectural instruction requires loops or branches to model the architectural semantics fully (e.g., `rep stosb`). Thus, an IR block is more akin to a sequence of instruction-level control-flow graphs, rather than a strict basic block.

After obtaining the SSA-form representation, we transform each statement into an e-graph, merge it into a block-level e-graph, and apply rewrite rules which cover algebraic identities and simplifications. Upon adding a new statement to the block-level e-graph, we greedily apply rewrite rules, and extract its most optimal representation with respect to Abstract Syntax Tree (AST) depth (i.e., the shallower, the better). By extracting statements in this way, we preserve the original IR ordering, and by constructing an e-graph for the entire block incrementally, we preserve the equivalence classes discovered, allowing each statement to benefit from any rewriting possibilities applied to its predecessors. Finally, we use liveness analysis [75] to identify redundant assignments and remove them.

Our optimizations preserve all side-effects to memory and registers encoded in our unoptimized IR and therefore will not negatively impact the accuracy of any downstream analyses performed. To the best of our knowledge, our use of e-graphs for optimizing IR for emulation is novel. In evaluating our approach (Section 2.5.1), we find that through IR optimization and designing our emulator specifically for rehosting, we achieve more than 400% performance improvement over Ghidra's lifter and emulator.

[2] An e-graph is a DAG representing IL terms combined with a congruence-closed equivalence relation over said terms.

2.4.2 Firmware Simulation

At the most fundamental level, the simulators MetaEmu generates load an input firmware, translate its instructions into our IR, and interpret each emitted IL operation. Within the output VXE, each simulator runs in parallel on a separate thread, and communicates using a shared analysis coordinator.

Execution Modes & Policies

The method each simulator uses for interpretation depends on the execution mode or policy it was synthesized with. In contrast to traditional emulation environments, MetaEmu supports a variety of execution modes to simulate the execution of a rehosted firmware. Among other scenarios, these modes allow us to address the lack of complete peripheral models and support analyses bootstrapped using incomplete execution contexts, which often occur when analyzing firmware extracted from end-user devices. Each policy also influences the kind of observations that attached execution observers (Section 2.4.3) can make and the underlying representation they act upon. We support five modes:

❶ **Concrete execution** follows the standard semantics of the firmware's Instruction Set Architecture (ISA). It represents state as a collection of byte arrays, and updates the state by directly interpreting each IL operation in a step-wise manner.

❷ **Concolic execution** [76] permits us to mark some input variables, i.e., memory locations, as symbolic, while treating the remainder as in concrete execution. We obtain concrete assignments for symbolic variables by querying the Satisfiability Modulo Theories (SMT) solver, Boolector [77]. By negating the path constraints associated with a branch, we can obtain an assignment that allows us to explore both branch targets. Under this policy observers can inject symbolic values into the firmware's state.

❸ **Forced execution** [78] follows concrete execution semantics, with the caveat that at conditional branches, we can "flip" the branch condition, such that the non-satisfying

branch target is taken.

❹ **Micro execution** [79] follows concrete execution, with the exception our observers can intercept invalid memory accesses, map the locations into our VXE state, and populate them by an analysis-dependent policy.

❺ **Flood execution** [80] follows forced execution, with the additional property that all branch targets, regardless of their satisfiability, are explored in accordance with an exploration policy, e.g., to a fixed bound or threshold branch coverage. In MetaEmu we implement a variant of flood execution based on micro execution, as proposed by Gotovchits et al. [81]. Under this policy, observers witness events from multiple execution paths simultaneously.

Architectural Nuances

To facilitate non-standard addressing, dynamic processor mode switches, and other architectural quirks, we provide an interface to implement IL intrinsic operations. This interface enables MetaEmu to support functionality that cannot be directly modelled in our IR, e.g., instructions for cryptographic primitives, as well as modify how low-level operations such as memory reads and writes are performed. We use this interface to support the *DPP override mechanism* address scheme for the C166 architecture [82], which requires us to alter how the simulator resolves memory accesses for a variable window of instructions.

2.4.3 Observers & Analysis Coordination

To extend our specification approach to support peripherals (Section 2.4.4), arbitrary extensions to our generated simulators (Section 2.4.5 and Section 2.4.6), and different kinds of analyses (Section 2.5.2), we use a concept called *execution observers*. Observers *attach* to our simulators and register to receive event notifications via each simulator's *introspector* (bottom blue box in Figure 2.1). In response to events, we can configure

observers to manipulate the running firmware's state and/or interface with the VXE *analysis coordinator* (gray box in Figure 2.1) to enable inter-device communication or analyses.

As noted in Section 2.4.2, the kind of events an execution observer can witness, and the firmware state representation it operates on, is tied to the *execution policy* of the simulator it is attached to. However, regardless of the execution policy a simulator is synthesized with, we can register observers to receive event notifications for a number of basic event kinds. These include: **register reads/writes**, **memory reads/writes**, **program counter changes**, **conditional branches**, and **function calls**. We associate each event kind with a corresponding response kind, which we use to facilitate common rehosting tasks, such as: instruction skipping, conditional branch flipping, function call replacement, and forking the simulator's execution state. We provide an exhaustive list of event kinds in Appendix A.2.

Inter-device analysis

Our analysis coordinator provides a generic message passing interface that routes arbitrary event notifications between each simulator. It also acts as an abstract inter-device communication channel, that allows each rehosted device to communicate with any other device inside the same VXE. For example, it enables us to easily connect devices by their Hardware Abstraction Layer (HAL) APIs, without resorting to manually implementing low-level cross-device communication channels using emulated peripherals.

2.4.4 Peripheral Support

As discussed in Section 2.2, correct and faithful firmware simulation depends on providing some degree of peripheral support. When rehosting, this support is still required: for example, to supply input to the firmware when fuzzing, or to ensure the firmware can execute at all if it is based on a RTOS.

In embedded firmware, peripheral interactions are frequently performed using MMIO.

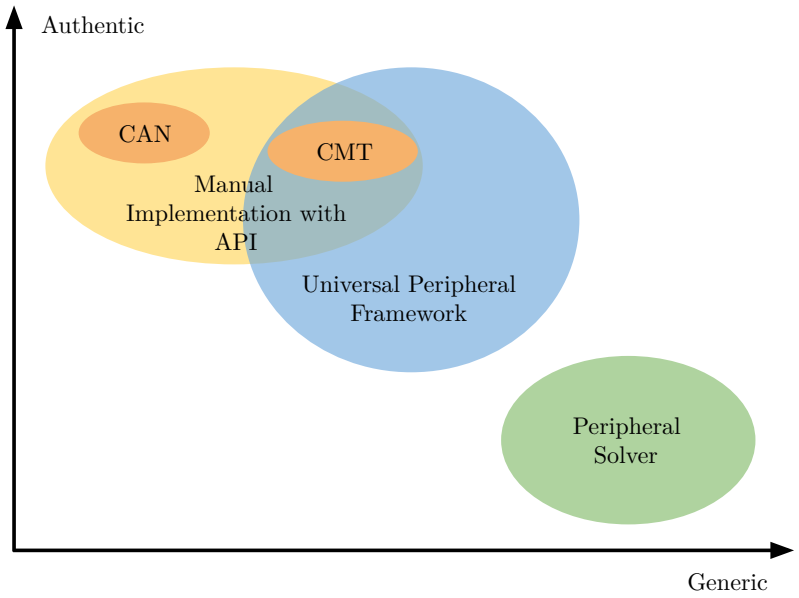


Figure 2.3: Types of peripheral support implemented in MetaEmu. ❶ Peripheral Bypass (green) can be used to handle peripherals that only interact with firmware in a limited way—e.g., via status checks. This kind of peripheral handling does not faithfully model real-world peripheral behavior, but can be used to automatically bypass checks that would otherwise stall emulation. ❷ Universal Peripheral models (blue) can be used to simulate *fundamental* peripherals, e.g., timers, in hardware-agnostic manner, but require some manual configuration, hence are less general, but provide more true-to-hardware behavior. ❸ Complete peripheral emulation (yellow) can be used to handle complex peripherals, such as those that perform I/O, e.g., over CAN, which require true-to-hardware behavior. These models cannot generalize beyond specific device families, and their behavior cannot be automatically inferred.

To facilitate this, such firmware has a continuous memory region allocated specifically for MMIO, which is divided into smaller regions for each peripheral. These regions contain memory mapped versions of a peripheral's control and status registers, as well as registers facilitating data transfers between the firmware and peripheral. This presents two challenges when rehosting firmware: ❶ When executing, firmware frequently checks its peripherals' statuses. If the status of a peripheral does not match what the firmware expects, e.g., due to being uninitialized, execution will "stall" in an infinite loop until the peripheral's status changes. Thus, without correct support, execution cannot proceed past such checks. ❷ A firmware's control-flow is often dependent on data read from its peripherals. For example, for ECUs that communicate using Unified Diagnostic

Services (UDS), the specific service handler executed will be determined by data received via a CAN peripheral. Hence, without peripheral support, it is impossible to faithfully explore such execution paths.

To overcome these obstacles, we provide three types of peripheral support in MetaEmu, as explained in Figure 2.3. They are all based on execution observers (Section 2.4.3). The first, inspired by Laelaps [60], is a **generic Peripheral Check Solver** that allows us to specify a MMIO address range and automatically bypass simple checking loops that would otherwise cause execution to stall. The second, is a ***Universal* Peripheral framework**, which provides a specification-based approach to build common peripherals by composing generic models. These models are *architecture-agnostic*, so can be reused for many different devices, with minimal reconfiguration. MetaEmu provides models to act as generic input sources, e.g., for fuzzing or to facilitate inter-firmware communication, timers of various kinds, and models that allow a peripheral to be associated with one or more interrupts (discussed in Section 2.4.5). Finally, as some peripherals cannot be implemented using automated or generic approaches, but are essential to enable security analyses, e.g., complex I/O devices for communication over CAN, we provide an observers-based API (Section 2.4.3) to **manually implement peripherals**. We demonstrate this in Section 2.5.2 by implementing CAN network interface controller.

Inferring Expected Peripheral Behavior

Our peripheral check solver can be applied in many rehosting scenarios, as firmware often only performs comparisons against a peripheral's status register to ensure it is initialized. These checks usually occur as tight loops whose exit condition depends on certain bits of a MMIO register being set or clear, similar to Listing 2.1. They generalize to the following: ❶ the firmware reads the peripheral's MMIO register, ❷ it checks the value read using a boolean predicate, ❸ if the predicate yields false, the loop repeats, otherwise execution continues past the check. We use this pattern to identify peripheral checking loops.

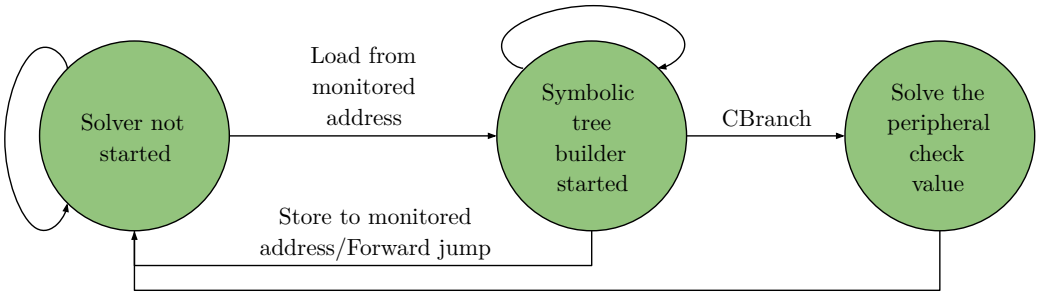


Figure 2.4: Peripheral model inference to bypass status checks. Our solver monitors loads from MMIO register addresses and builds an expression tree of the constraints imposed upon the value read. Upon reaching a conditional (`cbranch`), it uses SMT solver to compute an assignment for the register that allows us to exit the checking loop.

```
1 do {
2    mmioValue = read_4(0xfff81104);
3 } while ((mmioValue & 4) == 0);
```

Listing 2.1: Example of a peripheral checking loop.

Bypassing status checks To compute *satisfying* values to exit checking loops, we use localized symbolic execution, as shown in Figure 2.4. Our technique works by monitoring and symbolizing read accesses to our configured peripheral MMIO region. For each access, we track its data-flow and build an expression tree capturing the constraints imposed upon the read value. Upon reaching a conditional branch that is dependent on our read value, we use a SMT solver [77] to find an assignment that enables us to explore the "not taken" branch target. We then execute the branch; if it leads us to re-enter of the checking loop, we replace the value read from the MMIO register with the value computed using the solver.

Universal Peripheral Models

Building support for common peripherals for many different devices is both tedious and error-prone. Thus, it is desirable to build generic peripheral models instead. However, different Microcontroller Units (MCUs) use different MMIO registers and bit masks to perform similar tasks, complicating the implementation process. For example, to control

the start/stop of a CMT, the C167CR MCUs use bit 6 at address `0xff42` [82], while SH-2A MCUs use bit 0 at address `0xfffec000` [83]. Our *universal* peripheral models overcome this difficulty generically.

Within MetaEmu, we support *universal* peripherals using two methods. ❶ We provide an interface to intercept and redirect any function call, e.g., HAL API calls. This enables MetaEmu to handle many kinds of I/O-based peripherals using generic handlers, without concern for the low-level hardware details. ❷ We provide memory/register read/write hooking, which enables us to model MMIO-based peripherals generically. Our MMIO peripheral models implement data and status handling logic using a common interface that enables them to be used with our universal peripheral framework. Our framework then allows us to configure each model declaratively by specifying the address, bit masks, and handler functions associated with a given peripheral's MMIO registers. To facilitate model composition and architectural quirks, we provide a means to override each model's handler functions, which enables us to modify and combine their behaviors arbitrarily. Listing 2.2 demonstrates how the CMSTR register of SH2A CMT can be configured using our generic `CompareMatchTimer` model.

```
1 let mut cmt = CompareMatchTimer::default();
2 cmt.map_function_addr_read (ADDR_CMSTR, 0x01, &CMTFun::is_enabled);
3 cmt.map_function_addr_write(ADDR_CMSTR, 0x01, &CMTFun::set_enable);
```

Listing 2.2: Configuration of a Compare & Match Timer model using our universal peripheral backend.

2.4.5 Interrupt Support

While many peripheral interactions occur synchronously, i.e., exclusively via MMIO, others happen asynchronously. A common example is the *tick* of a timer peripheral used in RTOS-based firmware to implement task scheduling. Such asynchronous events manifest in firmware via interrupts, which are handled by inducing a context switch to a so-called Interrupt Service Routine (ISR). This type of context switch is almost always

facilitated by hardware, and works by first saving a snapshot of the current execution context to a vendor-defined area of memory, and then redirecting control-flow to the interrupt handler routine. To return from an ISR, either the interrupt routine or the micro-controller itself will be responsible for restoring the execution context depending on the implementation of the vendor. These type of interrupts are commonly used in event driven firmware, where most functionality is implemented via ISRs. To handle this diversity in MetaEmu, we provide a generic interrupt handling framework, implemented using observers.

Similar to our universal peripheral backend (Section 2.4.4), our interrupt handling backend is also specification-based. It enables us to specify an interrupt by configuring handlers for its triggering behavior and restoration logic. Within MetaEmu, we provide a number of *default* interrupt behaviors, e.g., to associate an interrupt trigger with a peripheral, and to perform context restoration based on specified registers or memory ranges. Our backend tracks each configured interrupt's *enabled* and *triggered* status, the ISR or analysis-specific functionality to perform when it is triggered, and the context restoration logic to execute after it has been handled. We provide a complete example in Section 2.5.2.

With our peripheral model, we can reduce the manual effort for implementing a common peripheral from scale of a few days to a few hours.

2.4.6 Integration with External Tools

As highlighted by Fasano et al. [58], rehosting is an iterative process that requires human intervention and debugging. To that end, we integrate MetaEmu with widely used binary analysis and reverse-engineering tools, to reduce the manual effort involved.

Trace visualization MetaEmu captures execution traces in a format supported by the widely used Binary Ninja and IDA Pro trace visualization plugin, Lighthouse [84]. As part of MetaEmu, we developed a Ghidra plugin providing comparable functionality. These plugins provide an intuitive view of the control-flow taken by our simulators, and

can be used to quickly diagnose the cause of unexpected behavior and pinpoint the firmware regions responsible. An example is shown in Appendix A.1.

Peripheral register name mapping When reverse engineering end-user firmware, we often encounter hundreds of functions that reference memory mapped peripheral registers. Unfortunately, tools such as Ghidra cannot automatically recover the names of all of these registers, and will often render them as raw addresses. When a firmware references many peripherals, this can be a significant hindrance. Thus, to aid our understanding of peripheral behavior when rehosting, we provide a peripheral register mapping plugin. Our plugin extracts a mapping of peripheral register names to addresses from C-header files distributed with vendor SDKs, which can be applied to Ghidra databases.

Fuzzing and coverage reporting Fuzz testing is fast becoming a de facto means of discovering security flaws in many kinds of software. LibAFL [59] provides a robust base for constructing fuzz harnesses in a modular way. We use LibAFL's `Executor` trait to implement support for fuzzing firmware rehosted in our VXEs. We report coverage to LibAFL using an execution observer, and mimic LAF-Intel-style [85] comparison splitting on our generated IR, which enables us to feedback finer grained coverage, without requiring *a priori* recompilation or rewriting of binary-only targets.

2.5 Evaluation

We evaluate MetaEmu using three criteria: ❶ performance, ❷ implementation flexibility, and ❸ real-world usability. For ❶ we benchmark MetaEmu's raw execution performance, and compare it to two existing tools for firmware emulation: Unicorn and Ghidra's emulator. For ❶ and ❸, we measure the extent our lifter is able to optimize its generated IR for large ECU firmware images, and compare the number of emitted IL operations to naive lifting without any optimization. We assess ❷ and ❸ together through six case-studies, which demonstrate using MetaEmu to perform different kinds of firmware analysis, including: inter-device analysis of two rehosted ARM firmware,

inference of undocumented memory mapped peripheral interactions in a Renault BCM, reverse engineering of the UDS handler functions of a Land Rover Discovery's TCU, fuzz testing to rediscover a backdoor in the "Security Access" UDS service of a Volkswaggen Passat's IC [57], and emulation of the ThreadX RTOS implementation of a Range Rover Evoque TCU. Across our case studies we demonstrate that MetaEmu is capable of enabling complex dynamic analyses built upon symbolic execution, fuzz testing, and micro execution [79].

We perform our experiments on a machine with a 32-core AMD Ryzen Threadripper 3970X and 128GB RAM.

2.5.1 Performance Benchmarks

We evaluate the performance of MetaEmu using two metrics: execution time and optimization of emitted IL operations. In the evaluation of execution time, we compare our tool with two existing emulation frameworks: Unicorn and Ghidra. We compare MetaEmu to Unicorn, as it is widely used for rehosting and performing security analyses [64, 65, 86], and to Ghidra's emulator, as it is used as the backend for `afl_ghidra_emu`, a fuzzer released by AirBus targeting esoteric architectures [87]. We use ARM programs for our head-to-head benchmarks based on the assumption that it is the most optimized embedded architecture all tools support.

We perform four sets of experiments: ❶ a micro-benchmark to evaluate the performance of each tool on various analysis tasks, ❷ a deeper inspection of the differences in performance between Unicorn and MetaEmu performing different kinds of hooking, ❸ an evaluation of the performance and trade-offs of performing IR optimization, and ❹ the effects of IR optimization on the number of IL instructions emitted.

Micro-benchmarks

We benchmark the performance of each tool on the programs described in Appendix A.3 and visualize the results in Figure 2.5. Our test harnesses measure execution time in

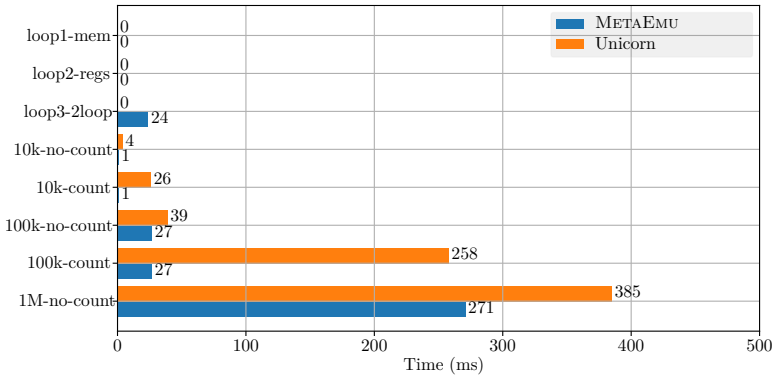


Figure 2.5: Performance comparison between MetaEmu and Unicorn. Tests 1-3 benchmark performance on tight loops updating state: registers or memory. Tests 4-8 benchmark programs of 10k, 100k, and 1M instructions.

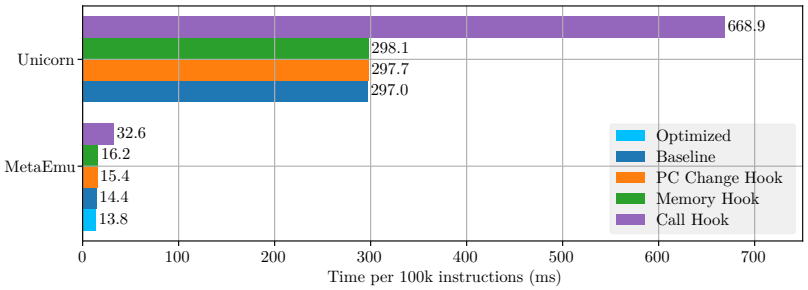


Figure 2.6: Comparison of MetaEmu and Unicorn whilst performing analyses using different kinds of hooks on a 100k instruction ARM-based sample program. We include two baselines for MetaEmu—one with IR optimization and one without—IR optimization gives a 4-5% improvement.

milliseconds; we run each test 10,000 times and report the average time taken. We omit the results for Ghidra's emulator, as they are significantly worse than both tools shown—on average by at least an order of magnitude, e.g., "1M-no-count" takes 3234ms, compared to 271ms for MetaEmu and 385ms for Unicorn.

We find that MetaEmu performs favorably compared Unicorn on all but one benchmark. We attribute the performance difference between MetaEmu and Unicorn on the "loop3-2loop" benchmark, to be due to the program being well suited to optimization by Unicorn's JIT compiler, as it contains a small number of basic blocks executed in two tight loops, where both loop bounds are hard-coded.

Deeper Analysis of MetaEmu and Unicorn

In this experiment, we benchmark the performance of MetaEmu and Unicorn while analyzing a sample program that consists of 100k instructions performing repeated memory transfer operations and function calls. In addition to baseline performance, we benchmark three different kinds of analyses: execution trace logging (`PC Change Hook`), memory access interception (`Memory Hook`), and function stubbing and call context logging (`Call Hook`). We include two baselines for MetaEmu—one with IR optimization and one without. We run each experiment 10,000 times and report the average time taken. We do not include Ghidra's emulator in these experiments, as it does not provide sufficient hooking support and is dramatically slower than both MetaEmu and Unicorn.

Results: We show the results of our benchmarks in Figure 2.6. Across all benchmarks, Unicorn is on average ∼20 times slower than MetaEmu. For `Memory Hook` and `PC Change Hook`, both MetaEmu and Unicorn have negligible performance overhead over baseline performance, while for `Call Hook`, both tools show larger overheads, which is due to the additional complexity of logging calling contexts. MetaEmu's IR optimizations provide an average of 4-5% performance improvement. Our benchmarks clearly demonstrate that our specification-based rehosting approach can perform at least as well as those based on Unicorn—without a performance trade-off—while also being much easier and flexible to extend.

IR Optimization Performance and Trade-Offs

In this experiment, we analyze the trade-offs of IR optimization by performing an extended analysis of the the "loop3-2loop" program from Section 2.5.1 compiled for both ARM and C166 (omitted for Unicorn as it does not support C166). We measure the time each framework takes to execute the program's loop 1k times and 65k times, and measure the overheads of performing MetaEmu's IR optimizations both online and offline against a baseline without any optimizations enabled.

Results: Figure 2.7 shows the results of our benchmarks. As mentioned in Section 2.5.1, Unicorn outperforms all frameworks on this benchmark, due to its JIT optimizer. We find that although MetaEmu and Ghidra's emulator share the same processor specifications, MetaEmu is ~400% faster on all benchmarks for both architectures. MetaEmu performs most favorably when applying its optimizations offline, and when applying its optimizations *online*, we only observe favorable performance on the longer running loop. On the shorter loop, the overheads induced by optimization (C166: ~15ms, ARM: ~2ms) result in diminished performance. Thus, we find that the choice of whether to apply optimizations depends on the kind of analysis being performed and the size of the program being analyzed. We provide an option to disable optimizations as part of our synthesizer specification. As a rule of thumb, optimizations make most sense when running loop-based firmware, fuzzing, or when performing symbolic execution (due to the decreased size of symbolic formulas), and otherwise may negatively impact performance.

Impact of Optimizations on Real Firmware

To evaluate the effectiveness of our optimizer on real-world firmware, we measure the total number of IL operations before and after optimization when lifting every basic block in each of our ECU firmware. The C166-based IC firmware is ~512KB and contains 2609 functions, the RH850-based TCU is ~32KB and contains 646 functions, the V850ES-based BCM is 544KB and contains 2987 functions, the SH-2A-based TCU is 8.4MB and contains 14,223 functions.

Results: We summarize MetaEmu's optimizer performance by IL elimination in Figure 2.8. Even though our results are a conservative approximation of our tool's real performance—accounting only for optimizations that lead to complete elimination of IL operations—across all firmware we see reductions of hundreds of thousands of operations. For our C166-based firmware, for instance, we observe that MetaEmu reduces the number of operations by ~27%: from ~10.8M to ~7.9M.

As our optimizations can be applied offline, they incur no overhead during simulation.

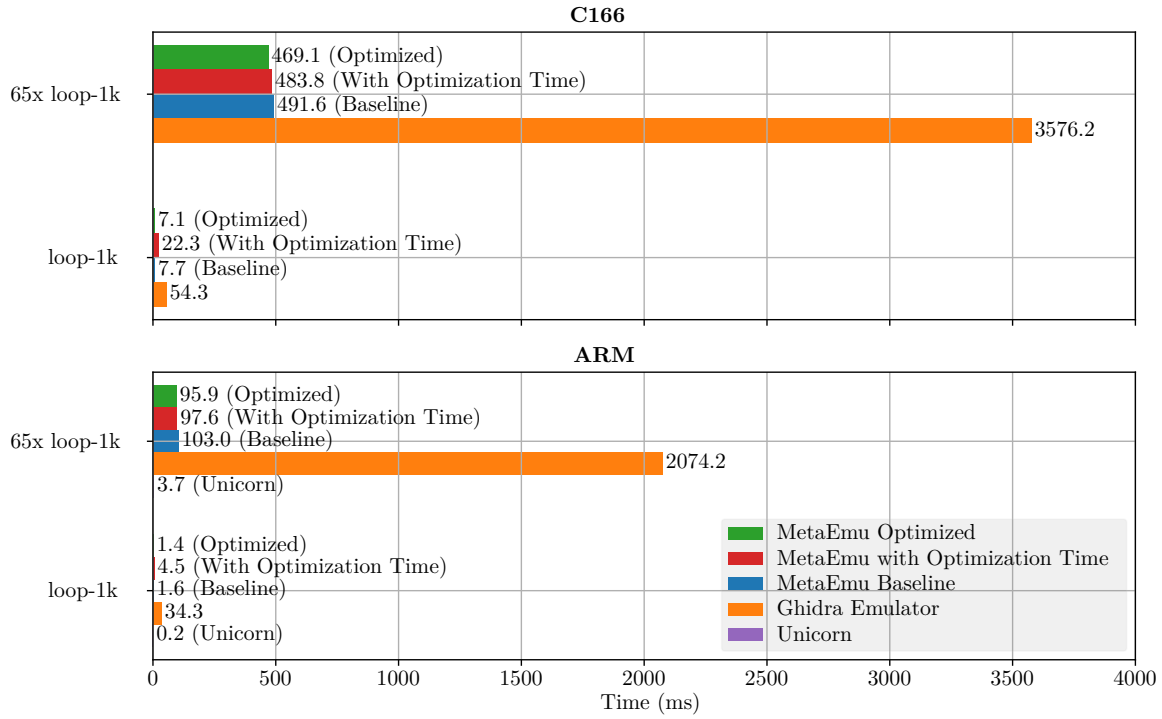


Figure 2.7: We benchmark Ghidra and MetaEmu's performance for both ARM and C166, and Unicorn for ARM, as it does not support C166. MetaEmu's optimization overhead is constant for both architectures on both test configurations—∼15ms for C166 and ∼2ms for ARM. As a result, optimization is more useful when running the program for a large number of iterations (e.g., fuzzing).

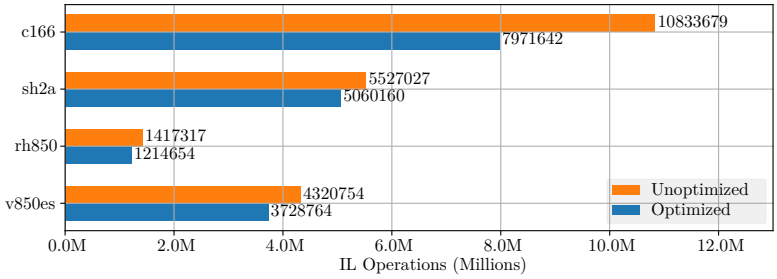


Figure 2.8: IR optimization performance for each ECU firmware. Total number of operations calculated by lifting every basic block of each firmware; reduction calculated by number of eliminated operations.

However, when lifting, applying IR optimization induces a non-negligible overhead of ∼700μs per IR block, compared to ∼6μs without. This can be seen in Figure 2.7, the optimization time is constant regardless of the number of iterations. We attribute the significant difference to our use of e-graphs and equality saturation [74] to find optimal IL sequences—our optimizer effectively computes all simplifications of each IR block and

Table 2.1: Summary of case studies performed. MetaEmu features evaluated: architecture support (**A**), inter/multi-device analysis (**M**), peripheral support (**P**), interrupt support (**I**), integration with external tools (**T**). Implementation size shows the approximate LoC for experiment (**E**) and specification for peripherals and execution mode (**S**).

Firmware	**ISA**	**Impl. Size**		**Firmware Evaluated**				
		E	**S**	**A**	**M**	**P**	**I**	**T**
Arduino & NuttX firmware	ARM Cortex-M3	300	180	✓	✓	✓	-	✓
P2IM firmware (averaged)	ARM Cortex-M4	200	170	✓	-	✓	-	✓
Volkswagen IC	C166	260	100	✓	-	✓	-	✓
Renault BCM	V850E2	160	<10	✓	-	✓	-	✓
Land Rover Discovery TCU	RH850	700$^{\delta}$	-	✓	✓‡	✓	-	✓
Range Rover Evoque TCU	SH-2A	370	110	✓	-	✓	✓	✓

δ: We manually implemented complex peripheral (CAN Bus) with hooking API
‡: We rehost multiple instances of the firmware to perform fuzzing.

extracts the most optimal with respect to minimizing the IR size.

2.5.2 Case Studies

In this section, we show how MetaEmu can be used to emulate and analyze firmware based on five different ISAs: ARM, C166, RH850, SH-2A, and V850E2—the latter four are not supported by any existing rehosting framework. We show that MetaEmu handles complex end-user firmware by rehosting four ECU firmware, extracted from real automotive components. We also perform two other case studies: inter-device analysis (Section 2.5.2), and fuzzing and rehosting non-automotive firmware (Section 2.5.2), using ARM-based firmware from the P^{2}IM data-set [62]. The firmware from this data-set are simpler than our ECU examples, and are chosen to enable us to more easily describe our experimental set-up and results. We summarize our case studies in Table 2.1.

Inter-Device Analysis

In this case study, we demonstrate MetaEmu's capability to rehost multiple firmware in the same VXE and perform an inter-device analysis. Each target firmware is ARM-based and adapted from the P^{2}IM [62] data-set—one is based on the Arduino SDK, and the other on NuttX, as shown in Figure 2.9.

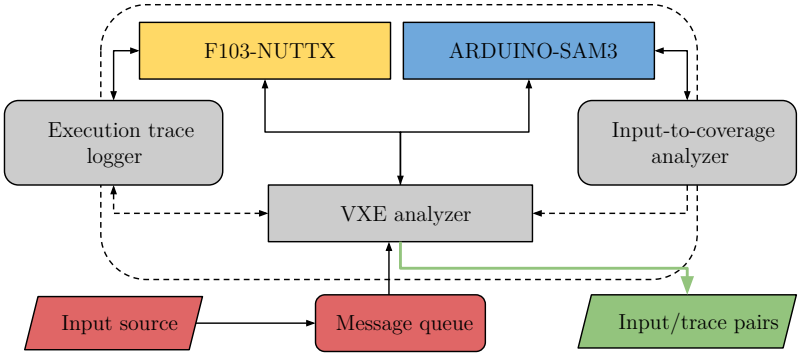


Figure 2.9: Inter-device analysis. Inputs are supplied outside the VXE via a message queue, received by the F103, and forwarded to the ARDUINO. The F103 has an observer that logs the execution path corresponding to each input received and sent. The ARDUINO has an observer that analyzes the branch coverage corresponding to processing received inputs; on new coverage, it notifies the shared VXE analyzer, which instructs the F103's observer to dump the execution trace matching the last sent input. The VXE analyzer outputs input/trace pairs that yield new coverage.

Objective: The goal of this case study is two-fold: First, to show how two rehosted firmware can communicate, where one firmware (F103) is supplied input from outside the synthesized VXE, and the other (ARDUINO) receives input from the other rehosted firmware (F103). Second, to demonstrate an analysis that captures execution traces from a rehosted firmware (F103) that leads to new branch coverage in another (ARDUINO). This case study models a typical scenario found in CAN networks, where one can only interact with a particular ECU by sending messages via another.

Set-up: We develop five observers. Three simulate peripherals: one to receive input via UART (outside to F103), one to transmit data via a TTY (from F103) and one to receive input via a serial port (into ARDUINO). Two perform analyses: an execution trace dumper that starts on a *read* from UART and stops on a *write* to a TTY, and a coverage tracker, which reports new branch coverage via the VXE's coordinator. The coordinator is configured to dump execution trace/input pairs that cause new branch coverage. We provide input to the F103 using random byte sequences, and specify a *concrete execution mode* for each firmware.

Discussion & results: Combined, our firmware configuration spans ~180 lines of code

(not including processor specifications), while our observers take a further ~300. Our configuration accounts for overriding firmware functionality unrelated to our analysis task, creating a memory mapping for each firmware, and specifying the behavior of our analysis coordinator. Each rehosted firmware executes in parallel and faithfully simulates its expected behavior. This case study demonstrates how our approach can enable an otherwise tricky to reproduce analysis scenario with very little manual overhead. Further, it shows how MetaEmu can facilitate modeling peripheral and inter-device interactions, akin to those that regularly occur in real automotive networks, without requiring any hardware or source-code access.

Volkswagen Passat Instrument Cluster (C166)

In this case study, we demonstrate how MetaEmu can be used with LibAFL to rediscover a previously reported backdoor in the UDS "Security Access" service of a Volkswagen Instrument Cluster [57]. The ECU is based on the Infineon C166 architecture. To support the C166 architecture, which is not currently distributed with Ghidra, we base our language definition on an open-source project [88]; our definition consists of 1737 lines. We implement two extensions to our simulator: an intrinsic to handle the C166's *DPP override* addressing [82], which enables the firmware to override which memory pages are accessed for a variable window of instructions, and an observer to implement its GPR bank switching.

Objective & set-up: The backdoor is embedded in the UDS handling routine. The routine receives two inputs: a buffer containing the UDS request, which we populate with input from LibAFL, and a request ID, corresponding to a UDS service, which we populate using the ID of the "Security Access" service (0x27). Authenticating against this service enables a client to access security critical services, e.g., "Request Download" (ID 0x34), which permits new software to be transferred to the ECU. To authenticate, a client must complete a challenge-response handshake by sending a correct *key* for a given *seed*. The backdoor enables this check to be bypassed by supplying a specific

hard-coded value in the request buffer (0xCAFFE012). The backdoor trigger is in the form of two 2-byte comparisons, and can easily be discovered by fuzzing. To do so, we configure our LibAFL harness to mark inputs that reach blocks corresponding to successful authentication attempts as if they induce a "crash".

Discussion & results: We were able to trigger the backdoor with our fuzzer, but not as easily as expected. In fact, our initial attempt was unsuccessful, as the firmware relies on a timer peripheral to correctly execute the "Security Access" check—instead of validating our fuzzer's supplied "key", it enters an infinite polling loop. On investigation, we found that this check first generates a challenge seed by repeatedly sampling from the timer's value register until a specific criterion is met. Rather than bypassing the seed generation, we attached a peripheral to the firmware's simulator based on our generic CMT model (Section 2.4.4). The specification takes just 6 lines; requiring masks for the enable bit, toggle on match bit, and current tick value, and the address range of the MMIO of the timer peripheral. Our peripheral model emulates realistic behavior in our simulator when accessing the timer's memory mapped registers, `TxCON` and `Tx`, successfully enabling us to fuzz the firmware to discover the backdoor key.

Renault Body Control Module (V850E2)

In this case study, we use MetaEmu to aid in identifying peripheral access checks in firmware from a Renault BCM. Even when an emulator is available for the architecture of a device, one of the major obstacles when performing rehosting is the lack of documentation for its peripherals. Usually, processor manuals provide the address range of MMIO registers, but as each device can have a diverse array of peripherals, inferring the meaning of each address often requires manual reverse engineering. Unfortunately, when these documents are not in the public domain, we need to reverse engineer the MMIO range itself to achieve even basic emulation. In this case study, we show how MetaEmu can help automate this task.

Objective & set-up: The objective of this case study is to facilitate basic emulation

of our BCM firmware, and identify peripheral access checks, without prior knowledge of specific peripheral register addresses. We use processor definitions adapted from those distributed with Ghidra. We obtain an approximate MMIO peripheral address range from the processor manual of a related MCU (V850E2/Fx4-G), as the peripheral manual for the MCU (V850E2/Dx4) used in the BCM is not publicly available.

Identifying peripheral status checks: We perform our analysis in two steps. We first use our peripheral check solver to identify peripheral checking loops using registers mapped to the MMIO range of a similar MCU (`0xff400000`–`0xffff8000`). This enables us to identify and bypass some peripheral checks. However, our firmware still *stalls* prior to reaching its main loop. To overcome this, we gradually widen our assumed MMIO range by specifying larger bounds in our solver's configuration. After a few iterations, we determined that the real MMIO range of the MCU is much wider than expected: we discovered checks in the range `0xffff8000`–`0xffffffff`. By reconfiguring our peripheral check solver to use this extended range, we were able to bypass the initialization checks performed by the firmware. This case study thus demonstrate that MetaEmu's specification-based approach can enable analyses of this kind without developing custom peripheral models, or device-specific heuristics.

Land Rover Discovery Telematics Unit (RH850)

In this case study, we use MetaEmu to rehost the TCU from a 2018 Land Rover Discovery in order to reverse-engineer its UDS handler routines. To aid our reverse-engineering, we use fuzzing to identify *valid* UDS requests. The ECU's firmware is based on the RH850 architecture. It's worth mentioning that despite the RH850 and V850 are from similar microcontroller family, there are still differences in instruction set and the peripheral layout.

Objective & set-up: TCUs are responsible for communicating with and monitoring other ECUs on the CAN bus, and reporting diagnostics data and metrics over other mediums, such as LTE, hence have a large attack surface. UDS is the protocol responsible

for transmitting diagnostic information and is implemented on top of CAN. Among other functionality, UDS can be used to re-flash or dump an ECU's firmware. The goal of our analysis is to reverse engineer the UDS handling functions in our target firmware. These functions take input from a CAN peripheral, process it, and act accordingly. To analyze them, we take a four-step approach:

1. Since UDS is a CAN-based protocol, to determine how to use it to communicate with the device, we first need to identify the device's CAN IDs.

2. Next, to interface with the device's rehosted firmware over UDS, we create a model for its CAN peripheral.

3. As the firmware interfaces with many peripherals unrelated to our analysis task, yet relies on them being successfully initialized to configure its internal state prior to executing its main loop, we attach a peripheral check solving observer to supply suitable values when accessing their MMIO registers.

4. Finally, to identify valid UDS requests, we use a LibAFL-based fuzzer to supply input to the firmware via its CAN peripheral, and track which inputs lead to new coverage.

CAN ID identification In a modern automobile, the CAN bus is the standard medium for ECUs to communicate with each other and the outside world. Thus, it is usually the first interface/input source we investigate when analyzing automotive firmware. On the CAN bus, each ECU is assigned one or more CAN IDs, and will only respond to messages sent to those IDs. As many manufacturers attempt to keep CAN IDs secret, it is a non-trivial task to identify them effectively without analyzing the ECU's firmware.

As mentioned previously, automotive microcontrollers usually interface with their peripherals and, thus, the CAN bus, by reading and writing to MMIO registers. Each such register controls either a peripheral's behavior or act as an input/output buffer. From our experience, for peripherals that interface with the CAN bus, one register will

contain the *listening* CAN ID. Hence, if we know the address this register is mapped to, we can use it to recover the IDs the ECU *listens* on by monitoring values written to it.

Unfortunately, exhaustively enumerating all execution paths that write to the register requires near complete emulation of the entire firmware—defeating the purpose of rehosting. Flood execution [80], however, provides a means to approximate all execution paths in a bounded manner, and the micro execution-based variant proposed by Gotovchits et al. [81] provides a means to perform flood execution in the presence of failing memory accesses. To recover CAN IDs, we use *flood execution* mode with our synthesized simulator to enumerate paths, and use an observer to log writes to the register documented to store CAN IDs. Through this process we recover three CAN IDs: `0x7df`, `0x18db33f1`, and `0x700`.

CAN peripheral modeling We model the device's CAN peripheral using observers, as shown in Figure 2.10, and interact with our model via SocketCAN [89]. We use an observer to hook *reads* and *writes* to the CAN registers to simulate receiving and sending CAN frames. When sending data, we build CAN frames using values from the CAN status register (`TMSTS`), the receive ID register (`RFID`) and receive data register (`RFDF`); after transmission, we set the firmware's transmission success bit. When receiving data, we read a frame from our input source, and use it to set the following registers: `TMC`, `TMID`, `TMPTR`, and `TMDF`.

Bypassing peripheral status checks To bypass the initialization checks of the device's other peripherals, we use our symbolic status check solver (Section 2.4.4). We configure it using the full peripheral address range: `0xFF400000`–`0xFFFFAFFF`. Alongside our CAN peripheral, this enables us to execute the firmware from reset to its UDS request processing loop.

Fuzzing for valid UDS requests To fuzz for valid UDS requests, we use *concrete execution* mode to run our rehosted firmware from reset to the UDS function, and then use an observer to take a snapshot of the simulator's state. We build a LibAFL-based harness which starts execution from our snapshot and supplies input to the UDS routine

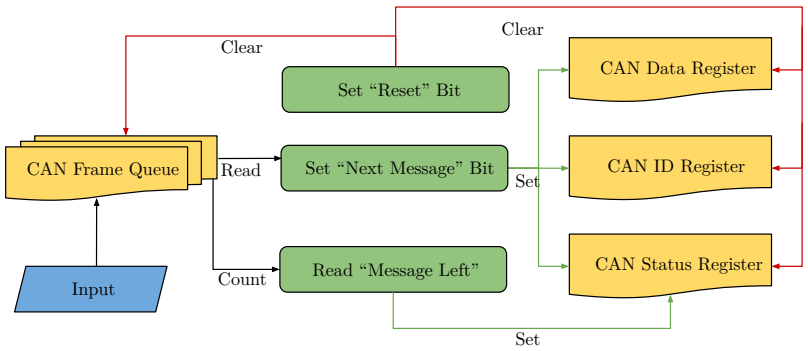


Figure 2.10: Data-flow diagram of RH850 CAN peripheral. We store incoming CAN messages in a FIFO queue (blue box). When the firmware sets the "next message" bit in its CAN MMIO register, we dequeue a CAN frame, use it to populate the CAN data and CAN ID registers, and then set the data available bit in the CAN status register. Upon a *reading message left* event, we update the "message left" counter in the CAN status registers with the length of the queue. When the firmware sets the peripheral's "reset" bit, we clear the queue and reinitialize the registers.

via our CAN peripheral. We implement a custom `EventManager` for LibAFL that supports sharing coverage and *interesting* inputs across multiple MetaEmu simulators via our *analysis coordinator*. This enables us to rehost many instances of our firmware and fuzz the UDS routine in parallel. Through our analysis, we generated inputs that explored execution paths covering 898 unique IL branches, which uncovered 8 UDS request handlers.

Range Rover Evoque Telematics Unit (SH-2A)

In this case study, we demonstrate how MetaEmu can be used to rehost a Range Rover Evoque TCU SuperH-2A firmware. The firmware is based on ThreadX RTOS, which executes multiple tasks concurrently. To rehost, it requires robust peripheral models to correctly perform task switching—something that cannot be achieved using simpler symbolic modeling approaches, such as the peripheral check solver used in our other case studies.

Objective & set-up: Through manual reverse engineering, we found that the firmware uses a timer interrupt to facilitate task scheduling, which is performed by a CMT peripheral. The goal of this case study is: First, to show how MetaEmu facilitates

implementing a complex peripheral that can faithfully trigger task switches. Second, to show how MetaEmu can emulate hardware-supported multi-tasking.

Discussion & results: Since the device's CMT peripheral acts as both a timer and interrupt source, we can implement it using a combination of our generic CMT model (Section 2.4.4) and our interrupt backend (Section 2.4.5), as shown in Figure 2.11. To configure our CMT model, we specify the mapping between the peripheral's registers and the model's set/unset actions. This enables our model to update and track its enabled status, interrupt status, and timer matching status. To simulate the timer's counting and interrupt triggering behavior, we implement a simple observer that increments the timer's value by one each time a new architectural instruction is lifted, and *fires* an interrupt using MetaEmu's interrupt backend. Our implementation (optimistically) assumes that each instruction takes one clock cycle, and the timer increases based on the device's clock. We configure our interrupt backend to trigger a jump to the firmware's timer ISR when it is instructed to *fire* by our timer observer. Since the context switch is normally performed in hardware, we configure MetaEmu to preserve the status register `SR` and program counter `PC` prior to jumping. The firmware uses the `rte` instruction to return from interrupt handlers, and its logic is implemented in the architecture's processor specification. Our peripheral requires just ~250 lines of specification and code. To test our peripheral model, we attach an execution tracer observer to our simulator and run the device from reset. Our traces show that the emulated firmware correctly performs task switches and mirrors the real firmware's behavior.

P^{2}IM dataset firmware

The P^{2}IM authors [62] provide a data-set consisting of 10 ARM-based firmware based on open-source projects, which all use a HAL to interact with their peripherals. In this case study, we use the `Gateway` and `Soldering_Iron` firmware to demonstrate the correctness of MetaEmu and show that it can achieve the same analysis outcomes as existing rehosting frameworks. We perform two experiments: ❶ we fuzz test the `Gateway`

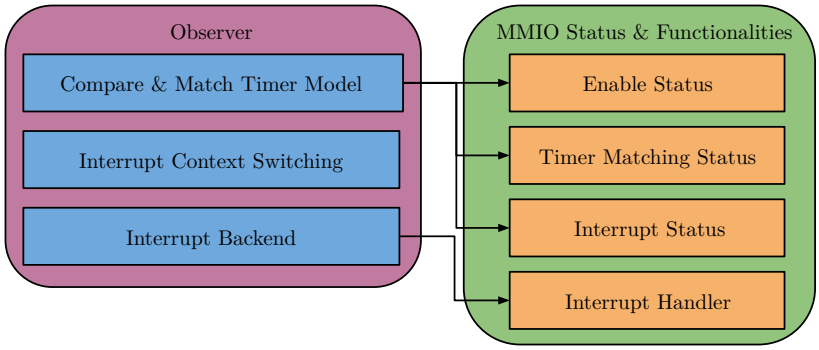


Figure 2.11: Overview of our SH-2A CMT model, implemented using an observer and our universal peripheral and interrupt backends. Our CMT model is responsible for tracking the enabled, timer matching, and interrupt status of the peripheral. We use our interrupt backend to specify a custom interrupt handler and ensure the PC and SP registers are preserved during the context switch.

firmware, and ❷ we use the `Soldering_Iron` firmware to test MetaEmu's support for handling tricky (DMA-based) peripherals.

Set-up: For experiment ❶, we set up a basic fuzz harness to supply I2C query packets by hooking the HAL of the firmware. Similar to past work [90], we mark part of the firmware's memory as a "red zone" (write permission disabled) to perform sanitization and detect any memory corruption bugs. For experiment ❷, we attempt to test if MetaEmu can run the firmware from its entry-point, allow it to perform peripheral initialization, and execute its main loop without crashing due to peripheral mishandling.

Discussion & results: For ❶, we were able to trigger an out of bounds write with inputs exceeding a length of 62 bytes (the first two bytes of `i2cRxData` are populated using `i2c_device_info.addr` and `i2c_device_info.reg`) after only three iterations of our fuzz harness. This demonstrates that MetaEmu is usable for analyzing firmware beyond our original use-case of automotive firmware, as well as its suitability for fuzz testing.

For ❷, the author's of P^{2}IM [62] describe two *false* crashes/hangs induced by their framework when rehosting the `Soldering Iron` firmware. The first is caused by misclassification of a peripheral register, and the second is due to P^{2}IM missing support for DMA-based peripherals. As MetaEmu can support peripherals generically, we were able to successfully rehost this firmware by intercepting DMA interactions by the firmware's

HAL interface in a manner similar to past work by Clements et al. [91]. We thus avoid the false crashes experienced by P^{2}IM and μEmu [61]. Since MetaEmu provides an API to intercept any memory read/write (i.e., `hook_memory_write/read`), it can be easily extended to provide DMA support similar to DICE [92]. We note that since the author's of P^{2}IM do not provide source code for this firmware[3], we had to manually stub out the functionality relating to AFL, e.g., the call to `aflCall`, as it was unnecessary for our experiment.

2.6 Human Effort

In this section, we discuss the human effort involved in using MetaEmu for rehosting. As with any kind of reverse engineering, rehosting necessarily requires manual intervention for tasks that cannot be automated—from implementing complex peripherals for I/O to identifying functions to hook and override. MetaEmu attempts to reduce this manual effort by moving towards a specification-based approach from one that is purely programmatic. In Table 2.1, we show the human effort involved in each of our case studies in terms of lines of code—all require under 1kloc—even in the case of complex inter-device analysis. Concretely, each case study took the author's less than one day to implement.

The advantage of using our approach is twofold: firstly, a specification-based approach provides a DSL which enables much faster iteration of manual tasks such as implementing missing ISA instructions or peripheral models. As we separate specification files from the implementation of our tool, we can prototype new features without recompiling the tool, something not possible with QEMU-based rehosting approaches. Secondly, by using the same ISA specification language as Ghidra, we benefit from the maintenance and testing efforts of a large and active community. Ghidra's repository contains specifications for 28 architectures and when combined with community projects [88], MetaEmu can rehost

[3] https://github.com/RiS3-Lab/p2im-real_firmware/issues/2

most ECUs without the need to implement any architectural support.

2.7 Limitations

In this section, we provide a discussion of MetaEmu's limitations. The correctness of MetaEmu's lifter depends on the processor definitions it uses as input. Fortunately, as they are based on Ghidra's language definitions, we benefit from Ghidra's large and active community that regularly contributes fixes.

The three peripheral interfaces in MetaEmu provide a simple way to implement peripherals for automotive microcontrollers of atypical architectures, however, we have less peripheral models for common microcontrollers such as ARM or MIPS based devices. This makes it difficult to produce a direct comparison with existing emulators such as QEMU. As future work, we intend to explore adding a compatibility layer to allow MetaEmu to use QEMU-derived peripheral models to address this issue, however, it would require implementing QEMU's Object Model, memory, sysbus, interrupt, and peripheral APIs in MetaEmu to do so. Our universal peripheral backend (Section 2.4.4) does not fully support DMA-based I/O, except when firmware interfaces with it using a HAL. Support for peripherals requiring this, however, can easily be added using MetaEmu's execution observers (Section 2.4.3).

While MetaEmu's runtime performance is comparable to Unicorn, there are possibilities for improvement, such as adding a JIT compiler for our IL. We believe such an enhancement will lead to even greater performance for our specification-based approach. However, it is unclear how to generically implement a JIT compiler that supports all of MetaEmu's different execution modes (Section 2.4.2).

2.8 Discussion & Related Work

Approaches to firmware rehosting cover a broad spectrum: in terms of the kind of analyses they facilitate, the degree to which they rely on hardware, the kinds of device they support, and the fidelity and faithfulness of the environments they transplant firmware into.

Avatar [93, 94] and Surrogates [95] propose hardware-in-the-loop analysis, which enables a device to be analyzed without handling many of the complexities of its peripherals. It permits a kind of hybrid methodology where a fast host can emulate most of the firmware and rely on the real device for peripheral I/O and interrupt handling. Many techniques have capitalized on these seminal works; for instance, Ruge et al. [64] use a hardware-in-the-loop approach to fuzz for vulnerabilities in bluetooth chips, while Gustafson et al. [63] use such an approach as a basis for their tool, Pretender, which automatically infers peripheral models from execution traces and device I/O behavior. Inception [96] and HardSnap [97] use a hardware-in-the-loop approach for their debugger component; both attempt to handle the nuances of complex firmware and devices with multiple peripherals under symbolic execution.

In contrast to hardware-dependent approaches, emulation-based approaches do away with hardware interaction altogether, and to varying degrees attempt to emulate a device and its peripherals. Of the approaches that make their implementations open-source, we observe that almost all rely on either QEMU (e.g.,PARTEMU [98]) or Unicorn (e.g., BaseSAFE [65]) as the basis for their VXE. Clements et al. [91] emulate peripherals by hooking HAL APIs provided by many vendor SDKs. They simulate peripheral interactions through generalized models that receive input and supply output via the hooked HAL functions. In [99] they extend their approach to support VxWorks-based devices. Feng et al. [62] rehost firmware to facilitate fuzz testing. To handle peripherals, they learn appropriate values for MMIO peripheral interactions based on device-specific *abstract peripheral models*. Mera et al. [92] propose a method to handle DMA-based

peripheral inputs, similarly targeted at fuzzing rehosted firmware. Liu et al. [100] use model-guided execution to generate QEMU peripheral models from kernel device-tree and source code. Cao et al. [60], Johnson et al. [101], and Zhou [61], all leverage symbolic execution to learn *satisfying* values to bypass peripheral checks. Hernandez et al. [102] achieve full-system emulation of closed-source Shannon baseband firmware by adding missing architectural and peripheral support in QEMU, they later demonstrate that such an approach can be extended to other basebands [103]. In contrast to the aforementioned approaches, Milburn et al. [104] build a custom emulator and peripheral models to rehost an automotive instrument cluster; they use their emulator to aid in reverse-engineering the firmware's UDS commands. Meanwhile, Davidson et al. [105] use full-system symbolic execution to discover vulnerabilities in PIC32 devices; they propose using specifications to configure interrupt and peripheral memory mappings, allowing their approach to be adapted to analyze different device configurations.

While pure emulation and hardware-in-the-loop approaches can achieve near complete support for a given device or family of devices, when based on commodity emulators, they are difficult to adapt to support devices based on esoteric architectures, such as automotive components, due to the substantial engineering effort required. Mera et al. [92] highlight this difficulty in their evaluation—to test their approach on both ARM and MIPS32-based devices, they need to build separate prototypes of their tool for two different forks of QEMU, as neither variant supports both architectures. Hernandez et al. [103] note the current impossibility of porting their baseband rehosting framework to work with Qualcomm basebands, due to lack of architecture support in the PANDA [106] QEMU fork.

As discussed in our evaluation, Ghidra provides a P-Code-based emulator, which can facilitate basic analysis tasks, including basic instruction hooking and manipulation of memory and register values. However, it lacks more advanced features for firmware reverse engineering, such as symbolic execution, and peripheral support. Although it is possible to add such functionality using its hooking API, its emulation performance

is insufficient to support intensive tasks, such as fuzzing. Further, adding functionality to support dynamic changes to addressing modes, e.g., to correctly emulate C166-based firmware, would require extensive changes to the core of Ghidra's emulator framework—a significant engineering task.

In Table 2.2, we provide a feature, architecture, and analysis-support comparison of MetaEmu with the state-of-the-art, using the framework classification proposed by Fasano [58]. The key difference between MetaEmu and the frameworks listed, is that our approach generically enables analysis of firmware not currently supported by other frameworks with little effort. Moreover, like Avatar, it is a general framework to build analysis tools, as opposed to a method to enable a specific type of analysis, e.g., fuzzing. As demonstrated through our case studies (Section 2.5.2), we can use MetaEmu to build analyses similar to those proposed by other approaches (e.g., Cao *et al.* [60] and Johnson *et al.* [101]) in a completely architecture-independent way, without resorting to implementing those techniques for many different emulators, or limiting the approach to a few architectures. As with other rehosting approaches, such as P^{2}IM, MetaEmu's peripheral support will lead to better coverage when performing analysis tasks, such as fuzzing, over an emulator with no peripheral support.

2.9 Conclusion

To conclude, we have presented the first architecture-agnostic framework capable of rehosting multiple devices simultaneously. Our IR lifter and simulators provide fully generic, extensible, architecture support, and our universal peripheral models and peripheral solver enable simulation of realistic execution environments. We also have tight integration with binary analysis tools to help manual analysis. Through our case studies, we have demonstrated that our tool is flexible, efficient, and can drive the analysis of real-world automotive firmware whose architectures are not supported by existing state-of-the-art rehosting approaches.

Table 2.2: A comparison of MetaEmu with the state-of-the-art. Layer refers to how the framework handles a rehosted device and its firmware: ε–untouched, α–passed through, β–emulated, ϕ–replaced, χ–symbolic model. Type refers to the kind of devices the framework can rehost; we use the classification proposed by Fasano et al. [58]: 3–bare-metal (no OS), 2–embedded-device specific OS (e.g., Zephyr), 1–general purpose OS retrofitted for embedded firmware (e.g., Linux). Peripheral classifies how peripherals are handled: ●–hardware, ◗–approximated (symbolic or inferred) ❍–modeled. Multi-device denotes if the framework can rehost multiple targets simultaneously. Analysis support refers to the kinds the framework analyses enables.

Framework	Layer				Target		Peripheral	Multi-device	Analysis Support
	Hardware	OS	Application	Function	Type	ISA(s)			
Avatar [93]	$\alpha/\beta/\chi$	ε/α	ε/α	$\varepsilon/\alpha/\chi$	2, 3	ARM	●	✓‡	Generic
Avatar^2 [94]	$\alpha/\beta/\phi$	-	ε/α	ε/α	3	ARM	●	✓‡	Generic
DICE [92]	β/ϕ	ε	ε	ε	2, 3	ARM, MIPS	◗	✗	Fuzzing
HALucinator [91]	β/ϕ	ε/ϕ	ε	ε/ϕ	2, 3	ARM, MIPS	❍	✗	Fuzzing
FirmWire [103]	β/ϕ	ε/ϕ	ε/ϕ	ε/ϕ	2, 3	ARM, MIPS	❍	✗	Fuzzing
Jetset [101]	ϕ/χ	ε	ε	ε	2, 3	ARM, Coldfire, x86	◗	✗	Fuzzing
Laelaps [60]	χ	ε	ε	ε	2, 3	ARM	◗	✗	Fuzzing
P^{2}IM [62]	β/ϕ	ε	ε	ε	2, 3	ARM	◗	✗	Fuzzing
μEmu [61]	χ	ε	ε	ε	2, 3	ARM	◗	✗	Fuzzing
Ghidra EmulatorHelper	β	ε	ε	ε	3	✻	-	✗	Generic
MetaEmu	$\beta/\phi/\chi$	ε/ϕ	ε	ε/ϕ	2,3	✻	◗/❍	✓	Generic (+inter-device)

‡: Avatar and Avatar^2 can, in principle, facilitate multi-device analysis by instantiating separate 'System'/'Avatar' objects for each device.

✻: MetaEmu supports all architectures supported by Ghidra, including: ARM, AArch64, C166, MIPS, PPC, SH-2, RH850, V850, x86, x86-64.

Chapter Three

Cutting Through the Complexity of Reverse Engineering Embedded Devices

Sam L. Thomas, Jan Van den Herrewegen, Georgios Vasilakis, Zitai Chen,
Mihai Ordean and Flavio D. Garcia

This chapter is based on the following publication:
Cutting Through the Complexity of Reverse Engineering Embedded Devices [2]
which is published in
IACR Transactions on Cryptographic Hardware and Embedded Systems, 2021(3)

3.1 Introduction

When performing security analysis of end-user embedded devices, reverse engineering is often required to gain a deep understanding of a device and its firmware. In general, this is a challenging and time-consuming task, even for experts. The core difficulties stem from how devices are composed and how tooling to simplify analysis can only go so far in terms of what can be automated. Developing tooling that generalizes over multiple devices is complicated and prone to scalability issues. Most moderately complex consumer devices are composed of numerous peripherals and CPU cores that interact with each other and their environment, often under hard real-time constraints, further complicating the process.

Recent work has sought to address the challenges involved in performing analysis by emulating firmware and peripherals [62, 63, 91, 92]. These approaches enable device firmware to be *rehosted*—either fully or partially—within a commodity emulator and then analyzed without interacting with the physical hardware. However, to rehost real firmware, it is often necessary to perform a non-trivial amount of reverse engineering of the device, its peripherals, and its firmware. Such techniques are, therefore, not always practical when time is a limiting factor in performing analysis.

Other techniques address the challenges by incorporating the device *in the loop* during analysis [93, 96, 97, 107]. Though without careful harnessing, for real-time constrained and complex bare-metal firmware, the timing overheads induced by these approaches can lead to unpredictable behavior in the target device due to violations of hard real-time constraints [95] and interference with timers. To identify the source of these violations and develop suitable harnesses, we must again resort to reverse engineering the device and its firmware.

Table 3.1 lists a broad selection of recently published articles proposing techniques for analyzing embedded firmware. Irrespective of the analysis goal or approach used, when applied to real-world targets, all require a non-negligible amount of manual reverse

engineering to perform. Without this upfront reverse-engineering, we would not know how to interact with the firmware or which parts of it to analyze. Thus, developing methods to reduce this reverse engineering effort is essential to use such automated analysis techniques to analyze end-user devices.

Table 3.1: Embedded firmware analysis approaches. Amount of prior reverse engineering (R-E) measured by: ✚ for identification of interrupts and handlers, stall states, and context switches, ✚✚ for peripheral interaction, cross-core communication, and ✚✚✚ for specific functionality (for stubbing/harnessing). Requirements refer to target support needed: (C)ompiler, (D)ebugger, (DI)sassembler, (E)mulator, (S)ource Code, (T)racer.

	R-E	Requirements						Objective	Enabling mechanism
		C	D	DI	E	S	T		
Avatar [93]	✚✚	○	●	●	●	○	○	Dynamic analysis	Hybrid execution
BaseSAFE [65]	✚✚✚	○	●	●	●	○	○	Fuzzing	Partial rehosting
DICE [92]	✚	●	●	●	●	●†	○	Fuzzing	Peripheral modeling
Frankenstein [64]	✚✚✚	○	●	●	●	○	○	Fuzzing	Partial rehosting
HALucinator [91]	✚✚✚	○	●	○	●	○	○	Fuzzing	HAL emulation
HardSnap [97]	✚✚	●	●	●	●	◐	○	Symbolic execution	Partial rehosting
Inception [96]	✚✚	●	●	●	●	◐	○	Symbolic execution	Hybrid execution
Laelaps [60]	✚	○	●	●	●	○	●	Dynamic analysis	Peripheral modeling
P^{2}IM [62]	✚	●	●	●	●	●†	○	Fuzzing	Peripheral modeling
Pretender [63]	✚	○	●	●	●	○	●	Rehosting	Peripheral modeling
Surrogates [95]	✚✚	○	●	●	●	○	○	Dynamic analysis	Hybrid execution
Incision ‡ (this chapter)		○	○	●	○	○	●	Reverse engineering	Trace comprehension

† DICE and P^{2}IM require firmware to be compiled with a call to `startForkserver` for fuzzing [108].
‡ Incision is used in the pre-analysis step to perform initial reverse engineering.

3.1.1 Our Contribution

In this chapter, we address the challenge of reverse engineering complex embedded device firmware to facilitate more straightforward manual analysis and application of automated analyses. We present Incision, a device in the loop reverse engineering framework, based on lightweight execution tracing and static firmware analysis. Our framework is both Operating System (OS) and architecture agnostic. It complements the standard reverse engineering workflow and approaches such as those listed in Table 3.1, by aiding functionality identification, control-flow recovery, and correct disassembly of binary blob firmware.

Our approach operates in feedback loops that progressively improve the reverse-engineered representation of the firmware analyzed. A single iteration consists of three

high-level steps: region inference, trace capture, and trace processing. Each iteration identifies firmware regions containing functionality related to an input reverse engineering policy. We use a machine encoding of the policy as input to a clustering algorithm to perform region inference. We use the outputted regions to decide which portion of firmware to focus on and what to trace. Due to interrupt processing and task switches, system-wide traces often end up being fragmented. Therefore, to recover control-flow information, we identify task switching logic and interrupt handlers and partition the traces based on this information. Then taking the resulting sub-traces, we group those that follow the same execution flow and combine them back into constituent traces. Finally, we refine our firmware representation with recovered control-flow information from those traces. We add new control-flow edges during refinement and use them to locate new functions, static data references and rectify incorrect disassembly. The process leverages statically inferred and dynamically witnessed control-flow information in tandem. At the end of the analysis, INCISION outputs the bounds of functionality corresponding to our reverse engineering policies and a firmware representation suitable for further analysis.

In summary, our contributions are:

1. An analysis of the specific challenges faced when reverse engineering complex embedded devices and their firmware.

2. Four techniques for automating aspects of the reverse engineering process: ❶ Firmware region inference. ❷ Identification of OS primitives for task and context switching from execution traces. ❸ Task-aware control-flow recovery from system-wide execution traces containing multiple interleaved tasks and interrupts. ❹ A feedback mechanism that combines the results of the previous techniques to improve our firmware representation and rectify disassembly errors.

3. An evaluation of INCISION against two complex real device firmware (an LTE baseband and an automotive BCM) and a set of firmware simulating a range of device

configurations. In full, our evaluation spans three different Real-Time Operating Systems (RTOSes), VxWorks, FreeRTOS, and Zephyr, a bare-metal OS, and two different instruction sets (ARM/Thumb2 and Renesas V850ES).

3.2 Background

In this section, we establish the scope of devices and firmware analyzed. We outline their composition, the standard approaches to analyze them, and the challenges faced.

The target of our approach is *hard to analyze* embedded device firmware. We define this as monolithic interrupt-driven software operating with or without an OS. For firmware with an OS, we restrict the definition to firmware that operates under real-time constraints. We refer to firmware without an OS as *bare metal*. We call the combination of a device and its firmware a System Under Test (SUT).

The execution of interrupt-driven software is directed by handling and processing interrupts. Interrupts are triggered by (on- and off-chip) peripherals, the OS, and system tasks. A task represents the execution of a part of software or firmware responsible for performing a specific function, such as input processing. Interrupt-driven software is composed of one or many tasks, which together constitute its functionality. An RTOS is a specific type of interrupt-driven software that operates under soft or hard real-time constraints. Time-sensitive tasks orchestrated by the OS must be completed within strict windows to adhere to these constraints. An RTOS manages the execution of many concurrent tasks. Each task can be interrupted by one running at a higher priority, and interrupts may be nested.

3.2.1 Reverse Engineering Embedded Firmware

Reverse engineering is the process of taking a SUT and determining properties about its inner workings. The principal objective of reverse engineering is to understand the SUT to explain how it works and extract specific implementation details. Therefore,

the process necessarily involves a high degree of human intervention—both during the analysis stage and afterward.

We perform reverse engineering using two complementary classes of analyses: static and dynamic. Static analyses form conclusions about a SUT by analyzing it at rest. In contrast, dynamic analyses allow us to reason about observations of its runtime behavior. Interactive disassemblers such as IDA Pro [109] and Ghidra [69] are at the core of most reverse engineering approaches. They enable an analyst to interact and manipulate a static representation of firmware to gain a deep understanding of its functionality. Similarly, debuggers and execution tracers fulfill that role for dynamic approaches by enabling us to reason about the SUT's execution at different points.

Interactive disassemblers represent programs using a database. This database typically contains each executable segment's disassembly, a control-flow graph, identified functions, and cross-references between code and data. For commodity software, interactive disassemblers can automatically populate the database. However, for most embedded firmware, database creation involves manual effort. RTOS-based and bare-metal firmware do not have standard container formats and vary widely in their composition. The only commonality is that their application logic, library code, static data, and OS (if present) all exist within the same binary blob. Thus, to populate the database, we need to determine the firmware's memory mapping (*i.e.*, where its regions are loaded) and ISA. As interrupt-driven firmware does not have a single entry-point, we must also identify interrupt vector tables, addresses of callbacks for tasks, and other initialization routines.

Disassembly describes the process of translating raw bytes into architecture-specific instructions. A disassembler requires viable instruction starting offsets within firmware to perform correct disassembly. For embedded ISAs, locating these offsets is not a trivial task. Many instruction sets are variable length encoded (VLE) with short instruction widths, so plausible disassemblies exist at many offsets, of which only one is correct. For some architectures, such as ARM/Thumb2, the correct instruction set to use for

disassembly is context dependent, further complicating the process. Identified function prologues are a common means to locate starting points. However, as non-standard calling conventions are commonplace in embedded firmware, we cannot always rely on known instruction patterns to determine starting points and often must identify the offsets manually.

To analyze the control-flow of firmware, we recover it from the disassembly. Control-flow recovery algorithms reconstruct a Control Flow Graph (CFG) containing the transitions between blocks of disassembled instructions. In this case, a block is a sequence of instructions terminated by a control-flow altering instruction, e.g., a branch or a call. Since interrupt-driven software contains a high degree of indirect control-flow, it is difficult to statically estimate the destinations of many of these branches and calls (a well documented problem [110, 111]). Therefore, static control-flow recovery for embedded firmware often results in highly incomplete CFGs [112].

When reverse engineering large and complex software, such as firmware, we must identify where to focus our analysis effort. Symbols (function names) are beneficial for this purpose. However, they are often not present in embedded firmware. Static data (debug strings) are also useful, as we can analyze code that references strings of interest under the assumption that code will reference strings related to its functionality. When all of the data exists in the same section, this is a trivial process. However, in firmware, it is often embedded inside code segments to avoid expensive references to locations at offsets exceeding what can be addressed by a single load instruction. Therefore, when disassembly and control-flow recovery results in an incomplete program database, the amount of data references to guide the reverse engineering process will be similarly limited.

To analyze a device dynamically, we cannot debug it *live* without interfering with how it communicates with its peripherals. To obtain useful analysis results, we must faithfully simulate a realistic external environment, e.g., by providing its peripherals with input. Without this, the firmware will stall until it receives an expected input.

Identifying stall points and handling them is necessary to enable most kinds of dynamic analysis.

When available, execution tracing facilities allow us to work around the problems associated with debugging. Execution tracing involves tracking part of the state of a SUT over an execution. The granularity of this tracking is dependent on the tracing technology available. Many devices offer hardware-based support for execution tracing [113]. While lightweight, this type of tracing has some limitations. Due to hardware constraints, traces are often limited in size and can only capture certain types of events, such as the program counter's value at branches. Under certain conditions, e.g., when firmware executes a tight loop or an interrupt handler, events may get dropped, resulting in fragmented traces. Some tracing technologies support starting tracing on a particular condition. However, we must specify these conditions before beginning tracing. As this kind of tracing is system-wide, it captures events across multiple tasks. Traces, therefore, represent a temporal ordering of events rather than real control-flow.

3.3 Overview

In this section, we detail the challenges tackled by our approach, Incision. We then list the assumptions made in its design and provide an overview of how it operates. As detailed in the previous section, embedded device firmware introduces significant complications to the reverse engineering process. The challenges below represent the difficulties present when reverse engineering embedded device firmware compared to commodity software:

1. State-of-the-art tools provide limited support for embedded architectures resulting in disassembly errors that propagate through the analysis process.

2. Embedded device firmware is monolithic and often contains no symbol information. When debug information is present, it is usually not located within a single

section but mixed with code and data, making it difficult to leverage for firmware comprehension.

3. Many end-user devices offer limited hardware tracing facilities. The available mechanisms are often error-prone and limited in terms of the amount and type of events they can capture.

4. Captured traces of interrupt-driven firmware do not reflect real control-flow and instead represent a temporal ordering of events.

5. Combining static and dynamic analysis methods is difficult due to the strong coupling of hardware and software. Analysis requires a human in the loop to orchestrate trace capture, provide input to the SUT, and reset it in case of error.

We build Incision around Ghidra [69], an open-source interactive disassembler. We integrate it into the standard reverse engineering workflow, where static and dynamic analyses are used in conjunction to form conclusions about the SUT. We make the following assumptions about the type of devices and firmware analyzed:

1. We know the ISA of the firmware, its memory mapping, and can identify at least one entry-point.

2. We can trace the execution of the device (e.g., using hardware-based trace facilities). The granularity of traces produced is at least at the basic block-level.

3. Captured traces may contain errors, *i.e.*, dropped events, but no active measures have been taken to restrict tracing. If present, these errors are reported.

Figure 3.1 provides an overview of our approach. We first import the firmware into Ghidra to produce a program database. We perform region inference (step ❶) based on an initial reverse engineering policy (an encoding of our initial analysis goals) using the database as input. Region inference identifies firmware regions corresponding to the policy. We use these regions' bounds to configure the SUT's tracing mechanism

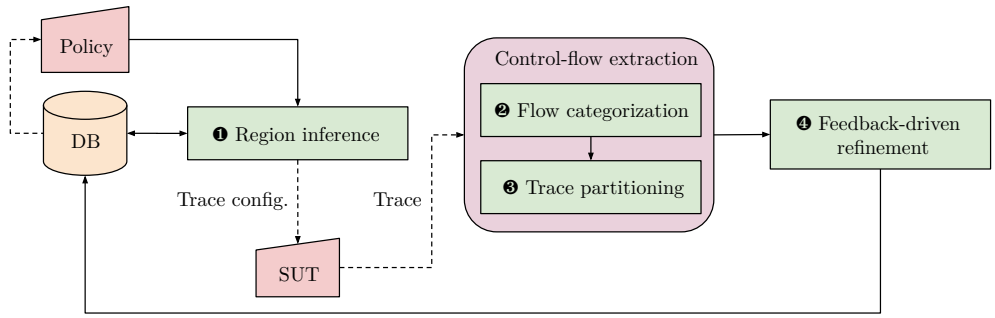


Figure 3.1: Overview of INCISION. We indicate data-flow by block arrows for automatic propagation and dashed arrows for manual propagation. INCISION operates in four step feedback loops. Region inference (step ❶) takes a policy and derives bounds to form a trace configuration. We use the bounds to take a trace of the SUT. From the trace, we perform control-flow extraction (steps ❷ and ❸) to identify task-switching logic and obtain task-level CFGs. Finally, we integrate the extracted CFGs into our DB and apply a static analysis pass to propagate the newly added information (step ❹).

and perform trace capture. From this, we obtain a system-wide trace containing the interwoven control-flow of multiple tasks. To extract each task's control-flow, we analyze the trace in two stages. In the first stage (step ❷), we determine which parts of the trace correspond to task switching logic and interrupt handling. In the second stage (step ❸), we split the trace based on the first stage, by extracting sub-traces separated by task switching or interrupt handling logic. We then recover each task's control-flow by combining sub-traces into a dynamic CFG per task. Finally, we update the program database using the extracted control-flow (step ❹) by performing disassembly of previously unexplored regions and adding control-flow edges. We follow this with a static control-flow recovery pass over the updated program database. This pass allows us to recover additional control-flow edges and data references not identified in traces or the initial database and correct disassembly errors.

We apply steps ❶-❹ iteratively. On each iteration, we refine our database, while at the same time, we *zoom* into the firmware regions corresponding to our reverse engineering goals. Each iteration necessarily requires some human intervention for processes that cannot be automated: to import the firmware into Ghidra, to perform trace capture (as the mechanism for doing so is generally hardware-based), to provide input to our region inference algorithm (which requires encoding of our reverse engineering goals), and

to resolve database conflicts we cannot address automatically. After several iterations, our framework's output (firmware database and traces) provides a basis for performing further analyses, such as those listed in Table 3.1.

3.4 Methodology

In this section, we describe the technical details of our approach. We discuss each component and the corresponding challenges it addresses.

3.4.1 Inputs

INCISION takes three kinds of input, a program database, execution traces, and reverse engineering policies. A program database corresponds to a Ghidra database. This database is created by loading the firmware into Ghidra at the correct image-base and triggering a basic *auto-analysis* pass. From this, we obtain an initial set of function starts, a call graph, and an intra-procedural control-flow graph for each identified function.

We base our execution trace format on the ARM ETB format [114]. It captures block-level control-flow event packets and encodes two high-level trace errors. The first kind of error is for trace capture toggles. We use this to flag that event packets are missing because of a trace parameter. The other type of error is for trace mechanism errors. We use it to flag when event packets are missing due to overflows. A common cause of this type of error are events that occur too frequently to be processed by the trace hardware.

We encode reverse engineering objectives as reverse engineering policies. We specify policies as a vector of one or more: keywords, symbol names, raw addresses, and instruction patterns. The first few parameters for the policy define the regions to reverse engineer and the last parameter defines the target to reverse engineer. Keywords, symbol names, raw addresses can be used in both region definition and target definition:

⟨<region definition 1>, <region definition 2>, ..., `<target>`⟩

Keywords match static data, symbol names match program database labels, raw addresses match referents of database cross-references, and instruction patterns match disassembled instructions that have specific syntactic properties. We write symbol and keyword matchers as regular expressions. An instruction pattern may be suffixed with `+` to indicate that it should only yield a successful match when it repeatedly occurs in a trace (e.g., as part of a loop). When no reverse engineering region is defined, we use the whole firmware as the region to apply the policy. The policy ⟨`(mov ?, 0xcafe)+`⟩, for example, matches instruction locations where the constant `0xcafe` is used as a source operand in a `mov` operation repeatedly over a trace. At the same time, ⟨"Rrc", "RrcSmc", `memcpy`⟩ encodes the goal of locating `memcpy` usage within a region with references to static data containing the strings "Rrc" and "RrcSmc". Policies are used as input to our region inference and feedback-driven refinement algorithms. They evolve across each iteration of Incision to become finer-grained, reflecting our increased understanding of the SUT over time.

3.4.2 Region Inference

To determine what to trace and which part of the firmware to focus on, we perform region inference. This process locates firmware region boundaries corresponding to functionality specified by the input reverse engineering policy. We perform inference in two steps: region identification and region grouping (shown in Algorithm 1). Region identification locates disparate regions (functions) corresponding directly to the input policy, while region grouping locates the boundaries of encompassing regions and includes transitively related functionality. Grouping is required as many embedded tracing mechanisms are limited in the number of tracing regions (inclusions or exclusions) that can be specified. By locating larger encompassing regions, we, therefore, ensure traces are complete and

specific and minimize overheads due to entering and leaving trace regions, which can cause trace errors.

Algorithm 1: Region inference.

In: D: program database, P: policy, R: max. number of regions, δ: grouping distance.
Out: M_r: at most R region bounds for P.

```
1  Rs ← staticAndSymbolRefs(D), M ← {}
2  foreach r ∈ Rs do // region identification
3  |  if match(D, P, r) then
4  |  |  k ← matched(D, P, r)
5  |  |  f ← enclosingFn(D, r)
6  |  |  M_k ← M_k ∪ {(r, f)}
7  M_r ← rankFnRefs(D, M, δ)
8  if |M_r| > R then // region grouping
9  |  M_r ← agglCluster(D, M_r, R)
10 return M_r
```

Region Identification



(a) (b)

Figure 3.2: **(a):** Region identification applied to LTE firmware using a policy of ⟨"NAS"⟩.[1] **(b):** Region grouping applied to the result of (a) for $R = 4$.

We perform region identification by identifying functions containing elements or references to elements matching the input policy. We then group the functions by their locality and rank them based on the number of times each group matches each input policy element. We consider references to be in the same group if they occur within δ bytes of each other; we set δ as twice the database's median function size. We rank policy elements associated with each region by the number of times they are referenced or occur, forming a ranking of all referencing regions of an element based on their relative frequency. This ranking metric is based on the observation that the functionality of a

[1]NAS (Non-Access Stratum) layer protocol is used to communicate control plane information in a telecom network.

region which heavily references a particular element is likely to be closely related to that specific element. Since we match static data and symbols based on both full and partial matches, we treat strongly related referenced data as a single entity. By doing so, we limit the amount of static data, such as debug strings, considered in isolation due to being embedded near the code that uses it and only references it once.

Region Grouping

Region identification locates sections of firmware containing functionality specific to a given policy. However, we may identify more regions than our trace mechanism can handle if the functionality is split over non-contiguous regions separated by more than δ bytes. We show such a scenario in Figure 3.2a, which depicts region identification applied using the policy ⟨"NAS"⟩ to an LTE baseband firmware. To overcome this, we combine the identified regions by grouping them into R larger regions.

To perform grouping, we use hierarchical agglomerative clustering [115] with proximity as the distance metric. This type of clustering works by iteratively merging regions based on their proximity until R remain. We base our grouping method on the observation that functions with related functionality (*i.e.*, that are used together in the same module to provide some higher-level functionality) are generally located close together in program binaries. We stipulate that this is due to how software is developed—the standard practice is to put related code into the same module, and this grouping gets preserved during compilation. Figure 3.2b shows region grouping applied to the result of region identification with $R = 4$. The clustering groups the disparate regions in Figure 3.2a into four larger regions containing related functionality.

3.4.3 Control-Flow Extraction

System-wide traces contain a mix of application-level code (*i.e.*, task code), interrupt handling, and OS-level task switching logic. To extract the control-flow for each of these sub-traces, we use the CFG representation from the program database to *follow* the

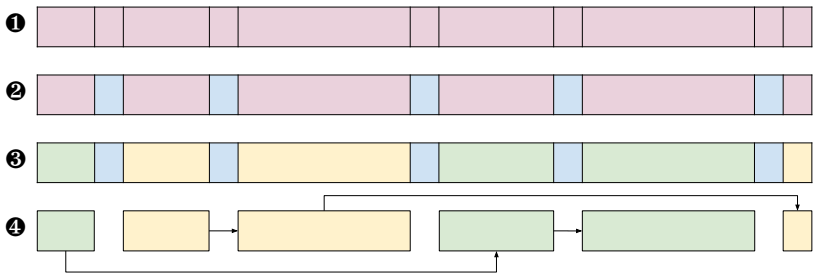


Figure 3.3: Visualization of control-flow extraction for two tasks. We first identify contiguous regions of viable control-flow in the input trace (step ❶), then identify task switching and interrupt handling logic (shown in blue in step ❷), identify the control-flow for each task (task 1 shown in green and task 2 in yellow in step ❸), and finally, perform task-level CFG extraction (step ❹).

execution of the trace. We first identify continuous flows of blocks such that there exist *real* control-flow transitions between each contained block. We then categorize blocks we could not determine viable transitions to or from, using the following labels: *breaks* for context-switch like behavior, *unresolved* for feasible transitions we could not yet resolve due to an incomplete database (e.g., indirect flows), and *errors* for blocks that do not exist within the database. Based on this categorization, we split the trace into flows that correspond to either task code and context switching logic. This enables us to isolate the control-flow of each trace part, and extract each task's control-flow. We visualize the complete process in Figure 3.3.

Flow Identification

We identify flows (visualized by step ❶ in Figure 3.3) by performing a single pass over the input trace. We process each block pairwise with its successor. For each pair of blocks, we use Algorithm 2 to categorize the transition between them as either *viable*, an *error*, or *unresolved*. We use the *error* label for transitions where either block failed to disassemble. We use *viable* when there exists an irrefutable flow between the blocks due to a branch, call, or return. To handle returns, we maintain a call stack by pushing the address following a call onto the stack and popping the top of the stack on each return. We consider a return between two blocks *viable* if the address at the top of the call stack

matches the second block's address. Since a trace may contain events for several tasks, we cannot rely on it to help us resolve indirect control-flow. Thus, at this stage, we conservatively handle indirect calls. To do so, we propagate constants referenced in the function enclosing the first block. If this enables us to resolve the call-site target to the second block's address, we label the transition *viable*. We handle additional cases later when performing control-flow graph extraction (Section 14).

Algorithm 2: Transition classification.

In: D: program database, $G^{\rightarrow}$: dynamic ICFG, b_i: first block, b_j: successor of b_i, S: call stack, N: reachability bound, T: execution trace.
Out: C: classification, $G^{\rightarrow}$: new dynamic ICFG, S: new call stack.

$G \leftarrow ICFG(D)$, $C \leftarrow unresolved$
if $disasmError(D, b_i) \vee disasmError(D, b_j)$ **then**
  $C \leftarrow error$
else if $traceError(T, b_i, b_j)$ **then**
  if $reachableIn(G^{\rightarrow}, b_i, b_j, S, N)$
  $\vee$ $reachableIn(G, b_i, b_j, S, N)$ **then**
    $C \leftarrow viable$
else if $directFlow(G, b_i, b_j, S)$ **then**
  $C \leftarrow viable$
else if $isIndirectFlowOut(G, b_i)$ **then**
  $f \leftarrow enclosingFn(D, b_i)$
  $f_{b_i} \leftarrow propagateConstsAndBounds(D, f)$
  if $indirectFlow(G, f_{b_i}, b_i, b_j, S)$ **then**
    $C \leftarrow viable$
if $C \neq error \wedge \neg traceError(T, b_i, b_j)$ **then**
  $G^{\rightarrow} \leftarrow updateDynamicICFG(G^{\rightarrow}, b_i, b_j)$
else if $C = viable$ **then**
  $S \leftarrow updateCallStack(G, b_i, b_j, S)$
return $\langle C, G^{\rightarrow}, S \rangle$

When we encounter a tracer-induced error between two blocks, it indicates that at least one block is missing from the trace. If context switching occurred between the blocks, then the second will not be reachable from the first by regular control-flow. However, if there should be a *viable* transition between the blocks, then there will be some path in the program-wide CFG connecting them in some bound N of intermediate blocks. To capture this, as we process traces, we construct a dynamic CFG containing transitions between blocks across untraced regions. We then use this CFG to approximate *viable*

transition targets when errors are present. As this may lead to unsoundness when we label error transitions as *viable*, to minimize false positives, N must be small; for all firmware analyzed herein, we set $N = 3$. When no viable flow can be identified based on the current database, we terminate the current flow and classify the transition as *unresolved*.

Flow Classification

To determine which flows correspond to task code or context-switching, we identify which *unresolved* transitions are *from* task-switching logic and *to* interrupt handlers that should be re-classifed as *breaks*. This process enables us to find out where task code starts and ends. To perform re-labelling, we analyze how the prefixes and suffixes of flows repeat across the entire trace. We base this on the observation that task switching and interrupt handling code exhibits execution patterns distinct from regular code. We use a suffix array to represent the input trace and an Longest Common Prefix (LCP) array to locate repeated prefixes and suffixes, constructed using Algorithm 3. An LCP array enables us to determine the length of the common prefix between two sub-sequences (or flows).

Algorithm 3: Suffix and LCP array construction.

In: T: execution trace, d: prefixes or suffixes.
Out: S: suffix array, L: LCP array w/o overlaps, M_T: address to suffix array mapping.

```
1  Σ ← makeAlphabet()
2  foreach b ∈ T do
3  |  if address(b) ∉ Σ ∨ hasError(b) then
4  |  |  k ← next(Σ)
5  |  |  Σ_b ← k
6  Σ^⊥ ← next(Σ) // compute sentinel
7  if computeBackwards(d) then
8  |  M_T ← makeReverseMapping(T, Σ, Σ^⊥)
9  else
10 |  M_T ← makeMapping(T, Σ, Σ^⊥)
11 S ← SuffixArray(M_T, Σ^⊥)
12 L ← removeOverlaps(LCPArray(M_T, S))
13 return ⟨S, L, M_T⟩
```

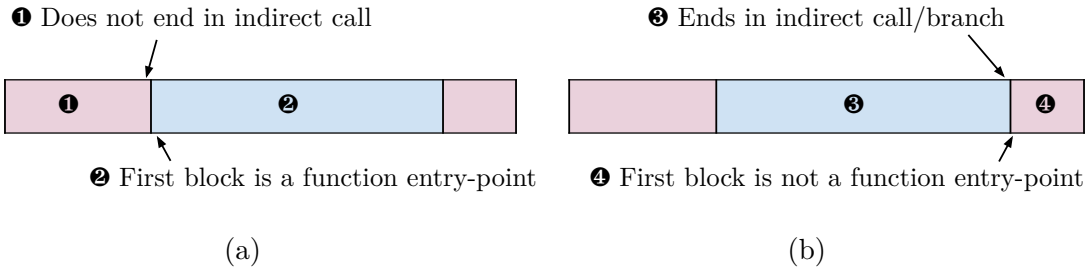


Figure 3.4: **(a)** When there is no viable control-flow between two flows (e.g., where ❶ does not end on an indirect call and ❷'s first block is a function start), we classify the second flow as an ISR (❷ in blue). **(b)** When there is viable control-flow between two flows, but it is not regular (e.g., where ❸'s last block ends on an indirect call and ❹'s first block is not a function start), we classify the first as task-switching logic (❸ in blue).

Neither suffix arrays nor LCP arrays are directly amenable for representing trace data. Thus, we perform some adaptations. Suffix arrays operate on sequences taken from a fixed alphabet Σ. For traces, we use an alphabet derived from block start addresses. We consider two blocks with the same address distinct if a trace error precedes or follows them. Suffix arrays also require us to signify the end of an input sequence, a so-called sentinel character $\Sigma^{\perp}$. For traces, we select this character as $\max(\Sigma) + 1$. By design, an LCP array enables us to find *all* repeated sub-sequences of an input sequence, including those that overlap. However, for execution traces, only those that do not overlap are meaningful. We thus compute the length of the common prefix between two sequences as the minimum of the reported LCP length and the difference between their indices in the suffix array.

Identifying Transitions to Interrupt-Handling Logic We identify the starting points of context-switching code by identifying transitions to ISRs. Our intuition for this is based on the fact that task switches as well as other kinds of context-switching are typically preceded by an Interrupt Request (IRQ) (e.g., a timer interrupt preempting a certain task). IRQs are then handled by an ISR or interrupt handler, which determines how the interrupt will be processed.

To locate ISRs in a generic way, we first identify pairs of neighboring flows, where ❶ the last block of the first flow does not end in an indirect call or branch, and ❷ the first

block of the second flow is the entry-block of a function, as shown in Figure 3.4a. To distinguish transitions to ISRs from other *unresolved* transitions matching this criteria, we filter them by only considering those that have execution patterns that appear frequently in the trace (which we identify using the LCP array). We detail the complete procedure in Algorithm 4.

Algorithm 4: Task switch start identification.

In: D: program database, L_p: LCP array for prefixes, C_u: unresolved transitions, N: min. prefix length, k: max. ISRs, δ: min. proportion of satisfying transitions.
Out: C: transitions into the ISR.

```
1  G ← ICFG(D)
2  M ← {}, P ← {}, T ← {}
3  foreach (b_i, b_j) ∈ C_u do
4  |   f ← enclosingFn(D, b_j)
5  |   if isFnStart(D, b_j) ∧ ¬reachable(D, b_i, f) then
6  |   |   M_{b_j} ← succ(M_{b_j})
7  |   T_{b_j} ← succ(T_{b_j})
8  foreach b_j ∈ M do
9  |   if M_{b_j}/T_{b_j} > δ then
10 |   |   v ← repeatedPrefixes(L_p, N, b_j)
11 |   |   if |v| = 1 then
12 |   |   |   P_{b_j} ← v
13 C ← take(k, sort(P))
14 return C
```

Identifying Transitions from Task-Switching Logic Similar to interrupt-handling logic, we locate task-switching code by identifying pairs of neighboring flows with specific properties, visualized in Figure 3.4b. Before task-switching happens, a task context will be set-up or restored from a global structure; then, control will transition to the newly created or resumed task using an indirect branch or call. We can distinguish these calls and branches from regular indirect calls and branches by their targets. For task-switching logic, the set of possible targets is unbounded, and they are unrestricted in their locality. Further, they can be function starts, block starts, and the middle of basic blocks. The number of firmware locations for performing task switching is small (usually one) and

frequently appears within traces.

Thus, to identify task-switching logic, we construct an ordering over the last block of each identified flow. We consider the highest-scoring block to be part of the flow performing task-switching. We score the blocks based on the following criteria: ❶ (greater) if the block occurs at the end of more than one flow and is followed by blocks that are function starts and blocks within functions, ❷ (greater) if there is no trace error between the block and the next, ❸ the total number of times the block occurs at the end of a flow, ❹ the number of unique blocks that follow the block, and ❺ the length of the longest common suffix of the flows terminated by the block (computed using the LCP array). Criteria ❺ reflects that set-up code for task switching is mostly branch free.

Task-Aware Control-Flow Extraction

To extract task-level control-flow, we first merge flows of the same kind (task-switching logic or task code). To do this, we use the labels applied to flows and transitions in the previous phases. We then merge non-adjacent flows to build flows for each task. Finally, we output the task-level control-flow graphs derived from the merged flows.

We always elect to merge adjacent flows that are both labeled as task-switching logic, which occurs when the task-switch code is itself interrupted (nested interrupts). We merge adjacent task flows separated by *unresolved* transitions if the first flow ends with a computed call (e.g., `call [R0, #offset]`). We also merge flows if the second flow starts at the entry of a function or if the first flow ends on a return and the second starts at a block whose predecessor ends on a computed call target.

To merge non-adjacent flows and hence detect task-resumption, we maintain a stack of flows while merging. At each point following an identified task switch, we attempt to merge the next flow with one of the unmatched flows in our stack. We consider two flows to match when the second starts (or resumes) in the same block as the first ends on, or when the second starts in a block reachable within a single transition from the block that the first ends on, modeling resumption into a successor block. When a trace

mechanism drops events, identifying task resumption is generally not possible because the ends and starts of flows may miss intermediate blocks. Further, due to code-sharing between tasks, and as traces only contain basic block starts, it is not always possible to discern which task resumed if the task-switch occurred when executing shared code (e.g., library code). We detail the complete procedure in Algorithm 5.

Algorithm 5: Control-flow extraction.

In: D: program database, F: identified flows, C: transition classifications, P: policy, L: LCP array, η: merge threshold.
Out: $F^{\rightarrow}$: flows grouped by tasks, $G^{\rightrightarrows}$: map of tasks to dynamic CFGs with policy labels.

$F^{\rightarrow} \leftarrow \{\}$, $G^{\rightrightarrows} \leftarrow \{\}$, $prev \leftarrow \bot$, $pKind \leftarrow \bot$
foreach $f \in F$ **do**
  $b_s \leftarrow startBlock(D, f)$, $b_e \leftarrow endBlock(D, f)$
  if $\neg isTask(pKind)$
  $\wedge$ $isBreaks(C, endBlock(D, prev))$
  $\wedge$ $isBreaks(C, b_s)$ **then**
    `// contiguous switching logic`
    $prev \leftarrow mergeFlow(prev, f)$
  else if $isBreaks(C, b_s) \wedge isBreaks(C, b_e)$ **then**
    $updateFlow(F^{\rightarrow}, G^{\rightrightarrows}, L, P, prev, pKind)$
    $prev \leftarrow f$, $pKind \leftarrow context\text{-}switch$
  else
    $updateFlow(F^{\rightarrow}, G^{\rightrightarrows}, L, P, prev, pKind)$
    $f' \leftarrow findMatchingFlow(F^{\rightarrow}, f, \eta)$
    if $f' \neq \bot$ **then**
      $pKind \leftarrow task\text{-}resume$
    else
      $pKind \leftarrow task\text{-}start$
    $prev \leftarrow f$
$updateFlow(F^{\rightarrow}, G^{\rightrightarrows}, L, P, prev, pKind)$
return $\langle F^{\rightarrow}, G^{\rightrightarrows} \rangle$

3.4.4 Feedback-Driven Refinement

At the end of each iteration performed by Incision, we add control-flow edges to the program database based on extracted task-level CFGs, correct disassembly errors, and propagate that information across the entire database. We detail the full procedure in Algorithm 6.

Algorithm 6: Feedback-driven refinement.

In: D: program database, $G^{\rightrightarrows}$: map of tasks to dynamic CFGs, C_e: error transitions.
Out: D: new program database.

```
1 foreach G ∈ CFGS(G⇉) do
2 |   D ← mergeDatabaseCFG(D, G)
3 foreach e ∈ C_e do
4 |   blocks ← disassembleFrom(D, e)
5 |   if blocks = ⊥ then
6 |   |   blocks ← updateViaOracle(D, e)
7 |   D ← updateDatabase(D, blocks)
8 D ← autoAnalyse(D)
9 return D
```

We apply Ghidra's auto-analysis to perform propagation, which identifies new function starts and code/data cross-references. Propagation acts as a feedback-mechanism. It enables us to incorporate reverse engineering policy-specific information extracted from the input trace (such as control-flow edges) and use it to drive further static analysis-based refinement, which will discover additional functions and cross-references.

To handle disassembly errors identified while processing the trace, we re-disassemble blocks using alternative disassembler options. For ARM, for example, we switch to/from Thumb mode. To ensure that incorrect disassembly does not propagate across the database, we re-disassemble blocks transitioning into "error blocks" and force Ghidra to rebuild their enclosing functions to correct the function-level control-flow. This automated process can fail when there is a conflict in the database. For example, when an incoming edge forces disassembly to be performed incorrectly with a specific set of options, or when a function is incorrectly labeled as non-returning causes disassembly to stop and miss code following calls to that function. To remedy this, we require human intervention to correct the error. In practice, the frequency that is required is low, and the intervention is usually minor (Section 3.5.4).

Following propagation, to locate repeated instruction patterns from our input policy, we label each function with any instruction patterns it matches. To find them, we use the LCP array from the previous phase to locate repeated sequences containing each

matched instruction's enclosing block. Upon completion of an iteration, we terminate Incision when our reverse engineering objectives have been satisfied (e.g., we identified the firmware region responsible for a given behavior). Otherwise, we update our reverse engineering policy and perform another iteration of Incision's feedback loop.

3.5 Evaluation

In this section, we evaluate the effectiveness of Incision. We first assess each of its components separately: region inference (RI), task-based control-flow extraction (CE), and feedback-driven refinement (FD). Then, we apply Incision in full by performing real reverse engineering tasks. Our evaluation assesses *correctness*, *real-world usability*, and *human effort* using the following criteria:

C.1 Correctness: ❶ For CE, the extent to which Incision extracts correct control-flow of input traces, its ability to discern between task and context-switching logic, and the correctness of the identified flow bounds. ❷ For RI, the extent to which the tracing bounds reported by Incision correspond to functionality relevant to the input policy. ❸ For FD, the extent to which Incision improves the firmware database. ❹ For FD, whether Incision correctly rectifies disassembly errors.

C.2 Real-world usability: ❶ The extent to which input policies reflect real reverse engineering goals. ❷ How effectively Incision applies to real-world SUTs.

C.3 Human effort: For FD, the proportion of errors that require manual intervention compared to what Incision can fix automatically, how many changes to the database that equates to and the complexity of those changes.

3.5.1 Firmware Data-Set

We conduct our evaluation using firmware extracted from end-user devices and firmware adapted from open-source projects. Our end-user devices consist of a VxWorks (RTOS)

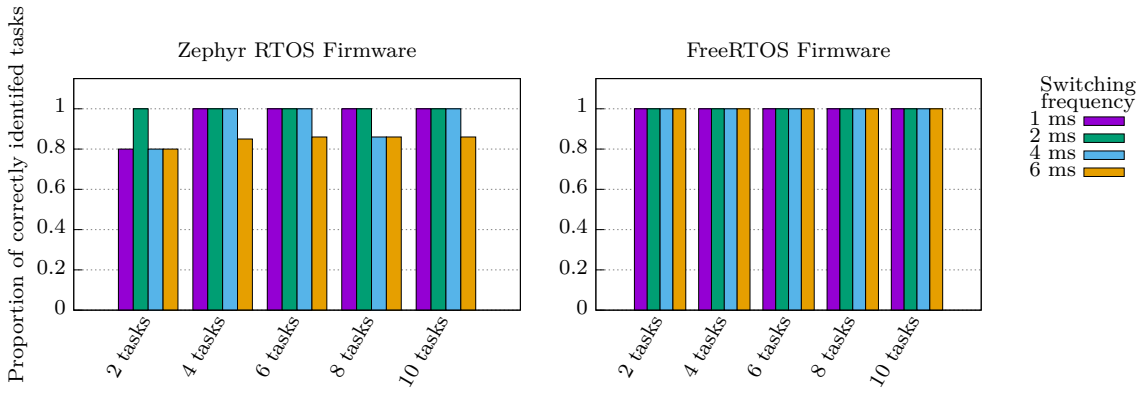


Figure 3.5: **Left:** Correctly identified tasks for Zephyr firmware; **Right:** Correctly identified tasks for FreeRTOS firmware. We group the results by task-switching frequency (1ms, 2ms, 4ms, and 6ms). We calculate the resulting correctness for each group as the average correctness over five trace samples of 50k blocks.

based Huawei LTE baseband targeting ARMv7 and a Renault BCM targeting Renesas V850ES. We use ARM CoreSight to trace the baseband, as detailed in Appendix B.1. For this particular device, CoreSight's trace capture buffer is limited to 1MB. For the BCM, we perform tracing using Renesas CubeSuite+ [116]. Both SUTs are of moderate complexity and interact with numerous peripherals. To establish a ground truth, we evaluate Incision using two open-source RTOSes, FreeRTOS, and Zephyr. To do so, we select existing example firmware from each project and expand its functionality by adding: encryption and decryption operations, sorting and searching operations using custom predicates (as a means to introduce many indirect calls, which represent an adversarial setting when performing control-flow extraction), and various data encoding and decoding schemes, each running as different OS tasks. In total, we construct ten firmware images, five for each OS. To capture execution traces, we use QEMU.

3.5.2 Correctness (C.1)

Control-Flow Extraction (C.1.1)

For this criterion, we evaluate using our FreeRTOS- and Zephyr- based firmware. We construct a ground-truth by manually identifying task code and the ISR responsible for triggering task switches from the firmware source code. We then match these locations

to regions in the compiled firmware. In the Zephyr-based firmware, the context switching starts with the clock interrupt `z_clock_isr` and for FreeRTOS-based firmware it starts with `xPortSysTickHandler`. In both OSes, the context switch ends with a `PendSV` exception. For each firmware, we vary the number of concurrent tasks from two to ten. We also configure the scheduling mechanism of each OS to vary the frequency of task switches from 1ms to 6ms.

Experiment: We trace each firmware and sample five non-overlapping sub-traces of 50k blocks from each trace with a different task switch frequency. Then, we compare the identified tasks, resumes, and switching logic to our ground truth.

Results: Figure 3.5 shows the proportion of correctly identified tasks averaged over five traces for each firmware. In over 80% of the cases, INCISION correctly identifies the task switching logic. Furthermore, when it correctly identifies the task switching behavior, INCISION can correctly discern between task starts and task resumes. The incorrectly identified task switches stem from a limited number of switches in the trace samples. The application-level indirect control-flow outnumbers the number of task switch targets, thus inducing false positives. This would not be an issue in practice, as traces from real-world devices have a higher prevalence of task-switching due to peripheral interaction.

Region Inference (C.1.2)

For this criterion, we assess the quality of the region bounds output by INCISION using the policies shown in Table 3.2. We use the Huawei LTE baseband for this experiment, as it is the most complex SUT in our data-set. We construct a ground truth by manually identifying regions based on symbols and debug strings in the firmware. We further support this evaluation in Section 3.5.3 by taking traces using the identified bounds for a real reverse engineering task.

Experiment: We measure quality by calculating the overlap between the regions identified in our ground truth and to those output by INCISION. For each policy, we limit

Table 3.2: Comparison of size and ratio of overlap of manually identified regions compared to regions identified by INCISION for policies used in Section 3.5.3.

Policy	Area of ground truth region	Area of overlap with identified region	Ratio of overlap
⟨"USIMM"⟩	95009	91677	96.49%
⟨"WAS"⟩	1412691	1116338	79.02%
⟨"NAS"⟩	946908	904951	95.57%
⟨"RRC"⟩	792107	722861	91.26%
⟨"RRC_SMC"⟩	9086	8524	93.81%

the number of regions to find to four to match the number of regions traceable using the SUT's CoreSight configuration.

Results: Table 3.2 shows that all of the ground truth bounds fall within regions output by INCISION, with at least 79% overlap. As demonstrated later in Section 3.5.3, these bounds enable us to acquire traces containing functionality matching the specified input policies. Further, those traces enable us to perform practical reverse engineering. In this way, INCISION remedies some of the limitations of hardware-based tracing mechanisms, such as trace buffer limits. It can narrow the number of regions to trace while still enable the capture of relevant functionality.

Database Improvement (C.1.3)

To assess this criterion, we compare the increase in identified function starts after each iteration of INCISION to a baseline of Ghidra's auto-analysis. We use our real-world firmware for this experiment. To establish a ground truth of function starts for the baseband, we use leaked firmware symbol information. For the BCM, we establish a ground truth by manually reverse engineering its firmware.

Experiment: We capture traces for each SUT and process them using INCISION. We use two traces for the baseband (Figure 3.6, left) and six traces for the BCM (Figure 3.6, right).

Results: For both firmware databases, the use of INCISION results in an increase in the number of identified functions over Ghidra. For the baseband, the increases are due

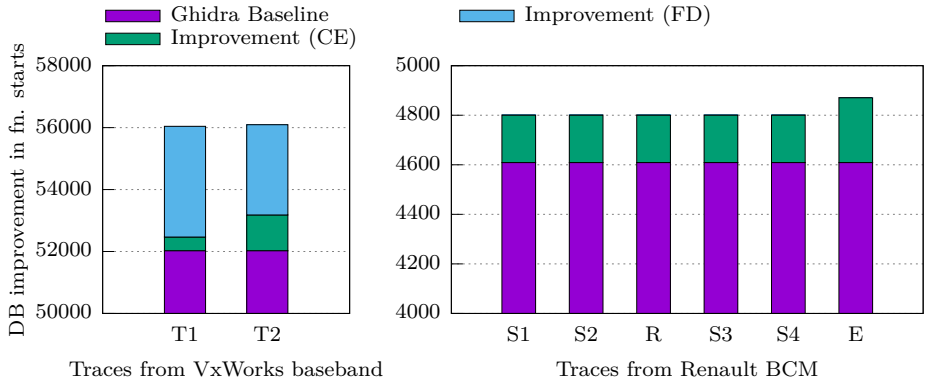


Figure 3.6: **Left:** Database improvement for VxWorks baseband firmware; **Right:** Database improvement for Renault BCM firmware. Improvement measured in correct function starts; categorized by: Ghidra's auto-analysis (Ghidra Baseline), improvement due to control-flow extraction (CE), and improvement due to feedback-driven refinement (FD).

to adding task-level CFGs into the database and the subsequent feedback process. The task-level CFGs trigger further discovery of functions not found in either the processed traces or original database. For the BCM, all increases are due to the extracted CFGs. Notably, by combining static and dynamic analyses in a feedback loop, INCISION recovers more function starts than present in either the traces or the baseline Ghidra database.

Database Error Correction (C.1.4)

We assess this criterion using the same firmware and traces as the previous section (Section 3.5.2), as shown in Figure 3.6.

Experiment: We process each trace using INCISION and assess how it rectifies reported disassembly errors. For each "error block" our tool reports it can fix automatically, we verify that the proposed correction is in line with how a human analyst would proceed.

Results: For the BCM firmware, we encounter no disassembly errors for any of the traces. For the baseband firmware, we encounter errors for both traces; trace T1 yields 53 errors out of 88k blocks, and T2 yields 188 out of 200k blocks. Our results are in line with our expectations. The BCM firmware uses the V850ES instruction set, which has a single configuration. Therefore if our traces contain correct block starts,

then no disassembly errors should be introduced by processing them. In contrast, the baseband firmware uses both the ARM and Thumb2 instruction sets. As reported by Friebertshäuser [112], most tools perform relatively poorly for firmware that mixes these instruction sets since the correct disassembly context is often only apparent at runtime.

For trace T1, INCISION reports a fix for 33 out of the 53 errors. Of these, only one is incorrect. On investigation, we find that this is because Ghidra reverts INCISION's fix. Ghidra performs a "Non-returning Function Detection" pass as part of its *auto-analysis*, which INCISION triggers as part of its feedback process. The result is that the thunk function inserted by INCISION gets removed by Ghidra. For trace T2, INCISION proposes fixes for 135 out of the 188 errors, all of which are correct. Overall, we find that disassembly errors rarely occur. However, when present, INCISION can apply automatic fixes that handle the majority of cases.

3.5.3 Real-World Usability (C.2.1, C.2.2)

Despite the large number of automated firmware analysis techniques proposed in the literature, it is rare to find any evaluated using firmware extracted from end-user devices. In practice, it is often the case that these techniques operate under specific assumptions, which are target-dependent and may not always hold. To apply these techniques to end-user devices, we must almost always perform preliminary manual reverse engineering to make necessary adaptations.

In this section, we show how INCISION can complement techniques by supporting that preliminary reverse engineering effort. To that end, we evaluate the firmware of two end-user devices. These devices and their firmware share several properties that make them challenging to analyze. ❶ Both firmware coordinate various complex layers and protocols (RRC for the baseband and an immobilizer [117] and diagnostics [57] for the BCM). ❷ Even with JTAG access, acquiring representative execution traces from either device remains a challenging task. Due to a lack of tracing hardware on the BCM, as it is not real-time constrained, we resort to emulation to acquire traces. For the baseband,

we reverse engineer and configure the on-chip tracing hardware for trace acquisition. However, our traces are often incomplete and limited in size. Thus, we must carefully select which regions to trace in order to perform practical analysis. ❸ Both devices rely on various on- and off-chip peripherals. The baseband incorporates an RF component for LTE radio communication. While the BCM includes an immobilizer base station, a low-frequency antenna, and peripherals to communicate over automotive buses (e.g., CAN).

Identifying Stall States to Emulate Renault BCM Firmware

In this experiment, we use INCISION to aid in locating stall states in the BCM firmware that prevent us from successfully emulating the device.

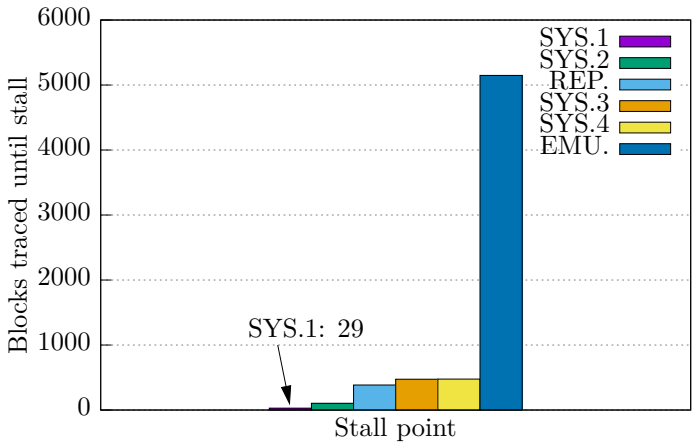


Figure 3.7: Traced block coverage (unique blocks) for BCM firmware for each stall point; tracing halted when buffer capacity (1MB) reached. After providing stimulus at each stall point (SYS.N, REP.), we are able to emulate the device firmware (EMU).

Set-up: As read/write protection is disabled on the BCM, we can obtain its firmware over JTAG. We use Renesas CubeSuite+ IDE to emulate the firmware and acquire traces.

Objective: We assess the effectiveness of INCISION to aid firmware emulation by detecting stall states in traces. An emulated firmware instance should be as true to the on-target execution as possible. However, emulation often runs into problems where the code hangs on a peripheral status register [62]. Thus, we need to handle such states

to facilitate emulation. We can identify such states in an execution trace by locating repeated flows that saturate the trace buffer. They might manifest as long repeated flows of code or short tight loops on particular blocks, e.g., when polling a peripheral register.

Experiment: We attempt to emulate the firmware by applying INCISION iteratively. In each iteration, we execute the firmware from the reset vector and halt when the trace buffer (limited to 1MB) is full. Since our objective is emulation, we do not restrict traces by a policy. Instead, we use policies to detect stall states; we define the following policies: ⟨`(tst1 ?, DAT_fffff*)+`⟩, to detect any repeated instructions which poll for a register bit in the memory region where peripherals reside, and ⟨`(cmp r0, r*)+`⟩, to detect if emulation stalls on the return value of a specific function.

Results: Figure 3.7 shows how INCISION gradually improves emulation over successive traces. The first trace makes little progress and stalls on an instruction that polls a bit in the *SYS* register. Since INCISION can identify the flow responsible for the stall, we can clear the corresponding bit in the register to advance the emulation past this point. Over all iterations, INCISION enables us to identify four stall-states waiting on the *SYS* register and one longer repeated sub-flow, which saturates the trace buffer. In the latter case, the repeated flow consists of two function calls. The emulation only proceeds if the second returns successfully. Even though approaches such as P^{2}IM [62] reportedly handle tight polling loops, such as those on the *SYS* register, it is unclear how they would identify the latter stall state. In our last trace, where we achieve emulation (EMU.), we observe a dramatic increase in the number of unique blocks executed after passing the initial states.

Analysing the Key Usage of the Huawei R216h

In this experiment, we use INCISION to aid in analyzing how the baseband handles sensitive key material.

Set-up: We obtain the baseband firmware over JTAG and configure tracing as detailed

in Appendix B.1. We also use JTAG to obtain RAM dumps. As mentioned previously, for this device, the CoreSight trace capture buffer is limited to 1MB, and the number of regions we can specify as inclusions or exclusions is limited to four. We provide input to SUT using an Ettus USRP B210 Software-Defined Radio (SDR). We detail the complete experimental set-up in Appendix B.2.

Objective: Our analysis goal is to identify how the firmware uses cryptographic keys and check that it sanitizes them correctly. We use INCISION to perform the first stage of analysis: to identify crucial functions and obtain relevant traces. We then use the improved database to help identify the buffers storing key material and analyze how the firmware uses them.

Experiment: To improve our initial database, we first attempt to capture traces of LTE layers known to handle cryptographic keys using the following policies: ⟨"NAS"⟩ and ⟨"RRC"⟩. From this we obtain traces T1 and T2, as shown in Figure 3.6 (left). These traces improve the quality of (relevant) information in our database (identified global buffers, static data references, resolved control-flow). They also enable us to form bounds to exclude task-switching behavior in subsequent traces. Next, we take traces excluding the task-switching region and the bounds obtained using the following (more specific) policies (traces shown in Figure 3.8): ⟨"RRC", "RRC_SMC"⟩ (traces R1 and R2), ⟨"NAS", "NAS_EMM", "NAS_LMM"⟩ (traces N1 and N2), and ⟨"USIMM"⟩ (trace U1). Using the extracted sub-flows from the traces captured, we identify firmware regions responsible for each protocol layer. Next, we use those regions and database cross-references to locate global buffers accessed by the code blocks present in identified sub-flows. The key material we are interested in (AES keys) should be at least 16 bytes in size and have high entropy. We use this as a filtering criterion: keeping only references to buffers with these properties that are referenced by code from our identified sub-flows. Finally, we perform a manual analysis of the key usage using the identified buffers and the improved database.

Results: In traces containing task-switching logic (i.e., T1 and T2), INCISION correctly identifies its bounds. It identifies the entry-point of `intEnt` as the ISR and a block in `reschedule` as the point responsible for task switching. In the remaining, finer-grained traces, we observe no switching logic as we can exclude its bounds. We find that all of our traces correspond to the policies used to take them. In this way, INCISION enables us to perform a reverse engineering task, which requires both static and dynamic analyses with minimal manual effort (see Section 3.5.4). It integrates information acquired dynamically into the firmware database, which allows us to form finer-grained reverse engineering policies. Further, it enables us to restrict taken traces to contain policy-relevant information by automatically locating the bounds of task switching/handling logic.

When performing the next stage of analysis (to identify key buffers), our improved database and traces enable us to reduce the number of buffers requiring manual analysis. The decrease is by two orders of magnitude—from thousands of buffers to tens, as visualized in Figure 3.8. Through our analysis, we locate the following keys: k_{nas_i}, k_{nas_e}, k_{enb}, k_{rrc_i}, k_{rrc_e}, k_{up_e}, ck, ik, and k_{asme}. After analyzing the code processing them, we find that, though all identified keys are considered session keys by the standard, only ck, ik, and k_{asme} are correctly sanitized.

3.5.4 Human Effort (C.3)

We evaluate this criterion using traces T1 and T2 for the baseband firmware. We compute the proportion of errors corrected automatically to those needing manual intervention. For each error requiring manual intervention, we measure its complexity ($\propto$ human effort) by the type and number of changes required to fix it in the database. We consider adding/removing a control-flow edge, label, or function start to be low-effort and performing multiple changes or other changes as high-effort.

Experiment: Using the same methodology as Section 3.5.2, we manually analyze each

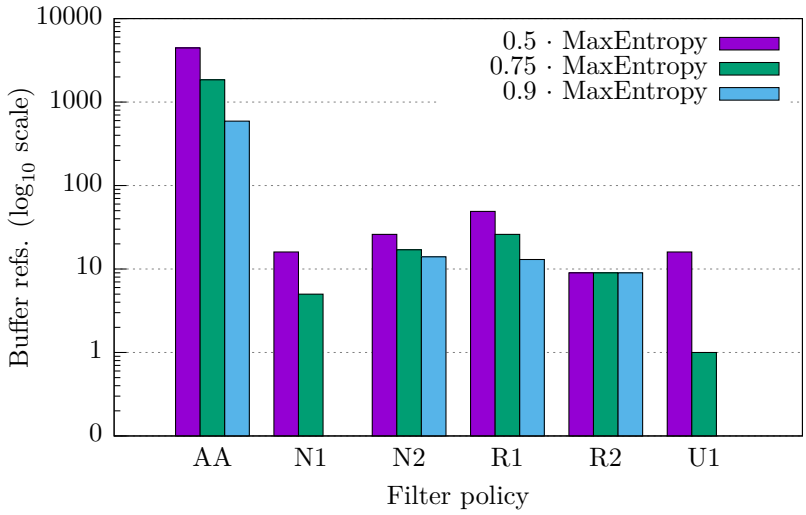


Figure 3.8: Number of key buffers to analyze. AA: initial database; Nn, Rn, Un: database after processing a trace with references kept if referenced by a block in the trace. We identify viable buffers as those with high entropy. We compute thresholds for keeping buffers as a proportion of their Max. (Shannon) Entropy (ME) relative to their size. Without INCISION, max. number of buffers to analyze is 4451 (AA, 0.5·ME), after using INCISION, max. number of buffers is 49 (R1, 0.5·ME). We use $\log_{10}$ scale for readability. Bars not shown in the diagram has value of 0

un-fixed error. We then apply a suitable fix manually and record the complexity of the fix performed in terms of database changes.

Results: We show the results in Figure 3.9. For trace T1, we encounter 53 disassembly related errors, and for T2, 188. The majority of errors could be corrected automatically by INCISION. Of those requiring manual intervention, all were "low-effort" fixes, requiring a single change. The majority consisted of creating a missing function start, and the rest, re-creation of a function.

Overall, INCISION dramatically improves the standard reverse engineering workflow for complex device firmware. Although its analyses are not entirely automated, it reduces human effort in almost all cases. Where manual intervention is needed (e.g., to perform tracing), it is unavoidable and not a limitation of our approach. Moreover, the manual effort required to use INCISION is minimal, consisting of minor changes to the database.

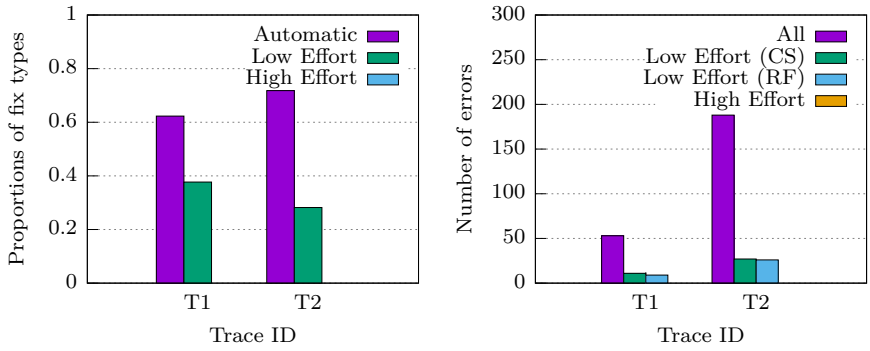


Figure 3.9: Number of disassembly errors resulting from baseband traces T1 and T2. **Left:** proportion of errors fixed automatically, errors fixed manually that are low effort, errors fixed manually that are high effort (0). **Right:** breakdown of errors; all errors reported, errors fixed: low effort where we create a function start (CS), where we force Ghidra to re-create a function (RF); No error requires high effort fix.

3.5.5 Discussion and Limitations

In this section, we discuss the possible limitations of our approach.

Firmware Composition In a scenario where several tasks within a trace heavily share code, Incision may incorrectly merge their flows. The may occur if one task resumes to a location where another was preempted (e.g., in library code). In this case, given the limited information available in our traces, there would be no way to distinguish between the flows without additional context.

Reverse Engineering Policy Granularity While a policy can encode realistic target-specific reverse engineering objectives, it cannot encode all possible goals. In particular, a policy cannot express relations between elements. While this could be considered a limitation, we do not consider it a limitation in the context of the tasks Incision is intended to perform, *i.e.*, the initial reverse engineering effort to enable other analyses.

Soundness and Completeness To handle trace mechanism errors, we estimate viable control-flow to make analysis possible. Therefore, we cannot guarantee soundness or completeness for analyses that build upon Incision's outputs where such errors are present. We consider this an acceptable trade-off as trace analysis would not otherwise

be possible.

Quantification of Human Effort When assessing **C.3**, to avoid biases based on prior human experience, we measure human effort as a function of the number of manual tasks performed and their complexity. This comes at the cost of not directly measuring the time taken to perform tasks (as that would depend on the subject's expertise). We argue that this metric is a reasonable alternative. The time taken to complete tasks is generally proportional to their complexity and amount, irrespective of prior human experience.

Tracing Mechanisms While many end-user devices provide tracing facilities, some vendors remove such features altogether. While it may be possible to alter a device's firmware to add instrumentation to provide a trace mechanism [118], the effort to do so is comparable to many everyday reverse-engineering tasks. Hence, removal of these facilities acts as a countermeasure against performing reverse-engineering using execution traces. Therefore, we consider devices without tracing features out of scope. For other devices, high-end or expensive equipment can overcome some of the problems we describe, such as limited trace buffers. However, for many end-user devices, these tools are often unavailable. We, therefore, exclude analysis scenarios involving them from our evaluation.

3.6 Related Work

Dynamic firmware analysis Avatar [93, 107] and Surrogates [95] introduced the concept of hardware in the loop analysis. This methodology allows dynamic analysis to be performed without handling many of the complexities of the device being analyzed. It enables hybrid analyses where a fast host can emulate most of the firmware and defer to the device only for I/O and interrupts. While these approaches reduce some of the complexities of analysis, interaction with the underlying device may introduce unacceptable overheads. For example, as Avatar relies on state transfer between the device and

host, the pause introduced during the transfer may lead to the analyzed core being reset. Furthermore, when using this kind of analysis, due to the way devices are interacted with, it is often not possible to achieve reproducible executions. PANDA [106] addresses this problem by shifting firmware execution to an emulator that can record and replay execution traces. Many approaches attempt to completely rehost the firmware using a generic emulator and provide peripheral models and simulated interrupts to mimic the hardware. To a greater [60, 62, 92] and lesser extent [63, 91, 98], techniques can generate these models automatically. Sun et al. [119] propose a domain-specific reverse engineering framework to extract the high-level semantics of control-logic from Internet of Things (IoT) device firmware. In contrast to INCISION, they require high-quality CFG reconstruction and fine-grained execution traces that include memory loads and stores, making their technique challenging to apply to the devices and firmware analyzed in this work. In order to test firmware for vulnerabilities, many approaches use symbolic execution. Davidson et al. [105], for example, demonstrate how for very small firmware, it can be used effectively to achieve complete coverage. Both INCEPTION [96] and HARDSNAP [97] focus on how to handle the nuances of complex firmware and devices with multiple peripherals under symbolic execution. They leverage the fact that source code is sometimes available for portions of firmware, which contains significantly more information about high-level constructs and data-types and can thus make symbolic execution more tractable. Other approaches use fuzz testing to assess devices for vulnerabilities. A naive approach to this has significant drawbacks, as highlighted by Muench et al. [90]. Therefore, for most approaches, the firmware is first manually reverse engineered, and then select parts are fuzzed under an emulated environment [64, 65, 120, 121]. Aside from academic work, both SYSTEMVIEW [122] and TRACEALYZER [123] support developers in debugging their firmware through trace analysis. However, they require access to the source code for the firmware, so that it can be compiled with software-based instrumentation. As such, the use of these tools for reverse engineering or analysis of end-user devices is limited.

Static Firmware Analysis While the majority of focus has been on dynamic techniques, a small number of approaches have used static analysis. However, almost all of these have not been applied to bare-metal or monolithic firmware. Costin et al. [124] perform a large-scale assessment of downloadable device firmware (mostly Linux-based) and apply simple static analyses to discover several vulnerabilities. Shoshitaishvili et al. [125] demonstrate how symbolic execution with human guidance can be used to discover authentication bypass vulnerabilities in a various firmware, including binary blob firmware. Cojocar et al. [126] use static analysis and machine learning to locate parsing routines in device firmware automatically. Similarly, Thomas et al. [127, 128], use a static approach to locate undocumented functionality in Linux-based firmware.

Control-Flow Recovery and Disassembly A significant body of research has contributed to the state-of-the-art in binary analysis. Much of this work has been towards improving control-flow recovery and disassembly. We refer the reader to the work of Shoshitaishvili et al. [129] for a more complete exposition. We summarize the contributions most relevant to our work in what follows. IDA Pro [109] and Ghidra [69] provide solutions to aid manual reverse engineering. They perform function start identification, disassembly, control-flow recovery, and cross-referencing. However, for complex firmware the quality of their analyses is highly dependent on human intervention. Andriesse et al. [130] provide an in-depth analysis of the problems faced when performing disassembly on real-world x86/x64 binaries. They attempt to remedy the situation with a compiler agnostic approach to control-flow recovery [131]. Muhui et al. [132] perform a similar analysis for ARM-based binaries. The authors of Polypyus [112] propose an approach leveraging past reverse engineering efforts on firmware from the same vendor to aid in control-flow and function start recovery. Their tool provides a step forward in addressing a problem which dramatically affects the correctness of disassembly for VLE instruction sets such as Thumb2.

3.7 Conclusion

Complex embedded devices present many challenges when performing a security analysis: undocumented peripherals, uncommon instruction sets, and limited tool support. Performing any kind of automated analysis, in particular, generally requires a non-trivial amount of upfront manual reverse engineering. In this chapter, we present Incision, a framework that reduces the manual effort required to perform these preliminary reverse engineering tasks. We design and implement four novel approaches that automate aspects of the typical reverse engineering workflow: firmware region inference, OS primitive identification, task-aware control-flow recovery from system-wide execution traces, and feedback-driven refinement. The latter combines static and dynamic control-flow recovery to improve our overall understanding of the firmware.

We demonstrate the effectiveness of Incision on the firmware of two end-user devices, a VxWorks-based LTE baseband and an automotive BCM, as well as a set of firmware based on widely used RTOS: FreeRTOS and Zephyr. We demonstrate Incision's effectiveness by performing real reverse engineering tasks. We use it to identify stall-points that prevent emulation and assess cryptographic key usage. Both tasks greatly benefit from the use of Incision. Further, we show that Incision does not introduce significant complexity to the standard reverse engineering process and requires little manual effort to use. Incision's analyses produce correct results with high confidence and are robust to different OSes, ISAs, and trace mechanisms.

Chapter Four

VoltPillager: Hardware-based Fault Injection Attacks against Intel SGX Enclaves

Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean,

David Oswald, and Flavio D. Garcia

This chapter is based on the following publication:

VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves [3]

which is published in

30th USENIX Security Symposium (USENIX Security '21)

4.1 Introduction

Modern computing platforms allow the operating system to self-regulate the processor's core frequency and voltage in order to manage heat and power consumption. Several authors [20–22] have shown that an adversary can abuse this feature to inject bit flips into computations, including those inside an Intel SGX enclave (cf. CVE-2019-11157). Using the software-exposed interface MSR `0x150`, attacks against Intel SGX were mounted by undervolting from (untrusted) software running with root privileges. Intel have addressed this vulnerability by providing features to disable software undervolting through this MSR. Because SGX was compromised, Intel have initiated Trusted Computing Base (TCB) recovery and modified remote attestation to verify that software-based undervolting is disabled. This requires Microcode (UCODE) and BIOS updates.

Hardware fault injection considers a different adversarial model where the adversary has physical access to the device under attack. When targeting an SGX enclave running on a fully patched system (with the latest UCODE and BIOS updates), software-based fault attacks have been fully mitigated and that is where hardware-based attacks become relevant. Fault attacks induce a computation fault in the target processor, such as skipping an instruction, by changing the physical operating environment of the chip, e.g., the supply voltage. They do not rely on the presence of a software vulnerability or any code execution privileges. Voltage fault injection (aka, glitching) in particular has the advantage of being very powerful whilst not requiring expensive lab equipment.

4.1.1 Our Contribution

In this chapter, we analyse the dynamic voltage scaling features of x86 systems at the hardware level. We found that a three-wire bus, SVID, is used to send the currently required voltage to an external Voltage Regulator (VR) chip on the motherboard. The VR then adjusts the voltage supplied to the CPU. We reverse-engineered the communication protocol of SVID and developed a small microcontroller-based board that can be

connected to the SVID bus. As there is no cryptographic authentication of the SVID packets, we were able to inject our own commands to control the CPU voltage. With this, we reproduced Plundervolt's [20] open-source PoC attacks, including against code running inside an SGX enclave. Beyond that, we also found (and document) faults not previously observed. These faults affect elementary operations such as memory accesses. Because the software interface MSR `0x150` is not used, Intel's countermeasures do not prevent this attack. The main contributions of this chapter are:

- We showcase the (to our knowledge) first hardware-based attack that directly breaches SGX's integrity guarantees. We demonstrate its practicality with end-to-end secret-key recovery attacks against mbed TLS and the unmodified `file-encryptor` sample enclave from Microsoft Open Enclave.
- We show that Intel's countermeasures for CVE-2019-11157 do not prevent fault-injection attacks from adversaries with physical access. This challenges the widely accepted belief that SGX can protect enclave integrity against a malicious cloud provider (cf. e.g., [133–136]).
- We demonstrate novel fault effects discovered through hardware-based undervolting, in particular by briefly delaying memory writes.
- We present VoltPillager, an open-source hardware device to inject SVID packets. VoltPillager is based on a low-cost, widely available microcontroller board, the Teensy 4.0, and can be built for approximately $ 30. We also document the internal power management interfaces on modern motherboards, SVID and SMBus.

4.1.2 Responsible Disclosure

We reported this issue to Intel on 13 March 2020. Intel evaluated our report and concluded on 5 May that "... opening the case and tampering of internal hardware to compromise SGX is out of scope for SGX threat model. Patches for CVE-2019-11157 (Plundervolt) were not designed to protect against hardware-based attacks as per the

threat model", and, therefore, they will not further address the issue. Intel have not requested an embargo for the vulnerabilities described in this chapter. We discuss the implications of Intel's response in relation to the widely adopted threat model of SGX in Section 4.1.4.

4.1.3 Related Work

Since their introduction by Boneh et al. [14], fault-injection attacks with physical access have been widely investigated in the context of embedded devices. Those attacks are based on the fact that the execution semantics of an IC can change when it is operated outside the specified operating conditions. Examples of fault injection include: over and undervolting ("voltage glitching"), overclocking, exposure to high or low temperature, or laser light [17, 18].

The fault injection threat model changed with the discovery of *software-based* attacks. In 2014, Kim et al. reported the Rowhammer effect: bits could be flipped in DRAM by accessing neighbouring rows but not the actual target location [19]. Several authors [137–139] have since discovered applications, variations, and improvements of the original attack, including the successful bypass of countermeasures in recent DDR4 DRAM chips [140]. While Rowhammer can be performed from unprivileged software, other classes of software-based fault injection attack may require the adversary to have root privileges. These generally target a TEE such as ARM TrustZone or Intel SGX, which should defend the code running inside the TEE even against a privileged adversary.

CLKSCREW [23] was the first attack of this type: it exploited the software-controlled overclocking features on the ARM processor of a Nexus 6 smartphone. CLKSCREW was able to extract cryptographic secrets from TrustZone and to bypass signature checks, leading to code execution inside TrustZone. Qiu et al. later found a similar attack, VoltJockey, against TrustZone, this time controlling the CPU's core *voltage* from privileged software [24]. This line of work continued with voltage fault injection attacks on Intel SGX enclaves [20–22], which use the software-exposed MSR `0x150` to undervolt

during enclave execution and thus trigger bit flips in certain operations, e.g., multiplications, vector instructions, and cryptographic operations.

Hardware-based attacks against TEEs have, so far, received less attention. Cui et al. showed that electro-magnetic fault injection can be used to bypass the TrustZone-based secure boot process of a Broadcom ARM CPU [141]. Similarly, Roth et al. presented fault injection attacks with physical access to ARMv8-M processors, among others breaking the TrustZone-M security on certain CPUs [142]. The approach used in this work also inspired a hardware-based fault injection attack against AMD SEV [39]

Lee et al. presented a side-channel attack [143] on SGX by physically connecting to, and eavesdropping on, the DRAM memory bus. They showed that by observing the pattern of the (encrypted) memory accesses, they can recover secret information from a range of example enclaves. Notably, their attack requires specialized and expensive test equipment (e.g., $ 170,000 for a JLA320A signal analyzer).

4.1.4 Attacker Model

We are using the widely adopted SGX adversary model with physical access to the target CPU and full control over all software running outside the enclave, including BIOS and operating system. Crucially, our attacks do not require expensive laboratory equipment (e.g., for invasive attacks on the CPU die), but can be mounted with an inexpensive microcontroller board and only require board-level access as opposed to e.g., chip decapsulation. It is sufficient that the adversary can connect two wires to the SVID bus on the motherboard.

In the research community, SGX is widely assumed to provide protection against such an adversary, e.g., in the form of an untrusted cloud provider with physical access to the server hardware. A substantial amount of research explicitly relies on SGX protecting integrity and confidentiality even for a malicious cloud operator, cf. e.g., [133–136]. Accordingly, cloud providers with SGX support, such as Microsoft Azure, state that SGX “safeguard[s] data from malicious and insider threats while it’s in use” [144,

145]. Fortanix, the developers of widely used runtime software for SGX, similarly claim that "Intel SGX allows you to run applications on untrusted infrastructure (for example public cloud) without having to trust the infrastructure provider with access to your applications." [146]. Similarly, the Enarx project considers the enclave host as untrusted in their threat model [147]. Finally, SGX was also originally designed for client-side applications such as DRM (e.g., in early versions of the Netflix 4K client), user authentication, and fingerprint matching [148].

4.1.5 Experimental Setup

For the experiments in this chapter, we mainly used three different systems, with 7th and 9th generation Intel CPUs and SGX support. We upgraded the BIOS and UCODE to the most recent available version. The systems are detailed in Table 4.1 and all use SVID as the main interface for controlling the supply voltage. We initially used a fourth motherboard, a Gigabyte Z170X Gaming 3 with an i3-7100, but this was damaged due to inadvertently short-circuiting the SVID lines, and is therefore not listed in Table 4.1. However, as we used this motherboard during SVID reverse engineering, we occasionally refer to it as i3-7100-GZ170 in Section 4.4.

Device	Motherboard	BIOS version	CPU	UCODE	VR IC	VR on SMBus
i3-7100-AZ170	ASRock Z170 E4	P7.50	i3-7100	0xca	ISL95856	`0x40`
i3-9100-MZ370	MSI Z370-A Pro	E7B48IMS.2B0	i3-9100	0xca	UP9508	`0x45`
i3-7100U-NUC	NUC7i3BNH	0082.2020.0505	i3-7100U	0xca	ISL96853	✗

Table 4.1: CPUs and motherboards used for experiments in this chapter, including BIOS and UCODE version, and the VR IC used. We also indicate if the VR is connected to the SMBus (and at which address) in addition to SVID. A 4^{th} system, i3-7100-GZ170, left out here as it was only used in initial reverse engineering.

We used 64-bit Ubuntu 18.04.3 LTS as our operating system with stock Linux 5.0.0-23-generic kernel, Intel SGX driver V2.6 and Intel SGX-SDK V2.8. We publicly release all source code at https://github.com/zt-chen/voltpillager.

4.1.6 Outline

The remainder of this chapter is structured as follows: first, in Section 4.2, we discuss Intel's mitigation for CVE-2019-11157. In Section 4.3, we then describe the two main interfaces for controlling CPU voltage on modern systems. We introduce our open-source tool VoltPillager for injection of SVID packets in Section 4.4. The use of VoltPillager for hardware-based undervolting attacks on SGX is detailed in Section 4.5 and Section 4.6. We discuss possible countermeasures and the implications of our findings in Section 4.7, before concluding in Section 4.8.

4.2 Intel's Mitigation for Software-based Undervolting Attacks on SGX Enclaves

The mitigation deployed by Intel to address CVE-2019-11157 effectively disables software access to the voltage control features of the system. It consists of two main parts: (*i*) a BIOS update supplied by the BIOS vendor to disable the undervolting functionality at boot, and (*ii*) a UCODE update to interact with the updated BIOS and include the software undervolting status (enabled or disabled) in SGX's remote attestation functionality.

The exact implementation of the BIOS update differs by vendor. On our test systems i3-7100-AZ170 and i3-9100-MZ370, even the most recent BIOS still allowed undervolting. In contrast, the latest update of our Intel-manufactured i3-7100U-NUC added a new BIOS option ("Real-Time Performance Tuning"), which, when disabled, removes the ability to undervolt via MSR `0x150`. We ran all experiments for the i3-7100U-NUC with software undervolting disabled via this BIOS option, and confirmed, practically, that writes to MSR `0x150` no longer cause voltage changes. We also verified that the OC mailbox interface (which is used for undervolting) is disabled on i3-7100U-NUC using `intel-oc-mbox` [149].

Other manufacturers, such as Dell [150], removed software-controlled undervolting with recent updates in response to CVE-2019-11157, without any configuration options available at the BIOS level. The UCODE update deployed by Intel includes the status of the software undervolting interface in the remote attestation process, similar to other functionality such as hyperthreading and the internal graphics card. Specifically, the SGX attestation service returns a `CONFIGURATION_NEEDED` response if software undervolting (or any of the other problematic features such as hyperthreading) is enabled [40].

4.3 Power Management Interfaces

In modern computers, there are usually one or more VRs connected to the CPU on the motherboard. They are used for managing the performance and power consumption of the system by changing the core voltage (and other voltages) supplied to the CPU. When the CPU runs at lower frequencies or is in idle mode, it sends commands to the VR to reduce the voltage. Vice versa, when the CPU operates under heavy load and/or at high frequency, it requests the VR to increase the voltage. We found two main interfaces to the VR that can be used for changing the CPU voltage and hence to conduct undervolting attacks: the SVID interface and the SMBus interface (more specifically PMBus in this context). The overall architecture of the voltage supply on an x86 system is shown in Figure 4.1. We now introduce those VR interfaces in detail and discuss their use for undervolting attacks.

4.3.1 Serial Voltage Identification

SVID is the interface used by the CPU to send the currently required voltage (and other related data) to an external VR IC. To the best of our knowledge, Intel does not provide any detailed documentation for this interface. From the CPU documentation [151], we found that SVID uses the three pins `VCLK`, `VDIO`, and `ALERT#`. The first two pins are used for a bi-directional serial interface similar to common serial protocols like I2C or

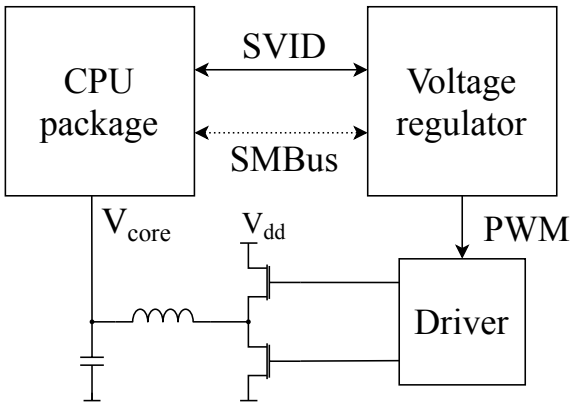


Figure 4.1: Architecture of voltage supply on x86 systems. The CPU has an SVID and (optional, dotted) SMBus connection to the VR. The VR drives a Pulse Width Modulation (PWM) output to generate the core voltage from the main supply voltage.

Serial Peripheral Interface (SPI). The `ALERT#` line is asserted by the VR when a voltage change has been completed [152]. The SVID bus uses voltage levels of 0 V (low) and 1 V (high) and is clocked at 25 MHz. Both clock (`VCLK`) and data (`VDIO`) lines are realised as open-drain outputs: by default, the lines are held high by pullup resistors to 1 V, and actively driven low by the CPU or VR when they exchange data. Note that this allows multiple devices to be connected to SVID, we will later use this to connect our own device for command injection.

Locating VR ICs The identification of the VRs on the motherboard is the first step required for further analysis of a particular system. Some vendors provide schematic diagrams of their boards, which greatly simplifies the process. Unfortunately, most vendors do not publish such detailed documentation of their hardware. However, in our experience, VRs are commonly placed in close proximity to the CPU and to large switching transistors and inductors, making them easy to identify by visual inspection and oscilloscope probing. Additionally, the SVID signals are commonly connected to small resistors and/or available on test pads, simplifying connecting devices for analysis and packet injection. Figure 4.2 shows the ISL96853 VR on i3-7100U-NUC. The large transistors generating the actual core voltage are visible on the right, while the relevant

SVID pins are located around the top-left corner to the VR IC.

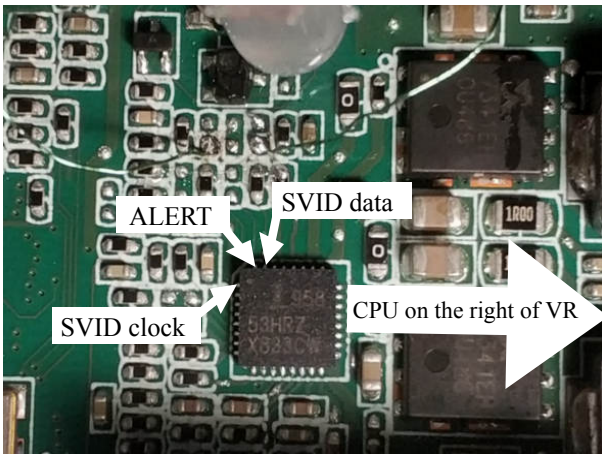


Figure 4.2: ISL96853 VR on the motherboard of i3-7100U-NUC with relevant SVID pins annotated.

Protocol Reverse-Engineering We identified the clock and data lines on the motherboard used for our experiments using a Rigol DS1074Z oscilloscope [153], and connected a DSlogic logic analyser [154] to the SVID bus.

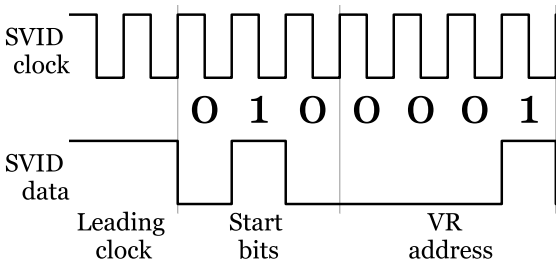


Figure 4.3: SVID data and clock lines during the first cycles of a command sent to the VR.

By observing the bus and setting known values for the voltage and with the help of a screenshot of a SVID protocol analyzer [155], we reverse-engineered the relevant commands used for configuring the voltage output by the voltage regulator. Note that this is only a one-time process required to understand the working of the SVID protocol and is not necessary for subsequent packet injection attacks. Figure 4.3 shows an example of the clock and data signals when transmitting a bit sequence over SVID. Based on this,

we analysed the command that configures the voltage and its response, and document their format in Figure 4.4 and Figure 4.5. Further details on SVID can be found in Appendix C.3.

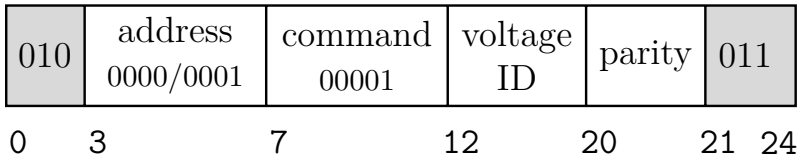


Figure 4.4: Format of 24-bit SVID command from CPU to VR to set the current core voltage. Gray background indicates fixed bits.

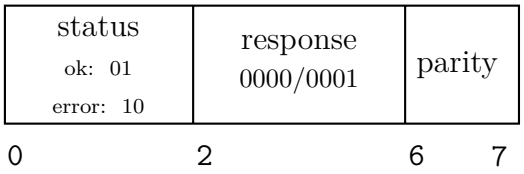


Figure 4.5: 7-bit response from VR to CPU to the set voltage command from Figure 4.4.

Bus Activity and Command Injection We found that the SVID bus on our test motherboards was active even when a fixed core voltage was configured in the BIOS. Regular commands from the CPU to the VR are still sent, although at a reduced rate compared to normal, dynamic voltage control.

Understanding bus activity is crucial for SVID. Our experiments (as described in Section 4.4) show that the CPU freezes if there are multiple subsequent collisions between an injected and a "real" SVID packet. From our experiments, we concluded that the CPU stops after less than a second of failed transmission attempts. Hence, it is crucial to limit SVID command injection to a short burst.

Conversely, for reverse-engineering and analysis of VR behaviour, the ability to temporarily "remove" the CPU from the bus was useful: once the CPU has frozen, we could send our own commands to the VR without interference from the CPU. This allowed us to verify our understanding of the SVID commands during the initial analysis of the protocol, and to test our tool, VoltPillager.

4.3.2 System Management Bus and Power Management Bus

Apart from SVID, some VRs support another voltage control interface named SMBus (also referred to as PMBus in this case) [156, 157]. Such VRs supporting SMBus are mainly found on server and gaming motherboards. Similar to SVID, SMBus is a serial two-wire interface consisting of a clock line (maximum frequency 1 MHz) and a data line. Multiple devices are connected to the SMBus, with the motherboard's Southbridge acting as bus master. Each device on the bus, including the VR if connected, is assigned a unique address. Typically, these addresses are assigned by the motherboard vendor and not publicly documented. Hence, determining the address of the VR on the SMBus requires probing the bus and/or analysis of manufacturer tuning software.

Among our test systems, the i3-7100-AZ170 and i3-9100-MZ370 had the VR chip connected to the SMBus. We found that on both systems, when setting a fixed core voltage through the BIOS, this configuration takes place via the SMBus. Subsequently, we determined the VR address (cf. Table 4.1) and the respective commands to adjust the CPU voltage. We also noted that any voltage change made through SMBus overrides any subsequent settings made through SVID.

While it is technically possible to inject packets into the SMBus, we opted for using SVID as our main attack interface for several reasons: First, with the higher clock frequency of SVID, voltage changes can be made more quickly, allowing for more accurate fault injection. Second, we found that SVID commands were the same across our test systems, while SMBus commands varied between VRs. Finally, SVID is used by all modern motherboards and CPUs, while SMBus is only present on certain motherboards and VRs.

4.4 VoltPillager for SVID Command Injection

In this section, we present VoltPillager, our custom device for SVID command injection based on a Teensy 4.0 microcontroller development board [158]. In contrast to other

widely used interfaces like SPI, we could not find a Microcontroller (muC) with a dedicated hardware peripheral for SVID. We initially attempted to implement the protocol in software using the General Purpose I/O (GPIO) pins, but we found that the relatively high clock frequency of 25 MHz of SVID makes this difficult in practise even on relatively high-end muCs like the ARM Cortex-M7 on the Teensy 4.0, which runs at 600 MHz.

However, as SVID is similar to SPI in so-called mode two (clock idles at high voltage and data line stable and sampled on the falling edge of the clock), we were able to use the SPI hardware peripheral of the Teensy 4.0 for packet injection. Note that an additional required feature is the support for large SPI frames, which allows for the transmission of a complete SVID transaction. With our implementation, a complete transmission of one SVID packet takes 96 clock cycles (3.84 µs), including 40 clock cycles before sending the data, the 24-bit data frame, and 32 clock cycles for receiving the response. To adapt

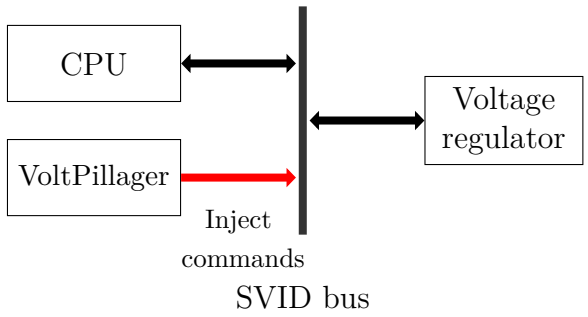


Figure 4.6: Command injection into SVID with the VoltPillager connected in parallel to the bus.

the output voltage levels of 3.3 V of the Teensy 4.0 to the 1 V levels of SVID, we used two open-drain drivers [159] for the data and clock lines. As shown in Figure 4.7, we also made sure to keep the wires between the actual bus lines and the driver ICs short to minimise additional inductive and capacitive load on the bus. Note that VoltPillager can be connected in parallel to the bus without—when inactive—affecting the normal SVID traffic from CPU to VR.

We also carried out initial experiments with a Field Programmable Gate Array (FPGA) board to obtain a man-in-the-middle position by splitting the SVID bus: in

this situation, the attacker simulates normal behaviour of the VR to the CPU, while injecting commands that are only visible to the VR. However, due to the strict timing constraints of SVID and the necessary level conversions, we did not further investigate this approach for the present chapter.

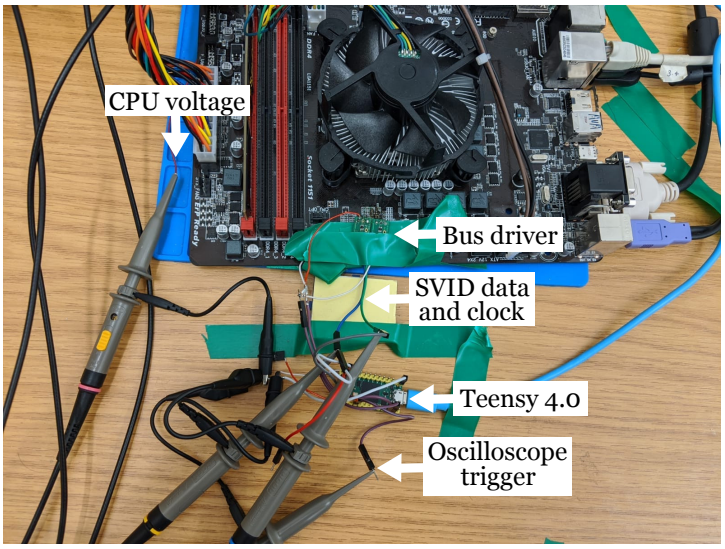


Figure 4.7: Hardware setup (i3-7100-GZ170) for analysis and packet injection into the SVID bus. Oscilloscope probes are attached to CPU core voltage and SVID lines, while Teensy injects SVID packets.

4.4.1 Implementation of VoltPillager

VoltPillager has two main components, the firmware of the Teensy 4.0 and PC-side software which controls the device. We developed the Teensy 4.0 firmware using the open-source Arduino IDE [160] and the official Teensyduino [161] library, which provides the basic SPI functionality. We configured this to match the structure of an SVID message. The controlling PC communicates with the VoltPillager through USB using the protocol shown in Figure 4.8. Firstly, the software on the controlling computer specifies the undervolting parameters, including the required voltage, number of SVID packets being sent, and additional parameters as detailed in Table 4.2. It then arms the glitch.

Parameter	Symbol	Description
Delay after trigger	T_d	The time between assertion of trigger and start of first undervolting.
Number of glitches	N	Number of times to repeat the undervolting.
Preparation voltage	V_p	Voltage before undervolting. Typically $\leq V_n$, but system stable at V_p.
Preparation width	T_p	Time of voltage slew to V_p plus time for which V_p is held.
Fault voltage	V_f	Voltage used for actual fault injection.
Fault width	T_f	Time of voltage slew to V_p plus time for which V_f is held.
Normal voltage	V_n	Stable operating voltage after V_f.
Slew rate	SR	Rate of voltage change (in mV/µs). Most VRs support a single SR.

Table 4.2: Parameters defining a specific undervolting experiment with VoltPillager

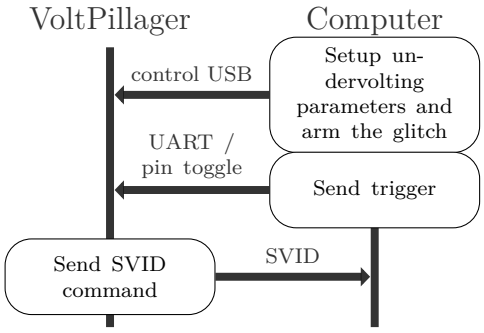


Figure 4.8: Protocol of VoltPillager for configuring and initiating undervolting from a controlling PC.

Trigger After that, VoltPillager is armed and waits for an active-low *trigger* input on a specific pin. When the trigger signal is asserted (*i.e.*, the trigger pin pulled low), VoltPillager sends the prepared SVID packets according to the configuration to the target VR, *i.e.*, initiates a hardware-based undervolting attack. The trigger is generated by the controlling computer from untrusted code directly before the target code executes. In our experiments, the control PC typically is the machine that also runs the target SGX enclave, although this is not a strict requirement.

To provide a precisely-timed trigger signal, we utilise the fact that some mother-boards, such as the ones used in i3-7100-AZ170 and i3-9100-MZ370, offer a legacy on-

board RS232 Universal Asynchronous Receiver Transmitter (UART) port, including the so-called `DTR` signal, which can be controlled from software through an `ioctl()` system call. Because this interface is controlled by the motherboard's Southbridge through the "super I/O" chip, it is more reliable in timing than sending a trigger command over the control USB interface.

However, for systems without hardware UART, e.g., i3-7100U-NUC, we additionally implemented a less timing-stable trigger over USB. After the undervolting has been triggered, the control program evaluates whether it has succeeded (*i.e.*, that an unexpected result occurred in the target code), and outputs the fault results or repeats the process until a fault has been found.

Adjustment of Undervolting Parameters The parameters from Table 4.2, defining the undervolting glitch shape, are illustrated in Figure 4.9. Note that these parameters affect the stability of the system-under-attack and hence, can be adjusted to minimise system crashes. Note that the VR changes the core voltage CPU with a finite slew

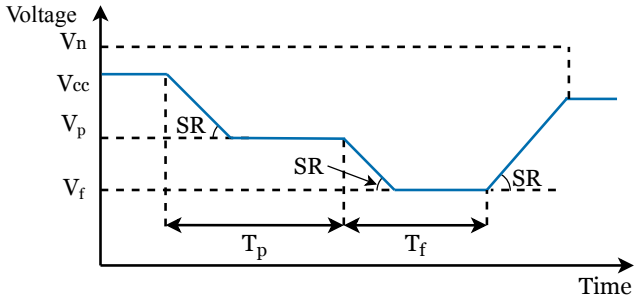


Figure 4.9: Undervolting waveform with the parameters described in Table 4.2. V_n can be $\leq V_{cc}$.

rate SR, typically 20 mV/µs, however on i3-7100U-NUC we observed a higher slew rate (approximately 40 mV/µs). The limited slew rate can reduce the timing precision of undervolting, as it adds delay between the reception of the injected SVID command from the VR and the physical change of the core voltage to the fault voltage V_f. The preparation voltage V_p reduces the time to reach the target voltage V_f before the actual attack: the system is still stable at V_p, and the adjustment to V_f at finite SR is quicker

compared to the higher default voltage V_{cc}. The reset voltage V_n is set after the fault. It can take any values and is used to stabilize the system. Figure 4.10 shows an oscilloscope capture of an actual undervolting injected by VoltPillager on i3-9100-MZ370. In this case, we set $V_p = V_n = V_{cc} = 1.050\,V$, $V_f = 0.810\,V$, $T_p = 10\,\mu s$ and $T_f = 32\,\mu s$.

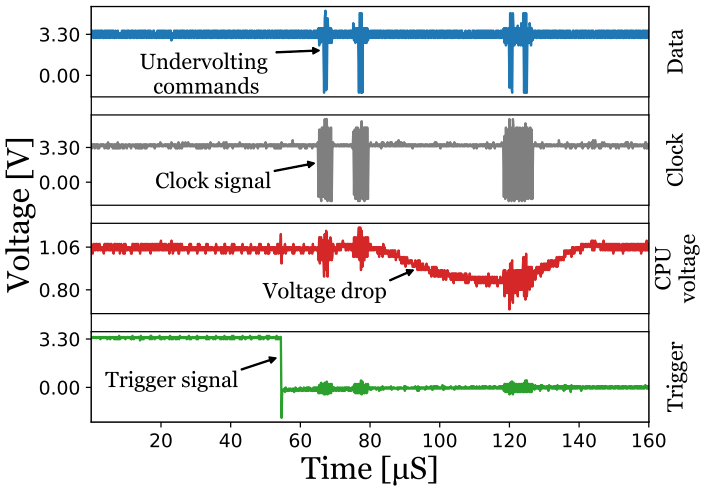


Figure 4.10: Oscillocope capture of undervolting. Data and clock captured before the voltage level shifters.

System Stability One of the major factors affecting the system stability is the amount of undervolting. When the voltage is too low, the CPU will "freeze" or crash, while a too high voltage will not yield successful faults. Figure 4.11 shows the undervolting for the fault to occur and the value at which the system crashes for different CPU frequencies. We observed crashes before a fault happens below 1.4 GHz on i3-7100-AZ170 and 2.9 GHz on i3-9100-MZ370. For both systems, there is a gap of $\approx 20\,mV$ (4 VID steps) which can be used for fault injection without affecting system stability.

4.5 Fault Injection into SGX Enclaves with Hardware-based Undervolting

In this section, we demonstrate how VoltPillager can be used to inject faults into the CPU even when the software-controlled interface has been disabled. To this end, we first

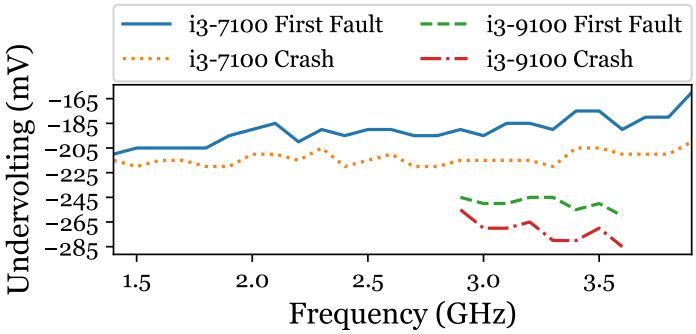


Figure 4.11: Undervolting for first fault (solid) and crash (dotted) for different frequencies on i3-7100-AZ170 (max. 3.9 GHz) and i3-9100-MZ370 (max. 3.6 GHz), using the PoC from Listing 4.1 and VoltPillager. 5 repetitions per frequency. $V_n = V_p = V_{cc}$.

show that we can reproduce faults observed with software-based fault attacks through hardware-based undervolting. We show how this can be used in an end-to-end attack scenario to extract in-enclave secrets. We then describe a novel fault attack based on briefly delayed memory/cache writes.

4.5.1 Reproducing Plundervolt Proof-of-Concepts

The authors of Plundervolt provide several PoCs on their Github repository [48]. We mainly focused on two of their PoCs, namely faulting integer multiplications (in userspace) and CRT-RSA decryption/signature (running inside an SGX enclave). We made this choice to (*i*) give examples both for SGX and non-SGX code and (*ii*) compare the behaviour of the well-documented faults on `imul` when using hardware-based undervolting (with various parameters shown in Table 4.2). Additionally, we also successfully reproduced Plundervolt's PoC for AES-NI.

Faulting Multiplications We integrated Plundervolt's `faulting_multiplications` PoC into our experimental setup as shown in Listing 4.1. This code segment is similar to the one used in Plundervolt: two multiply operations (compiled to `imul`) are executed with the same input in a tight loop and the result of the calculation is compared after each operation. However, before entering the loop, a trigger is generated to start the

```
TRIGGER_SET // Set trigger

do {
  i++;
  correct_a = operand1 * operand2;
  correct_b = operand1 * operand2;
  if (correct_a != correct_b) {
    faulty = 1;
  }
} while (faulty == 0 && i < iterations);

TRIGGER_RST // Reset trigger
// ... fault check omitted ...
```

Listing 4.1: Simplified C code used for demonstrating hardware-based fault injection into multiplication

hardware-based undervolting using VoltPillager.

Setting first the operands of `imul` to `0xAE0000` and `0x18`, respectively, we obtained the same faulty result (`0xC500000` instead of `0x10500000`) on all our test systems, using the parameters from Table 4.3.

		Multiplication				RSA-CRT			
Device	Clock	V_p	V_f	T_f	Temp.	V_{cc}	V_f	T_f	Temp.
i3-7100-AZ170	2 GHz	0.83 V	0.64 V	29 µs	23° C	0.83 V	0.63 V	29 µs	24° C
i3-9100-MZ370	3.4 GHz	1.050 V	0.81 V	83 µs	26° C	1.050 V	0.81 V	43 µs	27° C
i3-7100U-NUC	2 GHz	0.94 V	0.71 V	8 µs	32° C	0.94 V	0.75 V	9 µs	22° C

Table 4.3: Parameters for successful fault injection into the Plundervolt PoCs for userspace multiplication and SGX RSA-CRT. We also record the clock frequency and the actual CPU temperature when our program starts. All experiments are conducted with $N = 1$. For experiments with i3-7100-AZ170 and i3-9100-MZ370, $V_n = V_p$, for experiments with i3-7100U-NUC, $V_n = 1.05\,V$. See Appendix C.1 for full glitch details.

Cryptographic Operations inside SGX We then adapted Plundervolt's PoC `sgx_crt_rsa` to our setup. This program computes an RSA signature/decryption inside an SGX enclave, using the standard `ipps` library functions. Again, with the parameters shown in Table 4.3, we successfully obtained faulty signatures and confirmed that these faulty values can be used to factor the RSA modulus and recover the private key using the Lenstra attack [14]. Crucially, this attack also succeeded when the software-undervolting interface was disabled on i3-7100U-NUC through the respective BIOS option.

End-to-end Attack To demonstrate the real-world implications of successful fault injection, we developed an end-to-end attack on the mbed TLS library as used in Microsoft Open Enclave [162]. As a first step, we targeted mbed's `mbedtls_aesni_crypt_ecb()` API function, which internally is accelerated using `aesni`. We confirmed that we can mount a VoltPillager attack to inject a single-byte fault on i3-7100-AZ170 into the 8th round of AES and then perform a Differential Fault Analysis (DFA) to extract the full key [163].

Based on this, we developed an attack on an unmodified enclave, namely the `file-encryptor` sample from Open Enclave[1]. This enclave exposes `ecalls` to encrypt/decrypt files using AES in Cipher Block Chaining (CBC) mode with an in-enclave secret key. The enclave uses the respective mbed TLS function, that internally calls `mbedtls_aesni_crypt_ecb()`. For simplicity, we focus on encryption using a 128-bit key, but note that DFA can be extended to larger key sizes and decryption [164]. A challenge when attacking more complex modes of operations such as CBC is that the DFA requires both the correct and a faulty ciphertext for a given plaintext. However, the `file-encryptor` sample allows us to invoke the encryption `ecall` multiple times without resetting the CBC chaining. Hence, we can first invoke the `ecall` with a chosen plaintext p_0 and obtain:

$$c_0 = \mathrm{AES}_k\left(p_0 \oplus \mathrm{iv}\right)$$

where iv is a fixed value. Then, we can repeatedly invoke the `ecall` again to encrypt the value $p_0 \oplus \mathrm{iv} \oplus c_0$. By construction of CBC, this will repeatedly yield the same ciphertext c_0. We can now inject faults into the computation until we obtain a faulty ciphertext c_0', and then use DFA to recover the in-enclave key from c_0 and c_0' in $\approx 2\,\mathrm{min}$ of computation on a 16-core CPU.

We practically implemented this attack against the unmodified `file-encryptor` enclave. We verified on i3-7100U-NUC and i3-7100-AZ170 that we can successfully inject

[1] https://github.com/openenclave/openenclave/tree/98b71a/samples/file-encryptor

the desired faults and recover the secret key. On i3-7100U-NUC (with Plundervolt patches disabled), we used software-undervolting by -272 mV and invoked the respective `ecall` for a maximum of 100,000 times. On i3-7100-AZ170 we used hardware undervolting with VoltPillager and successfully injected faults while encrypting 10 blocks of data repeatedly in a loop of 700 iterations. After starting the program, it took less than 15 s to obtain a fault using the parameters $V_p = 0.7\,V$, $T_p = 30\,\mu s$, $V_f = 0.64\,V$, $T_f = 35\,\mu s$, $V_n = 0.83\,V$, $T_d = 600\,\mu s$, and $N = 1$. In both cases, triggering was performed outside the enclave from untrusted code.

4.5.2 Comparison with Software-based Undervolting

In this section, we compare VoltPillager to software-based undervolting through MSR `0x150` and discuss the advantages of our hardware-based approach. First of all, crucially, attacks with VoltPillager are not prevented by the mitigations deployed in response to CVE-2019-11157 (cf. Section 4.2), and hence can be mounted on systems with the most recent, patched UCODE and BIOS. This undermines the common assumption that SGX can protect against an attacker with physical access (e.g., a malicious cloud provider). Furthermore, as discussed in detail in Section 4.7, mitigating this issue will require, at least in the short term, substantial changes to enclave code to detect fault injections.

Timing Precision The authors of [20] reported that they required more than 100,000 iterations to fault an `imul`. Furthermore, the fault cannot target a particular loop iteration. VoltPillager can overcome both limitations: first, setting $T_d = 0\,\mu s$, $T_p = 26\,\mu s$, $V_p = 0.615\,V$, and $V_f = V_n = V_{cc} = 0.830\,V$, we were able to introduce a fault with as little as $i = 1,680$ iterations (for Listing 4.1) on i3-7100-AZ170.

To evaluate the precision of VoltPillager when targeting a particular loop iteration, we re-ran the multiplication experiment several times with fixed parameters and observed in which loop iteration (*i.e.*, at which value for `i` in Listing 4.1) the fault occurred. We conducted the following experiment on i3-7100-AZ170 at 2 GHz. Core 1 was isolated

with the kernel parameter `isolcpus=1` and used solely for running the target code. We then recorded the faulted loop iterations for the following undervolting parameters, repeating the experiment 60 times: $T_d = 10\,\mu s$, $N = 1$, $V_p = V_n = 0.830\,V$, $T_p = 35\,\mu s$, $V_f = 0.635\,V$ and $T_f = 24\,\mu s$. Out of the 60 runs, we observed a fault in 53. Within these successful faults, the median value for the faulted iteration was 14,634, with 21 faults within iterations 14,562 and 14,729. In fact, 75% of all faults occurred within iteration 14,634 ± 300 as shown in Figure 4.12.

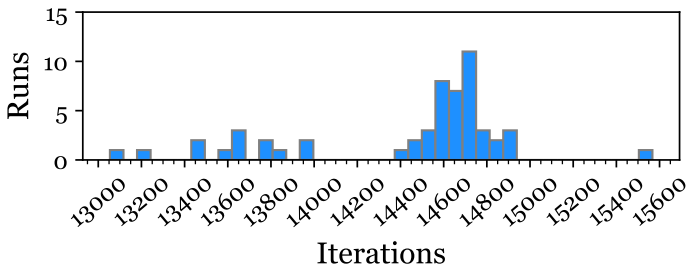


Figure 4.12: Histogram over affected loop iteration for 53 successful fault injections into multiplication on i3-7100-AZ170 at 2 GHz.

To compare the jitter of the different triggering methods, we used the following setup: we periodically toggle the trigger signal in a fixed-period loop on the controlling PC and let VoltPillager generate a pulse on a pin when it detects the trigger. To ensure consistent timing of the loop, we created a delay with `nanosleep()` and ran the program on a single core at priority 99 and with SCHED_RR policy. Furthermore, we set the following kernel parameters: `intel_pstate=disabled`, `intel_idle.max_cstate=0`, `isolcpus=1`.

Without any jitter, the period of the waveform would be constant. Any jitter added by the controlling PC, the interface, and the Teensy will lead to the period varying. Note that while the delay loop itself might introduce some jitter, this would be present for both triggering methods and hence still allows for relative comparison. Figure 4.13 shows the distribution (over 100 trigger period measures) for both `DTR` and USB trigger, with the loop period on the controlling Personal Computer (PC) set to 400 µs. The average deviation from the ideal period value, *i.e.*, the jitter, was measured as 4.521 µs (for `DTR`) and 54.442 µs (for USB). Clearly, the `DTR` trigger exhibits substantially less jitter than

USB.

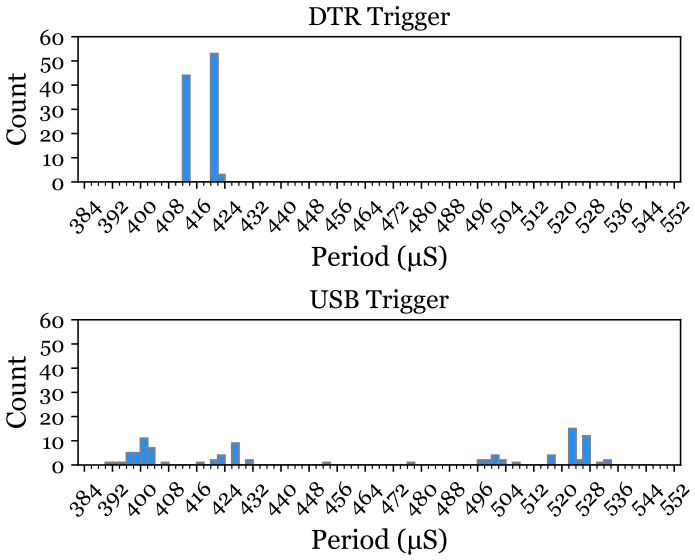


Figure 4.13: Histogram over 100 trigger period measured on i3-9100-MZ370 (2 GHz) with program running at priority 99 and SCHED_RR policy on core 1.

Glitch Width In software-based fault attacks, a long delay was observed between the MSR write and the actual voltage change. This limits the ability to generate short and potentially "deeper" glitches. In contrast, using VoltPillager, the only limitation on the glitch width is the slew rate SR: given SR, V_p, V_f and V_n, the minimal glitch width T_{min} is:

$$T_{min} = \frac{|V_p - V_f| + |V_n - V_f|}{SR}$$

Assuming $V_p = V_n = V_{cc}$, a typical fault voltage $V_f = V_{cc} - 200\,\text{mV}$, and typical $SR = 20\ \text{mV}/\mu\text{s}$, the minimal glitch width is hence $T_{min} = 20\,\mu\text{s}$.

The VR on some systems, e.g., i3-7100U-NUC, further supports a higher slew rate of $SR = 40\,\text{mV}/\mu\text{s}$. Thus, in this case, T_{min} can be further reduced to 10µs. However, during practical experiments with i3-7100U-NUC, we noticed that typically 2 µs after the voltage has reached V_f, the CPU emits an SVID packet, which negatively affects the ability to inject short glitches. The injected SVID packet to immediately reset the

voltage from V_f to V_n has a high probability of colliding with the CPU's command, leading to an ineffective packet injection and the voltage cannot be increased to V_n until the next SVID set voltage packet.

4.6 Delayed-Write Fault Attacks through Undervolting

In this section, we describe a novel class of undervolting-induced faults not reported in prior research. Specifically, we observed that undervolting appears to briefly delay memory writes to the cache, so adjacent instructions still read the previous value. We initially observed these faults with hardware-based undervolting using VoltPillager, however, could also later reproduce them on a i7-7700HQ in a Dell XPS15 9560 laptop (without the CVE-2019-11157 mitigations installed) through software-based undervolting. We refer to this system as i7-7700HQ-XPS in the following. For our first PoC, we compare two integers for inequality, as further explained in the following Section 4.6.1.

4.6.1 Initial Proof-of-Concept

The C code used for our PoC is shown in Listing 4.2. We developed this initial PoC in userspace and later verified that a realistic exploit also works when running inside an SGX enclave, cf. Section 4.6.2.

```
1 int i = faulty = 0;
2 int operand1 = ...;
3 int operand2 = operand1;
4
5 do {
6   if(operand1 != operand2)  {
7         faulty = 1;
8   }
9   operand1++;
10  operand2++;
11  i++;
```

```
12 } while(faulty == 0 && i < iterations);
13 // ... trigger code and fault check omitted ...
```

Listing 4.2: Simplified C code used for demonstrating fault injection into memory accesses

Note that the code from Listing 4.2, under normal non-faulty execution, never sets `faulty` to 1.This tight loop of memory writes (for incrementing) and reads (for comparison) was essential to discover this novel effect. After a pre-set number of executions (or if `faulty` is set), the loop terminates and proceeds to check whether a fault was injected (*i.e.*, `faulty` has been set). To avoid faulting adjacent instructions after the detection of a successful fault, we also inserted a group of `nop` instructions as a buffer between the loop and subsequent code.

If a fault has occurred, we output the values of both operands to ensure that the actual operand values have not changed (or e.g., the increment has been faulted). With this design, we can reduce the actual assembly instructions that can possibly be affected by a fault to the instructions on Lines 1, 3, 5, 9, 10 shown in Listing 4.3. Note that even though Listing 4.3 uses the 32-bit register `%eax`, all our code is compiled and runs in 64-bit mode. We have not observed any difference between 32 and 64-bit instructions. The code in Listing 4.3 is compiled with optimisation turned off (`-O0` option)

```
1 mov     -0x18(%rbp),%eax
2 // compare operand1 (%eax) and operand2
3 cmp     -0x14(%rbp),%eax
4 // continue at no_fault if equal
5 je      no_fault
6 // else set faulty = 1
7 movl    $0x1,0x20290f(%rip)
8 // Increment operands and counter
9 no_fault: addl    $0x1,-0x18(%rbp)
10 addl    $0x1,-0x14(%rbp)
11 addl    $0x1,-0x1c(%rbp)
```

Listing 4.3: Assembly compiled from Listing 4.2 with Lines 1, 3, 5, 9, 10 presumably affected by fault injection (AT&T syntax)

In other words, a fault injection that would lead to `faulty` being set could only affect one of the following common instructions: (*i*) a load from memory into a register, (*ii*) increment/write to memory, (*iii*) a comparison of a memory location with a register, or (*iv*) a jump-if-equal operation.

Initial Experiments We ran the target code described in Section 4.6.1 on the i3-7100-AZ170 at a clock frequency of 3 GHz and observed several successful fault injections (*i.e.*, `faulty` being set to 1) at 24° C with $V_f = 0.76\,V$, $V_p = 0.95\,V$, and $T_f = 29\,\mu s$ as shown in Figure 4.14. In all cases, the operands printed after the detection of the fault were equal, *i.e.*, the undervolting could have only affected the assembly instructions pointed out above.

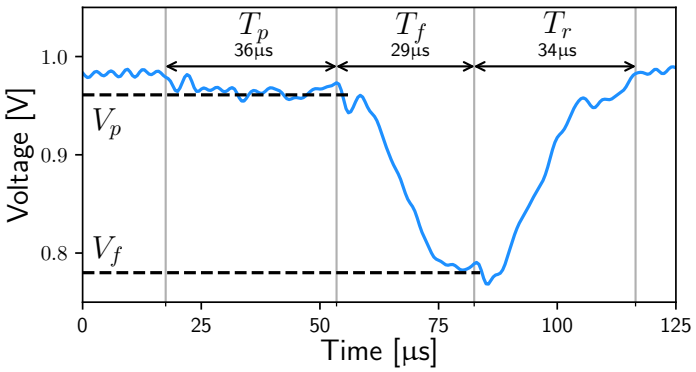


Figure 4.14: Oscilloscope capture of the CPU voltage during a successful fault injection. Signal low-pass filtered (Gaussian filter with $\sigma = 2$) for clarity. Actual V_f different to the value specified by VoltPillager due to physical effects. T_r indicates recovery time from V_f to V_n.

We found the parameters for successful fault injection through manual tuning combined with an exhaustive search in a specified region. Note that our experiments required a relatively low voltage, which in turn can affect system stability.

As reported by earlier work [20, 21], the CPU temperature affects the success rate and system stability. For faulting operations with (presumably) short critical paths such as memory accesses, we found this effect even more pronounced than for multiplications or AES-NI rounds (with longer critical path) on the i3-7100-AZ170. Hence, we

```
1 do {
2   if(operand1 != operand2) {
3       faulty = 1;
4   }
5   operand1++;
6   operand1--;
7   i++;
8 } while(faulty == 0 && i < iterations);
```

Listing 4.4: Modified C code used for demonstrating fault injection into memory accesses

ensured that we kept the core temperature at around 24° C with standard cooling. Subsequently, we successfully reproduced the same experiment on i7-7700HQ-XPS at 2 GHz with software-based undervolting through MSR `0x150`. The attack succeeded at normal core temperature of approximately 50° C without any additional cooling.

Further Analysis By modifying the code from Section 4.6.1, we obtained further insights into the likely cause of the fault (*i.e.*, the "operands not equal" branch being taken). To this end, we replaced the increment of `operand2` on Line 10 with a decrement of `operand1`, *i.e.*, `operand1/2` should be equal and constant throughout the loop, as shown in Listing 4.4:

However, in this case, fault injection led to `operand1` being only decremented, with the increment on Line 5 seemingly ignored. Conversely, when swapping the order and first decrementing followed by incrementing, `operand1` took the value `operand2 + 1` when undervolted. Furthermore, we modified the assembly code from Listing 4.3 to store the value of `%eax` used for the comparison on Line 3 when the "operands not equal" branch had been taken. We observed that, e.g., when a fault was found for `operand1` = `operand2` = 3, the value loaded into `%eax` was 2 *i.e.*, the increment had not taken effect.

From these observations, we conjecture that the most likely explanation for the observed faults is that *recent* memory (cache) writes are delayed and thus ignored in adjacent reads of the modified location. This suggests that the fault affects the load-store queue logic of the CPU, causing writes to be delayed for a few cycles while the execution of dependent instructions progresses with old values. For example, for the (undervolted)

```
uint32_t array[8] = { 0 };
// Attacker-supplied out-of-bounds size
int copy_size = 7;

// Ensure we stay within bounds
if(copy_size >= 5)
  copy_size = 4;

// overwrite elements 4, 3, 2, 1
while(copy_size >= 1)  {
  array[copy_size] = 0xabababab;
  copy_size--;
}
```

Listing 4.5: Proof-of-concept to demonstrate out-of-bounds memory accesses due to undervolting

code sequence `operand1++; operand1--;` the decrement operates on the previous value of `operand1`, ignoring the update through the preceding increment.

4.6.2 Practical Exploitation Scenario

We now consider a realistic scenario to exploit the effects from Section 4.6.1. The experiments described in this subsection were all performed inside an SGX enclave. We show how a delayed-write fault can be used to trigger out-of-bounds accesses in memory-safe code. To this end, the PoC code shown in Listing 4.5 initializes elements of an array to a fixed value. In Listing 4.5, `array[]` holds eight `uint32_t` elements all initially set to zero. The code then first ensures that the (potentially adversary-controlled) upper bound `copy_size` is ≤ 4, using a common code pattern that effectively implements `min(4, copy_size)`. It then proceeds to write `0xABABABAB` to array elements 4 to 1, leaving the other elements (0 and 5 ... 7) at their initial value of zero.

We are intentionally only writing to part of the eight-element array in this PoC to avoid triggering actual stack corruptions (and hence crashes). However, note that all experiments also apply to a real scenario, where the attacker would write beyond array bounds and corrupt the enclave stack, thus gaining control over the program counter and applying traditional exploitation techniques afterwards [165].

We undervolted the CPU whilst executing the above code in a loop within an `ecall` handler. The experiments were run on i7-7700HQ-XPS at a frequency of 2 GHz, under-

volting by -170 mV. The core temperature reported by the CPU varied between 44° C and 49° C. We observed two distinct effects induced by the fault (cf. Appendix C.2), as illustrated in Figure 4.15: (*i*) in addition to elements 4 to 1, element 0 was also overwritten (*i.e.*, an underflow) and (*ii*) the upper bound was not limited to 4 but stayed at 7, *i.e.*, an overflow into elements 5 . . . 7 occurred. In both cases, out-of-bounds accesses

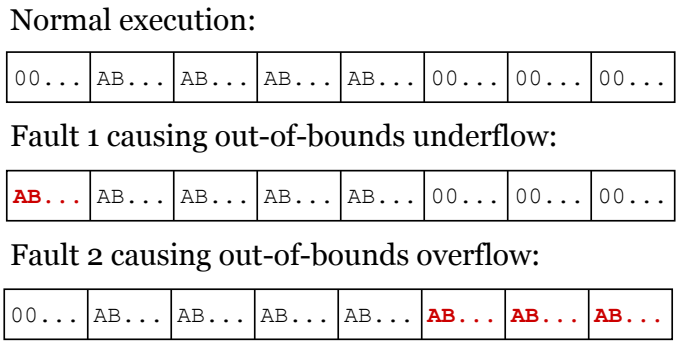


Figure 4.15: State of `array[]` after normal execution of Listing 4.5 and out-of-bounds under/overflow when undervolted. Faulty values in red bold font.

take place, leading to potential memory corruption and enabling further exploitation with traditional techniques, e.g., through stack buffer overflows. We describe the two observed fault types in the following.

Case 1: Out-of-Bounds Underflow As shown in Figure 4.15, undervolting caused `array[0]` to be incorrectly overwritten. Our analysis showed that this is due to a fault affecting the code responsible for decrementing and directly afterwards comparing the loop counter on Lines 12 and 10 in Listing 4.5, which translates to the assembly code shown in Listing 4.6.

```
1 // check for copy_size >= 1
2 copy_loop: cmpq   $0x0,-0x28(%rbp)
3 jle    exit_loop
4 // move copy_size into rax
5 mov    -0x28(%rbp),%rax
6 // move 0xabababab into array[copy_size]
7 movl   $0xabababab,-0x20(%rbp,%rax,4)
8 // copy_size--
```

```
1 movq    $0x7,-0x28(%rbp)
2 cmpq    $0x4,-0x28(%rbp)
3 // jump if copy_size less than or equal to 4
4 // THIS JUMP SHOULD NEVER BE TAKEN
5 jle     cont
6 // set copy_size = 4
7 movq    $0x4,-0x28(%rbp)
8 cont: // ...
```

Listing 4.7: Assembly affected by overflow

```
9 subq    $0x1,-0x28(%rbp)
10 jmp    copy_loop
11 exit_loop: // ...
```

Listing 4.6: Assembly affected by underflow

When undervolting, we observed the decrement of the loop counter on Line 9 in Listing 4.6 had not been committed by the time the comparison on Line 2 occurs. Thus, the loop performs one additional iteration for `copy_size` = 0. We found that the decrement *does* come into effect on the subsequent read into `%rax` on Line 5, which is the index into the array, hence overwriting `array[0]`.

Case 2: Out-of-Bounds Overflow In the second observed fault, elements 5–7 are incorrectly overwritten. In this case, we concluded that the fault affects the initialisation of the upper limit on Lines 6 and 7 in Listing 4.5. The respective assembly snippet is shown in Listing 4.7.

As with the previous fault, we conclude that the operation `copy_size` = 7 on Line 1 has not completed by the time the compare statement on Line 2 is reached. Consequently, `copy_size` is not limited to 4 but remains at the higher value of 7, triggering writes beyond the upper limit of 4. Note that in this example the initial value 7 is loaded as a constant, but it could equivalently be loaded from an attacker-controlled parameter, e.g., an untrusted length field passed to an `ecall`.

4.7 Implications and Countermeasures

To the best of our knowledge, this chapter presents the first practical attack that directly breaches integrity guarantees in the Intel SGX security architecture through a hardware-based attack. We show that the fix currently deployed by Intel—disabling the software undervolting interface— is insufficient when taking hardware-based attacks with physical access into account.

We also believe that these attacks might have implications for non-SGX programs, because the fault injection in principle does not require code execution on the CPU (in contrast to the software-based fault attacks [20–22]). It is conceivable that it could enable attacks on e.g., locked computers using disk encryption, similar to the attacker model for DMA attacks [166], where an adversary with physical access is able to bypass security mechanisms. However, an adversary mounting attacks would face substantial challenges, including proper triggering to fault the desired program while keeping the system stable, and the need to open the case and connect to SVID without powering down.

It is worth noting that the type of attacks described in this chapter could be applied to CPUs by other vendors: AMD uses a similar design with a VR connected to the CPU though their SVI bus [167].

Countermeasures against the attacks described in this chapter can be implemented at the level of (*i*) the SVID protocol, (*ii*) the CPU hardware or UCODE, (*iii*) the enclave code itself. In the following, we discuss mitigations in detail. Note that countermeasures cannot be implemented in the BIOS or in components outside the CPU package, because SGX regards the BIOS and external hardware as untrusted.

Mitigation through Changes to SVID In our opinion, the issue of voltage glitching with physical access cannot be effectively addressed by e.g., adding cryptographic authentication to the SVID protocol. As explained in Section 4.3, the VR essentially converts the SVID commands to a PWM-modulated waveform, which controls the tran-

sistors generating the actual core voltage. Hence, instead of injecting into SVID, an attacker could disconnect these control outputs and supply their own (malicious) PWM signal, bypassing any authentication of SVID. A well-resourced attacker could even completely replace the VR with a custom voltage glitcher, and a malicious cloud provider could use custom motherboards with built-in glitching functionality.

Mitigation in CPU Hardware or μCode To detect the adversary's injected SVID packets, the CPU could monitor the bus for packets that were not generated by itself. On detecting packet injection, the CPU could raise an exception and abort execution. But, as pointed out in Section 4.4, an adversary could split the connection between CPU and VR and act as man-in-the-middle, hiding malicious packets from the CPU. Hence, this countermeasure would only protect against basic SVID injection attacks with VoltPillager in parallel to the bus.

Secondly, the CPU could continuously monitor its own supply voltage and abort if the voltage falls below a safe threshold. When using existing functionality such as measuring the core voltage through the Running Average Power Limit (RAPL) interface and MSR `0x198` [168, 169], it should be taken into account that such interfaces have a low sample rate in the order of 1 kHz. Hence, they would not be fast enough to detect glitches shorter than the sampling window.

Therefore, future CPU generations could include dedicated hardware countermeasures [170], including e.g., voltage monitoring circuitry as commonly found in smartcards [171]. One could also consider running critical code paths on multiple cores and detect deviations through additional CPU logic as e.g., implemented by certain Infineon smartcard ICs [172]. However, such countermeasures would amount to substantial hardware changes and incur overheads. Another option would be the use of a Fully Integrated Voltage Regulator (FIVR), *i.e.*, the VR integrated within the CPU package, as used in 4th generation Intel CPUs, but later abandoned in newer generations [173]. However, as the input voltage for the FIVR is still supplied externally, such circuitry would have to

appropriately handle malicious modification of that input voltage.

Mitigation in Enclave Code For SGX enclaves that require immediate protection against fault injection, countermeasures can be implemented within the enclave code. According to our experience, it is highly unlikely to produce the same fault twice in adjacent or nearby instructions; however, this warrants further detailed investigation. Enclaves could duplicate potentially vulnerable instructions and compare the results. While manual insertion of such countermeasures might be feasible for comparatively small pieces of critical code (e.g., crypto functions or memory management), fully protecting an existing codebase would require excessive effort. However, prior research [174, 175] shows that automatic instruction duplication at the compiler level is feasible. Hence, porting such techniques to x86 architectures and in particular SGX poses an interesting problem for future work. Note that SGX enclaves cannot rely on mitigations based on measuring CPU voltage, as SGX does not offer a trusted way to access MSRs, so any such countermeasure could be bypassed by a compromised operating system.

4.8 Conclusions

In this chapter we identified a novel and powerful attack surface of Intel CPUs. We have shown how the SVID interface can be leveraged by adversaries with physical access to gain full control over the voltage regulator. We then demonstrate that dynamic voltage scaling can be reliably exploited to mount fault-injection attacks against the CPU. To the best of our knowledge, this represents the first hardware-based fault-injection attack against a fully-fledged CPU and also the first one that directly breaches the integrity and confidentiality of SGX enclaved computations on a fully patched system.

We have proven that this attack vector is practical by recovering RSA keys from an enclaved application, and have shown that other fundamental operations such as multiplication and memory/cache writes can be faulted as well. These lead to novel memory

safety vulnerabilities within SGX, which are not detected by SGX's memory protection mechanisms. The results in this chapter, together with the manufacturer's decision to not mitigate this type of attack, prompt us to reconsider whether the widely believed enclaved execution promise of outsourcing sensitive computations to an untrusted, remote platform is still viable.

Acknowledgments

This research is partially funded by the Engineering and Physical Sciences Research Council (EPSRC) under grants EP/R012598/1, EP/R008000/1, EP/V000454/1, by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 779391 (FutureTPM), and by the Paul and Yuanbi Ramsay Endowment Fund.

Chapter Five

BMC Firmware Analaysis and PMBus Based Power Attacks

Zitai Chen and David Oswald

This chapter is based on the following publication:

PMFault: Faulting and Bricking Server CPUs through Management Interfaces [4]

which is published in

IACR Transactions on Cryptographic Hardware and Embedded Systems, 2023(2).

5.1 Introduction

In recent years, the security implications of software-exposed power and clock management features have received substantial attention by the research community. Several attacks including CLKSCREW [176], Plundervolt [20], V0ltPwn [21], and VoltJockey [22] showed that undervolting or overclocking from software can be used to inject faults (e.g., bitflips) into computations and break TEEs like Intel SGX and ARM TrustZone. Subsequent attacks like VoltPillager [3] and the work by Buhren et al. [39] showed that similar attacks can be mounted with direct access to the computer hardware, physically connecting to the control interface of the VR.

In particular, Chen et al. targeted the SVID interface used by Intel CPUs to set the desired supply voltage. However, apart from SVID, many systems, in particular servers, support a second interface, the so-called PMBus, to control the VRM. PMBus is an open standard for digital power management [177] and has been adopted by more than 40 companies. It is based on the I2C bus and offers monitoring features apart from voltage and current control.

Another component usually presents on server motherboards is the BMC. This chip, intended to remotely manage the server even if e.g., the main CPU has crashed or is powered down, has connections to several buses and chips on the motherboard, including the I2C bus on which the VRM resides.

Previous research on x86 platforms has focused on the software-hardware interface provided by the CPU itself and on the security within the perimeter of each individual component, e.g., the BMC [43] or Intel ME [6–8]. There is a lack of board-level security analysis that reviews the system and motherboard design and interactions between the different components: even if an individual part of the system is secure within its individual threat model, the combination of it with other parts can cause security risks. In our PMFault attacks, the privileged position of the BMC, combined with its large attack surface, makes it interesting from an adversary's perspective to exploit vulnerabilities of

the system via power management features.

5.1.1 Our Contribution

Our main contributions in this paper are:

PMBus-based under/overvolting against server platforms: We first analyse the VRM management interface at the hardware level. We discovered that the semi-standardised PMBus can be used to control the CPU voltage. Using the case study of a widely-used server motherboard, the Supermicro X11SSL-CF, we explore this attack surface and show that software vulnerabilities in the BMC (or another programmable chip connected to the PMBus) can have severe consequences for the security and safety of the server platform. To determine if the vulnerabilities can affect other server motherboards, we also investigated the PMBus connections and usage on an ASRock E3C246D4I-2T and a Supermicro X12DPi-NT6.

PMBus access through BMC exploits: We then study the BMC firmware and—based on prior work in [178–180]—found that it can indeed be exploited to send arbitrary PMBus commands to control the voltage of the CPU. More precisely, several software vulnerabilities in the BMC, including incorrect firmware encryption and signing mechanisms, a lack of authentication for firmware upgrades and control interfaces, an attacker can manipulate the CPU voltage remotely because the PMBus is connected to the BMC and the SMBus by default.

PMBus-based undervolting against SGX enclaves: With this, we observed the same faults as with Plundervolt/V0ltPwn (CVE-2019-11157), including for code running inside an SGX enclave. As the BMC has an independent, external flash chip for its firmware, SGX attestation currently *does not* have the ability to verify its status. Crucially, because the software voltage-control interface in MSR `0x150` is not used, Intel's fix for CVE-2019-11157 does not address this attack.

Permanent denial-of-service through overvolting: We also discovered a novel overvolting attack: by sending a certain sequence of PMBus commands, we can set the CPU

voltage outside the specification (as high as 2.84 V) and permanently brick the Xeon CPU used in our experiments.

Countermeasures and mitigations: Finally, we develop the PMBusDetect tool for detecting if the VRM is connected to the PMBus, and then discuss countermeasures and challenges in securing server platforms. Importantly, we point out that TEEs like SGX must not only rely on the security of the CPU itself, but also of that of management components the hardware design of the platform.

The details of our experiments and source code can be found at: https://github.com/zt-chen/PMFault. CVE number CVE-2022-43309 has been reserved for PMFault.

5.1.2 Adversary Model

In this paper, we assume a privileged software attacker, *i.e.*, who has obtained `root` on the host CPU. This is the standard adversary model in the case of TEEs like SGX, and is also realistic in the case of overvolting to permanently destroy the CPU, which could be e.g., exploited by ransomware with `root` rights. Notably, our attacks do not require physical access (for additional hardware to be added to the system) and can thus be conducted remotely e.g., over SSH.

5.1.3 Responsible Disclosure

We have responsibly disclosed our findings to Intel and Supermicro in April 2022. We discussed the details of our methods in several calls with Supermicro, and they acknowledge the existence of the issue and are looking into deploying fixes for their 11th generation products like the Supermicro X11SSL-CF. Supermicro highlighted that the attacks do not replicate on their 12th generation, which e.g., include secure boot and update for the BMC and filtering on PMBus commands. Both of these features break the attack chains described in the paper. Intel considered the issue in the context of their own server motherboards and did not find them vulnerable. Intel did not comment on the

impact on SGX.

5.1.4 Related Work

Since Boneh et al.'s seminal work on fault injection [14], the research community has devoted substantial efforts to investigating fault attacks and developing according countermeasures (cf. e.g., [17] for an overview).

Software-based Fault Injection Often, fault injection was considered a technique limited to attacks with physical access to the target. However, with the discovery of the Rowhammer effect [19], it was shown that faults can also be injected from software (through specific memory access patterns in the case of Rowhammer). Then, in 2017, Tang et al. showed that the clock management features of ARM processors can be exploited to inject faults into computations shielded inside a TEE like ARM TrustZone [176]. Similarly, Plundervolt, V0ltPwn, and VoltJockey [20–22] (all tracked via CVE-2019-11157) use the software-exposed voltage control MSR in Intel processors to break the integrity guarantees of SGX enclaves. In response, Intel deployed a microcode update that disables the undervolting interface in MSR `0x150` and allows remote parties to verify correct configuration through SGX's remote attestation. Thus, purely software-based undervolting attacks against Intel processors were considered no longer possible.

Hardware-based Fault Injection on TEEs The second generation of undervolting attacks on TEEs like SGX and AMD SEV require physical access to the target motherboard. In the case of VoltPillager [3], the adversary attaches two wires to the data and clock lines of the SVID bus and can then control the VRM external to the CPU, enabling undervolting even if Intel's microcode fixes for CVE-2019-11157 are installed. For AMD SEV, the adversary does not glitch the actual CPU, but the separate security co-processor, the AMD Secure Processor (SP) [39]. The adversary then proceeds to upload custom firmware to the SP to leak memory encryption keys and also endorsement

secrets, which ultimately enable attacks without permanent physical access.

Security of servers and BMCs Independent of hardware-based attacks, the security of server platforms has received attention in the research community and wider society. In 2018, Bloomberg published a—since widely disproven—article that *incorrectly* claimed the inclusion of small backdoor chips on Supermicro motherboards [181]. However, at the same time, researchers at Eclypsium showed that it is indeed possible to maliciously manipulate the BMC firmware of Supermicro motherboards from 8th to 11th generation [178], without the need to add a hardware implant. They also demonstrated how flashing corrupted BMC firmware can "brick" the server system by preventing it to boot.

Niewöhner [180] subsequently published a tool to exploit the (weak) firmware encryption of Supermicro BMCs. Other work, for example by Waisman et al. [182] and Périgaud et al. [43], has shown that software weaknesses in BMCs are not limited to Supermicro motherboards, but also applied to Dell, HP, and Lenovo systems.

However, the implications of direct access to the PMBus from a compromised BMC have not been deeply studied to our knowledge.

5.1.5 Paper Outline

The remainder of this paper is structured as follows: in Section 5.2, we review the PMBus protocol and analyse its specific implementation and usage on Supermicro motherboards. Then, in Section 5.3, we describe Supermicro's BMC implementation and methods to modify the firmware. In Section 5.4, we experimentally investigate how a compromised BMC can interact with the VRM through the PMBus. We then use this to develop over/undervolting attacks in Section 5.5, before concluding in Section 5.7.

5.2 Analysis of Power Management Bus

We started our work by analysing how the PMBus is used on practical server motherboards. PMBus is an interface that is used to control the VRM, supplying the power to the CPU. The most recent public available specification is version 1.3 [177]. This specification standardises the physical interface, packet structure, and command set of the PMBus. However, some commands are left as "manufacturer specified", so that each VRM manufacturer can have a slightly different implementation of the command set. This matches what we found during our investigation of the MP2955 VRM on the Supermicro X11SSL-CF platform described in the following.

5.2.1 Experimental Setup

We carried out initial experiments with an Intel Xeon E3-1220 v6 (CPU family: 6, model: 158, microcode version: 0xea) on a Supermicro X11SSL-CF Rev 1.01 motherboard (BMC microcontroller ASPEED AST2400, firmware revision 01.63, BIOS version: 2.4).We used 64-bit Ubuntu 18.04.3 LTS with a stock 5.4.0-107-generic kernel, Intel SGX driver V2.11.0, and Intel SGX-SDK V2.15.100.3. We refer to this system as E3-1220V6-X11SSL-CF throughout the paper. An overview of the server motherboard representative for Supermicro's 11th generation products is shown in Figure 5.1. The target of the PMFault attack is an Intel CPU with SGX technology. As mentioned, our actual attacks do not require additional hardware or physical access to the system, though we soldered some wires to the motherboard during the analysis phase.

On Intel platforms, the voltage of the CPU is controlled by an external VRM IC. The CPU connects to the VRM via the SVID bus to control the voltage supplied by it. This interface for CPU voltage control is present on all desktop and server motherboards.

However, server VRMs—including the Supermicro X11SSL-CF—often have an additional I2C-based communication interface called PMBus. This interface allows e.g., overclocking or fine-tuning of the CPU voltage. One of the crucial steps in the PMFault

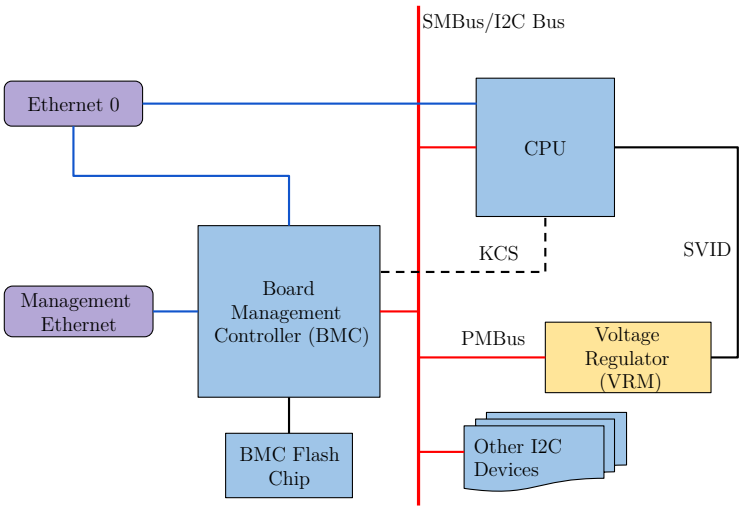


Figure 5.1: Overview of the connections on the server motherboard.

is to get access to this interface and understand the communication protocol, so that we gain full control of the CPU voltage.

One of the design issues we found on our server motherboard is that the PMBus can be directly connected to the more general SMBus. There are various components on the system on that bus, including the CPU, BMC, and other I2C devices. A compromise of any of these components leads to the takeover of PMBus and thus control of the CPU voltage.

In this paper, we use the BMC as the starting point of the attack, as it commonly exists on server platforms. In order to analyse the attack surface of the BMC, we further investigated its connection and hardware design on the Supermicro X11SSL-CF. First, we found that its firmware is stored in a SPI flash chip, separate from the BIOS flash. We also found there are two Ethernet ports on the system for communication with the BMC: one is called “Management Ethernet” and is dedicated for server management. The other port can be shared between CPU and BMC so that devices on this Ethernet port can communicate with both CPU and BMC. Finally, the BMC also has a Keyboard Controller Style (KCS) interface that enables direct access from the OS running on the CPU. These management interfaces open a large attack surface on the BMC, and make remote attacks possible.

5.2.2 Protocol Structure

To be able to eavesdrop and forge PMBus commands, knowledge of the protocol structure shown in Figure 5.2 is necessary. The PMBus is an I2C-based protocol (with clock speed of 100 kHz–1 MHz and an open-drain data pin) and uses a master-slave communication mechanism. The master device can query or change the setting of the slave device. Each slave device is assigned a unique 7-bit device address.

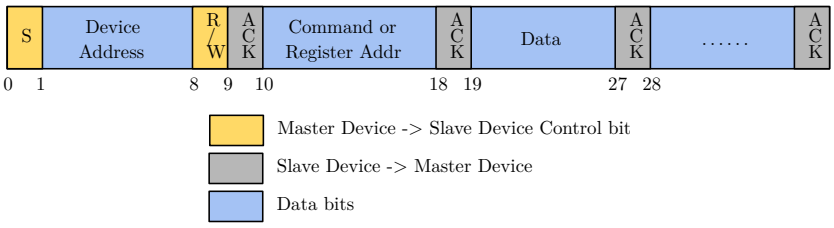


Figure 5.2: PMBus protocol structure.

The master device first sends a starting bit to initiate a transmission. During transmission, every group of 9 bits forms a segment, with the 9th bit indicating ACK (0) or NACK (1) for every 8 bits received. The starting bit and the (N)ACK mechanism are handled at hardware level and do not need to be handled manually.

The first segment is always sent by the master. The first 7 bits are the address of the target slave, and the 8th bit indicates whether this transmission is a read (1) or write (0). The second segment is the register address to operate on. In the PMBus specification, this segment is called the *PMBus command*. The segments after the second one contain the data read from or written to the register.

Interaction between PMBus and SVID Although the functionality of the PMBus protocol is similar to SVID, they have different specifications for the digital signal interface and command sets. A VRM can have both SVID and PMBus interfaces, with the SVID interface directly connected to the CPU and the PMBus interface connected to the SMBus. Both interfaces can be used to control the voltage of the CPU, and some implementations of the PMBus specification also have commands to override the

voltages set through the SVID interface.

5.2.3 PMBus Commands

For an adversary to communicate with the VRM and e.g., configure voltage levels, they also need to know the specific PMBus commands. As mentioned, the PMBus specification allows manufacturers to have custom implementations of PMBus commands. The E3-1220V6-X11SSL-CF motherboard features an Monolithic Power MP2955 voltage regulator. To understand the PMBus implementation of this VRM, we first started looking for its datasheet, but unfortunately, found that it is not publicly available. However, on the Monolithic Power website[1], we found the datasheet of an alternative VRM (MP2965) [183]. As both chips are manufactured by the same company, we used this datasheet as a reference and starting point to discover the available PMBus commands by analysing the PMBus traffic on the Supermicro X11SSL-CF.

We found the relevant PMBus commands by reading and analysing the response (ACK or NACK) of the registers, and validating found commands according to the PMBus specification and the MP2965 datasheet: Table 5.1 gives the command name, command code, and description of each commands. The first three commands in the table are implemented according to the PMBus 1.3 specification [177], while the rest are manufacturer-specific.

Command name	Command code	Usage
`CMD_PAGE`	`0x00`	Switch between different voltage rails
`CMD_OPERATION`	`0x01`	PMBus override
`VOUT_COMMAND`	`0x21`	Output voltage settings
`READ_VOUT`	`0x8B`	Voltage reading from sensor
`MFR_VR_CONFIG`	`0xE4`	Enable overclock mode
`MFR_OCP_TOTAL_SET`	`0xEE`	Over-current protection configuration

Table 5.1: Discovered PMBus commands on E3-1220V6-X11SSL-CF.

With `CMD_OPERATION`, we can configure the operation mode of the VRM. By setting

[1] https://www.monolithicpower.com/

bit 1 of this register, we can enable the PMBus override mode. In this mode, the voltage configured in the `VOUT_COMMAND` register will override the voltage configuration from the SVID bus. Another command that is useful for PMFault is `READ_VOUT`, as it allows us to read the current voltage of the CPU and establish a baseline for undervolting. The `MFR_VR_CONFIG` register is manufacturer-specific. By setting bit 3 or bit 10 and configuring `CMD_OPERATION`, we could enable the tracking or fixed voltage overclocking mode, respectively. Bit 8 `VID_STEP_SEL` of `MFR_VR_CONFIG` also allow us to use an alternative mode of SVID. In this mode, the VRM uses 10 mV Voltage Identifier (VID) steps instead of the default of 5 mV. This makes overvolting up to 3 V possible, which is well beyond the operating voltage range of the E3-1220 V6 Intel CPU, with a maximum voltage of 1.52 V [184]. We also discovered that the VRM has an Over Current Protection (OCP) circuit, which can be configured or disabled by another manufacturer-specific register (`MFR_OCP_TOTAL_SET`). Some VRM also support multiple voltage output rails. `CMD_PAGE` command is used to select the target rail to send the commands to.

With these discovered commands, we can now control the CPU voltage through the PMBus. In Section 5.4.1, we detail how this interface is used as part of attack chains for undervolting and overvolting attacks.

5.2.4 Jumper Settings

On the Supermicro X11SSL-CF motherboard, there are several jumpers that control different functionalities, including the connection of the VRM to other parts of the system. We kept all jumpers in the default status as delivered by the vendor. To avoid confusion, we still list the jumper settings in Table 5.2. During inspection of the jumper settings, we discovered that the `SMBDAT_VRM` and `SMBCLK_VRM` jumpers are neither mentioned in the user manual [185] nor in the quick reference guide [186]. Using an oscilloscope while sending PMBus commands, we found that these two jumpers can be used for (dis)connecting the VRM from/to the PMBus. The experiments and attacks described in this paper are conducted under the "connected" setting of both jumpers,

which according to Supermicro is the default.

We also found server motherboard without such jumpers, e.g., Supermicro X11SPG-TF and ASRock E3C246D4I-2T. For those, the VRM is always connected to the BMC. We detail our finding on other motherboards in Section 5.6. It is worth mentioning that to the best of our knowledge, SGX attestation does not have the functionality to include the configuration of these (external) jumpers.

Jumper name	Description
JPME2	Manufacturer mode normal (Default)
JPB1	BMC enabled (Default)
SMBDAT_VRM	Connect VRM data line to PMBus
SMBCLK_VRM	Connect VRM clock line to PMBus

Table 5.2: Jumper settings on Supermicro X11SSL-CF.

5.3 Supermicro's BMC and Server Management Interface

Having understood the basic PMBus protocol and commands, we next look at different ways to gain access to the PMBus and send commands to the VRM. To achieve that, an attacker needs access to the SMBus. As described in Section 5.2.1, on E3-1220V6-X11SSL-CF, one of the devices on the SMBus is the ASPEED AST2400 BMC controller. In this section, we introduce the functionalities and vulnerabilities in these management interfaces that allow us to achieve our main goal—to take control of the SMBus.

During the initial investigation of the BMC, we found there are mainly three services available: there is a web service running on port 80 (HTTP) and 443 (HTTPS), an Intelligent Platform Management Interface (IPMI) over LAN service on port 623, and the SSH service on port 22. Besides, we also found that the IPMI service can be accessed through the KCS interface from the CPU.

Some of these interfaces require authentication: to use HTTP, HTTPS, SSH, and IPMI -over-LAN, all exposed through Ethernet, one has to authenticate to the BMC. The used credentials in this authentication process are individual for each Supermicro motherboard. However, the IPMI-over-KCS interface does not require any authentication to the BMC. Instead, having root privileges on the host OS running on the CPU is sufficient to access this interface. One can also use the IPMI-over-KCS interface to add/remove/modify BMC credentials to subsequently login to the Ethernet-exposed interfaces.

5.3.1 SSH Shell

Since SSH is one of the most common interfaces that allows us to get a shell and possibly take over the system, we first started our investigation with it. However, the SSH service on E3-1220V6-X11SSL-CF provides a custom shell called "ATEN SMASH-CLP System Management Shell". It only provides limited commands that enable server monitoring and basic management. Previously, a vulnerability was reported in [187]: the command `shell sh` allows gaining root access from this shell, however, this command was not available on our system-under-investigation.

5.3.2 BMC Firmware Analysis

To further investigate the services running on the BMC and check if it is possible to enable an SSH root shell, we dumped the firmware of the BMC with a CH341A SPI flash programmer as shown in Figure 5.3. This procedure is only used once to assist our analysis, and is not necessary to execute the actual attack.

We found that the firmware stored in the SPI flash is neither encrypted nor signed. There are five partitions in the firmware, where the second one contains a Linux operating system. The SMASH shell is provided by `/SMASH/msh` and it is possible to change it to a different shell by replacing this file.

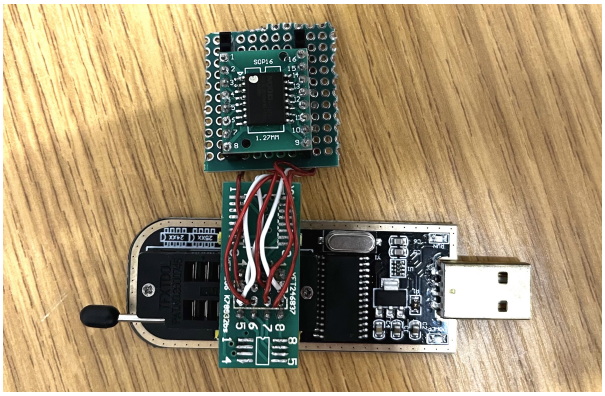


Figure 5.3: Dumping BMC firmware with a flash programmer.

The Linux operating system also has an I2C kernel module installed, which provides an interface to communicate with the SMBus. However, during our testing in Section 5.4.1, we found that the API provided by this kernel module is not compatible with the commonly used `libi2c` in `i2c-tool`[2]. As the result, in Section 5.4.1, we opted to write a custom library to use the I2C interface of the BMC and communicate with the VRM.

5.3.3 Firmware Upgrade

After analysing the firmware, we conclude that it is possible to enable an SSH shell by modifying the firmware. We then started to look for software methods to re-flash the BMC SPI flash chip. We found that the firmware upgrade functionality of the BMC provides a way to do this. There are two interfaces for firmware upgrade: one is through the web interface, the other through the KCS interface.

Through Web Interface The web interface has a firmware upgrade page that can switch the BMC into upgrade mode and allows the user to upload a BMC firmware update package. To prevents unauthorised user from upgrading the firmware, there is a login portal. The user is authenticated by the BMC. As the BMC is a system independent from the OS running on the CPU, users do not need to have privileged

[2] https://git.kernel.org/pub/scm/utils/i2c-tools/i2c-tools.git/

access to the OS to be able to use this method. Besides, this web interface can be accessed remotely through Ethernet. The remote BMC firmware upgrade attack chain described in Section 5.4.3 uses this method to upgrade the firmware.

Through IPMI-over-KCS Interface Crucially, the BMC firmware can also be updated through the KCS interface, using the following command:

```
AlUpdate -f firmware.bin -i kcs -r y
```

As mentioned, the KCS interface can be accessed from the OS running on the CPU, only requiring root access to the OS, *but not the BMC credentials*.

Firmware Upgrade Package After finding the firmware upgrade interface, the next step is to produce an upgrade package that can be uploaded to the BMC. We started with the analysis of the structure of the upgrade package. Figure 5.4 shows the layout of a firmware upgrade package. Previous work by [178] founds that in the firmware upgrade package, there is a region that contains a magic value (`ATENs_FW`), a half-length CRC checksum, and the length of each section. We call this part the firmware footer. There is also a region containing metadata of the firmware image, including the name of each region and their length and CRC, starting with "`[img]`". We refer to this region as firmware table. In the X11 series, the firmware table, the *file system header* of the root file system and the website *files system header* are AES-CBC encrypted. However, the files in these regions are not encrypted, but only LZMA compressed. As a result, the key of the AES-CBC encryption can be recovered from the `ipmi.so` file on the root file system.

With this information, we can modify the firmware and construct a valid firmware upgrade package for the web interface. We discuss firmware repacking in detail in Section 5.4.2.

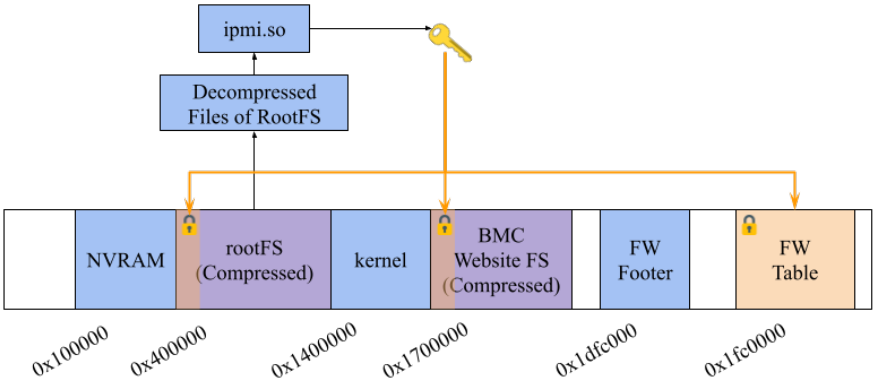


Figure 5.4: Layout of the BMC firmware upgrade package. The NVRAM region stores the current configuration of the BMC, the rootFS is a LZMA-compressed cramFS file system with only its header encrypted. The kernel region stores a Linux kernel image, while the BMC website FS is another compressed file system with only the file system header encrypted. The FW Footer starts with a magic value `ATENs_FW` and contain information about the firmware version, checksum, etc. The FW Table is an encrypted region and stores a table of the image layout. All encrypted region of the firmware can be decrypted with a key extracted from *ipmi.so* on the *rootFS*.

5.3.4 IPMI I2C functionality

When exploring the functionalities of IPMI, we also found that the interface also allows direct sending I2C packets with the `ipmitool i2c` command. This can be used either through the Ethernet or KCS IPMI channel. The authentication requirement for using IPMI-controlled I2C is the same as those described in Section 5.3.3. As shown in Section 5.4.3, we can use this functionality for direct access to the SMBus/PMBus *without* modifying BMC firmware.

5.4 Practical Experiments

Finally, using the results from the previous sections, we explain how to construct practical PoC attacks for PMFault. Some of our experiments require physical access to the system to understand the hardware configuration (with an overview shown in Figure 5.5). Note however that physical access is not required when performing PMFault attacks on a real-world system, as the hardware components and connections are identical for a given

motherboard model.

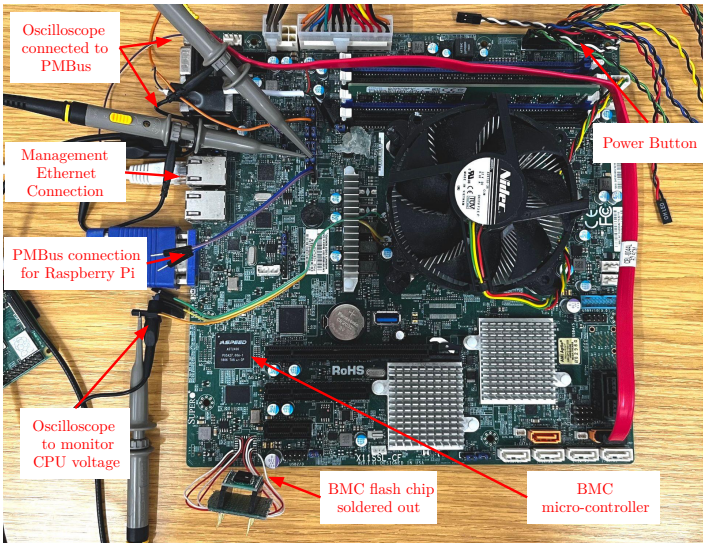


Figure 5.5: Setup of the E3-1220V6-X11SSL-CF for practical experiments. These connections are for experiments only; physical access is not required in the actual attack.

5.4.1 PMBus-based Voltage Control

To understand the configuration and capabilities of using the PMBus to control the CPU voltage, we conducted two experiments. Firstly, we used the "probe and verify" method to find the I2C address of the VRM. Then we tried different ways of sending commands to VRM to change the voltage.

Discovering the VRM Address Finding the I2C address of the VRM is the first step of PMFault. The easiest way to explore the I2C buses is to use the interface provided by the OS. There are two I2C buses that can be used from the OS running on the CPU: `i2c-0` is shown by default, while `i2c-1` requires the `i2c_i801` kernel module to be loaded. To find all available devices on both I2C buses, we ran the `i2cdetect` tool on them. We found that there are 12 devices in total connected to the I2C bus. The full list of device addresses can be found in Appendix D.1.

To then determine which device is a VRM, we use the result of the standard PMBus command, `READ_VOUT`, as an indicator. The Plundervolt [20] attack showed that the

normal operating voltage of the CPU should be greater than 0.55 V, thus, if the voltage read by `READ_VOUT` is within this range, it may be a VRM. Of the 12 devices detected, only one device with address `0x20` on I2C bus 1 responded with a value in this voltage range. We hence suspect this device is the VRM. To verify the result, we also used `MFR_ADDR_PMBUS` (`0xE1`) command found in the MP2965 datasheet [183] to read the PMBus address of the device. The result is `0x20`, which confirms our finding.

Changing CPU Voltage with PMBus Commands Having identified the VRM, one can next attempt to send commands to change the CPU voltage.



Figure 5.6: Command sequence to change the voltage via PMBus.

In the datasheet of the MP2965 [183], we found an "overclocking" procedure that can be used for this purpose. There are two overclocking modes, *tracking mode* and *fix mode*. In PMFault, we mainly use the fix mode to set a defined voltage.

In the fix overclocking mode, the VRM uses the VID configured with the PMBus command `VOUT_COMMAND` and ignores the configuration from the SVID bus. Figure 5.6 shows the steps of using this mode to change voltage. First, we need to configure two registers: The first one is `VOUT_OPERATION`; by setting the first bit of this register, we enable PMBus override mode. We also have to set bit 3 of `MFR_VR_CONFIG` to make the VRM act on these changes. After this, the voltage supplied to the CPU will be changed according to the configuration in `VOUT_COMMAND`. To send this PMBus command sequence and change the CPU voltage, we wrote a PoC with the `libi2c`. This PoC can be compiled and run under Linux.

"Stalls" caused by PMBus Commands The experiments in Section 5.4.1 also show that the VRM responds to the PMBus commands sent from the CPU. One may thus assume that it would then be straightforward to directly send PMBus commands to

change the CPU voltage with this method. However, we found that the CPU stalls after sending the `MFR_VR_CONFIG` command to actually configure the VRM to use the new voltage. This will make the CPU voltage being kept at the changed value with no way to change it back. This phenomenon raised two questions: Is the CPU stall caused by a crash or a recoverable halt? If it is caused by a recoverable halt, will this protect against targeted undervolting fault injection?

To answer this, we connected a Raspberry Pi to the PMBus to directly control the VRM. The I2C interface to the VRM is exposed with two pins, `SDA` and `SCL`. As shown in Figure 5.5, we connected the I2C interface of the Raspberry Pi to these pins.

In the first experiment, we sent a command to disable overclocking after the stall happens. It appears that with the VRM reconfigured to normal mode, the CPU recovers from the stall situation if the undervolting value is not too low. This shows that the stall is caused by a recoverable halt and not a crash. The second experiment is used to find out if the halt will prevent the fault from happening. In this experiment, we used the CRT-RSA PoC of the Plundervolt attack. With the CPU running this PoC, we used Raspberry Pi to send PMBus commands to produce voltage glitches. We found that with glitches with gradually lower voltage, an exploitable fault happens with the CRT-RSA calculation.

Hence, in summary, the "stall" phenomenon will prevent the PMBus attack from being conducted by the CPU-VRM I2C interface, but it does not prevent the fault caused by undervolting from having an impact on CPU calculations.

Voltage Control with BMC Because our attempt of voltage glitching failed with the PoC running on the CPU, we started to look into the BMC-VRM I2C interface. In the BMC firmware dumped in Section 5.3.1, we found the `i2c.ko` kernel module, which provides a driver for the I2C interface. However, this module does not implement a standard `ioctl()` for I2C devices, which is required for using `libi2c`. This means that the above PoC, which uses this standard I2C library, cannot be used to communicate

with this kernel module.

As the kernel module in the firmware did not implement the standard I2C API, we had to find another way to utilize the BMC's I2C interface. With the help of the I2C driver in the latest Linux kernel [188, 189], we found that there are 14 I2C interfaces on the AST2400 BMC controller. Each has a set of memory-mapped registers to control the interface. We also found the setup and message sending/receiving sequence of the I2C interface. We then created a small library to directly write these registers and send I2C bus commands from the BMC CPU to the address of the VRM. By monitoring the I2C activity with an oscilloscope (this was only required for debugging and during development), we found that the I2C bus 2 (counted from bus 0) of the BMC has the VRM connected.

5.4.2 Enabling SSH Access and Firmware Repacking

Modification of the firmware can be used to obtain a root shell on the BMC. With the "Supermicro BMC firmware image decryptor" [180] and a modified version of the "ipmi firmware tool" [179] with added support for X11 images, we were able to extract the firmware encryption key and decrypt the file system header. With these, we can unpack and modify the full root file system.

As described in Section 5.3.2, `/SMASH/msh` provides the shell for SSH service. To enable full root shell access, we replaced this file with a shell script with a single line to execute `/bin/sh`. Besides, as the SSH service is running with root privileges, with the shell redirected to `sh`, we could obtain a root shell once connected to the SSH.

To repack the image, we modified the "Supermicro BMC firmware image decryptor" tool to add firmware encryption support and constructed a firmware package with a valid footer and firmware table. We successfully tested and installed this modified firmware package both with the web firmware upgrade interface and the IPMI firmware upgrade interface via the `AlUpdate` tool.

5.4.3 Attack Chains for PMBus Access

In this section, we discuss three possible attack chains to take over the PMBus with the techniques shown in the previous sections. The attacker can use any of these attack chains and change the CPU voltage to perform PMFault attacks, *i.e.*, to over/undervolt the CPU.

Remote BMC Firmware Upgrade The first attack chain assumes a malicious insider threat model. This attack chain makes use of the web or IPMI interface through the BMC Ethernet connection. To use this interface, the attacker needs to have access to the BMC management Ethernet port or the shared management Ethernet port `eth0` on the system. Besides, the attacker needs to obtain valid credentials to login to the BMC.

In detail, the attacker can first use the method described in Section 5.4.2 to repack the `SMT_X11_163` firmware upgrade package from [190] to enable SSH `root` access to the BMC. Then, they can upload the firmware with the web management interface or the IPMI management interface over Ethernet. With the SSH interface enabled, the attacker can cross-compile the voltage-changing PoC described in Section 5.4.1 for the BMC, and then upload and execute it to send PMBus commands. We used `base64 -d > /tmp/i2c-pmbus-send` to upload our exploit code due to the unavailability of the SCP service on the BMC OS.

Local BMC Firmware Upgrade Similar to the first, this attack chain also involves a firmware upgrade for code execution on the BMC. However, we use the KCS interface discussed in Section 5.3.3 to upgrade the firmware. The attacker does not require access to the management Ethernet plane, instead, only `root` privileges on the OS running on the CPU is required. This is e.g., relevant for data centers that host bare metal machines for customers or for malware/ransomware that has obtained `root` through other exploits.

IPMI Interface The third attack chain uses the IPMI I2C functionality. An attacker with `root` access on the CPU OS or access to the management port of the BMC can use this interface to send commands to any I2C device that is connected to the BMC. The command used for sending the raw I2C packets is shown in Listing 5.1. The I2C mapping of this interface is the same as found during the initial investigation in Section 5.4.1. The VRM is at address `0x20` on bus 2. However, since the last bit of the first packet of I2C indicates the type of operation (read or write), we need to shift the device address left by one bit and set the last bit accordingly when using this interface to control PMBus.

```
1 ipmitool i2c bus=2 0x40 <PMBus Command> <PMBus Data>
```

Listing 5.1: IPMI command for sending I2C packets.

5.5 Undervolting and Overvolting Attacks

In this section, we show how under/overvolting through the PMBus leads to attacks on SGX and also permanent physical damage to the CPU. The attack requires any flaw that gives a software attacker access to the PMBus. As mentioned in Section 5.4.3, this can e.g., be a malicious firmware upgrade or the use of the IPMI-to-I2C functionality. The attack is generic in the sense that *various* flaws can lead to the same outcome: remote fault injection attacks on SGX and bricking the CPU. Figure 5.7 shows an overview of the attacks.

5.5.1 Undervolting Attack against Intel SGX

Adversary Model As mentioned in Section 5.1.2, we assumea threat model where an attacker (including a malicious insider) has full software access to the system but no (or limited) physical access. More precisely, the attacker has `root` access to the OS and software access to the BMC via the KCS interface or Ethernet. All attack chains described in Section 5.4.3 can generally be used under this threat model. It is worth mentioning that the attack that uses `ipmitool` through the KCS interface does not

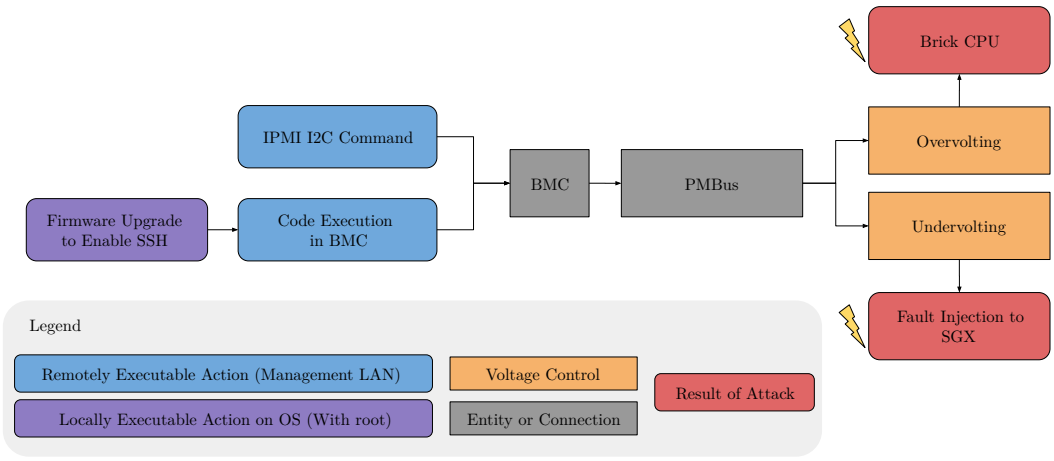


Figure 5.7: Overview of the PMFault attack. With root access to the OS or access to the BMC via Ethernet or KCS, the attacker can perform a malicious firmware upgrade of the BMC and then takeover the PMBus. The attacker can also use the `ipmi i2c` command to directly control the PMBus via BMC. With control over the CPU voltage, the attacker can overvolt to brick the CPU or undervolt to inject faults into SGX.

require knowledge of the BMC credentials. A privileged local user on a compromised host CPU can thus use `ipmitool` to inject fault into SGX purely from software.

Proof of Concept We used the same PoC code as Plundervolt/VoltPillager [20]. Before injecting the voltage glitch, we use the attack chain described in Section 5.4.3 to gain control of the PMBus.

To start with, we used the multiply operation as the first target, as it is a simple target to fault. By gradually lowering the CPU voltage with the PMBus commands sent by the BMC while running the Plundervolt/VoltPillager PoC on the CPU, we successfully injected faults into the multiply operation (in our experiments at voltage 0.845 V with the CPU running at 2 GHz.

To verify the fault injection also works for encryption operations running in SGX, we ran the CRT-RSA signature PoC from Plundervolt/VoltPillager, with an RSA signature computed inside an enclave using the Intel Integrated Performance Primitives (Intel IPP) cryptography library functions [191]. Again, we could obtain faulty signatures as shown in Listing 5.2. Furthermore, we confirmed that these faulty values could be used to factor

the RSA modulus and recover the private RSA key using the Lenstra attack [14].

```
// Faulty calculation 1
0x3f, 0xe0, 0xb8, 0x74, 0x04, 0x18, 0x9c, 0xed, 0x91, 0x1a, 0x02, 0x12, 0x2a,
0xce, 0x89, 0xf8, 0x32, 0x00, 0xdc, 0x05, 0x15, 0x53, 0x72, 0x8d, 0x84, 0x00,
0xd3, 0x67, 0xbe, 0xa1, 0xc2, 0x40, 0x76, 0xbc, 0x8c, 0xd8, 0xfe, 0xb1, 0x00,
0xd7, 0x9e, 0x0e, 0xb6, 0xac, 0x61, 0xc0, 0xec, 0x9c, 0xf7, 0x7e, 0xbc, 0x4b,
0xde, 0x18, 0xa5, 0xa4, 0x1c, 0x74, 0xc4, 0xb5, 0x6a, 0x8d, 0xd3, 0xb1, 0x35,
0xf9, 0xad, 0x0b, 0xe3, 0x4a, 0x01, 0x52, 0xd4, 0xc6, 0xb2, 0x95, 0xbc, 0xdc,
0xad, 0x61, 0x8e, 0x07, 0x84, 0x4d, 0xe3, 0xa7, 0xff, 0xf0, 0xd1, 0xa0, 0xd4,
0x58, 0x9f, 0xbc, 0x37, 0x0b, 0xa8, 0x91, 0x83, 0x15, 0x7b, 0xee, 0x28, 0x83,
0x12, 0x4a, 0x89, 0x61, 0x1e, 0x2c, 0xe1, 0x02, 0x2f, 0x08, 0x4d, 0x5b, 0x04,
0x92, 0x5e, 0x31, 0xd0, 0x7e, 0x94, 0x85, 0xd0, 0xce, 0x75, 0x4a, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
[... zeroes left out ...]
Incorrect result!
```

Listing 5.2: Faulty CRT-RSA decryptions/signatures generated by the respective ipps functions.

Reproducibility of CRT-RSA Fault Injection To further evaluate the reproducibility of the attack, we setup an automated testing environment by connecting a Raspberry Pi to an Ethernet port (`eth0`) and the power button of the motherboard. We ran a Python script to repeat the following steps numerous times:

1. Upload the exploit for controlling the CPU voltage to BMC via an SSH connection.
2. SSH into the OS running on the host CPU and trigger CRT-RSA signing in an SGX enclave.
3. Run the PMFault exploit on the BMC to gradually lower the CPU voltage while the signature is computed in the SGX enclave.
4. Stop lowering the CPU voltage when a fault occurs.
5. Record the result and cleanup.
6. If no faulty result is output, the system may have crashed due to too low voltage. In this case, we use the connection to the motherboard power button to reboot the system and wait to allow the system to boot into a stable status.

In total, we conducted 253 tests within 545 min. Of those, faults occurred in 194 tests. 66 of these faulty results could be used to successfully recover the correct RSA private key using the Lenstra attack, which translates to a success rate of 26%. On average, a useful fault could be obtained within 9 minutes.

5.5.2 Overvolting to Permanently Brick a CPU

Apart from the undervolting attack to extract keys from an SGX enclave, we also discovered another attack, which is an overvolting attack that can permanently destroy the CPU.

Adversary Model In this attack, as described in Section 5.1.2, we assume an attacker who has `root` privilege on the host CPU. For example, this could be in the case that an attacker has placed ransomware on a system and threatens to damage the CPU unless a ransom is paid. Clearly, `root` should have full control of all software running on the CPU, but *should not* be able to cause any physical damage to the system. The attack chain described in Section 5.4.3 using `ipmitool` with KCS can be used within this threat model.

Proof of Concept To overvolt the CPU, we firstly configure the `MFR_VR_CONFIG` register of the VRM to use the 10 mV SVID table. This allows changing the CPU voltage up to 3 V. We also disabled the over-current protection by reconfiguring the `MFR_OCP_TOTAL_SET` register. Then we used the voltage changing procedure to change the CPU voltage to a value much higher than the normal operating voltage.

We found that this procedure allows changing the CPU voltage up to ∼2.84 V for ∼1 ms, which is outside the typical operating range of Intel CPUs. By increasing the voltage beyond the specified operating voltage range (0.55 V–1.52 V) [184] of a 7th Gen Intel E3-1220V6 CPU two times, we permanently destroyed the CPU and left the system in an unbootable state within a few seconds. We successfully repeated the experiment

with a second, identical CPU. An example of overvolting is shown in Figure 5.8.

For environmental and financial reasons, we were satisfied after successfully destroying two CPUs and decided to not perform further experiments in that regard.

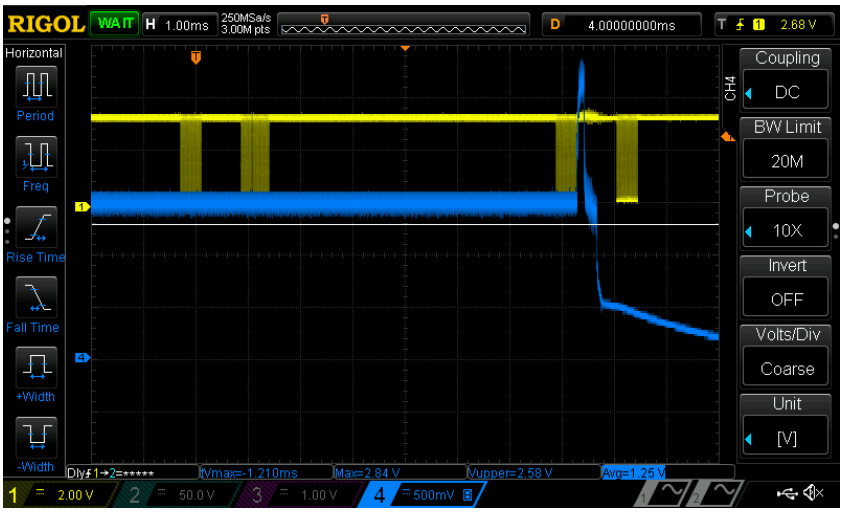


Figure 5.8: Oscillocope capture of voltage change during overvolting, `VOUT_COMMAND` set to `0xFF` (with 10 mV VID table). Yellow: PMBus clock, blue: V_{cpu}. V_{cpu} shoots up to 2.84 V during overvolting.

5.6 Evaluation of other Server Motherboards

As we found the PMBus to be a common interface present on server motherboard, we decided to investigate other manufacturers as well. To facilitate larger-scale testing of this, we wrote a tool called PMBusDetect. With this tool, we scan the system for a PMBus connection and try to detect the VRM address. We applied this tool to several other systems, including an ASRock rack motherboard (ASRock E3C246D4I-2T) and a Supermicro X12DPi-NT6 motherboard (kindly provided by Supermicro for testing). We then conducted further analysis of these systems to check if they are vulnerable to any PMBus-related attack.

PMBusDetect Tool for VRM Detection Based on the VRM detection process mentioned in Section 5.4.1, we built the PMBusDetect tool to automatically scan all

addresses of a specified I2C bus for VRMs. During testing, we found that the implementation of PMBus and usage of the VRM is different between motherboards, and the most stable command to identify a VRM is `READ_TEMPERATURE` (`0x8d`). We use the response to this command as an initial indicator to identify whether a VRM is present, and then use the VRM detection process from Section 5.4.1 to verify the result.

Moreover, as the capabilities and voltage changing sequence can differ between VRM vendor, we added an additional procedure to detect the vendor of the VRM. For this, we use the result of reading `ISL_DEVICE_ID` (`0xad`) as an indicator for Intersil VRMs and `SVID_VENDOR_PRODUCT_ID` (`0xbf`) for MPS, respectively. Detection based on `ipmi i2c` is also implemented for detecting the connection between VRM and the BMC as mentioned in Section 5.4.3. An example output of PMBusDetect with Supermicro X11SSL-CF is shown in Appendix D.2, while Table 5.3 shows a summary of the motherboard tested and the scan result for VRMs with PMBusDetect. We are aware that our testing—restricted by (lack of) access to server hardware— only gives a very limited picture of the use of PMBus and VRMs on server hardware. We hence decided to open-source PMBusDetect and build on community efforts in the future to obtain a better view of the PMBus landscape.

Name	BMC	Chipset	VRM Address	PMBus Connects to
Supermicro X11SSL-CF	AST2400	C232	`0x20`	BMC & CPU
Supermicro X12DPi-NT6	AST2600	C621A	`0x30` & `0x34`	—
ASRock E3C246D4I-2T	AST2500	C246	`0x60`	BMC & CPU

Table 5.3: Tested motherboards and their VRM detection result.

5.6.1 ASRock Power-Down Attack

The ASRock E3C246D4I-2T motherboard uses an Intel Xeon E-2124 CPU with an Intel C246 Chipset and ASPEED AST2500 BMC with login credentials defaulting to `ADMIN:ADMIN`. We used the PMBusDetect tool together with manual probing and found that the VRM of this motherboard is connected to both the BMC and I2C bus of the

CPU. In the following attack, we assume that the attacker is a user on a baremetal server with root access in the OS.

The VRM used on this motherboard is an ISL69138. Because it is made by a different manufacture compared to the MP2955, the voltage changing PMBus command sequence used for the MP2955 does not work with this VRM. Due to lack of documentation of this procedure, we at the moment could not precisely overvolt or undervolt the CPU via the PMBus. Yet, we discovered a new attack to disable the VRM and force power-down the CPU, leaving the system in a (temporary) inoperable state.

PMBusDetect shows that the VRM is at address `0x60` on I2C bus 2 of the host CPU. Different to the findings for the Supermicro X11SSL-CF, this VRM uses PMBus registers on page `0x1` instead of the default `0x0`. We then issue the `ON_OFF_CONFIG` (`0x02`) and `OPERATION` (`0x01`) commands: We configure the `OPERATION` to "Immediate Off" and set the "source of enable" only to `ON_OFF_CONFIG`. This results in a immediate power-off of the VRM and crashes the system.

During testing, we found the PMBus is only writable from the CPU with IPMI over KCS interface, but not from the BMC with `ipmi i2c` commands. As the result, it is not possible for the administrator of the system to remotely configure the VRM back to a normal state. Simply issuing the `ipmi powercycle` command with IPMI over LAN will leave the system in a infinite boot loop. To recover from this attack, the administrator has to physically power-cycle the system, which might increase downtime in a Denial-of-Service (DoS) scenario.

This shows that PMBus as an attack vector does not only affect Supermicro X11SSL-CF, but also can have impact on servers from other manufacturers. Besides we believe that it might also be possible to conduct CPU bricking attacks if the PMBus voltage changing sequence of Intersil VRM is known. We leave this for future work.

5.6.2 Other Supermicro X11 Motherboards

We also ran the PMBusDetect tool on X11SPG-TF and X11SSE-F Supermicro server motherboards—in both cases, the VRM was reachable in the default configuration. To test if they are vulnerable to PMFault, we sent PMBus commands through `ipmi i2c` commands and successfully undervolted them to crash the system. This shows that the attack chain through the IPMI interface is valid on these systems. As the systems were provided by a third party for remote testing, we were not able to attempt overvolting and similar, destructive experiments, but believe these motherboards to be equally affected.

5.6.3 Supermicro X12 Motherboards

We disclosed the vulnerability to Supermicro in May 2022. They confirmed the issue and also provided a X12 generation server for further testing. This system, Supermicro X12DPi-NT6, features a dual Intel Xeon Gold 6330 CPU, Intel C621A Chipset, and AST2600 BMC. Our investigation shows that mitigations has already been implemented on this motherboard to break the attack chain of PMFault before we reported the attack to Supermicro. Firstly, the firmware upgrade package is properly signed with RSA and verified during the firmware upgrade process, which prevents malicious firmware uploads to the BMC via IPMI. This breaks the attack chain though firmware upgrade. Secondly, I2C packet filtering has been implemented in the BMC, which prevents IPMI commands to directly send packets to the PMBus. Moreover, our PMBusDetect tool shows that the VR is not connected to the CPU, which prevents an attack directly from the operating system. In conclusion, to the best of our knowledge, we believe that Supermicro X12DPi-NT6 is not directly vulnerable to the attacks described in this paper. However, we note that as-of-yet unknown vulnerabilities might remain in the firmware update process and the complex software stack running on the BMC, which warrants further investigation.

5.7 Conclusions and Countermeasures

In this paper, we demonstrated two remote attacks that use the PMBus interface to control the CPU voltage. An undervolting attack can be used to inject fault to the SGX enclave of the CPU and e.g., recover a secret key used in cryptography algorithms. The overvolting attack causes permanent damage to the CPU.

The attack affects, to our knowledge, all 11th generation Supermicro systems. It also impacts ASRock (tested with ASRock E3C246D4I-2T), though as described the VRM behaves differently to Supermicro. We suspect that the attack might also affect other vendors (given that BMCs are often similar), but could not further investigate this and thus leave it for future work.

5.7.1 Server Platform Security and Embedded System Security

We first discuss the security considerations for server platforms. Previous security research on computer platforms were mainly focused on the security of the software (either running on the CPU or the management controller). However, each subsystem on a server platform does not act in isolation. Instead, they may interact with each other via the physical connections on the motherboard. In our attacks, we show that the hardware design of the system with a correctly implemented `ipmitool` can lead to severe security issues and damage to the system.

Apart from the components on the motherboard, one should also take "plugin" devices into consideration when analysing the security of server platforms. During our investigation of the system, we found that when a Peripheral Component Interconnect Express (PCI-E) device is plugged onto the motherboard, it is also connected to the I2C bus of the motherboard. However, if the firmware of a PCI-E device is compromised, it can gain access to the PMBus to perform the same attacks described in this paper. On E3-1220V6-X11SSL-CF, this connection can be configured with a jumper named `JI2C`. Although this jumper is disconnected by default, the user may not be aware of the

security implications of connecting this jumper.

In summary, the server platform is a system that has multiple components and microcontrollers. The security of the platforms is not only down to ensuring the security of the software running on it, but the overall design of the hardware and embedded systems on the motherboard should also go through a thorough security review. Securing such a system needs collaborative effort of both software developers and hardware engineers.

5.7.2 SGX Security

Our attack on SGX enclaves shows that a privileged local attacker can inject a fault to the enclave and recover secret information with the server management interface, effectively reviving Plundervolt-like software undervolting attacks on Supermicro X11 motherboards. We also demonstrate that a malicious service provider (e.g., cloud hoster) can use the attack chains described in the paper to break the security guarantee provided by SGX. Moreover, the vulnerability currently cannot be detected/mitigated by SGX attestation, because the BMC and its firmware are not within the scope of SGX attestation.

A supply chain attack is also possible: as the firmware is not securely verified, it is possible for a third party to implant malware into the BMC and later launch remote attacks on SGX and/or damage the CPU. Such a firmware modification is also conceivable while the device is being shipped to the end user. Detecting such attack would be hard, as the firmware of the BMC is stored in a separate flash chip. The software running on the BMC is thus usually out-of-scope of traditional malware detection methods.

5.7.3 Countermeasures

Overvolting Attack According to our experiments, PMBus-based overvolting can lead to permanent damage to the CPU and thus permanent DoS of the system.

The fundamental issue that leads to this attack is the lack of a hardcoded voltage

limit of the VRM. Simply adding signature verification of the BMC firmware or using secure boot to break the attack chain might not be sufficient to prevent overvolting, as other, future attacks might also yield PMBus access. Besides, configuring software-based PMBus read/write limitations of the VRM through the `MFR_PWD_USER` command is also insufficient to stop the attack. This is because this features only sets a 16-bit passcode, which is prone to brute force attack. We suggest the following mitigations be implemented for this attack to break the attack chain:

1. In the short term, the user manual of the relevant system(s) should be updated to describe the usage and suggested configuration of the `SMBDAT_VRM` and `SMBCLK_VRM` jumpers, if they are present on a specific model.
2. In the long term, an alternative VRM with a hardwired voltage safety limit should be used to replace the current VRM.
3. Another mitigation would be implementing an I2C filter to detect and block malicious PMBus packets. `MFR_VR_CONFIG`, which can be used to set a 10 mV VID table, is one of the main commands that need to be blocked. Optionally, other commands that involved in the overclocking procedure could be blocked, however, this may affect users who actually want to use this feature. Such a filter could be implemented in a small microcontroller that listens to the I2C bus and "jams" malicious commands by actively pulling the bus low once the command has been detected but before its transmission has been completed.

PMBus-based SGX Undervolting To the best of our knowledge, PMFault represents the first attack that directly breaches integrity guarantees in the Intel SGX security architecture through the PMBus interface. We believe that the fix currently deployed by Intel against Plundervolt/V0ltPwn (CVE-2019-11157)—disabling the SVID undervolting interface—is insufficient when a remote attacker can get access to the PMBus through the BMC or I2C interface of the CPU, as is the case for Supermicro X11 motherboards. We note that there might be many other devices connected to the bus, including PCI-E

devices like graphic cards. It is thus also possible for a compromised PCI-E device to send malicious commands to control the CPU voltage.

Given the potential impact of our findings regarding fault injection into SGX enclaves, in the short term, we recommend inserting software-based fault injection countermeasures into cryptographic computations in enclaves (e.g., the quoting enclave). However, we note that such fixes can only serve as mitigations, but not fully eliminate this attack vector.

We would like to highlight that in our opinion, this attack surface *cannot* be easily addressed by jumpers to disconnect the VRM from the SMBus or adding signature verification of the BMC firmware, as we believe that SGX attestation cannot independently verify the relevant system configurations:

1. The existence of a PMBus/SMBus interface to the VRM and whether it can be controlled through the I2C interface of the CPU;
2. The existence of an external microcontroller on the motherboard and if it has the functionality to control the VRM (e.g., BMC or other PCI-E devices);
3. The firmware security status of the BMC and other devices on the PMBus.

This will make it impossible to give SGX assurance of the trust status of the system. We believe that in the long term, appropriate hardware countermeasures *inside* the CPU package is required: this could on the one hand include continuous monitoring of the received supply voltage, as recently presented by Intel for critical parts of their systems [192], and on the other the use of fully-integrated voltage regulators.

Acknowledgements

This research is partially funded by the Engineering and Physical Sciences Research Council (EPSRC) under grants EP/R012598/1, EP/R008000/1, and EP/V000454/1. The results feed into DsbDtech. We would also like to thank Supermicro for providing a X12DPi-NT6 server for further investigation of the issue.

Chapter Six

Conclusion and Future Work

As more and more features are built into the computer system, there can be multiple microcontrollers on a PCB which can affect each other. This type of design is not only commonly seen in automotive systems, but also in PCs and servers. In order to secure such systems, the program running in each microcontroller or CPU needs to be evaluated for its security.

In this thesis, MetaEmu provides a universal environment for emulating devices of different architectures. This tool opens the door to conducting large-scale dynamic analysis of the esoteric firmware. Before the creation of MetaEmu, dynamic analysis was impossible because of the lack of emulation support for these architectures. The new firmware partitioning and functionality inferring tool introduced in Incision provides a way to identify functions of interest in larger firmware blobs. This tool, together with MetaEmu, greatly reduced the manual effort needed for analysing firmware and makes large-scale security analysing of esoteric firmware possible.

In the work of VoltPillager, I developed the first hardware-based fault injection attack to Intel SGX. It shows that when the service provider is considered malicious (the threat model of TEEs), simply disabling the in-package voltage control interface is not enough to stop the hardware-based attacks. The discovery of the VoltPillager attack ultimately leads to a rethink of the threat model of SGX and makes Intel exclude "attackers with physical access to the system" from their threat model. Also, this work provided a deeper

analysis of voltage control interfaces on an Intel motherboard at hardware level, which provides the basis for further attacks on x86 systems that make use of VRM.

As shown in PMFault, the voltage-control hardware connection can be exploited through a firmware vulnerability in other microcontrollers of the system. There are multiple factors that cause this issue, including insecure hardware layout and insecure design of the firmware running in BMC. This attack lead to a rethink of approaches to be used when analysing the security of modern computers and servers and opens up a new area of security research of modern computing systems which is board-level security analysis. Simply securing the CPU is not enough to protect the system against attacks where there are multiple microcontrollers on the same motherboard. In summary, a more holistic method, including a combination of software analysis, firmware analysis and board-level hardware design analysis, is required to build a complete picture of the security status of the TEE and systems that use it.

Future Work Based on the work in this thesis, there are several directions for future work. In the area of firmware analysis, MetaEmu has provided a building block for dynamic firmware analysis and been especially useful in automotive firmware. With the larger range of processor architecture that MetaEmu supports, this framework can not only be used for automotive firmware, but also be used to emulate microcode in x86 systems. More analysing tools can be built on MetaEmu with the powerful hooking and introspecting interface it provides. The research in VRM and PMBus have shown a new attacker surface for modern computer systems. These attacks make use of the venerability in PCB design and firmware of BMC. With the growing complexity of computer systems, there are other ICs on the motherboard which can lead to similar attacks. Reviewing systems with such designs requires a collaborative effort from both hardware and software security researchers.

Appendix One

Appendix of MetaEmu

A.1 Execution Trace visualised in Ghidra

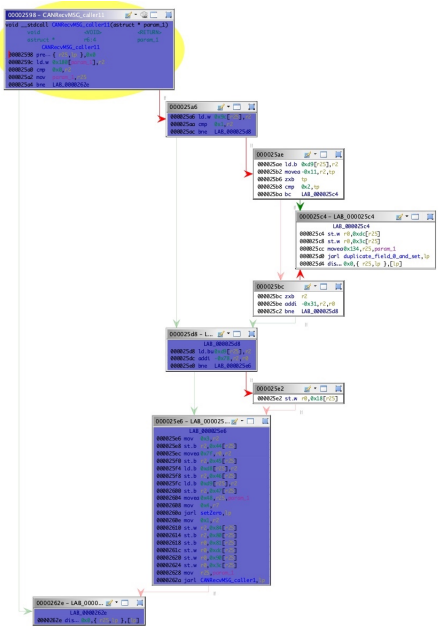


Figure A.1: Example Function Graph processed with trace visualizer. The executed flow is colored with purple

A.2 Execution Observer Event API

Name	Purpose
hook_operand_read	Intercept a register or memory read.
hook_operand_write	Intercept a register or memory write.
hook_memory_read	Intercept a memory read.
hook_memory_write	Intercept a memory write.
hook_register_read	Intercept a register read.
hook_register_write	Intercept a register write.
hook_architectural_step	Intercept a program counter change and the lifted instructions at the new address.
hook_operation_step	Intercept a single IL operation step.
hook_cbranch	Intercept a conditional branch.
hook_call	Intercept a function call.

A.3 Benchmark Programs

Name	Description
loop1-mem	Fibonacci sequence computed in a tight loop with a large N; multiple small blocks; uses stack to store all computed values.
loop2-reg	Fibonacci sequence computed in a tight loop with a large N; multiple small blocks; uses registers to store all computed values.
loop3-2loop	loop1-mem wrapped in a loop of 1k iterations: two tight loops. Maximizes optimization potential for Unicorn's JIT.
10k-no-count	10k instructions (multi-block), no execution bound set.
10k-count	10k-no-count with a counter updated after each processed instruction. We implement counting for Unicorn by hard-coding parameter 4 of `start_emu` to be the number of instructions to process (i.e., 10k).
100k-no-count	100k instructions (multi-block), no instruction bound.
100k-count	100k instructions (multi-block), with a counter update after each instruction, 100k instruction bound.
1M-no-count	1M instructions (multi-block), no instruction bound.

A.4 List of Automotive Firmware

Firmware	ISA	Firmware Extraction Method
Volkswagen IC	C166	Through JTAG with OpenOCD
Renault BCM	V850E2	Extracted with Renesas Flash Programmer
Land Rover Discovery TCU	RH850	Extracted with Renesas Flash Programmer
Range Rover Evoque TCU	SH-2A	Firmware extraction over CAN [193]

Appendix Two

Appendix of Incision

B.1 Huawei R216h 4G/LTE CoreSight Configuration

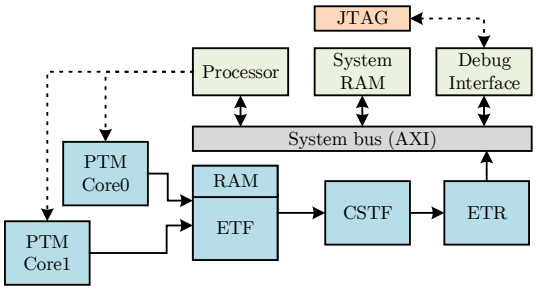


Figure B.1: CoreSight configuration as used on the Huawei R216h 4G/LTE modem. Trace data from the two PTMs which monitor cores 0 and 1 of the ARM CPU is initially stored in the local memory of the Embedded Trace FIFO (ETF) module. From there trace data is fetched via the CoreSight Trace Funnel (CSTF) by the Embedded Trace Router (ETR) module and transferred to the system RAM. Finally, the trace data in RAM is accessed externally through the JTAG interface.

The HiSilicon ARM processor on the Huawei R216h 4G/LTE baseband has tracing capabilities provided by ARM CoreSight [194]. We can configure and access the tracing peripherals through the JTAG interface. Figure B.1 depicts the configuration for our device. In our setup, we configure the trace data to flow from the Program Trace Module (PTM), which traces the VxWorks core, through the Embedded Trace Funnel (ETF) and CoreSight Trace Funnel (CSTF), to the Embedded Trace Router (ETR), which finally

writes the data to a buffer we specify in the RAM. We can then halt the device and access the trace data in the buffer through JTAG. The hardware also contains a Trace Port Interface Unit (TPIU), which outputs the trace as a stream. However, since we did not know the pin-out for this trace port, and capturing the stream would require specialized (and expensive) hardware, we did not attempt to trace the device in this way.

Our setup facilitates two types of traces: ❶ *complete traces* which contain trace data for all the code blocks starting from a specific given start address to either a given stop address or to when the trace buffer is exhausted, and ❷ *address range traces* which contain trace data for code blocks that fall within a specified range, similarly capped to the size of the storage space. A combination of ❶ and ❷ is also possible where ❷ is used as an exclusion range such that trace data can be obtained for everything that is not contained inside the excluded range(s).

B.2 Analyzing the Huawei R216h Baseband with INCISION

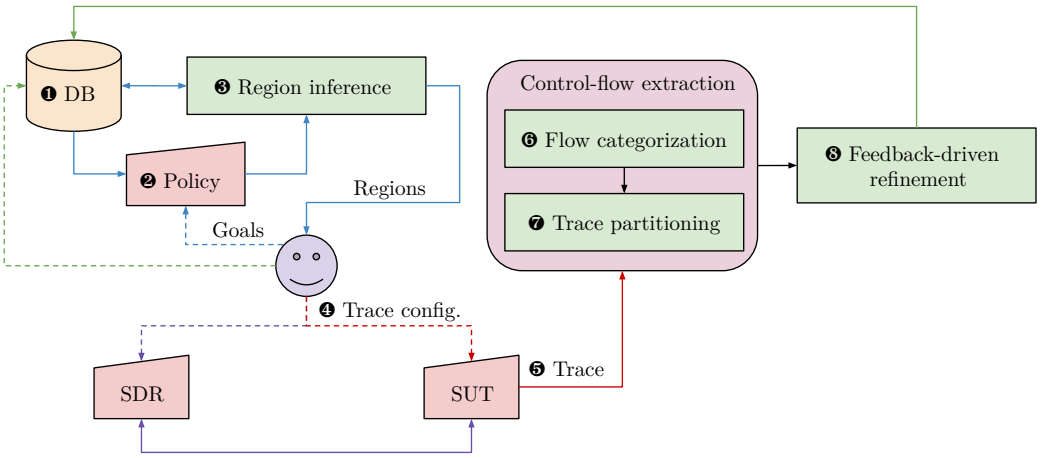


Figure B.2: Overview of INCISION applied to analyze the Huawei R216h baseband firmware. We indicate the human-in-the-loop by a purple face. We depict data-flow by block arrows where propagation is automated by INCISION, and dashed arrows where manual intervention is required.

In this section, we provide a break-down of the steps taken to reverse engineer the Huawei R216h baseband firmware using INCISION. We visualize the analysis set-up and data-flow in Figure B.2, which expands on Figure 2.1 to show the additional manual intervention required to use INCISION with this particular device.

As described in Section 3.5.3, the hardware set-up consists of an Ettus USRP B210 SDR (SDR in Figure B.2), the Huawei R216h configured as detailed in Appendix B.1 (SUT in Figure B.2), and a laptop with Ghidra and INCISION installed. The analysis process is orchestrated by a human-in-the-loop, depicted by the purple face in Figure B.2. As in the *standard* workflow to reverse engineer devices of this kind, we proceed by obtaining a dump of the device firmware and bootstrap the reverse engineering process by loading the dump into Ghidra and triggering its auto-analysis pass (step ❶ in Figure B.2). Following this set-up stage, we proceed to reverse engineer the device firmware by applying INCISION in feedback loops.

In the next step, we form an initial reverse engineering policy based on our goal

to analyze how the device handles cryptographic keys (step ❷). We encode this policy using a simple Python-based API provided by INCISION, and use it as input to the region inference algorithm (step ❸). As the device's trace hardware is limited in the number of regions that can be traced, we supply the algorithm with an upper bound on the number of regions to output (a value of 4). We depict the data-flow of this stage in blue.

The result of applying region inference is a set of region bounds, which we use to configure the CoreSight-based tracing mechanism on the device (step ❹). Since the analysis goal is to capture traces where the device performs operations using cryptographic keys, we configure the SDR to provide suitable triggering inputs to the device (data-flow depicted in purple). Following this, we perform trace capture (data-flow in red), which we terminate upon observing sufficient I/O interaction between the SDR and device. This provides us with a trace (step ❺), which we use as input to INCISION control-flow extraction procedures (steps ❻ and ❼).

The final step in the loop (step ❽) integrates the extracted control-flow information back into the firmware database (data-flow in green). For the first two loop iterations, this process requires manual intervention, as described in Section 3.5.4. We continue to apply INCISION, as outlined in Section 3.5.3 to capture additional traces which further refine the firmware database and provide a basis for analysis of the key usage in the firmware.

Appendix Three

Appendix of Voltpillager

C.1 Glitch Configuration and Results for Multiplication and CRT-RSA

Device	Clock	T_d	N	V_p	T_p	V_f	T_f	V_n	Iteration	Temp.
i3-7100-AZ170	2 GHz	1000 µs	1	0.83 V	35 µs	0.64 V	29 µs	0.83 V	90236	23° C
i3-9100-MZ370	3.4 GHz	300 µs	1	1.050 V	35 µs	0.81 V	83 µs	1.05 V	126490	26° C
i3-7100U-NUC	2 GHz	200 µs	1	0.94 V	35 µs	0.71 V	8 µs	1.05 V	41827	32° C

Table C.1: Parameters for successful fault injection with the Plundervolt PoC for userspace multiplication

Device	Clock	T_d	N	V_p	T_p	V_f	T_f	V_n	Temp.
i3-7100-AZ170	2 GHz	100 µs	1	0.83 V	35 µs	0.63 V	29 µs	0.83 V	24° C
i3-9100-MZ370	3.4 GHz	10 µs	1	1.050 V	10 µs	0.81 V	43 µs	1.05 V	27° C
i3-7100U-NUC	2 GHz	10 µs	1	0.94 V	35 µs	0.75 V	9 µs	1.05 V	22° C

Table C.2: Parameters for successful fault injection with the Plundervolt PoC for CRT-RSA

Run	T_d	N	V_p	T_p	V_f	T_f	V_n	Temp.	Iteration
1	200 µs	1	0.95 V	35 µs	0.76 V	29 µs	0.95 V	24° C	97702
2	200 µs	1	0.95 V	35 µs	0.76 V	29 µs	0.95 V	23° C	92286
3	500 µs	1	0.95 V	35 µs	0.76 V	29 µs	0.95 V	23° C	174087

Table C.3: Parameters for fault injection into the initial memory access PoC on i3-7100-AZ170 at 3 GHz

```
0x3c, 0xf7, 0x21, 0x56, 0xe7, 0x59, 0x69, 0x06,
0x08, 0x06, 0x01, 0x69, 0xf0, 0xa3, 0x0c, 0xb9,
0x0d, 0x3b, 0x75, 0xe9, 0x02, 0xb3, 0xe0, 0x05,
0xef, 0x59, 0xbf, 0x05, 0x54, 0x0f, 0xec, 0xc3,
0xc8, 0x90, 0x7b, 0x45, 0x90, 0x9c, 0x4b, 0x4e,
0xfc, 0x8d, 0xed, 0x0f, 0x31, 0xaa, 0xad, 0xae,
0x40, 0x0d, 0xf3, 0xc4, 0x6c, 0x00, 0x3b, 0xdd,
0x7a, 0xf6, 0x22, 0x61, 0x53, 0x2a, 0xcc, 0xf2,
0x16, 0xb9, 0xa7, 0x3e, 0x98, 0xbb, 0x8f, 0x56,
0xad, 0x4c, 0x35, 0xa2, 0x6e, 0x47, 0xd8, 0x80,
0x36, 0x4c, 0x9a, 0x2b, 0xab, 0x25, 0x08, 0x63,
0x93, 0x28, 0x6b, 0x98, 0xad, 0xda, 0x74, 0xab,
0x8b, 0xd2, 0x04, 0xeb, 0x4e, 0x76, 0xc5, 0x09,
0xe7, 0xd8, 0x5f, 0x97, 0xf3, 0x13, 0x75, 0x29,
0xd3, 0xa6, 0x07, 0xb5, 0x1f, 0x9f, 0x07, 0xfc,
0x82, 0x19, 0x70, 0x04, 0xda, 0x12, 0x71, 0x3e
```

Listing C.1: Example faulty result obtained on i3-7100-AZ170 for CRT-RSA

```
0x41, 0xbf, 0xa9, 0x4d, 0x96, 0xae, 0x2d, 0x35,
0xe4, 0xa8, 0xc7, 0x24, 0xaa, 0x8c, 0xc2, 0x05,
0x0f, 0x32, 0x56, 0xe5, 0x37, 0x56, 0x5d, 0x94,
0x31, 0x82, 0x62, 0xd8, 0xbc, 0x32, 0x34, 0xc0,
0x70, 0xdb, 0xfe, 0x98, 0xcc, 0x6e, 0x26, 0x75,
0x58, 0xa8, 0x2a, 0x84, 0xe7, 0x14, 0xe2, 0x4a,
0x93, 0x3b, 0xc2, 0x4d, 0xe9, 0xcb, 0xa2, 0x61,
0x07, 0x62, 0x88, 0xcb, 0x01, 0x36, 0x58, 0x1d,
0x8d, 0x09, 0x9b, 0x0a, 0x0b, 0x7e, 0x42, 0xd0,
0x68, 0xbb, 0x16, 0x28, 0x60, 0x14, 0x78, 0x3d,
0x73, 0x0a, 0xf5, 0x62, 0x2d, 0xbd, 0x22, 0xf0,
0x59, 0x96, 0x39, 0x5c, 0xbc, 0xe1, 0x46, 0x0b,
0x99, 0x3e, 0x04, 0x4a, 0x69, 0xbc, 0xdf, 0xc0,
0x5b, 0xb3, 0x98, 0x11, 0x56, 0xea, 0x03, 0xa2,
0x3a, 0x80, 0xc9, 0xd3, 0xe0, 0x7c, 0x55, 0xe6,
0x5c, 0x20, 0x13, 0x86, 0x7b, 0xba, 0x87, 0x6d
```

Listing C.2: Example faulty result obtained on i3-9100-MZ370 for CRT-RSA

```
0x6e, 0x35, 0xea, 0x8c, 0xac, 0xe4, 0xe8, 0x1d,
0xc0, 0x3f, 0x52, 0xe7, 0xf8, 0x27, 0x21, 0xd1,
0x75, 0x86, 0x1e, 0x30, 0xe7, 0xe6, 0x90, 0x07,
0x5a, 0xc6, 0xed, 0x97, 0x21, 0x59, 0xad, 0x4d,
0x61, 0x64, 0x43, 0x5f, 0x70, 0x78, 0xbc, 0x78,
0x1a, 0x82, 0x1e, 0x0d, 0x8f, 0xd3, 0x6d, 0x27,
0x78, 0x28, 0x72, 0x6f, 0xf9, 0x63, 0x6e, 0x8e,
0x92, 0x98, 0x40, 0x96, 0x2e, 0xde, 0x28, 0x0a,
0x14, 0x1d, 0xc0, 0xc3, 0x27, 0xf3, 0x44, 0xa8,
0x8d, 0xf5, 0xb5, 0xe5, 0x1c, 0x96, 0xed, 0xe4,
0xf6, 0x11, 0xa4, 0xa6, 0x26, 0x7f, 0xf1, 0x82,
0xaf, 0x33, 0x85, 0x24, 0xc5, 0x3d, 0x67, 0x2a,
0x55, 0x69, 0xd9, 0xc3, 0x9b, 0xcb, 0x25, 0xfc,
0xa4, 0x9a, 0x2a, 0x5d, 0x6e, 0xa6, 0x92, 0x97,
0xf0, 0x14, 0x3f, 0x8e, 0x91, 0x33, 0x65, 0xa1,
0x61, 0x0f, 0x75, 0xbf, 0xc1, 0x08, 0xec, 0x61
```

Listing C.3: Example faulty result obtained on i3-7100U-NUC for CRT-RSA

C.2 Example Results for Faults during Memory Accesses

The following out-of-bounds overflow fault happened at iteration 769170 with -172 mV undervolting and the CPU running at 2 GHz on i7-7700HQ-XPS during computation inside SGX.

```
1 [Enclave] FAULT: array[00]:    0x00000000
2 [Enclave] FAULT: array[01]:    0xabababab
3 [Enclave] FAULT: array[02]:    0xabababab
4 [Enclave] FAULT: array[03]:    0xabababab
5 [Enclave] FAULT: array[04]:    0xabababab
6 [Enclave] FAULT: array[05]:    0xabababab
7 [Enclave] FAULT: array[06]:    0xabababab
8 [Enclave] FAULT: array[07]:    0xabababab
```

Listing C.4: Overflow on i7-7700HQ-XPS

The following out-of-bounds underflow happened at iteration 210612 with -175 mV undervolting on the same system during computation inside SGX.

```
1 [Enclave] FAULT: array[00]:    0xabababab
2 [Enclave] FAULT: array[01]:    0xabababab
3 [Enclave] FAULT: array[02]:    0xabababab
4 [Enclave] FAULT: array[03]:    0xabababab
5 [Enclave] FAULT: array[04]:    0xabababab
6 [Enclave] FAULT: array[05]:    0x00000000
7 [Enclave] FAULT: array[06]:    0x00000000
8 [Enclave] FAULT: array[07]:    0x00000000
```

Listing C.5: Underflow on i7-7700HQ-XPS

C.3 Details of SVID

The 1-byte VID value for a target voltage U (in volt) is computed as:

$$\text{VID} = \left\lfloor \frac{U - 0.245}{0.005} \right\rfloor$$

SVID commands are 5 bit. We used SetVID-Fast (`0x01`) for setting the voltage. We also discovered other commands shown in Table C.4 with the help of a screenshot of an SVID protocol analyzer [155].

Command name	Value
Extended	0x00
SetVID-Fast	0x01
SetVID-Slow	0x02
SetVID-Decay	0x03
SetPS	0x04
SetRegADR	0x05
SetRegDAT	0x06

Table C.4: 5-bit SVID commands based on [155]

Appendix Four

Appendix of PMFault

D.1 `i2cdetect` Result for Supermicro X11SSL-CF

```
~$ sudo i2cdetect 0
          0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
[00-20]: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30:      -- -- -- -- -- -- -- 37 -- -- -- -- -- -- -- --
40:      -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50:      50 -- -- -- -- -- -- -- 58 -- -- -- -- -- -- --
60:      -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70:      -- -- -- -- -- -- -- --

~$ sudo i2cdetect 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- 08 -- -- -- -- -- -- --
10: 10 -- -- -- -- -- -- -- -- 19 -- -- -- -- -- --
20: 20 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: 30 -- -- -- -- 35 36 -- -- -- -- -- -- -- -- --
40: -- -- -- -- 44 -- -- -- -- -- -- -- -- -- -- --
50: -- 51 -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

D.2 PMBusDetect Result for Supermicro X11SSL-CF

```
$ sudo modprobe i2c_i801
$ sudo ./pmbusdetect -d /dev/i2c-1
Device 0x20              READ_TEMPERATURE success: 0019
!!!!!!!!!!! Detected! Device addr: 20 !!!!!!!!!!!
Device 0x20              SVID_VENDOR_PRODUCT_ID success,
data: 2555 This device is likely to be a MPS VRM
# Save the page
Device 0x20 : 00         READ_PAGE success

Page: 00
Device 0x20 : 00         WRITE_PAGE success
Device 0x20 : 00         READ_VOUT success: 00D8

Page: 01
Device 0x20 : 01         WRITE_PAGE success
Device 0x20 : 01         READ_VOUT success: 0001
# Restore the page
Device 0x20 : 00         WRITE_PAGE success
```

References

[1] Z. Chen, S. L. Thomas, and F. D. Garcia, "MetaEmu: An Architecture Agnostic Rehosting Framework for Automotive Firmware," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22, Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 515–529, ISBN: 9781450394505. DOI: 10.1145/3548606.3559338.

[2] S. L. Thomas, J. Van den Herrewegen, G. Vasilakis, Z. Chen, M. Ordean, and F. D. Garcia, "Cutting through the complexity of reverse engineering embedded devices," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 360–389, 2021.

[3] Z. Chen, G. Vasilakis, K. Murdock, E. Dean, D. Oswald, and F. D. Garcia, "VoltPillager: Hardware-based fault injection attacks against Intel SGX enclaves using the SVID voltage scaling interface," in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, Aug. 2021, pp. 699–716, ISBN: 978-1-939133-24-3. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/chen-zitai.

[4] Z. Chen and D. Oswald, "PMFault: Faulting and Bricking Server CPUs through Management Interfaces: Or: A Modern Example of Halt and Catch Fire," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2023, no. 2, pp. 1–23, Mar. 2023. DOI: 10.46586/tches.v2023.i2.1-23.

[5] S. Furst, "Challenges in the design of automotive software," in *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, 2010, pp. 256–258. DOI: 10.1109/DATE.2010.5457201.

[6] A. Tereshkin and R. Wojtczuk, "Introducing ring -3 rootkits," Black Hat USA, 2009. [Online]. Available: https://www.blackhat.com/presentations/bh-usa-09/TERESHKIN/BHUSA09-Tereshkin-Ring3Rootkit-SLIDES.pdf (visited on 01/06/2023).

[7] M. Goryachy and M. Ermolov, "How to hack a turned-off computer, or running unsigned code in Intel Management Engine," Black Hat Europe 2017, Nov. 2017. [Online]. Available: https://www.blackhat.com/docs/eu-17/materials/eu-17-Goryachy-How-To-Hack-A-Turned-Off-Computer-Or-Running-Unsigned-Code-In-Intel-Management-Engine.pdf (visited on 01/06/2022).

[8] *CVE-2017-5689*, Available from MITRE, CVE-ID CVE-2017-5689. Feb. 2017. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5689 (visited on 10/31/2022).

[9] Eclypsium, *Remotely Bricking a Server*, https://eclypsium.com/2018/12/19/remotely-bricking-a-server/, Dec. 2018. (visited on 10/31/2022).

[10] Tencent Keen Security Lab, *Tencent Keen Security Lab: Experimental Security Assessment on Lexus Cars*, https://keenlab.tencent.com/en/2020/03/30/Tencent-Keen-Security-Lab-Experimental-Security-Assessment-on-Lexus-Cars/, 2020.

[11] ——, *Mercedes Benz MBUX Security Research Report*, https://keenlab.tencent.com/en/whitepapers/Mercedes_Benz_Security_Research_Report_Final.pdf, 2021.

[12] F. Bellard, *QEMU: A generic and open source machine emulator and virtualizer*, https://www.qemu.org, 2003.

[13] N. A. Quynh and H.-V. Dang, *Unicorn: Next Generation CPU Emulator Framework*, https://www.unicorn-engine.org, 2015.

[14] D. Boneh, R. A. Demillo, and R. J. Lipton, "On the Importance of Checking Cryptographic Protocols for Faults," in *Eurocrypt'97*, 1997, pp. 37–51.

[15] N. Timmers and C. Mune, "Escalating Privileges in Linux Using Voltage Fault Injection," in *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2017, pp. 1–8. DOI: 10.1109/FDTC.2017.16.

[16] J. G. 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*, 2011, pp. 91–99. DOI: 10.1109/FDTC.2011.12.

[17] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The sorcerer's apprentice guide to fault attacks," *Proceedings of the IEEE*, vol. 94, no. 2, pp. 370–382, 2006.

[18] B. Yuce, P. Schaumont, and M. Witteman, "Fault Attacks on Secure Embedded Software: Threats, Design, and Evaluation," *Hardware and Systems Security*, vol. 2, no. 2, pp. 111–130, 2018. DOI: 10.1007/s41635-018-0038-1.

[19] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *ISCA*, 2014.

[20] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based Fault Injection Attacks against Intel SGX," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[21] Z. Kenjar, T. Frassetto, D. Gens, M. Franz, and A.-R. Sadeghi, "V0LTpwn: Attacking x86 Processor Integrity from Software," in *USENIX Security '20*, Boston: USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/kenjar.

[22] P. Qiu, D. Wang, Y. Lyu, and G. Qu, "VoltJockey: Breaking SGX by Software-Controlled Voltage-Induced Hardware Faults," in *AsianHOST '19*, 2019, pp. 1–6.

[23] A. Tang, S. Sethumadhavan, and S. Stolfo, "CLKSCREW: Exposing the perils of security-oblivious energy management," in *USENIX Security '17*, Vancouver, BC: USENIX Association, Aug. 2017, pp. 1057–1074, ISBN: 978-1-931971-40-9. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang.

[24] P. Qiu, D. Wang, Y. Lyu, and G. Qu, "VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-core Frequencies," in *CCS '19*, London: ACM, 2019, pp. 195–209, ISBN: 978-1-4503-6747-9. DOI: 10.1145/3319535.3354201.

[25] J. Gyselinck, J. Van Bulck, F. Piessens, and R. Strackx, "Off-Limits: Abusing Legacy X86 Memory Segmentation to Spy on Enclaved Execution," in *Engineering Secure Software and Systems: 10th International Symposium, ESSoS 2018, Paris, France, June 26-27, 2018, Proceedings*, Paris, France: Springer-Verlag, 2018, pp. 44–60, ISBN: 978-3-319-94495-1. DOI: 10.1007/978-3-319-94496-8_4.

[26] J. V. Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution," in *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC: USENIX Association, Aug. 2017, pp. 1041–1056, ISBN: 978-1-931971-40-9. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/van-bulck.

[27] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, Dallas,

Texas, USA: Association for Computing Machinery, 2017, pp. 2421–2434, ISBN: 9781450349468. DOI: 10.1145/3133956.3134038.

[28] D. Moghimi, T. Eisenbarth, and B. Sunar, "MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations," *International Journal of Parallel Programming*, vol. 47, Aug. 2019. DOI: 10.1007/s10766-018-0611-9.

[29] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache Attacks on Intel SGX," in *Proceedings of the 10th European Workshop on Systems Security*, ser. EuroSec'17, Belgrade, Serbia: Association for Computing Machinery, 2017, ISBN: 9781450349352. DOI: 10.1145/3065913.3065915.

[30] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How SGX amplifies the power of cache attacks," in *International Conference on Cryptographic Hardware and Embedded Systems*, Springer, 2017, pp. 69–90.

[31] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using SGX to conceal cache attacks," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2017, pp. 3–24.

[32] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 991–1008.

[33] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[34] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-privilege-boundary data sampling," in *Proceedings of*

the 2019 ACM SIGSAC Conference on Computer and Communications Security, 2019, pp. 753–768.

[35] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue in-flight data load," in *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019, pp. 88–105.

[36] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lippi, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens, "LVI: Hijacking transient execution through microarchitectural load value injection," in *2020 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2020, pp. 54–72.

[37] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre Returns! Speculation Attacks using the Return Stack Buffer," in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, Baltimore, MD: USENIX Association, Aug. 2018. [Online]. Available: https://www.usenix.org/conference/woot18/presentation/koruyeh.

[38] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, "CacheOut: Leaking data on Intel CPUs via cache evictions," in *2021 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2021, pp. 339–354.

[39] R. Buhren, H.-N. Jacob, T. Krachenfels, and J.-P. Seifert, "One Glitch to Rule Them All: Fault Injection Attacks Against AMD's Secure Encrypted Virtualization," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21, Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 2875–2889, ISBN: 9781450384544. DOI: 10.1145/3460120.3484779.

[40] Intel, *Intel SGX Technical Details for INTEL-SA-00289 and INTEL-SA-00334*, accessed June 5, 2020, https://cdrdv2.intel.com/v1/dl/getContent/619320.

[41] M. Lipp, A. Kogler, D. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss, "PLATYPUS: Software-based power side-channel attacks on x86," in *2021 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2021, pp. 355–371.

[42] J. Rutkowska, "Intel x86 considered harmful," *the Invisible Things Lab*, 2015. [Online]. Available: https://blog.invisiblethings.org/papers/2015/x86_harmful.pdf.

[43] F. Périgaud, A. Gazet, and J. Czarny, "Subverting your server through its BMC: the HPE iLO4 case," in *Recon Brussels '18*, 2018.

[44] *CVE-2019-6260: Gaining control of BMC from the host processor*, Jan. 2019. [Online]. Available: https://www.flamingspork.com/blog/2019/01/23/cve-2019-6260:-gaining-control-of-bmc-from-the-host-processor/ (visited on 10/31/2022).

[45] A. Matrosov, A. Ermolov, Y. Vasilenko, and S. Thomas, "Breaking Firmware Trust From Pre-EFI: Exploiting Early Boot Phases," Black Hat USA, 2022. [Online]. Available: https://www.blackhat.com/us-22/briefings/schedule/index.html#breaking-firmware-trust-from-pre-efi-exploiting-early-boot-phases-27229.

[46] Intel, "Intel® software guard extensions trusted computing base recovery," 2019. [Online]. Available: https://community.intel.com/legacyfs/online/drupal_files/managed/01/7b/Intel-SGX-Trusted-Computing-Base-Recovery.pdf.

[47] *CVE-2019-6260*. Available from MITRE, CVE-ID CVE-2019-6260. Jan. 2019. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-6260 (visited on 10/31/2022).

[48] K. Murdock, *Plundervolt*, accessed June 18, 2020, commit 3bb0295, https://github.com/KitMurdock/plundervolt.

[49] D. Lange and F. Domke, *The exhaust emissions scandal ("Dieselgate"): Take a deep breath into pollution trickery*, https://media.ccc.de/v/32c3-7331-the_exhaust_emissions_scandal_dieselgate, 2015.

[50] K. Koscher, A. Czeskis, F. Roesner, S. N. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage, "Experimental security analysis of a modern automobile," in *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, IEEE Computer Society, 2010, pp. 447–462. DOI: 10.1109/SP.2010.34.

[51] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno, "Comprehensive Experimental Analyses of Automotive Attack Surfaces," in *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*, USENIX Association, 2011. [Online]. Available: http://static.usenix.org/events/sec11/tech/full%5C_papers/Checkoway.pdf.

[52] C. Miller and C. Valasek, *Remote Exploitation of an Unaltered Passenger Vehicle*, http://illmatics.com/Remote%20Car%20Hacking.pdf, 2015.

[53] F. D. Garcia, D. F. Oswald, T. Kasper, and P. Pavlidès, "Lock it and still lose it - on the (in)security of automotive remote keyless entry systems," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, T. Holz and S. Savage, Eds., USENIX Association, Aug. 2016. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/garcia.

[54] R. Verdult, F. D. Garcia, and J. Balasch, "Gone in 360 Seconds: Hijacking with Hitag2," in *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, T. Kohno, Ed., USENIX Association, 2012, pp. 237–252. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/verdult.

[55] R. Verdult, F. D. Garcia, and B. Ege, "Dismantling Megamos Crypto: Wirelessly Lockpicking a Vehicle Immobilizer," in *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, S. T. King, Ed., USENIX Association, 2013, pp. 703–718. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/verdult.

[56] J. V. Bulck, J. T. Mühlberg, and F. Piessens, "Vulcan: Efficient component authentication and software isolation for automotive control networks," in *Proceedings of the 33rd Annual Computer Security Applications Conference, Orlando, FL, USA, December 4-8, 2017*, ACM, 2017, pp. 225–237. DOI: 10.1145/3134600.3134623.

[57] J. V. den Herrewegen and F. D. Garcia, "Beneath the bonnet: A breakdown of diagnostic security," in *Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part I*, J. López, J. Zhou, and M. Soriano, Eds., ser. Lecture Notes in Computer Science, vol. 11098, Springer, 2018, pp. 305–324. DOI: 10.1007/978-3-319-99073-6_15.

[58] A. Fasano, T. Ballo, M. Muench, T. Leek, A. Bulekov, B. Dolan-Gavitt, M. Egele, A. Francillon, L. Lu, N. Gregory, D. Balzarotti, and W. Robertson, "SoK: Enabling Security Analyses of Embedded Systems via Rehosting," in *ASIA CCS '21: ACM Asia Conference on Computer and Communications Security, Virtual Event, Hong Kong, June 7-11, 2021*, ACM, 2021, pp. 687–701. DOI: 10.1145/3433210.3453093.

[59] A. Fioraldi and D. Maier, *LibAFL*, https://github.com/AFLplusplus/LibAFL, 2021.

[60] C. Cao, L. Guan, J. Ming, and P. Liu, "Device-agnostic firmware execution is possible: A concolic execution approach for peripheral emulation," in *ACSAC*

'20: Annual Computer Security Applications Conference, Virtual Event / Austin, TX, USA, 7-11 December, 2020, ACM, 2020, pp. 746–759. DOI: 10.1145/3427228.3427280.

[61] W. Zhou, L. Guan, P. Liu, and Y. Zhang, "Automatic firmware emulation through invalidity-guided knowledge inference," in *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, M. Bailey and R. Greenstadt, Eds., USENIX Association, 2021, pp. 2007–2024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/zhou.

[62] B. Feng, A. Mera, and L. Lu, "P^{2}IM: scalable and hardware-independent firmware testing via automatic peripheral interface modeling," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds., USENIX Association, 2020, pp. 1237–1254. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/feng.

[63] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel, and G. Vigna, "Toward the analysis of embedded firmware through automated re-hosting," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2019, Chaoyang District, Beijing, China, September 23-25, 2019*, USENIX Association, 2019, pp. 135–150. [Online]. Available: https://www.usenix.org/conference/raid2019/presentation/gustafson.

[64] J. Ruge, J. Classen, F. Gringoli, and M. Hollick, "Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds., USENIX Association, 2020, pp. 19–36. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/ruge.

[65] D. Maier, L. Seidel, and S. Park, "BaseSAFE: baseband sanitized fuzzing through emulation," in *WiSec '20: 13th ACM Conference on Security and Privacy in Wire-*

less and Mobile Networks, Linz, Austria, July 8-10, 2020, R. Mayrhofer and M. Roland, Eds., ACM, 2020, pp. 122–132. DOI: 10.1145/3395351.3399360.

[66] B. Feng, A. Mera, and L. Lu, *P²IM documentation*, https://github.com/RiS3-Lab/p2im/blob/master/docs/add_mcu.md, 2020.

[67] W. Zhou, L. Guan, P. Liu, and Y. Zhang, *μEmu documentation*, https://github.com/MCUSec/uEmu/blob/main/docs/Configuration.md, 2021.

[68] M. Zalewski, *American Fuzzy Lop*, https://lcamtuf.coredump.cx/afl, 2013.

[69] National Security Agency (NSA), *Ghidra*, https://ghidra-sre.org/, Accessed: 2020/07/27.

[70] ——, *P-Code Reference Manual*, https://ghidra.re/courses/languages/html/pcoderef.html, 2017.

[71] M. Jung, S. Kim, H. Han, J. Choi, and S. K. Cha, "B2R2: Building an efficient front-end for binary analysis," in *Proceedings of the NDSS Workshop on Binary Analysis Research*, 2019.

[72] C. G. Nelson, "Techniques for Program Verification," Ph.D. dissertation, Stanford, CA, USA, 1980.

[73] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality saturation: A new approach to optimization," in *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Z. Shao and B. C. Pierce, Eds., ACM, 2009, pp. 264–276. DOI: 10.1145/1480881.1480915.

[74] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "egg: Fast and Extensible Equality Saturation," *Proc. ACM Program. Lang.*, vol. 5, no. POPL, Jan. 2021. DOI: 10.1145/3434304.

[75] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006, ISBN: 0321486811.

[76] K. Sen, "Concolic testing," in *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, R. E. K. Stirewalt, A. Egyed, and B. Fischer, Eds., ACM, 2007, pp. 571–572. DOI: 10.1145/1321631.1321746.

[77] A. Niemetz, M. Preiner, and A. Biere, "Boolector 2.0," *J. Satisf. Boolean Model. Comput.*, vol. 9, no. 1, pp. 53–58, 2014. DOI: 10.3233/sat190101.

[78] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "X-force: Force-executing binary programs for security applications," in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, K. Fu and J. Jung, Eds., USENIX Association, 2014, pp. 829–844. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/peng.

[79] P. Godefroid, "Micro execution," in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, P. Jalote, L. C. Briand, and A. van der Hoek, Eds., ACM, 2014, pp. 539–549. DOI: 10.1145/2568225.2568273.

[80] J. Wilhelm and T. Chiueh, "A forced sampled execution approach to kernel rootkit identification," in *Recent Advances in Intrusion Detection, 10th International Symposium, RAID 2007, Gold Goast, Australia, September 5-7, 2007, Proceedings*, C. Krügel, R. Lippmann, and A. J. Clark, Eds., ser. Lecture Notes in Computer Science, vol. 4637, Springer, 2007, pp. 219–235. DOI: 10.1007/978-3-540-74320-0_12.

[81] I. Gotovchits, R. Van Tonder, and D. Brumley, "Saluki: Finding taint-style vulnerabilities with static property checking," in *Proceedings of the 2018 NDSS Workshop on Binary Analysis Research (NDSS BAR 2018)*, 2018.

[82] Infineon Technologies AG, *C166S V2 16-Bit Microcontroller User Manual V1.7*, https://www.infineon.com/dgdl/c166sv2um.pdf?fileId=db3a304412b407950112b41d4ea32fe 2001.

[83] Renesas Electronics, *SH7266 Group, SH7267 Group User's Manual: Hardware*, https://datasheet.octopart.com/R5S72661P144FP%23VZ-Renesas-datasheet-11803191.pdf, 2010.

[84] M. Gaasdelen, *Lighthouse - A Coverage Explorer for Reverse Engineers*, https://github.com/gaasedelen/lighthouse, 2017.

[85] *Circumventing Fuzzing Roadblocks with Compiler Transformations*, https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/, 2016.

[86] D. Maier, B. Radtke, and B. Harren, "Unicorefuzz: On the viability of emulation for kernelspace fuzzing," in *13th USENIX Workshop on Offensive Technologies, WOOT 2019, Santa Clara, CA, USA, August 12-13, 2019*, A. Gantman and C. Maurice, Eds., USENIX Association, 2019. [Online]. Available: https://www.usenix.org/conference/woot19/presentation/maier.

[87] Airbus CyberSecurity, *Fuzzing exotic arch with AFL using ghidra emulator*, https://airbus-cyber-security.com/fuzzing-exotic-arch-with-afl-using-ghidra-emulator/, 2018.

[88] A. Esaulenko, *C166 Ghidra language definition*, https://github.com/esaulenka/Ghidra_C166, 2021.

[89] Volkswagen Research, *Linux-CAN / SocketCAN user space applications*, https://github.com/linux-can.

[90] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, The Internet Society, 2018. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018%5C_01A-4%5C_Muench%5C_paper.pdf.

[91] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, "Halucinator: Firmware re-hosting through abstraction layer emulation," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds., USENIX Association, 2020, pp. 1201–1218. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/clements.

[92] A. Mera, B. Feng, L. Lu, and E. Kirda, "DICE: Automatic Emulation of DMA Input Channels for Dynamic Firmware Analysis," in *Proceedings of the 42nd IEEE Symposium on Security and Privacy*, ser. S&P/Oakland'21, May 2021.

[93] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, The Internet Society, 2014. [Online]. Available: https://www.ndss-symposium.org/ndss2014/avatar-framework-support-dynamic-security-analysis-embedded-systems-firmwares.

[94] M. Muench, D. Nisi, A. Francillon, and D. Balzarotti, "Avatar^2: A Multi-target Orchestration Platform," in *Proceedings of the 2018 NDSS Workshop on Binary Analysis Research (NDSS BAR 2018)*, 2018.

[95] K. Koscher, T. Kohno, and D. Molnar, "SURROGATES: enabling near-real-time dynamic analyses of embedded systems," in *9th USENIX Workshop on Offensive Technologies, WOOT '15, Washington, DC, USA, August 10-11, 2015*, A.

Francillon and T. Ptacek, Eds., USENIX Association, 2015. [Online]. Available: https://www.usenix.org/conference/woot15/workshop-program/presentation/koscher.

[96] N. Corteggiani, G. Camurati, and A. Francillon, "Inception: System-Wide Security Testing of Real-World Embedded Systems Software," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, W. Enck and A. P. Felt, Eds., USENIX Association, 2018, pp. 309–326. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/corteggiani.

[97] N. Corteggiani and A. Francillon, "HardSnap: Leveraging Hardware Snapshotting for Embedded Systems Security Testing," in *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020, Valencia, Spain, June 29 - July 2, 2020*, IEEE, 2020, pp. 294–305. DOI: 10.1109/DSN48063.2020.00046.

[98] L. Harrison, H. Vijayakumar, R. Padhye, K. Sen, and M. Grace, "PARTEMU: enabling dynamic analysis of real-world trustzone software using emulation," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds., USENIX Association, 2020, pp. 789–806. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/harrison.

[99] A. A. Clements, L. Carpenter, W. A. Moeglein, and C. Wright, "Is Your Firmware Real or Re-Hosted? A case study in re-hosting VxWorks control system firmware," in *Proceedings of the 2021 NDSS Workshop on Binary Analysis Research (NDSS BAR 2021)*, 2021.

[100] Q. Liu, C. Zhang, L. Ma, M. Jiang, Y. Zhou, L. Wu, W. Shen, X. Luo, Y. Liu, and K. Ren, "FirmGuide: Boosting the Capability of Rehosting Embedded Linux Kernels through Model-Guided Kernel Execution," in *2021 36th IEEE/ACM In-*

ternational Conference on Automated Software Engineering (ASE), 2021, pp. 792–804. DOI: 10.1109/ASE51524.2021.9678653.

[101] E. Johnson, M. Bland, Y. Zhu, J. Mason, S. Checkoway, S. Savage, and K. Levchenko, "Jetset: Targeted firmware rehosting for embedded systems," in *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, M. Bailey and R. Greenstadt, Eds., USENIX Association, 2021, pp. 321–338. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/johnson.

[102] G. Hernandez, M. Muench, T. Tucker, H. Searle, W. Zhu, P. Traynor, and K. Butler, "Emulating Samsung's Baseband for Security Testing," *Black Hat USA*, 2020.

[103] G. Hernandez, M. Muench, D. Maier, A. Milburn, S. Park, T. Scharnowski, T. Tucker, P. Traynor, and K. R. B. Butler, "FirmWire: Transparent Dynamic Analysis for Cellular Baseband Firmware," in *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, February 27 - March 3, 2022*, The Internet Society, 2022.

[104] A. Milburn, N. Timmers, N. Wiersma, and R. Pareja, *There Will Be Glitches: Extracting and Analyzing Automotive Firmware Efficiently*, https://www.riscure.com/uploads/2018/11/Riscure_Whitepaper_Analyzing_Automotive_Firmware.pdf, 2018.

[105] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution," in *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, S. T. King, Ed., USENIX Association, 2013, pp. 463–478. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/davidson.

[106] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan, "Repeatable reverse engineering with PANDA," in *Proceedings of the 5th Program Protection and Reverse Engineering Workshop, PPREW@ACSAC, Los Angeles, CA, USA, December 8, 2015*, J. T. McDonald, M. D. Preda, and N. Stakhanova, Eds., ACM, 2015, 4:1–4:11. DOI: 10.1145/2843859.2843867.

[107] M. Muench, A. Francillon, and D. Balzarotti, "Avatar^2: A multi-target orchestration platform," in *BAR 2018, Workshop on Binary Analysis Research, colocated with NDSS Symposium, 18 February 2018, San Diego, USA*, San Diego, 2018.

[108] B. Feng, A. Mera, and L. Lu, *P^{2}IM documentation*, http://archive.is/ydJ6x, Accessed: 2020/07/30.

[109] Hex-Rays, *IDA Pro*, https://www.hex-rays.com/products/ida/, Accessed: 2020/07/27.

[110] L. Cojocar, T. Kroes, and H. Bos, "JTR: A binary solution for switch-case recovery," in *Engineering Secure Software and Systems - 9th International Symposium, ESSoS 2017, Bonn, Germany, July 3-5, 2017, Proceedings*, E. Bodden, M. Payer, and E. Athanasopoulos, Eds., ser. Lecture Notes in Computer Science, vol. 10379, Springer, 2017, pp. 177–195. DOI: 10.1007/978-3-319-62105-0_12.

[111] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite: Statically instrumenting COTS binaries for fuzzing and sanitization," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, IEEE, 2020, pp. 1497–1511. DOI: 10.1109/SP40000.2020.00009.

[112] J. Friebertshäuser, *Polypyus – The Firmware Historian*, https://github.com/seemoo-lab/polypyus/, Accessed: 2020/07/25.

[113] ARM, *CoreSight Program Flow Trace*, http://infocenter.arm.com/help/topic/com.arm.doc.ihi0035b/IHI0035B_cs_pft_v1_1_architecture_spec.pdf, Accessed: 2020/06/29.

[114] ——, *Embedded Trace Buffer Technical Reference Manual*, https://developer.arm.com/documentation/ddi0242/b/, Accessed: 2020/11/05.

[115] J. H. Ward, "Hierarchical Grouping to Optimize an Objective Function," *Journal of the American Statistical Association*, vol. 58, no. 301, pp. 236–244, 1963. [Online]. Available: http://www.jstor.org/stable/2282967.

[116] Renesas, *CubeSuite+ (CS+)*, https://www.renesas.com/us/en/products/software-tools/tools/ide/csplus.html, Accessed: 2020/07/31.

[117] L. Wouters, J. Van den Herrewegen, F. D. Garcia, D. Oswald, B. Gierlichs, and B. Preneel, "Dismantling DST80-based Immobiliser Systems," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, no. 2, pp. 99–127, Mar. 2020. DOI: 10.13154/tches.v2020.i2.99-127.

[118] D. Selianin, *Researching Marvell Avastar Wi-Fi: From zero Knowledge to Over-the-air zero-touch RCE*, https://2018.zeronights.ru/wp-content/uploads/materials/19-Researching-Marvell-Avastar-Wi-Fi.pdf, Accessed: 2020/07/30, 2018.

[119] P. Sun, L. Garcia, and S. A. Zonouz, "Tell Me More Than Just Assembly! Reversing Cyber-Physical Execution Semantics of Embedded IoT Controller Software Binaries," in *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019*, IEEE, 2019, pp. 349–361. DOI: 10.1109/DSN.2019.00045.

[120] P. Srivastava, H. Peng, J. Li, H. Okhravi, H. E. Shrobe, and M. Payer, "FirmFuzz: Automated IoT Firmware Introspection and Analysis," in *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things, IoT S&P@CCS 2019, London, UK, November 15, 2019*, P. Liu and Y. Zhang, Eds., ACM, 2019, pp. 15–21. DOI: 10.1145/3338507.3358616.

[121] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation," in *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, N. Heninger and P. Traynor, Eds., USENIX Association, 2019, pp. 1099–1114. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/zheng.

[122] SEGGER, *SystemView*, https://www.segger.com/products/development-tools/systemview/, Accessed: 2020/07/25.

[123] Percepio, *Tracealyzer*, https://percepio.com/tracealyzer/, Accessed: 2020/07/25.

[124] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A large-scale analysis of the security of embedded firmwares," in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, K. Fu and J. Jung, Eds., USENIX Association, 2014, pp. 95–110. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/costin.

[125] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware," in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, The Internet Society, 2015. [Online]. Available: https://www.ndss-symposium.org/ndss2015/firmalice-automatic-detection-authentication-bypass-vulnerabilities-binary-firmware.

[126] L. Cojocar, J. Zaddach, R. Verdult, H. Bos, A. Francillon, and D. Balzarotti, "PIE: parser identification in embedded systems," in *Proceedings of the 31st Annual Computer Security Applications Conference, Los Angeles, CA, USA, December 7-11, 2015*, ACM, 2015, pp. 251–260. DOI: 10.1145/2818000.2818035.

[127] S. L. Thomas, T. Chothia, and F. D. Garcia, "Stringer: Measuring the importance of static data comparisons to detect backdoors and undocumented functionality,"

in *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II*, S. N. Foley, D. Gollmann, and E. Snekkenes, Eds., ser. Lecture Notes in Computer Science, vol. 10493, Springer, 2017, pp. 513–531. DOI: 10.1007/978-3-319-66399-9_28.

[128] S. L. Thomas, F. D. Garcia, and T. Chothia, "Humidify: A tool for hidden functionality detection in firmware," in *Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings*, M. Polychronakis and M. Meier, Eds., ser. Lecture Notes in Computer Science, vol. 10327, Springer, 2017, pp. 279–300. DOI: 10.1007/978-3-319-60876-1_13.

[129] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Krügel, and G. Vigna, "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, 2016, pp. 138–157. DOI: 10.1109/SP.2016.17.

[130] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos, "An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, T. Holz and S. Savage, Eds., USENIX Association, 2016, pp. 583–600. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/andriesse.

[131] D. Andriesse, A. Slowinska, and H. Bos, "Compiler-agnostic function detection in binaries," in *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*, IEEE, 2017, pp. 177–189. DOI: 10.1109/EuroSP.2017.11.

[132] M. Jiang, Y. Zhou, X. Luo, R. Wang, Y. Liu, and K. Ren, "An empirical study on ARM disassembly tools," in *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, S. Khurshid and C. S. Pasareanu, Eds., ACM, 2020, pp. 401–414. DOI: 10.1145/3395363.3397377.

[133] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure Linux Containers with Intel SGX," in *Usenix OSDI '16*, Savannah, GA: USENIX Association, Nov. 2016, pp. 689–703, ISBN: 978-1-931971-33-1. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov.

[134] A. Baumann, M. Peinado, and G. Hunt, "Shielding Applications from an Untrusted Cloud with Haven," in *Usenix OSDI '14*, Broomfield, CO: USENIX Association, Oct. 2014, pp. 267–283, ISBN: 978-1-931971-16-4. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/baumann.

[135] R. Kunkel, D. L. Quoc, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer, *TensorSCONE: A Secure TensorFlow Framework using Intel SGX*, arXiv 1902.04413, 2019.

[136] C.-c. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX," in *USENIX ATC '17*, Santa Clara, CA: USENIX Association, Jul. 2017, pp. 645–658, ISBN: 978-1-931971-38-6. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai.

[137] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip Feng Shui: Hammering a Needle in the Software Stack," in *USENIX Security '16*, Austin: USENIX Association, Aug. 2016, pp. 1–18, ISBN: 978-1-931971-32-4.

[Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/razavi.

[138] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom, "Another Flip in the Wall of Rowhammer Defenses," in *S&P '18*, 2018, pp. 245–261.

[139] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "RAMBleed: Reading Bits in Memory Without Accessing Them," in *S&P '20*, 2020.

[140] P. Frigo, E. Vannacci, H. Hassan, V. van der Veen, O. Mutlu, C. Giuffrida, H. Bos, and K. Razavi, "TRRespass: Exploiting the Many Sides of Target Row Refresh," in *S&P*, May 2020. [Online]. Available: Paper=https://download.vusec.net/papers/trrespass_sp20.pdf%20Slides=https://download.vusec.net/slides/trrespass_sp20.pdf%20Web=https://www.vusec.net/projects/trrespass%20Code=https://github.com/vusec/trrespass.

[141] A. Cui and R. Housley, "BADFET: Defeating Modern Secure Boot Using Second-Order Pulsed Electromagnetic Fault Injection," in *Usenix WOOT '17*, Vancouver, BC: USENIX Association, Aug. 2017. [Online]. Available: https://www.usenix.org/conference/woot17/workshop-program/presentation/cui.

[142] T. Roth, *TrustZone-M(eh): Breaking ARMv8-M's security—Hardware attacks on the latest generation of ARM Cortex-M processors.* presentation at 36C3, 2019.

[143] D. Lee, D. Jung, I. T. Fang, C.-c. Tsai, and R. A. Popa, "An Off-Chip Attack on Hardware Enclaves via the Memory Bus," in *USENIX Security '20*, Boston: USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/lee-dayeol.

[144] Microsoft, *Azure confidential computing*, 2020.

[145] ——, *Azure confidential computing*, version of Dec 6, 2019 retrieved from archive.org, https://web.archive.org/web/20191206233429/https://azure.microsoft.com/en-gb/solutions/confidential-compute/.

[146] Fortanix, *Intel SGX FAQ*, accessed June 2, 2020, https://fortanix.com/intel-sgx/.

[147] Enarx, *Threat model*, accessed June 17, 2020, revision 678e2c2, https://github.com/enarx/enarx/wiki/Threat-Model.

[148] Synaptics, *Lenovo, Intel, PayPal and Synaptics announce collaboration to bring FIDO authentication to laptops*, accessed June 4, 2020.

[149] Fortanix, *intel-oc-mbox*, accessed September 21, 2020, https://github.com/fortanix/intel-oc-mbox/tree/jb/initial.

[150] Github, *intel-undervolt issue 43*, accessed June 18, 2020, https://github.com/kitsunyan/intel-undervolt/issues/43#issuecomment-619373836.

[151] 4th Generation i7 Datasheet: https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/4th-gen-core-i7-lga2011-datasheet-vol-1.pdf. (visited on 09/10/2022).

[152] Infineon, *IR35204 3+1 Dual Output Digital Multi-Phase Controller datasheet*, 2016.

[153] Rigol, *DS1074Z 70MHz Digital Oscilloscope*.

[154] DreamSource Technology Co., Ltd., *DSLogic Plus*, accessed June 2, 2020.

[155] ZEROPLUS, *Protocol Analyzer SVID 1.04.00*, http://www.zeroplus.com.tw/logic-analyzer_en/news_detail.php?news_id=1755, 2014.

[156] System Management Interface Forum, Inc, *PMBus™ Power System Management Protocol Specification, Part I, Revision 1.2X*, accessed June 9, 2020, Sep. 2010.

[157] ——, *System Management Bus (SMBus) Specification Version 3.0*, accessed June 9, 2020, http://smbus.org/specs/SMBus_3_0_20141220.pdf.

[158] PJRC, *Teensy 4 development board*, accessed June 2, 2020, https://www.pjrc.com/store/teensy40.html.

[159] Texas Instruments, *SN74LVC1G07 Single Buffer/Driver With Open-Drain Output*, 2016.

[160] Arduino, *Arduino IDE*, https://www.arduino.cc/en/Main/Software.

[161] PJRC, *Teensyduino arduino library*, https://www.pjrc.com/teensy/td_download.html.

[162] Microsoft, *Open Enclave SDK*, accessed September 23, 2020, https://github.com/openenclave/openenclave.

[163] M. Tunstall, D. Mukhopadhyay, and S. Ali, "Differential Fault Analysis of the Advanced Encryption Standard Using a Single Fault," in *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication*, C. A. Ardagna and J. Zhou, Eds., Springer, 2011, pp. 224–233, ISBN: 978-3-642-21040-2.

[164] Y. Lu, "Attacking hardware AES with DFA," *arXiv preprint arXiv:1902.08693*, 2019.

[165] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang, "Hacking in darkness: Return-oriented programming against secure enclaves," in *USENIX Security '17*, 2017, pp. 523–539.

[166] A. T. Markettos, C. Rothwell, B. F. Gutstein, A. Pearce, P. G. Neumann, S. W. Moore, and R. N. M. Watson, "Thunderclap: Exploring Vulnerabilities in Operating System IOMMU Protection via DMA from Untrustworthy Peripherals," in *NDSS '19*, San Diego, CA, USA, 2019.

[167] Renesas, *ISL95712 Datasheet*, available at https://www.renesas.com/eu/en/www/doc/datasheet/isl95712.pdf, 2015.

[168] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 4: Model-Specific Registers*, May 2019.

[169] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, and Z. Ou, "RAPL in Action: Experiences in Using RAPL for Power Measurements," *ToMPECS*, 2018.

[170] D. Karaklajić, J.-M. Schmidt, and I. Verbauwhede, "Hardware designer's guide to fault attacks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 12, pp. 2295–2306, 2013.

[171] W. Rankl and W. Effing, *Smart Card Handbook*, 4th. Wiley, 2010, ISBN: 0470743670.

[172] Infineon, *Integrity guard*, online, accessed 2020-04-05: https://www.infineon.com/dgdl/Infineon-Integrity_Guard_The_smartest_digital_security_technology_in_the_industry_06.18-WP-v01_01-EN.pdf?fileId=5546d46255dd933d0155e31c46fa03fb, 2018.

[173] E. Burton, G. Schrom, F. Paillet, J. Douglas, W. Lambert, K. Radhakrishnan, and M. Hill, "FIVR — Fully integrated voltage regulators on 4th generation Intel Core SoCs," in *IEEE APEC '14*, Mar. 2014, pp. 432–439, ISBN: 978-1-4799-2325-0. DOI: 10.1109/APEC.2014.6803344.

[174] J. Maebe, R. D. Keulenaer, B. D. Sutter, and K. D. Bosschere, "Mitigating Smart Card Fault Injection with Link-Time Code Rewriting: A Feasibility Study," in *Financial Cryptography*, 2013.

[175] T. Barry, D. Couroussé, and B. Robisson, "Compilation of a countermeasure against instruction-skip fault attacks," in *CS2 '16*, ACM, 2016, pp. 1–6. DOI: 10.1145/2858930.2858931.

[176] A. Tang, S. Sethumadhavan, and S. Stolfo, "CLKSCREW: Exposing the perils of security-oblivious energy management," in *USENIX Security '17*, Vancouver, BC: USENIX Association, Aug. 2017, pp. 1057–1074, ISBN: 978-1-931971-40-9.

[Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang.

[177] *PMBus Power System Management Protocol Specification, Part II – Command Language*, https://470q2hhkn9g15l4bc2btbal1-wpengine.netdna-ssl.com/wp-content/uploads/2022/01/PMBus-Specification-Rev-1-3-1-Part-II-20150313.pdf.

[178] Eclypsium, *Insecure Firmware Updates in Server Management Systems*, https://eclypsium.com/2018/09/06/insecure-firmware-updates-in-server-management-systems/, Sep. 2018. (visited on 10/31/2022).

[179] B. Rak, *Github repo: ipmi_firmware_tools*, https://github.com/devicenull/ipmi_firmware_tools, 2015. (visited on 09/15/2022).

[180] M. Niewohner, *Supermicro BMC firmware image decryptor*, https://github.com/c0d3z3r0/smcbmc, 2020. (visited on 09/08/2022).

[181] J. Robertson and M. Riley, *The Big Hack: How China Used a Tiny Chip to Infiltrate U.S. Companies*, https://www.bloomberg.com/news/features/2018-10-04/the-big-hack-how-china-used-a-tiny-chip-to-infiltrate-america-s-top-companies#xj4y7vzkg, Oct. 2018. (visited on 09/19/2022).

[182] N. Waisman and M. S. Soler, "The Unbearable Lightness of BMC's," in *BlackHat '18*, 2018.

[183] Monolithic Power Systems, Inc., *MP2965 Datasheet*, https://www.monolithicpower.com/en/mp2965.html.

[184] Intel Corporation, *Intel Xeon Processor E3-1200 v6 Product Family for S Platforms*, https://www.intel.co.uk/content/dam/www/public/us/en/documents/datasheets/xeon-e3-1200v6-vol-1-datasheet.pdf, Jan. 2018. (visited on 09/16/2022).

[185] I. Super Micro Computer, *X11SSL-CF X11SSL-nF USER MANUAL Revision 1.1*, https://www.supermicro.com/manuals/motherboard/C232/MNL-1782.pdf. (visited on 09/10/2022).

[186] ——, *X11SSL-CF(-nF) Quick Reference Guide*, https://www.supermicro.com/QuickRefs/motherboard/C232/QRG-1782.pdf. (visited on 09/13/2022).

[187] J. Vazquez, *Exploiting the Supermicro Onboard IPMI Controller*, Nov. 2013. [Online]. Available: https://www.rapid7.com/blog/post/2013/11/15/exploiting-the-supermicro-onboard-ipmi-controller/ (visited on 09/12/2022).

[188] *Linux device tree file: aspeed-g4.dtsi*, https://github.com/torvalds/linux/blob/133d9c53c9dcbb1b8f317e402e79c44d9eb725c9/arch/arm/boot/dts/aspeed-g4.dtsi#L438.

[189] *Aspeed 24XX/25XX I2C Controller Linux Kernel 5.16 Driver*, https://elixir.bootlin.com/linux/latest/source/drivers/i2c/busses/i2c-aspeed.c. (visited on 09/16/2022).

[190] https://drunkencat.net/misc/SupermicroBIOS.html. (visited on 11/18/2022).

[191] Intel Corporation, *Cryptography for Intel Integrated Performance Primitives Developer Reference—RSA Primitives*, https://www.intel.com/content/www/us/en/develop/documentation/ipp-crypto-reference/top/public-key-cryptography-functions/rsa-algorithm-functions/rsa-primitives.html. (visited on 01/05/2023).

[192] D. Nemiroff and C. Tokunaga, "Whitepaper: Fault Injection Countermeasures, Deployed at Scale," Tech. Rep., 2022. [Online]. Available: %5Curl%7Bhttps://www.intel.co.uk/content/www/uk/en/architecture-and-technology/hardware-shield/fault-injection-countermeasures-white-paper.html%7D.

[193] J. Van den Herrewegen, "Automotive firmware extraction and analysis techniques," Ph.D. dissertation, 2021.

[194] ARM, *CoreSight Trace Memory Controller*, http://infocenter.arm.com/help/topic/com.arm.doc.ddi0461b/DDI0461B_tmc_r0p1_trm.pdf, Accessed: 2020/06/29.