DISS. ETH NO. 30823

Trustworthy Platform Management

A thesis submitted to attain the degree of

Doctor of Sciences
(Dr. sc. ETH Zurich)

presented by

Daniel David Schwyn

born on 08.05.1990

accepted on the recommendation of

Prof. Dr. Timothy Roscoe
Prof. Dr. Donald E. Porter
Prof. Dr. Gustavo Alonso

2025

A thesis submitted to ETH Zurich to attain the degree of Doctor of Sciences.

Examiner:
Prof. Dr. Timothy Roscoe

Co-examiners:
Prof. Dr. Donald E. Porter, Prof. Dr. Gustavo Alonso

Examination date:
9 December 2024

Trustworthy Platform Management.

DISS. ETH NO: 30823

DOI: 10.3929/ethz-b-000715675

Abstract

It goes without saying that modern computer systems are enormously complex. There are however less well-known aspects to this complexity that are never visible to the operating system or firmware running on the main processors. Modern server boards include systems called Baseboard Management Controllers (BMCs) that manage this complexity. They orchestrate power and clock delivery to turn the machine on and off, manage firmware for other components on the board and offer remote management capabilities. However, despite their critical role in the safety and security of our modern compute infrastructure, BMCs are understudied in the academic community. This is in part due to the fact, that until recently these systems were proprietary and closed-source. Open-source systems have started to appear, but transparency alone does not solve the fundamental problem with BMCs: the lack of high-assurance engineering techniques is at odds with the criticality of these systems. The analysis presented in this dissertation shows, that there is a steady stream of newly discovered vulnerabilities affecting BMCs. Driven by the experience of building a BMC for Enzian, a heterogeneous server platform for systems research, this dissertation makes contributions to improving the state of the art for BMCs, both in terms of safety and security.

A BMC needs to manage vital aspects of a computing platform like power and clock delivery and firmware provisioning. Without correct implementations of these functions, the hardware cannot be operated safely. In this dissertation I propose an approach to creating such implementations using formal hardware models. I demonstrate the approach using power and clock delivery as an example. A power and clock manager has to correctly configure the network of regulators that supply power and clocks to the various components. Misconfiguring this network can cause permanent hardware damage. It is therefore arguably the most critical BMC function. I present a model that describes these networks declaratively. From this, a tool then generates correct configurations for the power and clock delivery topology, including a sequence of steps to apply these configurations. I show that full power sequences for a real two-socket

server can be generated in a matter of seconds. The generated sequences are used productively in Enzian's power management stack.

The configuration steps then need to be communicated to the regulators over bus-based chip-to-chip protocols such as Inter-Integrated Circuit (I^2C). These protocols are notorious for their interoperability issues between different devices attached to the buses. This dissertation addresses these issues with a framework for generating provably correct I^2C communication stacks. The framework can generate both software and hardware components from model-checked specifications. The resulting I^2C stacks have comparable performance with handwritten solutions in off-the-shelf systems.

Unfortunately, correct implementations of BMC functions are not enough: the analysis of BMC vulnerabilities presented in this dissertation shows that a majority of these vulnerabilities are privilege-escalation bugs. I address the security aspect of building a BMC by proposing a system design based on seL4, a provably correct microkernel. The design addresses the need for isolation between critical and untrusted BMC components. Furthermore, the design offers an incremental retrofit strategy for existing BMC systems.

In summary, this dissertation not only identifies the problems with existing BMC systems but also contributes several concrete solutions towards a future of trustworthy platform management stacks.

Zusammenfassung

Es ist allgemein anerkannt, dass moderne Computersysteme enorm komplex sind. Es gibt allerdings weniger bekannte Aspekte dieser Komplexität, die weder für das Betriebssystem noch für die Firmware auf den Hauptprozessoren sichtbar sind. Auf modernen Serverplatinen wird diese Komplexität von Board Management Controllern (BMCs) verwaltet. Sie orchestrieren die Strom- und Taktzufuhr um die Maschine ein- und auszuschalten, verwalten Firmware für Platinenkomponenten und bieten Fernwartungszugriff. Trotz ihrer zentralen Rolle für die Sicherheit unserer modernen Recheninfrastruktur sind BMCs kaum Gegenstand akademischer Forschung. Dies ist teilweise der Tatsache geschuldet, dass diese Systeme bis vor Kurzem proprietär und nicht quelloffen waren. Unterdessen gibt es quelloffene Systeme, aber Transparenz alleine löst das fundamentale Problem mit BMCs nicht: Die Tatsache, dass bei BMCs kaum Methoden für hochkritische Software zur Anwendung kommen, verträgt sich nicht mit der zentralen Rolle dieser Systeme. Die Analyse, die ich in dieser Dissertation präsentiere, zeigt, dass fortlaufend neue Sicherheitslücken in BMCs entdeckt werden. Basierend auf der Erfahrung mit dem Bau eines BMCs für Enzian, einer heterogenen Serverplattform für die Computersystemforschung, leistet diese Dissertation einen Beitrag zur Verbesserung der Sicherheit von BMCs.

Ein BMC verwaltet grundlegende Aspekte von Computerplattformen wie Strom- und Taktversorgung und die Bereitstellung von Firmware. Die Hardware kann ohne korrekte Implementationen dieser Funktionen nicht sicher betrieben werden. In dieser Dissertation schlage ich einen Ansatz vor, der solche korrekte Implementationen durch den Einsatz von formalen Hardwaremodellen ermöglicht. Ich demonstriere den Ansatz mit Strom- und Taktzufuhr als Beispiel. Die Strom- und Taktverwaltung muss das Netzwerk der Regler, die die verschiedenen Komponenten versorgen, richtig konfigurieren. Fehlkonfigurationen können zu bleibenden Hardwareschäden führen. Diese Funktion ist deshalb wohl die wichtigste eines BMCs. Ich präsentiere ein Modell, das diese Netzwerke deklarativ beschreibt. Ein Generator kann daraus korrekte Konfigurationen für die Strom-

und Takttopologie erzeugen, inklusive einer Sequenz von Konfigurationsschritten. Ich zeige, dass ganze Ein- und Ausschaltsequenzen für einen realen Server mit zwei Sockeln innert weniger Sekunden generiert werden können. Die erzeugten Sequenzen sind Teil der produktiven Stromüberwachungssoftware für Enzian.

Die Konfigurationsschritte müssen via busbasierte Protokolle wie Inter-Integrated Circuit (I^2C) an die Regler übertragen werden. Diese Protokolle sind bekannt für ihre Interoperabilitätsprobleme zwischen verschiedenen angeschlossenen Geräten. In dieser Dissertation werden diese Probleme mit einem Framework behoben, das beweisbar korrekte I^2C-Kommunikationsstapel generieren kann. Das Framework kann aus modellgeprüften Spezifikationen sowohl Software- als auch Hardwarekomponenten erzeugen. Die Leistungsfähigkeit der generierten I^2C-Kommunikationsstapel ist vergleichbar mit handgeschriebenen Lösungen in handelsüblichen Systemen.

Leider reichen korrekte Implementationen von BMC-Funktionen nicht aus: Die Analyse von BMC-Sicherheitslücken, die ich in dieser Dissertation präsentiere, zeigt, dass die Mehrheit dieser Sicherheitslücken Rechteausweitungsfehler sind. Ich befasse mich mit dem Sicherheitsaspekt von BMCs, indem ich ein Systemdesign vorschlage, das auf seL4 basiert, einem beweisbar korrekten Mikrokernel. Das Design bietet die notwendige Isolation zwischen kritischen und nicht vertrauenswürdigen BMC-Komponenten. Zusätzlich bietet das Design eine inkrementelle Nachrüstungsstrategie für bestehende BMC-Systeme.

Zusammengefasst zeigt diese Dissertation nicht nur die Probleme mit bestehenden BMC-Systemen auf, sondern präsentiert auch mehrere konkrete Lösungen für eine Zukunft mit vertrauenswürdigen Plattformverwaltungsstapeln.

Acknowledgements

This dissertation would not be what it is without the involvement of many people. More importantly, my *doctorate* would not have been the same.

First, I would like to thank my advisor, Mothy Roscoe, for giving me the opportunity to do a doctorate in the Systems Group at ETH Zurich and convincing me to pursue it. Thank you for your support and advice and for sharing your deep insights into computer systems.

I would also like to thank Gustavo Alonso and Don Porter for agreeing to be on my committee. Your valuable feedback helped me to improve this dissertation.

A big thank you goes to everyone in the Enzian Team. There are not many research groups in the world crazy enough to build a computer like Enzian. I am grateful and proud to have done my doctorate in one that is. Enzian taught me things about computers I never knew I wanted to know and that I could not have learned any other way. Special thanks go to David Cock without whom I doubt Enzian would exist.

Further, I would like to thank Michael Giardino for being ever optimistic and for encouraging me when I sometimes was not convinced that my research was worth pursuing. This was especially true during the pandemic when it was way too easy to lose sight of the goal while sitting alone in front of a screen at home.

Thank you as well to all my other friends and colleagues in the Systems Group: Abishek, Adam, Ana, Anastasiia, Andrea, Ben, Benjamin, Dario, Fabio, Foteini, Hidde, Jasmin, Lazar, Lukas, Max, Maxi, Melissa, Michael, Michal, Monica, Nicolas, Nora, Pengcheng, Reto, Roni, Roman, Sam, Tom, Vasilis, Yazhuo, Zhenhao and Zikai. It was a privilege working with you all. Your insights and perspective improved my work and made me a better researcher. A huge thank you also to the Systems Group's amazing admin team: Jena, Macy, Nadia, Natasha, and Simonetta. Thank your for always finding answers to any administrative questions, and for knowing exactly when that chocolate was needed the most. The Systems Group is a special place because of its members. Thank you for six years without a single boring lunch break.

I also had the pleasure to work with a number of brilliant students during my doctorate: Alessandro, Axel, Aya, Cedric, Constantin, Dennis, Edem, Georg, Jan Nino, Julian, Linus, Manuel, Moritz, Sarah, Thomas, and Tobias. Not only did your work contribute to the research in this dissertation, but working with you probably taught me more about research than I taught you. Thank you all.

To all my friends – many of whom knew I was going to do a doctorate before I did: Thank you for distracting me when things got stressful, and sorry for all the rants you had to listen to.

To my family: thank you for your support and for always believing in me. To my parents: thank you for instilling me with the curiosity and fascination for science that led me to do a doctorate.

And finally to Kim: Thank you for keeping up with me in the stressful times. Thank you for always knowing that I could do it, especially in the times when I thought I could not.

Zürich, February 2025

Contents

Abstract iii

Zusammenfassung v

Acknowledgements vii

1 Introduction 1
- 1.1 Structure of the dissertation . . . 3
- 1.2 Notes on collaborative work . . . 4

2 Case Study: The Enzian BMC 7
- 2.1 Existing BMC systems . . . 10
- 2.2 Adapting OpenBMC . . . 11
  - 2.2.1 Prototyping the power sequencer . . . 11
  - 2.2.2 Turning Enzian on . . . 15
- 2.3 Conclusion . . . 15

3 Critique of the State of the Art 17
- 3.1 BMC vulnerability analysis . . . 18
  - 3.1.1 Methodology . . . 18
  - 3.1.2 Vulnerabilities over time . . . 18
  - 3.1.3 A taxonomy of vulnerabilities . . . 19
  - 3.1.4 A closer look at non-critical vulnerabilities . . . 22
- 3.2 Conclusion . . . 24

4 Declarative Power Sequencing 25
- 4.1 Background . . . 27
- 4.2 Experience . . . 29
- 4.3 Model . . . 31
- 4.4 Algorithms . . . 36
  - 4.4.1 Computing the platform state . . . 36

4.4.2 Computing the sequence . . . 38
4.4.3 Full power-up sequence . . . 38
4.5 Evaluation . . . 40
4.5.1 Generating working power sequences . . . 40
4.5.2 Efficient state generation . . . 42
4.5.3 Efficient Sequence Generation . . . 45
4.5.4 Re-computing sequences for new revisions . . . 47
4.5.5 Adapting the tool . . . 47
4.6 Related Work . . . 48
4.7 Conclusion . . . 50

5 Dynamic Power Management **53**
5.1 Changes to the model . . . 54
5.2 New algorithms . . . 56
5.2.1 Computing the platform state . . . 57
5.3 Evaluation . . . 58
5.4 Conclusion . . . 60
5.4.1 An alternative to online sequence generation . . . 60

6 A Trustworthy I^{2}C Stack **63**
6.1 Background and problem statement . . . 65
6.1.1 The importance of I^{2}C and related protocols . . . 65
6.1.2 What makes I^{2}C different? . . . 66
6.1.3 The I^{2}C protocol stack and ecosystem . . . 68
6.2 Efeu design and implementation . . . 70
6.2.1 Specifying the driver stack . . . 70
6.2.2 Efeu compiler overview . . . 73
6.2.3 C Backend . . . 74
6.2.4 Verilog backend . . . 76
6.2.5 Generating hybrid hardware/software drivers . . . 77
6.2.6 Promela backend . . . 80
6.3 Verification . . . 81
6.3.1 Approach . . . 81
6.3.2 Verification Code Size . . . 83
6.3.3 Verification Runtime . . . 83

6.3.4 Scalability . . . 85
6.3.5 Non-Standard Devices . . . 85
6.4 Evaluation on real hardware . . . 87
6.4.1 Source code size . . . 88
6.4.2 Achievable Bus Speeds . . . 89
6.4.3 CPU Usage . . . 91
6.4.4 FPGA resource utilization . . . 94
6.4.5 Discussion . . . 96
6.5 Related Work . . . 97
6.5.1 Hardware/software co-design . . . 97
6.5.2 Driver synthesis . . . 98
6.5.3 Driver verification . . . 98
6.6 Conclusion . . . 99

7 System Design **101**
7.1 Providing isolation . . . 102
7.1.1 Physical separation . . . 102
7.1.2 Software isolation . . . 105
7.1.3 Summary . . . 108
7.2 A trustworthy BMC design . . . 108
7.2.1 BMC cyber retrofit . . . 110
7.2.2 Creating trusted BMC components . . . 111
7.2.3 Trusted BMC hardware components . . . 112
7.3 Preventing vulnerabilities by design . . . 113
7.3.1 Preventing vulnerabilities in critical components . . . 113
7.3.2 Preventing privilege escalation vulnerabilities . . . 113
7.3.3 Containing non-critical vulnerabilities . . . 114
7.4 Conclusion . . . 114

8 Future Work **115**
8.1 Cyber-retrofitting BMCs . . . 115
8.1.1 More trustworthy components . . . 116
8.1.2 Component interfaces . . . 116
8.1.3 BMC interfaces . . . 116

8.2 Hardware topology and schematics 117
8.2.1 Extracting topology information 117
8.2.2 Generating netlists from specifications 118
8.3 Opening BMCs for research 119
8.4 BMCs and the *de-facto* OS 121

9 Conclusion **123**

1

Introduction

Modern servers are enormously complex systems. Many parts on them are outside the control of what is traditionally seen as the OS [65]. This starts with the software that runs before the OS even boots. It is the norm for a modern CPU to run two or more stages of firmware before loading the OS kernel: the first stages initialize the hardware and the later stages provide facilities to load the OS kernel. The standard for Arm CPUs is to first load Arm Trusted Firmware (ATF), which consists of two to three stages. ATF then loads UEFI [12]. On AMD CPUs it is an initializer built with AMD Generic Encapsulated Software Architecture (AGESA) [2] followed by UEFI [2]. Firmware often also stays resident to offer services to the OS like turning cores on and off [193, 13]. This is just (a small part of) the complexity on the *application* processor. Much less well known however, is the complexity further below. A modern server is not just a single CPU. It is a collection of multiple CPU sockets and peripherals and with the trend to more heterogeneous computing also specialized processors like GPUs and more recently FPGAs. All these components need power, often multiple inputs with different voltages [26]. The result is a complex network of regulators that transform the output from the main power supply to the various voltages needed by these consumers. The need for energy efficiency also requires individual power domains to be turned on and off dynamically [15].

The complexity however, goes beyond power distribution: each of these components runs their own firmware and software that needs to be provisioned and serviced *remotely*. Together with monitoring requirements this has led to modern servers containing an embedded system, a computer in the computer,

that manages all this complexity. These embedded systems are called Baseboard Management Controllers (BMCs).

This is not just the case for servers, however. Increasingly, embedded systems and systems-on-chip (SoCs) contain their own service processors. The i.MX 8X family of SoCs by NXP contains a microcontroller it calls the "System control unit" and the AMD Zynq MPSoC SoC has a "Platform Management Unit". The same is the case for mainstream CPUs. They frequently contain *non-architectural cores*. These are auxiliary cores that run platform management functionality. The firmware running on these cores is vendor-supplied and invisible to user code running on the *application cores*. Intel calls their management core the "Intel Management Engine" while AMD processors have a "Platform Security Processor".

It is not uncommon for these service processors to run a complete OS like Linux or, in the case of the Intel Management Engine, Minix [194]. Full-blown server BMCs additionally are usually connected to the network and run a webserver for remote management.

Narayanan et al. [138] argue that the resulting inherent complexity of the firmware code routinely introduces bugs and vulnerabilities. The general state of firmware running on BMCs is at odds with the high level of privilege with which it executes [21].

Almost no attention has been paid in the research community to rigorously engineering the software for BMCs. This is despite the fact that a BMC has almost complete control over the server, is accessible over the network, and runs completely independently of any OS, hypervisor, or firmware on the main CPU. This has largely to do with the fact that until a few years ago, BMC firmware was proprietary and not publicly available at all [67].

Recent projects such as OpenBMC [64] have disclosed some implementations to the public, with the aim of providing more transparency and in the hopes of collectively finding and fixing bugs and vulnerabilities [68].

While openness helps with transparency and development, it does not eliminate vulnerabilities, and there have been significant critical flaws discovered in recent years [48, 47, 50]. Some of these vulnerabilities are related to OpenBMC itself, which is implemented by a large collection of C++ programs, supplemented by some Python and shell scripts, all running over the Linux D-Bus communications framework [147]. While these underlying technologies are not

insecure *per se*, the system as constructed does not provide high assurance of correctness or security.

I was made aware of this enormous complexity under the hood of modern computers when joining the effort to build a server system: Enzian, a heterogeneous server platform for systems research [39, 186]. The experience with building the BMC for Enzian yielded the following three main research questions addressed in this dissertation:

- How do we specify correct behavior of a BMC function, and how do we obtain a correct implementation?
- BMCs need to interact closely with hardware; how do we ensure the BMC's drivers are correct so that the correctness of a BMC function translates to the hardware?
- Verifying every component in a BMC system is likely to be prohibitively expensive; how do we keep critical, trusted BMC functions secure in the presence of less trusted components?

1.1 Structure of the dissertation

I start in Chapter 2 with an account of the challenges we encountered when building the Enzian BMC. The closed nature of BMCs, especially the low-level functions like power management, made it necessary to reverse-engineer some of the knowledge that I suspect is available in industry.

In this process I also became aware of the poor state of BMC security. Every server relies on its BMC for its correct operation. This is an enormous amount of implicit trust that our modern compute infrastructure puts into BMCs. However, as the vulnerability analysis in Chapter 3 shows, BMCs are far from *trustworthy*. I found 400 vulnerabilities and classify them to inform a safer and more secure system design.

The first part to this design is producing correct implementations for critical BMC functions. The biggest challenge in building the Enzian BMC was configuring the aforementioned power distribution network to turn the board on. The lack of principled approaches in the literature, paired with some close

calls during the bringup of the first Enzian board, inspired the work I present in Chapter 4. We developed a declarative model for power distribution topologies and demonstrated that we can efficiently synthesize correct configurations and instructions to transition these networks into the desired power states. In Chapter 5, I then explore how to use this model not just for offline sequence generation but for online dynamic power management.

The sequences generated from our model then need to be correctly communicated to the hardware. This communication happens over bus-based chip-to-chip protocols like Inter-Integrated Circuit (I^2C). Many I^2C devices suffer from so-called *quirks* which can cause interoperability issues between devices on the same bus. I discuss this problem in Chapter 6. In our approach, we specify I^2C topologies in a domain specific language (DSL). We then model-check the system to prove the absence of interoperability bugs. From the same specification we can also generate a driver stack for the I^2C topology. We generate both drivers and hardware controllers from these specifications. The generated stacks work on real hardware and can run at the same frequency and with the same CPU efficiency as stacks based on off-the-shelf I^2C hardware controllers.

While the critical BMC functions need to be correct, there will likely be components on a BMC like a webserver that are too expensive to build with the same high assurance. In Chapter 7 I present a system design that addresses the need for isolation between trusted and untrusted components.

Finally, I summarize the remaining challenges and other interesting research questions encountered along the way in Chapter 8 before I conclude in Chapter 9.

1.2 Notes on collaborative work

The research in this dissertation was done within the Enzian project. Building a server system is too large a task for a single doctoral student. This dissertation therefore contains results from collaborations with other team members. Some parts of this dissertation have also been published in some form or been accepted for publication.

The experience report in Chapter 2 has in part been published in a peer-reviewed publication. Section 8.3 reports on an experiment conducted in the context of the same publication:

[39] David Cock, Abishek Ramdas, Daniel Schwyn, Michael Giardino, Adam Turowski, Zhenhao He, Nora Hossle, Dario Korolija, Melissa Licciardello, Kristina Martsenko, Reto Achermann, Gustavo Alonso, and Timothy Roscoe. "Enzian: An Open, General, CPU/FPGA Platform for Systems Software Research". In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS 2022. Mar. 2022.

The power topology model presented in Chapter 4 was developed in collaboration with Jasmin Schult in the context of her Bachelor's thesis. It was also published in a peer-reviewed publication:

[161] Jasmin Schult. "A model-based approach to platform-level power and clock management". Bachelor's Thesis. ETH Zurich, Aug. 2020.

[163] Jasmin Schult, Daniel Schwyn, Michael Giardino, David Cock, Reto Achermann, and Timothy Roscoe. "Declarative Power Sequencing". In: *ACM Transactions on Embedded Computing Systems* 20.5s (Sept. 2021).

The extensions to the model and algorithms presented in Chapter 5 were developed in collaboration with Roman Meier in the context of his Master's thesis:

[131] Roman Meier. "Declarative Dynamic Power Management". Master's Thesis. ETH Zurich, Oct. 2022.

The I^2C framework in Chapter 6 is a continuation of earlier work. The main part of it was developed in collaboration with Zikai Liu in the context of his Master's thesis. It was then extended, and the resulting work was accepted for publication at a peer-reviewed venue:

[121] Zikai Liu. "Generating Trustworthy I^2C Stacks Across Software and Hardware". Master's Thesis. ETH Zurich, Sept. 2023.

[165] Daniel Schwyn, Zikai Liu, and Timothy Roscoe. "Efeu: generating efficient, verified, hybrid hardware/software drivers for I^2C devices". In: *Proceedings of the Twentieth European Conference on Computer Systems*. EuroSys '25. Mar. 2025.

The prototype for the Enzian BMC running on seL4 mentioned in Chapter 5 was built by Cedric Heimhofer and Zikai Liu in the context of their respective semester projects:

[78] Cedric Heimhofer. "Towards high-assurance Board Management Controller software". Master's Thesis. ETH Zurich, Mar. 2021.

[122] Zikai Liu. "Towards Trustworthy BMC Software with Virtualization on seL4". Semester Project. ETH Zurich, Feb. 2023.

Finally, some of the exploratory work described in Chapter 7 and Chapter 8 is a collaboration with other students in the context of their theses and semester projects:

[190] Sarah Tröndle. "Real-time Board Management using an FPGA". Bachelor's Thesis. ETH Zurich, Apr. 2021.

[205] Pengcheng Xu. "Enzian Firmware Resource Interface". Semester Project. ETH Zurich, Feb. 2023.

[200] Georg Wehrli. "Generating Platform Configuration from Netlists". Bachelor's Thesis. ETH Zurich, May 2024.

2

Case Study: The Enzian BMC

I learned about the state of the art for BMCs and the challenges in building one the hard way when I joined the effort to build a computer: Enzian, a heterogeneous server platform for systems research [39, 186]. I realized that while every modern server has a BMC, they are a class of systems that is understudied in the academic world. In this chapter I report on the experience of building a BMC for a fully fledged server.

Enzian's defining feature is the cache-coherent link between a 48-core Marvell Cavium ThunderX-1 CPU and a Xilinx Ultrascale+ XCVU9P FPGA with four DRAM channels on both sides. In that regard Enzian is unique. From a platform management perspective however, it looks very similar to a commercially available, off-the-shelf server: a two socket system, with DDR 4 DRAM on both sides. It has dedicated fans for each socket and four chassis fans. The board is powered by an ATX compliant power supply with a 12 V supply for each socket and several auxiliary inputs with lower voltages. Just like an off-the-shelf server it features a BMC.

Enzian's BMC is a daughter board connected to the mainboard. This design allowed us to start developing the BMC stack on an evaluation board while the printed circuit boards (PCBs) for Enzian were still in the design and manufacturing phase. The daughter board in the original design is an Enclustra ZX5 system-on-module (SoM) [133] which features a Zynq-7000 SoC [4]. This is an SoC with an Armv7 Cortex-A9 dual-core processor and a cache coherent FPGA (the Zynq systems were one of the inspirations for building Enzian). We chose a platform with an FPGA to offload certain I/O functions like SPI and

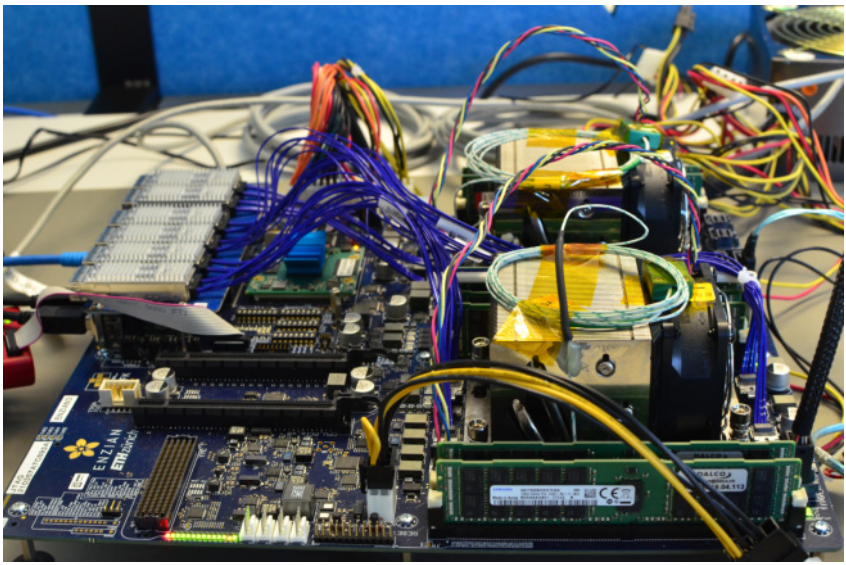


Figure 2.1: Picture of the Enzian board. The BMC module is the green PCB with a blue heatsink under the blue flyover cables leading to the QSFP+ cages on the left.

I^2C controllers to hardware and to be able to correct for potential design errors in the PCB by rerouting signals in the FPGA fabric. The Armv7-based BMC platform turned out to be a limitation for some of the research presented in this dissertation (more on this in Chapter 7). To address this, we now also have prototype machines with Enclustra XU5 modules [132] which are built around the Zynq Ultrascale+ MPSoC SoC [3] featuring an Armv8 Cortex-A53 quad-core processor and a larger FPGA. Figure 2.1 shows a picture of the Enzian board with the original BMC daughter board.

This might seem overengineered for a BMC; indeed we selected the platform in accordance to one of the design principles of Enzian: "If in doubt, overengineer" [39]. However, it is not too far from what is used in off-the-shelf servers: almost all BMC platforms by the two most common manufacturers ASPEED and Nuvoton [68] feature Armv7 or even Armv8 cores [89, 45]. There are also BMC platforms with integrated FPGAs [175] with a very similar rationale to why we chose to include an FPGA in our BMC. While a possibly slightly more powerful than standard BMC allows for more experimentation, it does not change how the platform is managed.

Enzian is built from off-the-shelf components, but the mainboard is custom designed. We specified the requirements and contracted a PCB manufacturer to design and manufacture it. They took care of the low-level design like power delivery circuits, and the physical layout of the board. To turn an Enzian board on – or in fact any such system – the power delivery circuits and the regulators that drive them need to be correctly configured.

It came as a surprise to us that while the manufacturer produced the computer-aided design (CAD) model and schematics of the board, it was our job to produce the software that configures all these regulators on the board correctly. This software turns the board on. Without it, the manufacturer could not run the integration tests on the board to verify the correct functioning of the design or rule out manufacturing defects. This is likely somewhat different for off-the-shelf servers, where both the design and the firmware are handled by the server manufacturer. Anecdotally however, there are at least different teams involved. The next section contains more details on how board management firmware is built.

At the time, we assumed that there were off-the-shelf BMC stacks that take care of powering the board and other configuration tasks. We investigated what

the options were for Enzian and I present the findings in the next section.

2.1 Existing BMC systems

Traditionally, BMCs are proprietary, closed-source systems [68]. Server manufacturers either have their own implementation or use a generic solution that they adapt to their servers. Examples for the former are Dell Remote Access Controller (DRAC)[54] HPE's Integrated Lights-Out (iLO) [83], Lenovo's xClarity Controller (XCC) [110] and Supermicro's Intelligent Management [180]. The most prominent solution in the latter category is MegaRAC SP-X offered by AMI [11]. It is used in a wide range of servers from most major manufacturers (including some that also have their own proprietary implementation) [18].

However, the traditional model of manufacturers building servers and selling them to customers started changing in the last decade: the demand for so called original design manufacturer (ODM) servers has grown [8]. These are servers that are designed and built directly to customer specification. This trend is largely driven by the *hyperscalers*[1] and in 2023 over a third of all servers sold were ODM-direct designs [187]. In a traditional server where the BMC is built by the manufacturer, implementing any bug fixes or additional features at least involves the hardware manufacturer and possibly the vendor of the generic BMC stack. With ODM servers it became possible for the hyperscalers to take control over the BMC stack to be able to "troubleshoot [their] own system" and increase "feature velocity" [64]. The first open source system was OpenBMC. It was initially developed by Meta (then Facebook) and IBM but is now a Linux Foundation project and backed by Microsoft, Intel, IBM, Google and Meta [120].

Since the inception of OpenBMC, many vendors of BMC software have started to offer OpenBMC-based solutions [10, 202, 84]. This is in part due to a growing customer demand for open-source BMC solutions in the light of growing security concerns about proprietary firmware [208]. I will come back to the security of BMCs in Chapter 3.

There is another open-source BMC project called u-bmc [191]. Its aim is to address some of these security concerns. It is also a Linux distribution but its

[1] The term "hyperscalers" describes cloud providers that operate infrastructure at very large horizontal scale. This includes companies like Alibaba, Amazon, Google, IBM, Meta and Microsoft.

userspace is fully written in Go. It also replaces the notoriously insecure Intelligent Platform Management Interface (IPMI) [21] with Google's gRPC [16]. The project is however in a much less mature state than OpenBMC [60].

With most BMC solutions being proprietary and closed-source, and u-bmc not being more than a proof of concept at the time, we were left with OpenBMC as the only viable option.

2.2 Adapting OpenBMC

OpenBMC is an embedded Linux distribution built with the Yocto project [66]. It supports platform management (power, clock and temperature control) and firmware provisioning. It also has remote management capabilities through industry standards like IPMI [90] and Redfish [58] including a web interface. OpenBMC is organized as a collection of services, each in their own process. Initially, a large part of the services were implemented in Python, but this has shifted to C++. The services communicate over D-Bus [147] and are orchestrated using systemd [184]. Examples for services include voltage and fan controllers, sensor monitors and a webserver for remote management. All these services need topology information of the hardware to function: information about voltage regulators, processing units, fans, etc., and how they are connected. This information has to be obtained from the platform schematics. While we have the schematics for Enzian and have also open-sourced them [62], schematics for other platforms that are supported by OpencBMC are not public. The documentation on how to port OpenBMC to a new platform is sparse. Neither did we have a way to compare existing platform implementations to their schematics. We were essentially on our own.

2.2.1 Prototyping the power sequencer

As mentioned before, the first thing we had to deliver to the board manufacturer was the sofware that turns the Enzian board on. Without it, the board manufacturer could not verify that the first PCBs function correctly. Figuring out how to turn on an Enzian turned out to be the biggest challenge from a board management perspective in getting the platform operational. It not only entails

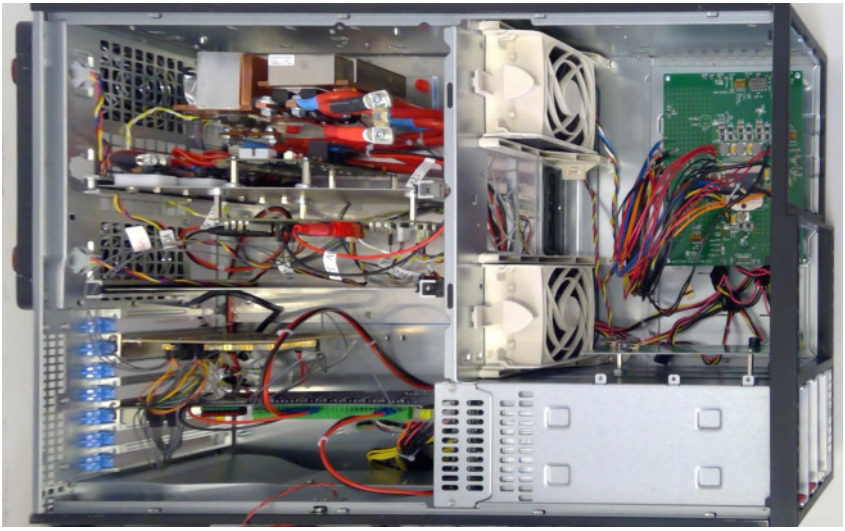


Figure 2.2: The testbed used to prototype the power sequencing software. The copper block with an attached fan is in the top left corner. Next to it is an evaluation board for one of the regulators. On the bottom left is a custom-designed power distribution board with safety fuses. The BMC evaluation board is not connected on this picture.

configuring all the rails to the correct voltages, the rails also need to come up in the right order. Additionally, the regulators' response to critical events like overcurrents or high temperatures need to be configured. This is generally referred to as *power sequencing*, and we will cover it in detail in Chapter 4. To the best of our knowledge these sequences are usually derived by experienced experts and there are no examples to learn from as for commercial platforms the crucial information is proprietary.

To build expertise in the matter, we built a testbed that consisted of evaluation boards for Enzian's BMC SoM and the regulators used on the board. To simulate the loads of the CPU and FPGA, we used a copper block with a heat sink and fan attached. The testbed is depicted in Figure 2.2.

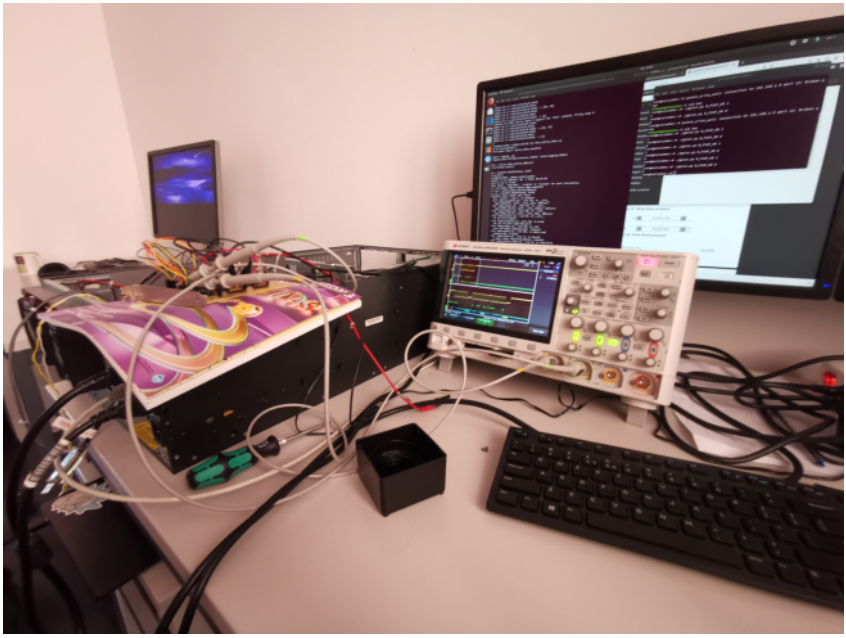


Figure 2.3: Debugging power sequencing code using our testbed and an oscilloscope

This testbed allowed us to prototype the power sequencing code before we had any finished Enzian boards. We could both test communication to the regulators over I^2C and develop and understanding of how to configure each individual regulator. The debugging of either often involved an oscilloscope to check the signal integrity on communication buses, and measure ramp times and fault response reaction times of regulators. Figure 2.3 shows a picture of such a debugging session where we were checking that the waveforms on the I^2C bus were correct.

The main goal for this prototype was maximum flexibility in our experiments. Instead of implementing the full regulator drivers inside the Linux kernel, we therefore chose to only use the Linux kernel's bus drivers for I^2C. All regulator specific drivers are implemented in userspace in Python. This turned out to be a good decision as we could focus on how to interact with the regulators without having to deal with the idiosyncrasies of Linux kernel drivers. However, the subtly different ways some regulators implement the I^2C standard still posed a challenge. This experience made us realize both the importance and fragility of these low-level protocols, and led to the work I present in Chapter 6.

We ultimately retired the testbed in favor of two Enzian boards from the first iterations that have design defects and are not be able to power the CPU and FPGA. As they are still populated with all the regulators they serve as a much more faithful testing platform for BMC development, especially the power sequencing stack. Using these boards we, e.g., found an issue where the communication between the BMC and some power regulators would spuriously stop working. It again took an oscilloscope to find out that this was due to a signal integrity issue on the I^2C bus caused by missing pull-up resistors. While our testbed had included them, the actual Enzian boards did not, but instead expected the internal pull-ups in the BMC FPGA to be enabled. A design choice by our manufacturer that had escaped us and led to quite a bit of frustration when previously working power sequencing code would suddenly fail.

The initial goal was to retire the Python code in favor of a power sequencer implemented using OpenBMC's facilities. However, we have since abandoned the idea in favor of a higher-assurance power management stack.

2.2.2 Turning Enzian on

The board manufacturer used our power sequencing stack to electrically test the boards. It was however our task to perform the functional tests as bringing up the CPU and FPGA was beyond the expertise of the board manufacturer. Despite having gained quite a bit of confidence in our power sequencing code, the bringup of the first board that had passed the manufacturer's tests was a nail-biting moment. We used current-limiting power supplies to minimize the chance of frying the board in the event of a power sequencing failure. However, the ThunderX-1 CPU on Enzian draws about 200 A on its main 0.9 V power rail during the boot process. Supplied into a short circuit that amount of current would easily destroy the board. This initial bringup happened mid 2020 during the COVID-19 pandemic. Due to supply chain issues and lockdowns it would take another year for the other Enzian boards to arrive, so we could not afford to lose this first one. We increased the current limit step by step, every time verifying that all voltages and temperatures were within specification. Figure 2.4 shows our setup for the CPU bringup. While we ultimately managed to boot the CPU, this experience convinced us there needed to be a less ad-hoc approach to power sequencing. I describe our solution in Chapter 4.

2.3 Conclusion

Building our own computer has changed our perspective on how modern computing systems work, especially in terms of board management. While probably none of the lessons we learned are surprising to professional server designers, they were eye-opening to us as systems researches. Judging from the lack of scientific literature on many of these topics I suspect this would be the case for many other researchers as well. The work presented in this dissertation is a result of these lessons. I start with a critique of the state of the art for board management in the next chapter. The experience with power sequencing has resulted in a more principled approach that is described in Chapter 4 and the experience with I^2C has spawned the work described in Chapter 6.

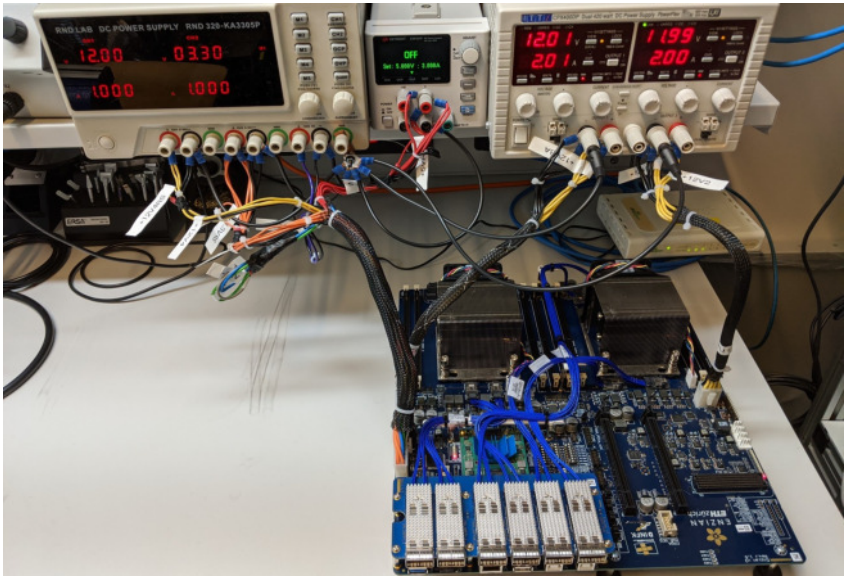


Figure 2.4: Picture of the setup for the first bringup of an Enzian board.

3

Critique of the State of the Art

The experience with building a BMC for Enzian did not only make me aware of how critical these systems are, but also raised concerns about the way they are designed. I am not the first to realize this. As I pointed out in Section 2.1, vendors have started to offer open source solutions due to customers raising concerns about the state of BMC firmware. Indeed, early BMC systems were not designed with security in mind [68] and a number of vulnerabilities have gotten public attention [174, 18, 61, 188, 152, 173, 199]. There is further evidence that the security requirements for BMCs were not taken seriously enough: in one of the vulnerability database entries a vendor is quoted stating that their BMCs "are intended to be on a separate management network; they are not designed nor intended to be placed on or connected to the Internet" [46]. In the disclosure of another vulnerability it was reported however, that at the time, over 92 000 systems had their BMCs exposed to the internet (over 47 000 of them vulnerable to the disclosed vulnerability) [61]. The disclosure also points out that while it is certainly not a good idea to expose BMCs to the internet, these vulnerabilities also increase the impact of an attacker gaining access to a corporate network. The United States National Institute of Standards and Technology (NIST) released guidelines for firmware resiliency [151]. In light of recent attacks by state-level actors, it has been pointed out that these firmware vulnerabilities are a threat to national security and that the NIST guidelines have critical gaps [178].

To gain more insights into the nature of BMC vulnerabilities, I conducted an analysis. I will present the results in the following.

3.1 BMC vulnerability analysis

3.1.1 Methodology

To collect BMC-related vulnerabilities I used the CVE Program's datebase. The CVE Program has been cataloging vulnerabilities in computer systems since 1999 [149]. The database is publicly available on GitHub[1]. I downloaded a snapshot version on June 26, 2024[2], and did a keyword search with some general BMC related terms (BMC, Baseboard) and the names of the most widely deployed systems (OpenBMC, iLO, (i)DRAC{0-9}, MegaRAC, XCC). Many of these systems are Linux-based. I intentionally did not explicitly include bugs in the Linux kernel in my analysis as they are fairly well studied already [29, 171]. I did however include Linux vulnerabilities if they matched the keywords. The search yielded 476 hits and after manually reviewing them and removing false positives I was left with 400 vulnerabilities affecting all mainstream BMC systems. I then analyzed the distribution of these over time (Section 3.1.2) and developed a taxonomy for the nature of the vulnerabilities (Section 3.1.3).

3.1.2 Vulnerabilities over time

The first analysis provides insights into how BMC security developed over time. I use the number of vulnerabilities discovered per year as a proxy measure for security. To be precise, I use the number of CVE records *reserved* per year, as the discovery dates are not available in the database. Due to embargoes mandated by responsible disclosure practices, the two dates can differ by several months. Despite this difference, the analysis is still able to show the overall trend in BMC security.

Figure 3.1 shows the results of the analysis. The CVE records that match the keywords are distributed over 20 years, from 2004 to 2024. While there are very few vulnerabilities in the earlier years, the number starts to rise around 2020. The number peaks in 2021 with 117 vulnerabilities and is again lower in the most recent years.

[1] https://github.com/CVEProject/cvelistV5
[2] Git hash: ac6fec85ca8551b9680bb125347af88a8a9fe906

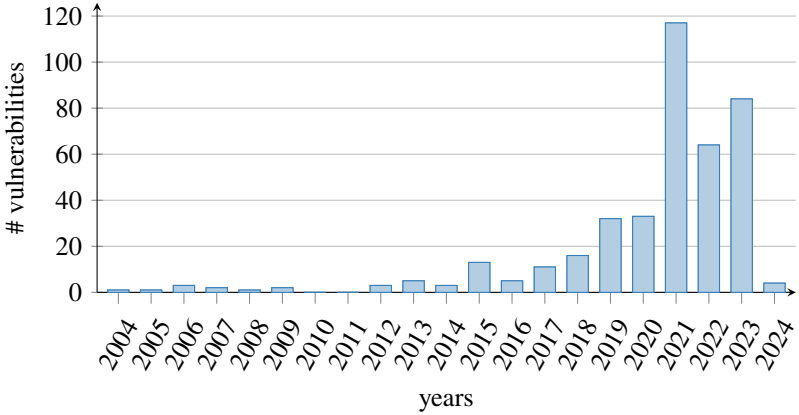


Figure 3.1: Number of vulnerabilities in BMCs over time

I attribute the increase in vulnerabilities in the past years to increased attention to BMCs rather than worsening security. However, the decrease in the past 2 years is probably due to variance in when the vulnerabilities are published. There are a few sets of vulnerabilities that, telling from how similar the reports look, were published together. Some of them state that they were discovered in internal security reviews. This leads to bursts in the number of vulnerabilities. The low number in 2024 is due to the database snapshot only covering half the year.

I conclude that while the security of BMCs is probably not getting worse, it is also not getting better. I argue that this is due to how BMCs are designed: instead of ruling out bugs and vulnerabilities by design they follow a reactive approach where bugs are fixed when they are discovered. In this dissertation I propose an alternative to this "patch and pray" approach.

3.1.3 A taxonomy of vulnerabilities

To inform a more secure design for BMCs I also analyzed the nature of the vulnerabilities. The goal is to design a system that rules out classes of vulner-

abilities. The taxonomy therefore focuses on how an attacker can compromise a critical function of the BMC using the vulnerability, rather than the vector that enables the vulnerability (e.g., a buffer overflow or flawed cryptography). I manually categorized the vulnerabilities into the following classes:

Vulnerabilities in critical components. These are flaws in critical BMC components that directly compromise the BMC's ability to safely manage the platform. Examples include bugs in the firmware manager that allow flashing compromised updates or a driver bug that misconfigures a power regulator. The category also includes weaknesses in the authorization mechanism like not forcing system administrators to change default credentials.

Privilege escalation vulnerabilities. These are vulnerabilities in components that themselves are not critical to the core task of the BMC but can be used to then gain enough privilege to affect platform management. This can either be through a further bug in the BMC's OS that lets an attacker compromise a critical component or through using ambient authority of the flawed component to bypass authorization. Examples include path traversal or code injection bugs in a webserver.

Non-critical vulnerabilities. Like privilege escalations, these are vulnerabilities in non-critical components, but the vulnerability stays contained in the component. The CVE records are not always clear on whether a compromise can spread. I therefore remained conservative and only classified a vulnerability in a non-critical component as a privilege escalation if the record explicitly mentions that sensitive data can be obtained or be tampered with. An example of this kind of vulnerability is a bug in the HTTP header parser of a webserver that crashes the server but does not allow an attacker to further affect the system. Calling such vulnerabilities non-critical might be counter-intuitive. The classification only refers to the fact that these vulnerabilities cannot affect components that are *safety-critical* for the platform. I will revisit this category and the potential consequences of such vulnerabilities in Section 3.1.4.

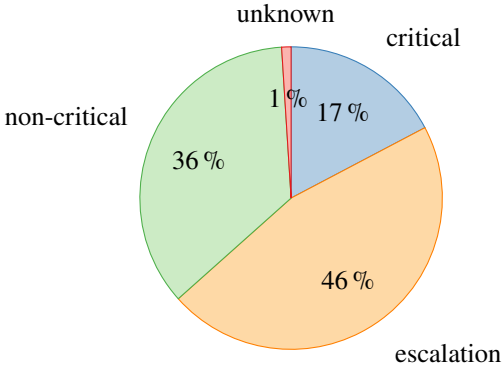


Figure 3.2: Percentage of BMC vulnerabilities per category.

The percentages of vulnerabilities in each class are shown in Figure 3.2. 46% of the vulnerabilities are privilege escalations (“escalation”) a further 36% are non-critical vulnerabilities, bringing the total of vulnerabilities in non-critical components to over 80%. The rest are vulnerabilities in critical components (“critical”) except for 1% of vulnerabilities where the CVE records did not provide enough information for a classification. Most of the vulnerabilities in critical components are related to authentication and authorization, e.g., insecure cyphers, default credentials or session management flaws. 4 of the total 69 are flaws in firmware update validation and a further 3 are low-level bugs related to restricting hardware access.

The fact that the overwhelming majority of vulnerabilities are in non-critical components is not surprising: critical components are usually not directly accessible from outside the BMC and the majority of vulnerabilities are discovered in remotely accessible components like webservers or IPMI protocol handlers. This also explains why a high fraction of vulnerabilities in critical components is related to authentication and authorization as it is the only critical function that – by its nature – deals with outside access to the BMC. However, almost half the vulnerabilities allow an escalation of privileges. I attribute this to the fact that the components that offer remote management have a lot of ambient authority:

they can trigger any management action that they offer, e.g., turning the server on or off or update firmware. They are, however, only supposed to do so after checking with the authorization component if the requesting user is allowed to perform the respective action. If a vulnerability somehow allows bypassing the authorization check, an attacker can potentially abuse all this ambient authority. OpenBMC for example by default runs its webserver with root user privileges. As it uses D-Bus for inter-process communication and connects all services to the same bus daemon, it can also not enforce strict communication patterns between components. Instead, every component can call any method offered by any other component. In the light of these issues it is worth further analyzing the vulnerabilities that were classified as non-critical.

3.1.4 A closer look at non-critical vulnerabilities

As mentioned, I was conservative in the classification and only classified a vulnerability in a non-critical component as a privilege escalation if the CVE explicitly stated so. However, knowing how little privilege restrictions some BMC systems have, it is worth taking a closer look.

I further divided the class of non-critical vulnerabilities according to the effect an attacker can achieve with an exploit. With the information in the CVE records I was able to distinguish three classes of exploits: code execution, client-side exploits and denial of service (DoS). Code execution vulnerabilities let the attacker inject code into the compromised component that can then be executed with the privileges of the component. Client-side vulnerabilities include cross-site scripting (XSS) and content injection into user interfaces and can be used to trick a user into executing commands on behalf of an attacker. Finally, DoS vulnerabilities allow the attacker to render the compromised component useless by, e.g., crashing it. I only classified a vulnerability as a DoS vulnerability if it did not also fit one of the other two categories.

The distribution of the sub-categories of non-critical vulnerabilities is shown in Figure 3.3. Note that 100% here only correspond to the 142 non-critical vulnerabilities, not the total of 400 BMC vulnerabilities. For 19% of the non-critical vulnerabilities the CVE records did not contain enough information to reliably classify them. 12% of the vulnerabilities allow an attacker to gain

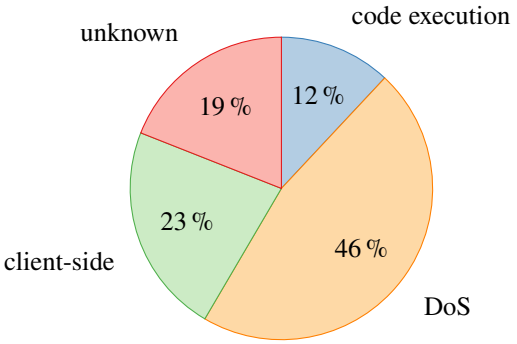


Figure 3.3: Breakdown of BMC vulnerability types in the non-critical category.

code execution. 23% are client-side vulnerabilities and 46% allow "only" DoS attacks.

While client-side vulnerabilities are by no means harmless, their prevention is out of the scope of this dissertation as I focus on the system design of the BMC itself not the client-side interfaces to it. I therefore concentrate on the other two classes of non-critical vulnerabilities here. According to our classification, the CVE records for the code execution vulnerabilities do not mention any escalation of privilege. In a system where, e.g., a webserver runs with high privilege, it is, however, very likely that the injected code could affect components other than the vulnerable one. An attacker could, e.g., inject shell code that instructs the power manager to turn off the machine via a D-Bus call. Similar caution has to be taken if a vulnerability can "only" be used for a DoS attack on the component. Firstly, crashing, e.g., the webserver still means that the system can not be accessed anymore. Unless there are recovery mechanisms, the BMC might need to be physically reset. In a large deployment this can be very expensive. Secondly, the attacker might be able to bring the component to monopolize a resource. One example is sending the component into a busy loop to consume all the CPU time. Such an attack will still affect the rest of the system. Critical functions like thermal management might degrade, threatening the safety of the managed platform.

3.2 Conclusion

I draw two main conclusions from the analysis. First, vulnerabilities in BMCs are unlikely to disappear with the current system designs. Second, missing or weak isolation and low privilege restrictions for non-critical components are major contributors to the continuing appearance of these vulnerabilities. I will present the proposed system design to address these issues in Chapter 7.

A vulnerability analysis is however less suited to assess the correctness and safety of components that are critical to a BMC's core task: managing the platform. This is in part due to the fact that these components are not directly accessible to an attacker. Obvious flaws in a component that is critical to the safe operation of the hardware, like the power manager, would surface during a manufacturer's tests and therefore never make it out in to the wild. There could however still be more subtle flaws, which could be exploited (or triggered by accident) to damage hardware. There are security vulnerabilities that exploit power and clock management features [185, 150]. My main critique here is, that there is very little public literature on how these critical components are built. At the same time very few platforms with public documentation at the necessary level exist. These factors make it almost impossible to assess the threat posed by BMC systems. In the next chapters I will shine a light on some of these issues.

4

Declarative Power Sequencing

We have seen that BMCs have to deal with a lot of complexity that is invisible to the OS or firmware running on the application cores of a server. In Chapter 3 we have seen that despite this critical role BMCs are far from high-assurance systems. I will focus on the security aspects of BMCs in Chapter 7. Here the focus is on the complexity and correctness of one of the most important tasks of a BMC: turning on the machine it manages, a surprisingly subtle problem.

I encountered this problem myself when I was involved in the process of bringing up BMC software for Enzian, a new, large, server-class board intended for research. I gave an account of building its BMC in Chapter 2. The biggest challenge was to figure out how to turn an Enzian board on. This involves a process called power sequencing: configuring all rails and clocks on the board to the right voltages and frequencies and turning them on in the right order. The work in this chapter was driven by this experience as well as 3rd-party accounts of power sequencing-related problems.

Incorrect behavior of the BMC in controlling voltages on the board can render the hardware permanently unusable, essentially destroying the board [30]. This, in turn, means that developing and validating the power sequences for a new board is a cautious, time-consuming process. Even if there is no damage to hardware, voltage and clock control features can be used to attack the system [185, 150].

The aim is to create a rigorous foundation for engineering power control software that gives OS and firmware developers assurance that the underlying hardware can be trusted to behave as advertised. This is an ambitious goal, but

the first step is to define the BMC's hardware environment and the constraints that capture the machine's safe operation.

We model the power trees of server machines and use these models to synthesize power-on sequences for the server, given a specification of the desired powered-up state of the system.

The state-of-the-art approach, as illustrated by publicly available systems such as OpenBMC, is handwritten sequences derived from schematics and datasheets. In this chapter I describe how we improve on this in the following ways:

- Identifying that platform description is a problem of *declarative specification*, and can (and should) be separate to the *mechanism* of generating imperative sequences.

- Recognizing that the important features of power tree nodes (e.g., regulators) are consistent across a wide range of components and topologies, allowing any system to be specified with a small set of *universal primitives*, expressed as a declarative specification language.

- Illustrating that the sequence generation mechanism maps well onto a well-known and widely-studied problem (constraint satisfaction), with existing mature tools.

- Demonstrating that this implementation scales efficiently to the *large*, *complex* systems, and works on *real production hardware*.

The benefits of the approach I describe are allowing the engineer to separate *"what their platform looks like"* from *"how it is controlled"* allowing firstly a division of labor, and more significantly the opportunity to delegate the mechanical, time-consuming, and error-prone second problem to a mature, well-studied, and high-performance algorithm. This promises to reduce not only the time needed to derive a sequence for new hardware, but the maintenance effort as hardware evolves. In the end, we get both the correct sequence and the reasons (often buried in datasheets) why the sequence is the way it is.

4.1 Background

I start by surveying the power sequencing problem, and how it has become so complex.

In the past, it was enough to use careful circuit design to power on a system which needed only a single 5 V or 3.3 V power rail, supplied directly from an AT or ATX power supply. As processor frequencies increased, however, the supply voltage decreased: because the power consumed by a CPU can be approximated by $P_{CPU} \propto V^2 \cdot f$ where V is voltage and f is clock frequency, the increased frequency of processors necessitated a decrease in voltage in order to keep power budgets reasonable. To obtain these lower voltages, *step-down regulators* take an input voltage from a power supply, and step it down to a stable voltage appropriate for the consuming device.

As boards become more complex, this leads to the situation in modern computing platforms where power and clock management is not a trivial matter, and the platform consists of numerous power and clock domains. For example, an AMD Zen processor requires three separate voltage domains (VDDCPU, VDDIO, and VDDSOC) [26], in some cases drawing hundreds of amps. Each channel of DDR4 memory requires two different voltages (1.2 V VDD and 2.5 V VPP) [136]. This means that for a two-socket system, each with two channels of DDR DRAM, a minimum of 14 different voltage regulators are needed to supply the necessary voltages, not to mention the needs of storage, networking, and accelerators. These devices each rely on a tree of regulators, each supplying voltage within a safe range, often needing to be supplied and activated in a specific order during the bring-up process. As an example for a modern power tree, the one for Enzian is shown as Figure 4.1.

As most voltage regulators are able to accept ranges of input voltages and, depending on configuration, produce a range of output voltages, getting a correct configuration is critical. Misconfiguration of voltage levels is an obvious source of failure, but there are additional failure modes associated with power electronics. Latch-up is one such failure mode that can occur due to incorrect power sequencing, when the input to a CMOS device is greater than VDD, causing a low impedance path that can cause the circuit to malfunction or be destroyed completely [135]. As the number of transistors increases and their size decreases the danger of latch-up and other sequencing failures increases [59]. Borderline

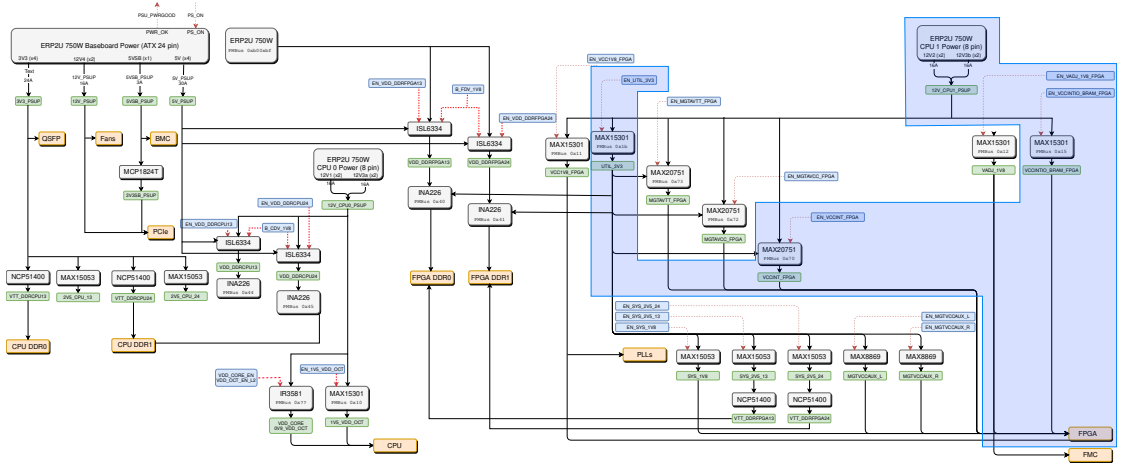


Figure 4.1: The power tree of a modern two socket server system contains dozens of regulators, illustrated here with the one for Enzian. Individual regulators are shown in white boxes and their control signals are in blue with red dotted lines. The black lines connecting boxes are supplied voltages and the orange boxes are end consumers of one or more power sources. The blue highlighted region is used as an example for the remainder of this chapter.

power and clock levels can be exploited as security vulnerabilities [185, 150].

In order to deal with these more complex requirements, several solutions have been developed, beginning with specialized hardware for power sequencing [77, 9]. These integrated circuits are called complex programmable logic devices (CPLDs). They are reliable and inexpensive but still require configuration with the correct sequence when the platform is manufactured. Moreover, system designers, especially with an eye on the data center, demand more complete and configurable board management than simple power sequencing, such as monitoring and remote management, which is beyond the capabilities of hardware regulator controllers.

This, coupled with the need to orchestrate increasing numbers of regulators via software over networks such as low-speed serial buses (e.g., I^2C) leads ultimately to modern servers using a BMC as a complete platform management system.

To my knowledge, the current published state of the art in industry does not go beyond manually coded point solutions and no attempts at auto-generating power sequencing code, let alone formal verification, have been made.

4.2 Experience

I was made aware of the complexity of power sequencing while involved in building Enzian. [39] (cf. Chapter 2). Helping to design and build a system as large as Enzian has exposed me to the importance and difficulty of correct power sequencing in a modern computer. This problem is not apparent in the existing public literature concerning smaller, simpler systems.

The Enzian mainboard is an eATX-format 22-layer board which has a per-socket thermal design power (TDP) of more than 100W, and a total system TDP of around 600W. The power tree consists of 25 discrete voltage regulators and more than 30 separate power rails with complex sequencing requirements. This complexity is increased by the heterogeneity of the system's main components leading to very different power sequencing constraints for the two sockets.

Much early ad-hoc work led us to appreciate the importance of separating concerns: dividing the task of correct *sequence generation* from both the

specification of *platform parameters*, and the labor-intensive task of faithfully modeling *regulator behavior*, including quirks.

Even though we (like most other groups who face this problem) are only working on a single design, both the design and our understanding of it have evolved over time. This persuaded us that a more generalizable approach was, indeed, worth the extra effort for us.

Platform parameters consist of minimum and maximum voltages on nets, power topology information, and the like. Some of these parameters we could control, but others were liable to change at short notice. For example, in our case an error in board layout led to some regulators being replaced with a different model partway between early prototypes and final hardware, requiring an updated power tree. Also, as we evaluated the prototypes, and the tightness of regulation (e.g., $I \cdot R$ losses) and transient behavior of the power nets, we gradually tightened (or occasionally loosened) the upper and lower voltage and current limits for specific nets relative to the datasheet values, again requiring updated parameters.

Between the first design and working hardware, the power sequences needed to be redesigned and/or adapted repeatedly, and doing this manually required a lot of time spent tediously changing one voltage after another and examining the resulting current levels in the system.

The modeling of regulators, particularly those for the critical processor core voltage supplies (our main chips draw more than 100 Amps at less than 1 Volt), was a very time-consuming process.

Device quirks and minor areas of non-compliance with the standard for Power Management Bus (PMBus) [182, 183], the protocol stack used by the BMC to communicate with the regulators, can have dramatic consequences: In one case, due to an interaction between the CPU core regulator's PMBus interface and its more fine-grained proprietary control registers, it would default to an output of 1.2 V on enable, well outside the maximum rating of the CPU, even though it was notionally programmed to 0.9 V.

This was a hair-raising realization, as turning the system on with this faulty sequence would have resulted in overvolting the CPU on Enzian. This would have likely destroyed the only existing board at the time, throwing the project back months. Luckily, we caught the mistake with our careful testing. The solution here was to reprogram the output voltage *after* the regulator's logic

supply was enabled, but *before* its output was enabled. This is now incorporated as a sequencing requirement in this device's model, guaranteeing that future generated sequences automatically incorporate this hard-won knowledge, which would not be explicit in a hard-coded power-up sequence.

The large investment we had to make in faithfully modeling the behavior of these components is preserved, and applied automatically to any updated sequence for this board, or indeed wherever the same components are used in future designs.

Enzian is designed with redundant configuration mechanisms: While the current firmware sequences its power tree in software over PMBus, the hardware itself also supports a traditional 'hardwired' power sequence where the enable input of a regulator is driven (via a CPLD) by some logical combination of control signals and the 'power-good' signals of other regulators.

However, whether the sequence is programmed in software, or wired-in directly, exactly the same problems exist, namely: what should the sequence be, and how do we know that it is safe?

4.3 Model

In this section I describe how we developed our model of a power tree, with reference to the full power tree of Enzian in Figure 4.1. Figure 4.2 shows a representative section of this full tree, and will serve as our running example. We consider the power tree as a directed graph, with two types of nodes: components (`IC1-4`) and nets (`12V_CPU1_PSUP`, `UTIL_3V3`, `VCCINT_FPGA`, `VCCO_FPGA`, `EN_UTIL_3V3`, `EN_VCCINT_FPGA`, `EN_VCCO_FPGA`). A directed edge exists from a net to the input of either a regulator or load component (e.g., CPU). Likewise, an edge exists from a regulator output to the net it supplies. Each net may only be driven by a single output.

While there are a huge variety of regulators, load devices and system designs, the basic electrical laws, together with practical considerations, mean that at the level of detail we need, all regulators and devices are essentially equivalent: Regulators convert a small number of input voltages (power and logic) into a small number of outputs, and with only limited exceptions (e.g., USB OTG

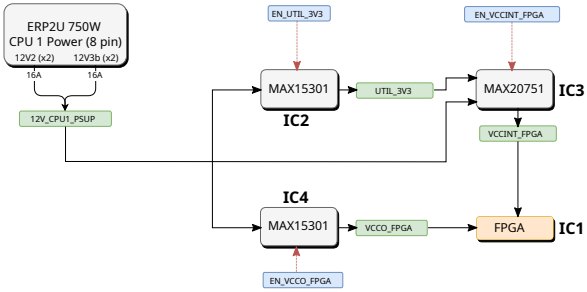


Figure 4.2: Detail view of a power tree (highlighted in blue in Figure 4.1) that illustrates sequencing requirements and high currents: `IC1` (`FPGA`), `IC2` (`MAX15301`) that supplies `UTIL_3V3`, `IC3` (`MAX20751`) which supplies `VCCINT_FPGA`, `IC4` (`MAX15301`) that supplies `VCCO_FPGA`, and the `12V_CPU1_PSUP` rail. The regulators are controlled by enable signals: `IC2` is enabled by `EN_UTIL_3V3`, `IC3` is enabled by `EN_VCCINT_FPGA`, and `IC4` is enabled by `EN_VCCO_FPGA`.

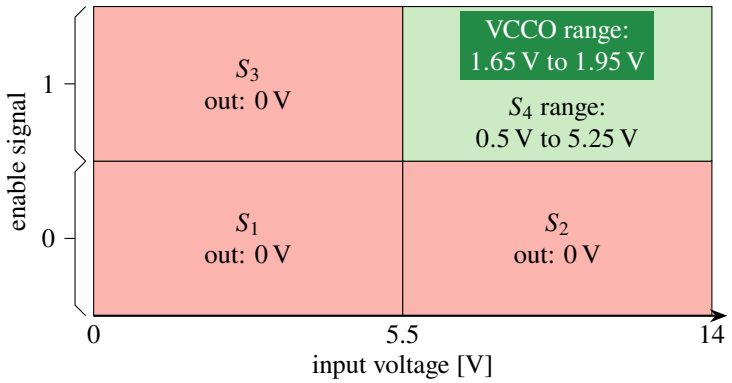


Figure 4.3: IC output voltage range as a function of inputs, for LVCMOS18 IO.

charging), power only ever flows from source to sink. Devices (e.g., CPUs) accept some range of voltages, and require some ordering between the rails.

Except for systems with rechargeable batteries, which are beyond the scope of the current work, the power rails thus always form a directed acyclic graph. In all systems of which I am aware, regulators can be characterized by a range of permissible input and output voltages and currents, and load devices by their allowed supply voltages and a partial order between them that power-up must respect. This model is applicable to a large range of systems, from embedded devices with only a handful of regulators, up to the large server-class system on which we evaluate our approach in Section 4.5.

The state of a net is its current nominal voltage (the model only considers its DC value and ignores $I \cdot R$ drops). The state of a regulator is the combined state of its inputs, both physical (supply voltage, on-chip enable pin) and logical (PMBus-commanded output voltage or enable signal).

The output of a regulator is a function if its inputs. Figure 4.3 illustrates this for regulator `IC4` which supplies net `VCCO_FPGA` (the I/O pin supplies), whose value depends on the I/O standard in use, generally either 1.2 V (e.g., for DRAM), 1.8 V or 3.3 V (e.g., for legacy CMOS ICs). Here, the IO standard is LVCMOS18 which requires an I/O bank voltage between 1.65 V and 1.95 V.

The inputs to this regulator are two-dimensional: supply voltage and enable signal. With enable deasserted, the regulator outputs 0 V. With enable asserted, it can generate any voltage between 0.5 V and 5.25 V. Thus, its output range (the set of outputs that we can instruct it to generate) is $[0\text{V}] \cup [0.5\text{V}, 5.25\text{V}]$. The valid ranges for the output driving a net and all inputs supplied by the net are intersected to compute the range of target voltages for that net, in this case $[1.65\text{V}, 1.95\text{V}]$. If this regulator were able to supply only up to, say, 1.90 V, the target interval would be reduced to $[1.65\text{V}, 1.90\text{V}]$.

From the target range for a net (the *static* constraint) and the state diagram for a regulator, we can infer *dynamic* constraints on its inputs which in turn become the target for regulators higher in the tree, possibly after intersection with the input requirements of other regulators/loads sharing the same net. These constraints are then iteratively filtered back toward the root nodes, such that every net has a target range. In this example, to produce an output in the 1.65 V–1.95 V range, the supply net for `IC4` (`12V_CPU1_PSUP`) must be between 5.5 V and 14 V, and its enable signal must be asserted. In this instance the supply net's requirement is satisfied by the EPS12V power supply's guaranteed output range of $[11.4\text{V}, 12.6\text{V}]$.

Looking again at Figure 4.2, `IC3` introduces a different type of constraint: ordering. This regulator has two supply inputs: one for the supply net to be regulated down, and the other for its internal logic. Until its internal logic is powered, it's impossible to communicate with the regulator, and thus enable its output, even if the main supply is available. In this case, we've introduced a recursive element to sequence construction once this logic supply is produced by another regulator that we must configure. This case is easily handled by the model: this regulator's equivalent of Figure 4.3 has an additional axis, corresponding to its logic supply. This in turn generates a dynamic constraint on the logic regulator (`IC2`) output, forcing us to enable it before attempting to enable `IC3`.

The final constraint imposed by the model is an ordering between the voltage rails for a given IC. While the supply nets for `IC3` may be safely enabled in any order, this is not true for the CPU and the FPGA. As already mentioned, incorrect power sequencing can lead to latch-up and instantaneous destruction for ICs whose *normal* power consumption is in the hundreds of amps: The core regulators will happily supply 200 A into a short circuit.

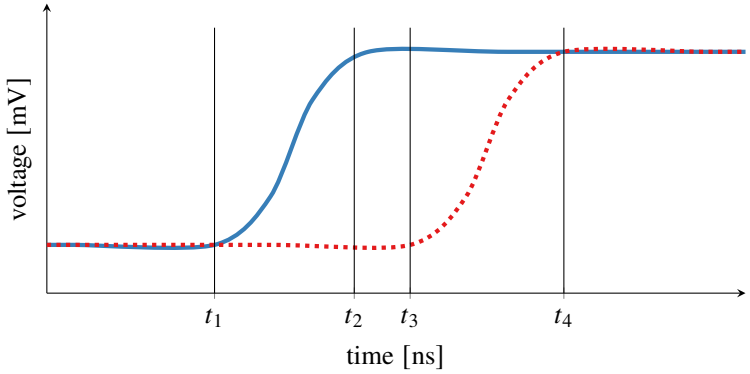


Figure 4.4: Illustrative example of a manufacturer-supplied sequencing graph for an IC. The voltage signal represented by the dotted line must only be ramped up once the other signal has stabilized, i.e., $t_3 > t_2$.

The sequencing requirements for an IC are generally supplied by the manufacturer, either as a recommended/mandated power sequence or in a diagram such as Figure 4.4.

To model this we associate each state change of a net with an *initiate event* (e.g., enable signal asserted) which triggers the state change and a *complete event* (e.g., n consecutive voltage measurements within range after which the conductor has stabilized in the new state).

For the CPU's main power supplies, the `VCCINT_FPGA` initiate event must happen after the `UTIL_3V3` complete event:

$$T(\text{Initiate}(\texttt{VCCINT\_FPGA})) > T(\text{Complete}(\texttt{UTIL\_3V3}))$$

Additionally, we have the natural condition that complete events always happen after the initiate event for the same net:

$$T(\text{Complete}(net)) > T(\text{Initiate}(net))$$

This gives us a partial order on events.

The goal of a power sequence is to place leaf components – those with no outputs, e.g., the CPU – in a specified power state. We term these leaf components *consumers*. To generate a valid power sequence we must:

1. Find a platform state that satisfies all consumer constraints and all other input constraints (static and dynamic).

2. Find an order of actions that transitions the platform into that state while observing the partial ordering of all relevant events.

4.4 Algorithms

As discussed in Section 4.3 there are two correctness criteria for a power-up sequence that we tackle separately. First, we compute a valid platform state by casting the problem as a constraint satisfaction problem and using a constraint solver. After that, we use the partial order on the events to derive a sequence that transitions the platform from its current state to the new state. If no such sequence exists, we compute a new solution for the platform state and try again.

4.4.1 Computing the platform state

Given a set of consumer constraints that describe the desired power states of the consumers (CPU and FPGA in the case of Enzian), we first need to compute a platform state that satisfies these constraints. This means we need to propagate these constraints back through the tree and for each output, select a state that satisfies the static and dynamic input constraints for the attached net.

A constraint satisfaction problem is defined by a set of variables, a set of domains that define what values each variable can take, and a set of constraints that define relations between subsets of variables that any assignment for the variables must satisfy. We cast the problem of computing a valid platform state as a constraint satisfaction problem as follows: The set of variables consists of a variable $w_i, 1 \leq i \leq$ (number of nets) for each net in the platform, that represents the state of the output connected to that net. The variables can take on integer values that represent the voltage of the output in millivolts (mV) or 0

and 1 in the case of logical signals. The set of constraints is composed of static constraints, dynamic constraints, and consumer constraints.

The static constraints ensure that all the maximum ratings for the connected inputs are observed, i.e., for each net we have a constraint

$$\bigwedge_{j=1}^{\text{number of inputs}} \text{low}(M_j) \leq w_i \leq \text{high}(M_j)$$

For each input j connected to net i this ensures that the net's state is within the range of the input's maximum rating, i.e., at least its minimum rating $\text{low}(M_j)$ and at most its maximum rating $\text{high}(M_j)$.

The dynamic constraints connect the components' outputs to their inputs: they ensure that for each output that gets configured into a specific state the inputs of the same component have the appropriate values. In the example in Figure 4.3 showing the possible states for `IC2`, this means the output can fall into one of four regions. Each one of these regions then in turn imposes dynamic constraints on the inputs if the output value falls within the region. In general for each component we add constraints of the following form:

$$\bigvee_{S_i \in \text{state regions}} \text{low}(S_i) \leq w_i \leq \text{high}(S_i) \wedge D_{S_i}$$

Each element in the disjunction represents the situation wherein the output connected to net w_i is in a particular state region S_i, i.e., between $\text{low}(S_i)$ and $\text{high}(S_i)$. The term D_{S_i} in each element represents the region-specific dynamic constraints on the inputs: they impose the requirements of the state region on the component's inputs and propagate them back up the tree. Each D_{S_i} is of the following form:

$$\bigwedge_{j=1}^{\text{number of inputs}} \text{low}(I_j)_{S_i} \leq w_j \leq \text{high}(I_j)_{S_i}$$

Every element of the conjunction takes care of propagating the dynamic requirements to one of the inputs I_j by ensuring that it will be configured inside its required bounds for state region S_i, i.e., between $\text{low}(I_j)_{S_i}$ and $\text{high}(I_j)_{S_i}$.

Finally, we add the consumer constraints which for each consumer constrain the values of the nets that they are connected to according to their desired power state:

$$\bigwedge_{j=1}^{\text{number of inputs}} \text{low}(I_j) \leq w_j \leq \text{high}(I_j)$$

Now that we have encoded the problem of finding a platform state that conforms with the consumer constraints for a specific power state into a constraint satisfaction problem, we can use standard constraint solving techniques to obtain a state for every output. We now compute a set of actions that ensure every output is configured into this state and all sequencing requirements are observed.

4.4.2 Computing the sequence

We have seen in Section 4.3 why it is important for a sequence that transitions the platform from a current state into a new state to observe the sequencing requirements. We have also seen that every change of output state is associated with an initiate and a complete event and that there is a partial ordering between those events. Every initiate event then translates to an action that triggers a state change and every complete event to a check that a state change has successfully completed, e.g., reading a voltage sensor. By topologically sorting the events we obtain a sequence that transitions the platform from the current state into the new state, observes the partial order between the events and therefore also conforms with the sequencing requirements of the platform.

4.4.3 Full power-up sequence

Given a power state for each consumer in the platform and corresponding consumer constraints we can now compute a platform state that satisfies these constraints. Given two of these states we can also compute a correct sequence that transitions the platform from one to the other. The primary chips on Enzian transition through multiple intermediate power states between being off and fully operational or vice versa. This is illustrated in Figure 4.5.

The columns are the CPU's power states and the rows are the FPGA's. Each tile of the grid then represents a potential platform state which satisfies the

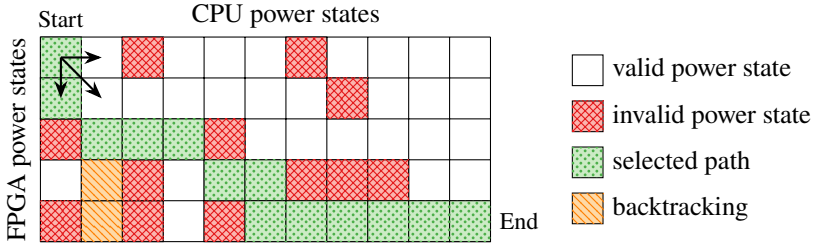


Figure 4.5: Finding a path through the state table.

consumer constraints for the corresponding power state of both CPU and FPGA. The upper left corner is the state when both chips are off and in the lower right one both chips are fully operational. We can compute such a platform state using the approach in Section 4.4.1.

To transition the platform from powered off to fully operational now means finding a path through the grid. In each step we have the choice of only advancing in one of the sockets' power state sequence or both at once. In the grid this corresponds to going right or down vs. diagonally right and down. Some combinations of consumer constraints might be mutually exclusive, i.e., there is no platform state that can satisfy both consumer constraints at once. These combinations are illustrated with a red cross-hatched square in the figure. This means we can also get stuck and need to backtrack to the last tile in the grid where we still had a choice and try a different path. Such a situation is illustrated in the figure towards the lower left with the orange hatched tiles. If we end up with no choices left, it means that the state transition is infeasible. This can either be sign of a badly designed platform or a bug in the model.

Once we have found a path from the origin state to the desired state, in our example from "off" in the upper left to "fully operational" in the lower right, we can compute the partial sequences that transition the platform between tiles using the approach in Section 4.4.2.

We obtain the full sequence by concatenating all the partial sequences from the individual steps.

4.5 Evaluation

I begin the evaluation by showing that using the model and a conventional constraint-satisfaction algorithm, we can indeed generate a working power sequence which successfully configures the voltage regulators of Enzian (Section 4.5.1). I then show that this is not only possible, but that the event sequence can be derived efficiently from the model with reasonable computational effort (Section 4.5.3 and Section 4.5.2). Finally, I provide a user-experience report about the efforts needed in using our tool to support a new hardware platform topology (Section 4.5.4) and adapting it to a new power management interface provided by the BMC (Section 4.5.5).

We built an implementation prototype in Python which is capable of deriving correct power sequences for Enzian. The tool converts the event graph produced by the constraint solver operating from the platform description into a sequence of power management API invocations on the BMC. We executed all performance experiments on a desktop machine with an Intel Core i7-6700K CPU @ 4 GHz and 32 GB DDR4 RAM. We then ran the generated Python program on Enzian's real BMC to configure the power sequence and bring up the processors of the board.

4.5.1 Generating working power sequences

In this qualitative evaluation I demonstrate that our tool is able to generate a working power sequence capable of bringing up the Enzian platform.

We modeled the power tree of Enzian using the above-described semantics and used the tool to generate a power sequence. We then ensured that the platform is turned off, i.e., all voltage rails are off except for the stand-by power. Next, we executed the generated Python program containing the command sequence to power up Enzian. This transitioned the platform from the off-state to the on-state where both sockets are fully powered on and operational. We then verified that the voltages are correctly set according to the specification.

First, we analyze the generated power sequence and the resulting Python program. The initial lines of the power sequence can be seen in Figure 4.6. Overall, the generated sequence consists of 28 discrete steps. These are implemented as several calls to the power management API that executes initiate and

```
1  init_device('isl6334d_ddr_v')
2  init_device('pac_cpu')
3  init_device('pac_fpga')
4  gpio.set_value('C_RESET_N', False)
5  gpio.set_value('C_PLL_DCOK', False)
6  gpio.set_value('B_PSUP_ON', True)
7  wait_for_voltage('3v3_psup', v_min=3.135, v_max=3.465,
       device='pac_cpu', monitor='VMON3_ATT')
8  wait_for_voltage('12v_cpu0_psup', v_min=4.702, v_max=5.197,
       device='pac_cpu', monitor='VMON1_ATT')
9  wait_for_voltage('12v_cpu1_psup', v_min=4.702, v_max=5.197,
       device='pac_fpga', monitor='VMON1_ATT')
10 wait_for_voltage('5v_psup', v_min=4.750, v_max=5.250,
       device='pac_fpga', monitor='VMON2_ATT')
11 wait_for_voltage('5v_psup', v_min=4.750, v_max=5.250,
       device='pac_cpu', monitor='VMON2_ATT')
12 init_device('clk_main')
13 init_device('clk_cpu')
14 init_device('ir3581')
15 init_device('ir3581_loop_vdd_core')
16 init_device('ir3581_loop_0v9_vdd_oct')
17 power.device_write('ir3581_loop_vdd_core', 'VOUT_COMMAND',0.96)
18
19 # more lines follow ...
```

Figure 4.6: First lines of the generated power sequence

wait-for-completion events. This results in a Python program with a total of 74 lines for Enzian. The majority of these lines (61 in total) are directly executing power sequencing actions on the voltage regulators. The remaining 13 lines are initializing the power management framework of the BMC and creating required software objects.

When executing the generated Python program, we observed that the power rails of Enzian were configured correctly and the CPU and FPGA were brought into an operational state.

Comparing the generated power sequence to the manually derived one we observe three key differences:

Ordering The order of the executed steps differs between the manually written program and the generated one. This difference is due to the partial ordering of the sequence steps. Consequently, there are multiple correct sequences to bring up a platform. Our tool is even capable of generating

multiple correct sequences by selecting a slightly different path through the power states resulting in a different power sequence. However, the end state is always the same and thus for the purpose of this work all sequences are equivalent.

Checks The manual sequence always inserts checks to verify that the complete events actually have happened before proceeding to the next step. The generated sequence will do multiple steps in parallel if the sequencing requirements allow it and only then insert checks to verify the steps have completed.

Default States The manual sequence explicitly sets the voltage for every regulator. The generated sequence omits this if our tool could infer from the model that the regulator was already configured correctly at that stage in the sequence.

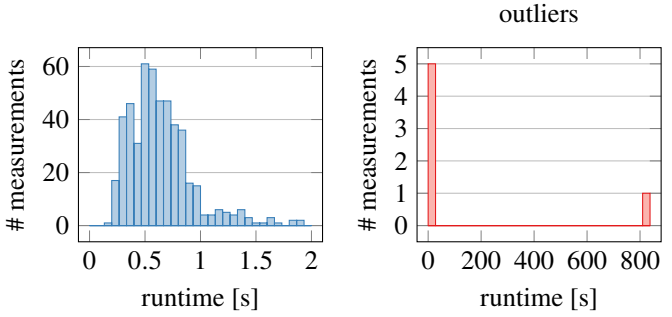
Based on the results obtained in this evaluation we can conclude that the tool is indeed capable of generating a working power sequence that correctly powers up Enzian.

4.5.2 Efficient state generation

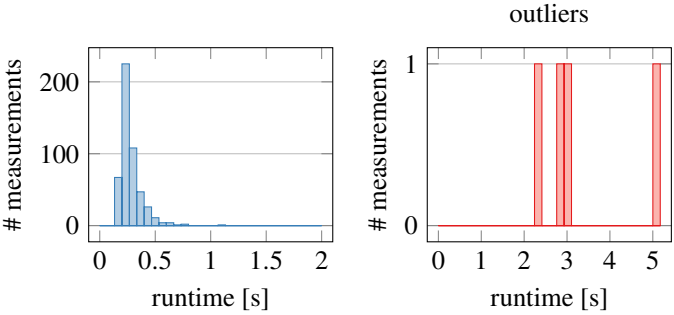
We now quantitatively evaluate the state generation process and will see that computing a platform state satisfying a set of consumer demands is efficient and feasible within acceptable time limits.

We populated the model with the power tree description of Enzian. We then measured the time it takes to evaluate the model and to compute the new platform state for the three combinations of consumer demands listed in Table 4.1. This evaluates the algorithm of Section 4.4.1.

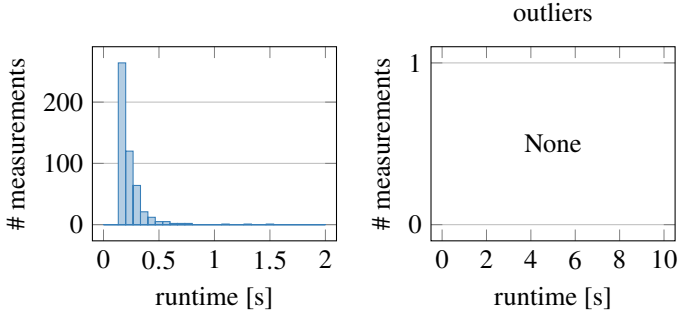
For each problem P1 to P3, we measured 500 runs of the experiment. Note that the constraint solver uses backtracking and thus may explore the search space in a different order each time, depending on the order in which the constraints are presented. To get a better representation of the expected runtime we randomized the order of constraints to compensate for better and worse paths through the search space.



(a) Solving times for problem P1



(b) Solving times for problem P2



(c) Solving times for problem P3

Figure 4.7: Histograms of solving times (to find one solution) for the three problems P1, P2, and P3.

Problem	CPU power state	FPGA power state
P1	Powered on	Powered on
P2	Powered on	Powered off
P3	Powered off	Powered off

Table 4.1: An overview of problem instances P1 to P3

The results are shown in Figure 4.7. For better visibility of the data, there are two histograms: one with the regular measurements showing runtime on the x-axis and the number of runs on the y-axis and one with the outliers above two seconds execution time. For all problems, we observe that the majority of experimental runs completed in less than 1.0 second with just a few outliers.

The complexity of the problems decreases from P1 to P3 as fewer components need to be powered, thus reducing the total number of constraints in the system (P1 has both processors on, while P3 has both switched off). This is reflected in the results of P1 to P3, where the median execution time decreases with fewer powered-on components.

As mentioned in the experimental setup we randomized the order in which the constraints are presented to the solver. The outliers correspond to runs where the solver happened to explore the search space in a particularly inefficient way, such that it had to backtrack more often. We also see a larger number of outliers with more components turned on. This is due, in part, to the DRAM voltages being included in those configurations. While they are showing up as leaves in the power tree, exploring their state is mostly irrelevant to finding a power sequence for initializing the board. However, the general algorithm is not aware of this and can spend time exploring the DRAM voltage regulators resulting in runtime outliers shown in Figure 4.7. A possible solution to this would be to add additional constraints to avoid these types of situations, but we did not explore this option further as the number of outliers is very small.

I have shown that the evaluation of the power state space search algorithm from Section 4.4.1 is efficient, usually taking less than a second, and thus it is feasible to evaluate during runtime in response to user demands for re-configuring the power state of the platform.

4.5.3 Efficient Sequence Generation

We now quantitatively evaluate the time it takes to compute a complete boot sequence or a partial reconfiguration using the algorithm from Section 4.4.3. In other words, I will show that it is possible to efficiently compute a sequence from one platform state to another.

We expressed the states of the main processors as either powered on or off which we call the initial state of the system. Then we toggled the power state of one or both of the chips to obtain the target state. Note that the underlying model captures all intermediate power states including the initial and target states. By enumerating all possibilities we obtain a total of 16 power states, four of which do not change the power state at all and are not of interest. From the remaining twelve states, we can further eliminate the ones where the FPGA is powered without the CPU being powered. This was a constraint on Enzian at the time the experiment was performed. This leaves us with six total combinations of initial and target states shown in Table 4.2 to evaluate. For each configuration P1-P6 we measured the time to generate the transition sequence between the two states. We repeated each measurement three times.

I present the runtime measurements for all six configurations in Table 4.3. Overall we observe that in tendency the runtime grows with the number of transitions, and that ON transitions are more expensive than OFF transitions. Additionally, finding a transition sequence for the FPGA is more expensive than for the CPU. However, even in the worst case, the execution time is less than three seconds.

The dependence on the number of transitions is intuitive: When only one chip has to be transitioned, the state table illustrated in Figure 4.5 collapses to a single dimension and the problem is reduced to computing intermediate platform states. ON transitions are more expensive than OFF transitions as the components on Enzian have more ordering constraints when turning on than when turning off, hence it is more likely backtracking is required. Finally, the power sequence for the FPGA involves more components on the board and finding a correct sequence for it is therefore harder.

With a measured worst case execution time of three seconds, pre-computing sequences offline is certainly feasible. Even online calculation at boot would be acceptable compared to other boot steps such as RAM initialization which can

Problem	Consumer	Initial state	Target state
P1	CPU	Powered off	Powered on
	FPGA	Powered off	Powered on
P2	CPU	Powered off	Powered on
	FPGA	Powered off	Powered off
P3	CPU	Powered on	Powered on
	FPGA	Powered off	Powered on
P4	CPU	Powered on	Powered off
	FPGA	Powered off	Powered off
P5	CPU	Powered on	Powered on
	FPGA	Powered on	Powered off
P6	CPU	Powered on	Powered off
	FPGA	Powered on	Powered off

Table 4.2: Overview of problem instances P1 to P6

Problem	Runtime [s]	#Transitions
P1	2.7	2x ON
P2	1.3	1x ON
P3	1.7	1x ON
P4	0.4	1x OFF
P5	1.3	1x OFF
P6	1.5	2x OFF

Table 4.3: Measurements (average of three runs) obtained for the six different combinations of consumer transitions possible on the Enzian platform

take a couple of minutes to complete.

In this evaluation I have shown that the entire power sequence of a platform can be generated within a few seconds and thus presents a viable option for both offline and online evaluation.

4.5.4 Re-computing sequences for new revisions

New board revisions or platforms have different power trees which must be modeled to generate a power sequence. I now elaborate on our experiences in expressing the Enzian platform using the modeling language.

There are essentially two steps involved: *1)* obtaining the constraints of the different voltage regulators on the board, and *2)* capturing the regulator topology in the power tree. We had to do both steps for both the manually-derived sequence and the model population.

For populating the model, we can independently focus on specifying power-tree topology and the voltage-regulator constraints. In contrast, when manually deriving the power sequence we had to pay attention to timing requirements and other constraints, as well as the power-tree topology. Adapting an existing platform to a new revision can be done by simply replacing the description of a voltage regulator with the new one in isolation without worrying about the effects it has on the power sequencing commands to bring up the board.

When using our tool one can express each regulator in isolation and form the topology step-by-step without worrying about timing and voltage constraints. I expect this approach to be less susceptible to errors than designing the sequence manually.

4.5.5 Adapting the tool

In this part of the evaluation we qualitatively evaluate the user experience, specifically the efforts needed to adapt the tool to the BMC-specific power management interface.

In its initial version, the tool was built against a different firmware image. When upgrading the firmware, the tool was no longer compatible with the Enzian BMC's firmware, and thus needed to be adapted to support this major

change in the power management API. We adapted both the manually derived sequence and our tool to the new interface.

Adapting the sequence manually required consulting various datasheets to obtain knowledge about the sequencing requirements of the various components and how they can be expressed using the API provided by the firmware. The previous sequence did not provide enough information to adapt it and required a significant amount of work reading the datasheets and carefully examining the schematics.

In contrast, we did not have to adapt the model itself for this change in the power management API because all knowledge about the power tree (with its components and constraints) were already encoded in the model. This completely avoids consulting the platform datasheets. All that is left to do is adapting the code generator of the tool to the new API.

Thus, adapting the code generator took roughly a single person-day, while understanding and adapting the entire power sequence manually to the new API consumed over three person-weeks. Our experience shows that supporting a new power management interface resulted in significantly less work than to manually deriving and adapting the bring-up sequences to the new firmware.

4.6 Related Work

As I remarked in Section 4.1, there is a dearth of published work on board management software. Nevertheless, our work is closely related to other, neighboring fields which I discuss here.

The problem of deriving a correct power-up sequence bears some similarity with the problem of a device driver correctly initializing and controlling the operation of a device. Device drivers not only contribute a large amount of code to systems software [93], but are also a significant contributor to bugs and errors [31]. Dingo formalizes driver protocols to make the interaction with devices unambiguous [155].

Writing device drivers is inherently tied to the operating system architecture, but device driver synthesis [156, 157, 198] enables the generation of OS-specific device driver code based on a specification, and thus automatically generating the right control sequence for the device. Beyond drivers, there is early work on

synthesizing most of the hardware-specific parts of an OS based on specifications [85]. Our work similarly applies program synthesis techniques to derive power sequences for the BMC.

Inside the OS (as opposed to BMC firmware), constraint solving has been applied to a variety of OS techniques both online and offline to select a whole-system configuration which satisfies current requirements.

For example, the problem of configuring PCI Express devices under a set of root complexes has been expressed in Prolog and solved using constrained logic programming techniques [164]. Similar methods have been applied to data center network configuration [137] or synthesizing cluster management code in distributed systems [181]. Spex [204] goes further by attempting to infer configuration constraints from the program source code, which is not possible in our case. Cocoon [154] uses a hierarchical design process to specify the configuration of software defined networks to obtain a correct-by-construction initialization of the network controller.

Outside the field of OS design, software-based industrial control systems consist of a controller and a plant forming a closed loop system; software running on the controller must correctly configure the plant. QKS synthesizes correct-by-construction control software from the specification of a plant model, an implementation specification and the safety and liveness requirements [7, 128].

Closer to our goal of platform power management, for modern servers and phones it is usually the OS's responsibility to implement power management *policies* – deciding which components to power on or turn off is important to minimize the power requirements. Xu et al. [204, 203] argue for a centralized power management agent, which decides when devices should switch between the discrete enabled or disabled states based on quality of service (QoS) requirements and specification of power states. QoS can also be used by agents in embedded systems to automatically find appropriate dynamic power states [72]. Benini et al. [20] provide a survey of design techniques for system-level dynamic power management.

In contrast, our work is not trying to decide *when* devices should transition to different power states but provide help in *how* these transitions are implemented at a lower level.

We are also not the first ones to apply more formal techniques to power management: Gupta et al. [75] applied formal methods to dynamic power

management with the goal to minimize the overall power consumption, which as already stated above is policy that could be implemented using our mechanisms. p-FSMs model system-level power management including control mechanisms and operating states [172]. Like us, the authors argue that the application of formal methods is essential to cope with the complexity of system-level power management in order to meet energy, power, and thermal constraints. They focus on the design of power management systems, and it is not clear whether their model is able to handle individual regulators as is required for our work. While they model-check their representation of an SoC, they do not demonstrate controlling physical hardware with their technique.

Other approaches focus on providing an interface between the main OS and the BMC or other power sequencing functionality. The Advanced Configuration and Power Interface (ACPI) [192] defines mechanisms to control the power state of the entire system, e.g., transitioning to sleep or waking up. Moreover, the ACPI tables include information about the power states of the motherboard devices and their connections, including methods to change the power state of the devices. Like the work mentioned above, ACPI operates at a higher level than our tool: it provides the OS on the CPU(s) of a platform with information about the power management capabilities of the platform but does not deal with how those capabilities are implemented.

Similarly, Devicetree [55] provides information about the hardware platform such as device addresses, amount of memory, processors, and existing power and clock domains to system software. The OS uses this information to find the device and its power and clock domains. However, information about power domains encoded in the devicetree does not include voltage levels or supported input/output voltages and is thus not suitable for our purpose.

4.7 Conclusion

In this chapter, I have shown how we applied computer science techniques to an understudied problem in building and operating a computer: how to turn the machine on in a safe and efficient manner.

While not well-known in the systems community, the power sequencing problem is real and becoming more significant as systems become increasingly com-

plex, and the consequences of getting it wrong become more serious (whether these consequences are security vulnerabilities or permanent damage to the hardware).

I have shown that generating a correct power-on sequence can be reduced to a constraint satisfaction problem, and that even a relatively unoptimized solver can compute a solution for a realistic, complex server in a relatively short amount of time – to the extent that it would be practical online at boot time. I will discuss further improvements to the sequence generator in the next chapter.

However, this solution can only be achieved if the problem can be posed to the solver in a suitable manner. Consequently, I have presented a representation of a machine's power tree that captures both the detailed topology of a modern server platform, and the behavior of individual power regulators and other components at a sufficient level of detail to generate useful results. I look forward to the release of more well-documented open hardware on which to evaluate our approach.

Even with a single system design though, our direct experience has been that this effort to create a more general solution has already paid off. We were prompted to explore it by the effort (and nerve) required to create a power-on sequence manually and interactively, and by the lack of any existing automated solutions to this problem, regardless of whether the resulting sequence was to be executed in software by a BMC or programmed into hardware in a CPLD. Having done the work to model hardware platforms, I am confident that applying it to another server design would be both valuable (in time saved) and low-effort.

All the code, including the power topology model for Enzian, is available as open source[1].

[1] https://github.com/Sockeye-Project/decl-power-seq

5

Dynamic Power Management

In the last chapter we have seen how we can generate power sequences from a platform model. The presented approach works well for generating sequences offline. For managing a platform at runtime however, we need to be able to react to changes in the hardware. If, e.g., the current draw on a power rail is too high, we need to be able to bring the platform back into a safe state. One way of achieving this is to also use our model at runtime. Once we detect that the platform has entered a degraded state, we could then generate a sequence on demand that transitions the platform back into a safe target state. For this we need two things. We need to be able to infer the platform model state that the hardware is in from sensor readings. We then also need faster sequence generation time to be able to react to the changes in time. There is however a caveat: The BMC is connected to the regulators via low speed buses like I^2C. A current spike can destroy expensive hardware much faster than we can react in software. For the immediate response we will therefore always need the regulators.

In this chapter I am going to present additions to the model from Chapter 4 and a new algorithm for state generation in order to explore the possibility of online sequence generation. In the end I will also present an alternative solution for a power manager and discuss the trade-offs.

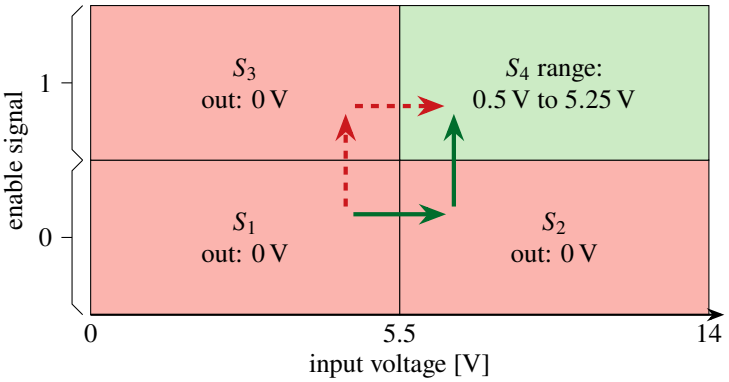


Figure 5.1: Updated regulator model. The arrows depict the partial order between states. The solid green arrows mark the natural sequencing path. The dashed red ones the alternative path we exclude to turn the partial order into a total order.

5.1 Changes to the model

We leave the structure of the model described in Section 4.3 unchanged: a directed graph with components and nets as nodes and edges between components and nets if the component is connected to a net. A net can also still only be driven by a single component i.e., a net always connects to exactly one output of a component and potentially many inputs of other components. We will however change the component model.

First, we introduce a partial order on a component's states. The states are ordered from lower power states to higher power states. What constitutes a lower power state is component specific, but usually it means fewer of the component's inputs are powered or enabled. In Figure 5.1, the partial order is as follows

$$S_1 < S_2 < S_4 \qquad \text{and} \qquad S_1 < S_3 < S_4$$

State transitions can only happen along the edges of the graph induced by

the partial order but both from smaller to larger, and from larger to smaller states. A *monotonic sequence* is a sequence where each component only either transitions towards higher states or lower states. A sequence is still monotonic if some components transition upwards and others downwards, as long as no components change direction.

We also allow states to be excluded. The more natural way to sequence the component depicted in Figure 5.1 is to first power it and then enable it. To model this, we can exclude state S_3 which turns the partial order on the component states into a total order. Total orders on component states are desirable as they simplify the sequencing problem by reducing the search space. However, they also restrict the expressiveness of the model. Enzian can be modelled with all components having total orders on their states and this is likely the case for most platforms. However, with no other platform available with the necessary documentation, I could not validate this assumption.

The second change we introduce to the model is a restriction on the requirements that component states can impose on nets. We impose that the requirements for a net can only change in one of the transitions between states. For regulators (non-leaf components), this for one means that every input axis can only have a single transition point (this is already the case in Figure 5.1). It also means that the output capabilities can only change once, usually from "no output" to "output in a specified range". These requirements are too restrictive for consumers: consider a consumer with an "off" state, an intermediate power state and a high power state. Further, assume that there is an input which needs to be off for the "off" and high power states but on for the intermediate power state. With the single-change requirement, this consumer could not be modelled. While this might be a constructed example, these requirements also prohibit an input from needing a low voltage for a lower power state and a higher one for a higher power state. This would exclude dynamic voltage and frequency scaling (DVFS) power states. We therefore do not enforce this single-change requirement on consumers. A *single-change sequence* is a sequence where all components obey the single change criterion. A platform with components that do not obey the single-change criterion can still have single-change sequences between states that do not need any requirement on nets to change more than once. All regulators on Enzian fulfil the single-change requirement. Again, while not validated, this is likely the case for most platforms. Enzian's

main consumers, the CPU and FPGA, do not support DVFS and also fulfill the criterion.

In the next section we will see, how monotonic and single-change sequences help improve the sequence generation time.

5.2 New algorithms

As in described Chapter 4, there are several steps to compute a full power-up sequence: First, we need to extend a partial platform state with constraints for the consumers to a full platform state. For this step, I will present a new algorithm in Section 5.2.1. We then need to compute a sequence to transition the platform into this new state. We do this with the same algorithm as before (Section 4.4.2). This algorithm however can only find sequences between two states where any given net only changes its value once. There is nothing however, that guarantees this in the case when multiple consumers transition through a sequence of intermediate states to their desired target states. In Chapter 4 we therefore needed to piece together the sequences between intermediate consumer power states. As described in Section 4.4.3, this involved finding a feasible interleaving of consumer power state transitions. Monotonic single-change sequences however do fulfil this requirement. We can therefore generate any such feasible sequence between two platform states in a single invocation of the sequence computation algorithm. This also means that given the current platform state and some target consumer states we also only need a single invocation of the state computation algorithm. On Enzian all sequences are monotonic and single-change. However, even on platforms that do, e.g., support DVFS, the requested transitions would normally be between two DVFS states. Such a transition is still feasible with a monotonic single-change sequence. Only transitioning through several DVFS states in the same sequence would need a fallback to the approach in Section 4.4.3. In practice, such sequences are however unlikely to be useful.

5.2.1 Computing the platform state

Compared to Chapter 4 we recast the problem of computing platform states as an integer linear program. Similar to a constraint satisfaction problem a linear integer program is defined by a set of integer variables and linear constraints on these variables. Additionally, an integer linear program has a cost function consisting of weights for the variables. The cost function is computed by summing up the products of variable weights and variable values. Integer linear program solvers can then find solutions that optimize (in our case minimize) this cost function.

The variables in our program are the states of the component. There is a variable $s_{i,j}$ for each state S_j of each component C_i. The variables are boolean, so $0 \leq s_{i,j} \leq 1$. The first constraints we add are to ensure that each component is only assigned exactly one state. For each component C_i we add a constraint of the form

$$\sum_{j=1}^{\text{number of states}} s_{i,j} = 1$$

Next, we need to make sure that the states of components connected to the same net are compatible. So for each pair of components C_1 and C_2 that are connected to the same net (input or output), we exclude combinations of states that are incompatible. Two states $s_{1,j}$ and $s1, k$ are incompatible if they have non-intersecting requirements for the same net. For all of these we add constraints of the form

$$s_{1,j} + s_{2,k} \leq 1$$

Lastly, we need to make sure that the consumers are in the desired states. For each consumer C_i that needs to be in state S_j we add a constraint of the form

$$s_{i,j} = 1$$

The weights for the cost function we assign such that lower power states (according to the partial order on component states) are preferred. This is to prevent a solution from arbitrarily turning on regulators if they are not actually needed to satisfy the consumer state requirements. Concretely, the weight $w_{i,j}$ for state S_j is then assigned as follows

$$w_{i,k} = d_j \cdot v$$

The value d_j is the distance of state S_j to the closest minimal state in the directed graph induced by the partial order (0 for minimal states). The value v encodes the strength of the preference for lower power states. We used 1000 in our prototype.

We recover the platform state from a solution to this integer linear program by assigning each component C_i the state S_j for which $s_{i_j} = 1$. While solutions to the constraint satisfaction problem described in Chapter 4 directly yielded the values for the nets, we need an additional step here: for each conductor, we find the valid range of values by intersecting the requirements that the states of the connected components impose. We can then, e.g., pick the midpoint of the interval to maximize the margins for the net.

The motivation for recasting the problem as an integer linear program is that we can easily encode the preference for lower power states for components using the partial order on states that we introduced. The additional step to recover the full platform state from the solution of the integer linear program can be efficiently computed: for each net we only need to intersect as many intervals as there are components connected. The less efficient part is adding the incompatibility constraints for component states on the same net: here we need to pairwise compare the requirements of all states of the connected components. This however, only has to be done once and can be pre-computed. We will see its impact on solving time in the next section.

5.3 Evaluation

We implemented a prototype of the new sequence generator in Rust. To estimate the new implementation's applicability to online power management we need to know how fast it can generate sequences and how well it scales with platform size.

We therefore conduct the following experiment: We create a synthetic platform parametrized in the number of sockets. The power tree for each socket closely resembles the CPU socket's power tree on Enzian. We instantiate the platform with 1 to 10 sockets and generate sequences to transition the platforms from an all off state into an all on state and vice versa. We run the generation for each platform 5 times and record the average running time for the differ-

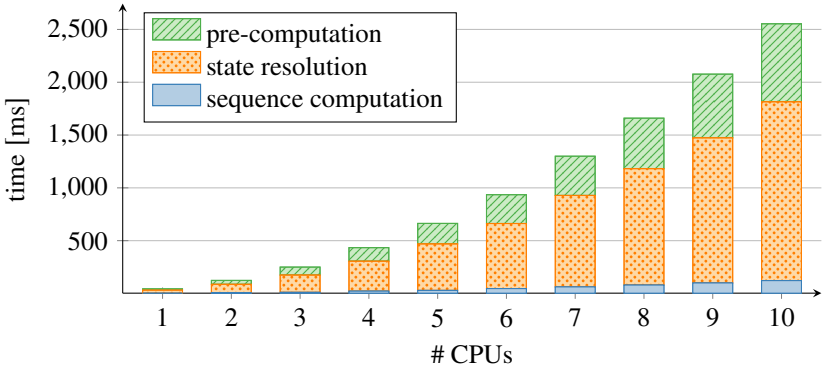


Figure 5.2: Scaling behavior of sequence generation time with number of sockets

ent stages of the sequence generation: pre-computation, state resolution and sequence computation. For real-world applicability of the results we run the experiment on the Enzian Zynq-7000 BMC with two Armv7 Cortex-A9 cores and 1 GB of DRAM.

The results are shown in Figure 5.2. I only show the plot for the power-on sequences as there is no performance difference between the two directions. Error bars are also omitted as the variance is negligible. We can see that the pre-computation always takes about 30% of the time, state resolution about 65% and sequence generation 5%. For up to 4 CPUs sockets, the sequences can be generated in under 0.5 s and for up to 6 in under 1 s. For 10 sockets the generation takes about 2.5 s.

The new implementation can generate sequences for moderately sized systems with 1 to 6 sockets in under a second. The pre-computation time only has to be spent once and can be done in advance, e.g., at BMC boot time. Given that emergency fault responses have to be implemented in the regulators themselves anyway, this is acceptable performance. The sequences we generate here correspond to the transitions for P1 and P6 in Section 4.5.3. However, our synthetic platform has only CPU sockets and the FPGA socket on Enzian is more complex to sequence. Nevertheless, the prototype implementation presented in

this chapter can generate sequences for a platform with 7 sockets in the same time the implementation from Chapter 4 can generate a power-off sequence for Enzian, a two socket system. Looking at power-on sequences this increases to 10 sockets. Additionally, the experiment here ran on much less powerful hardware than the experiments in Chapter 4. I speculate that a significant amount of the performance boost is due to the choice of programming language (Rust vs. Python). Another important contribution is the insight that for practical sequences we do not have to perform piece-wise sequence generation as described in Section 4.4.3. This avoids backtracking and multiple invocations of the state computation algorithm.

5.4 Conclusion

We have seen that we can push power sequence generation to sub-second runtimes for moderately sized platforms. Running sequence generation online however, also comes at the cost of significantly increased complexity in the power manager. Even though we did not *prove* our sequence generator to be correct, we are confident that any sequences generated adhere to the specifications. There are however cases where sequence generation can fail, e.g., if there is no valid platform state to satisfy some combination of consumer constraints. Furthermore, integer linear programming is an NP-complete problem. Solvers use heuristics to find solutions for many problem instances fast. However, it is possible to run into a case where the solver takes a much longer time to find a solution. While this did not happen in the evaluation for this chapter, we have seen such effects in Chapter 4 (see the outliers in Figure 4.7). In an offline scenario, these issues are not severe, as long as the generation is sound, meaning no incorrect sequences are generated. However, in an online scenario they can lead to unpredictably high reaction times to incidents. I therefore advocate for an alternative solution to build a model-based power manager.

5.4.1 An alternative to online sequence generation

To avoid having to generate sequences online, sequences could be pre-generated offline. However, generating and storing sequences to be able to transition

between any two platform states is prohibitively expensive for realistic server platforms. This is especially true as we do not just need sequences between valid states but also from degraded states back to valid platform states. I therefore propose the introduction of *checkpoint states*. These are user defined states where pausing power sequencing makes sense. For Enzian these states are, e.g., the following

1. Everything off
2. Regulators that are shared between the CPU and FPGA on
3. CPU on
4. FPGA on

The last two states are only partial platform states. However, because the CPU and FPGA do not share any power topology beyond what is already turned on in state 2, they can be turned on independently once state 2 has been reached. The parts of the platform that are not shared between the two sockets can therefore be treated like sub-platforms, with their own states.

To turn Enzian on, the platform now first transitions to state 2. Once the shared power topology is on, the sockets can also transition into their powered states. These 3 power-on sequences and the 3 corresponding power-down sequences can be generated offline. In fact, this is how the generated sequence in the current Enzian power manager is partitioned. If a fault occurs in, e.g., the CPU specific part, we can now just run the power-down sequence for the CPU socket, to transition back into a state where the degraded part of the platform is powered down. If a fault occurs in the shared part, we would first need to run the power-down sequences for both sockets before powering down the shared regulators. By picking more checkpoint states we can introduce finer-grained fault handling. Crucially however, with this approach we do not need to exactly know, in what state the platform is after the fault occurred. Instead, we just reset the fault part by powering it down. Furthermore, this approach keeps the number of required sequences low, making it possible to generate them offline. I believe that this solution strikes a good balance between flexibility at runtime and keeping unpredictable behavior out of the runtime power management stack.

Implementing and analyzing the behavior of this alternative solution remain however future work.

With the approach presented in this and the last chapter we can generate correct power sequences. In this chapter I also discussed options for building a complete power manager using the power topology model. To make sure that the sequencing instructions are correctly communicated to the hardware however, we also need reliable drivers for the regulators. In Chapter 6 I will present a framework for generating provably correct drivers for this purpose.

6

A Trustworthy I^{2}C Stack

We have seen in Chapter 4 how crucial correct power management is. However, the correctness of the power sequence does not matter much without a driver stack that correctly confers sequencing instructions to the power regulators. These regulators are predominantly controlled over an inter-chip protocol called I^{2}C. In this chapter, I present Efeu, a system for generating drivers for complete I^{2}C subsystems from formal specifications. The resulting software stacks are not just suitable for server BMCs, but also for embedded controllers in mobile phone SoCs, or resource-constrained internet of things (IoT) devices. Moreover, the I^{2}C drivers are high-performance and verified to behave correctly using a model checker, even when the system includes devices which *do not correctly follow* the I^{2}C standard. Finally, the generated drivers can be in C, or Verilog for FPGAs, *or a hybrid of the two*, enabling efficient determination of the optimal hardware/software split for a given platform.

The I^{2}C protocol is at the heart of almost all modern computer systems and critical to their correct behavior. Bugs in I^{2}C can result in inefficiencies in energy usage, hardware lockups, and in some cases permanent hardware damage. A correct I^{2}C network within a phone or server is essential.

At the same time, I^{2}C has features that make creating high-assurance driver software particularly challenging. It is a bus-based protocol but unlike, say, PCIe or USB it does not feature hardware facilities for isolation. This means that a bug in the driver for a single device can disable the entire subsystem at runtime. Unfortunately, an I^{2}C subsystem for a typical machine includes dozens of devices from many vendors, which (like PCIe) often exhibit *quirks*: deviations from the standard which can confuse other devices or controllers.

For this reason, hardware I²C controllers may interoperate with only a limited number of empirically compatible devices, and are thus frequently replaced with "bit-banging" drivers which directly manipulate bus signals from handwritten software. This impairs protocol performance, increases CPU load, and reduces energy efficiency.

Despite this, the design and implementation of a trustworthy I²C software stack has received relatively little attention from the research community. Efeu addresses this challenge.

Efeu provides a language for specifying I²C devices, which includes the ability to express known deviations from the protocol (quirks) by the devices. A complete hardware platform's I²C network can be expressed by composing such specifications within the language.

The Efeu compiler can then generate a driver for the I²C host controller together with drivers for the all the attached devices. This generated driver suite can be in C, or alternatively in Verilog for synthesis onto an ASIC or FPGA.

Crucially, the Efeu compiler can place boundaries between generated software and hardware functionality *between any two layers* in the I²C protocol stack. In its simplest form, this enables the optimal split between hardware and software implementation to be empirically determined, without writing any additional code. It can also be used for debugging, for example by replacing an optimized hardware layer with a more instrumented software layer.

The Efeu compiler also generates a specification of the complete I²C subsystem (with quirks) in Promela [176], allowing correctness to be model-checked using SPIN [81].

This is our second attempt at engineering verified I²C stacks, following a somewhat simpler previous approach [86]. Efeu uses the same Promela specifications that the lower 3 layers (`Symbol`, `Byte`, `Transaction`) of the I²C stack are verified against, but is otherwise completely new. In contrast to our previous approach, Efeu is designed to enable the verification of realistic I²C topologies with multiple attached devices, and targets the generation of both realistic software *and hardware* components for implementing efficient, usable I²C stacks. We provide a detailed evaluation of the verification and driver performance in Section 6.3 and Section 6.4 respectively.

In the next section, I elaborate on why I²C matters, what makes it different from driver assurance for interconnects like PCIe and USB, and the canonical

structure of an I^{2}C stack. Following this, in Section 6.2, I describe the Efeu language, compiler, and verifier, and how it addresses the challenges I have laid out. In Section 6.3 I explain how the I^{2}C stack is model checked, and show that it can be done in practical time, and in Section 6.4 I show that Efeu allows a range of different trade-offs in hardware/software implementations to be generated from the specification of a real hardware platform, and also that the resulting drivers are comparable with hand-tuned software and hardware in terms of throughput, CPU cycles, and hardware footprint. I survey the broader landscape of high-assurance device drivers in Section 6.5, and conclude in Section 6.6.

6.1 Background and problem statement

In this chapter, I address how to create high-performance, correct driver stacks for bus-based devices, in particular those using I^{2}C. By high-performance, I mean competitive with state-of-the-art handwritten drivers in terms of throughput, latency, and CPU usage. By correct, I mean that the driver is proven to function as specified and not interfere with other devices sharing the bus.

Driver defects have long been identified as a major cause of system failures and vulnerabilities [31, 69, 148], resulting in much work on improving driver assurance via synthesis of drivers from specifications, post-hoc verification of manually written drivers, and formally-derived hardware/software co-design. I survey this work in Section 6.5, but focus here on what makes the high-assurance I^{2}C case different.

6.1.1 The importance of I^{2}C and related protocols

Despite receiving much less attention in the literature than devices using, e.g., PCIe or USB, I^{2}C and the related protocols SMBus [160] and PMBus [182, 183] are fundamental to the operation of almost all computers today, from small IoT devices through mobile phone SoCs and platforms to large-scale servers and rack-scale systems [201].

Whether controlled by the conventional OS kernel, or by "hidden" parts of the *de facto* OS [65] like monitor code or BMC firmware, I^{2}C is the base protocol used to control almost all the components of a machine: configuring voltage and

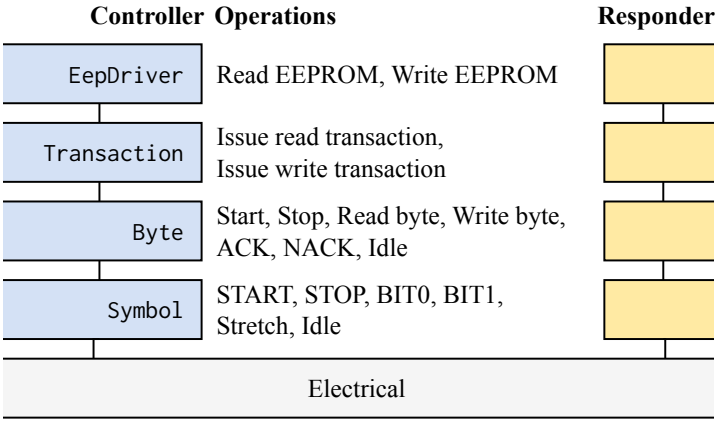


Figure 6.1: The I²C stack with an EEPROM driver as an example application.

clock frequency, monitoring temperature and power, etc. I²C bugs lead to board lockups, pathological power inefficiencies, or at worst hardware damage [23, 5, 30].

6.1.2 What makes I²C different?

I²C [88] is a serial bus protocol using two wires, the6 serial clock line (SCL) and the serial data line (SDA). An I²C device is either a *controller* or a *responder*[1] (Figure 6.1). Controllers initiate and control data transfers between themselves and responders, identified by 7-bit addresses. An additional bit distinguishes read transfers (a responder transmits data to the controller) and write transfers (the controller transmits data). SDA is driven by the transmitting device while SCL is normally only driven by controllers, but responders can "stretch" the SCL clock if they cannot keep up.

What makes I²C specifically challenging is the need for interoperability. I²C connects many components in an SoC or motherboard, and these are designed

[1] The standard refers to them as "master" and "slave"

and supplied by many different vendors. A correctly functioning I^2C subsystem depends not only on the driver for each individual device being correct, but also on all the devices and drivers interoperating correctly.

Of course, interoperability is not a challenge restricted to I^2C – USB and PCIe devices must also work together, for example. However, both USB and PCIe controllers by design provide isolation in hardware between devices, such that drivers do not need to be aware of the whole protocol stack. Even so, deviations from standards are common: the Linux kernel contains over 6000 lines of code for handling so-called *quirks* [164, 119] in PCI(e) devices alone.

Interoperability is different for I^2C devices. The bus-based nature of the protocol means that a misbehaving device or controller can prevent all other I^2C devices on the bus from working. A single driver bug can render the entire I^2C bus unusable. Unfortunately, like PCIe, numerous I^2C devices have quirks [6, 51]. For example, the KS0127 video decoder [103] expects its "stop streaming data" command to appear in a non-standard position [118] and blocks the bus indefinitely if the controller is unaware of this.

This means that a "one-driver-one-device" approach to high-assurance drivers is insufficient. It also means that any formal approach must be able to handle device quirks.

Moreover, in practice I^2C controllers are *often implemented in software*. A hardware implementation is usually provided, but often unused due to a lack of confidence in interoperability. The Raspberry Pi I^2C controller, for example, does not correctly handle clock stretching. Controller and responders can thus desynchronize, leading to lockups and data corruption [130]. This issue is *not* a driver bug in the traditional sense: the I^2C controller driver can correctly program the controller and, as long as no responder uses clock stretching, the system works correctly.

Consequently, many I^2C controllers consist of low-level "bit banging" software directly driving the SCL and SDA signals, even if a hardware controller is available. This allows post-hoc workarounds for quirks, but has a cost. While I^2C bandwidth requirements are relatively modest this still results in slowdown (see Section 6.4.2) and the heavy use of CPU cycles (and associated energy) becomes an issue for low-end embedded devices. Recently, some vendors have proposed using reconfigurable logic to help with I^2C functionality [175, 63].

The bug in the Raspberry Pi also shows that relying on hardware manufac-

turers to fix bugs once discovered is not a solution: the bug was originally discovered in the first Raspberry Pi model in 2013. Newer models released in 2020 are however still affected by it [94].

6.1.3 The I²C protocol stack and ecosystem

I now describe the I²C protocol bottom up (paraphrasing the standard [88]), alongside our model of the protocol (Figure 6.1). I show a complete, end-to-end example in Figure 6.2. Both controllers and responders have the same layers described below, but differ in their implementations.

At the **`Electrical` layer**, both SCL and SDA have external pull-up resistors, and devices may only drive the lines low. Multiple clock speeds are defined, but Efeu targets the commonly used Fast Mode (400 kbit/s, or 400 kHz SCL). In the Efeu model, the `Electrical` layer represents the levels with 0 and 1 and models the pull-down behavior with bit operations. We do not model the precise bus timing, but assume a bus adapter that translates bits into half cycles on the bus, allowing the stack to work with discrete time. Currently, this adapter is written by hand, but could be synthesized using a hardware/software co-design approach like Chinook [34].

The **`Symbol` layer** converts between I²C symbols (START, STOP, BIT0, and BIT1) and the SCL and SDA electrical levels using the encoding in Figure 6.2. Two further operations, IDLE and STRETCH, are defined: IDLE is a no-op to the bus, and STRETCH performs clock stretching, pulling SCL low for one cycle. This is the only operation with which a responder can drive SCL. Otherwise, responders passively respond to clock cycles. In our model, controllers handle clock stretching at the `Symbol` layer, waiting for its completion before returning to the upper layers.

I²C is byte-oriented. The **`Byte` layer** encodes and decodes bytes to and from bits, as well as acknowledging each byte or not: ACK is encoded to the BIT0 symbol and NACK to BIT1. `Byte` also detects arbitration loss if multiple controllers collide on the bus, reporting this upwards.

Above this at the **`Transaction` layer**, a transaction starts with a START symbol, the 7-bit target device address, and a read/write bit. Payload bytes follow, supplied by either the controller or the responder depending on the transaction type and are ACKed or NACKed by the receiving device. A transaction is ter-

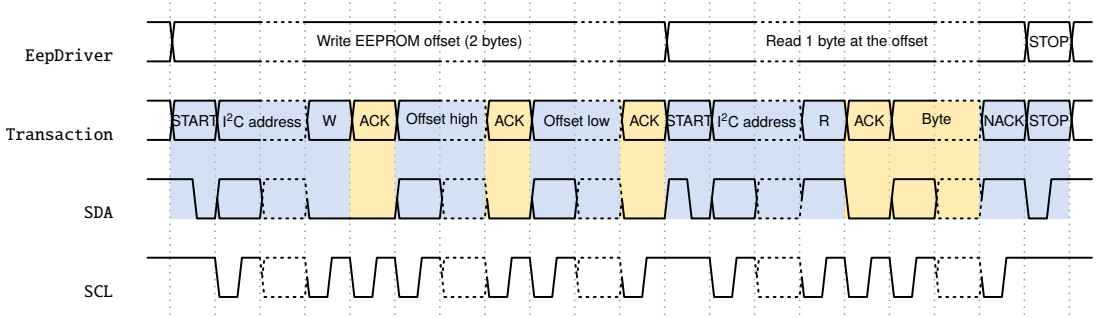


Figure 6.2: Timing diagram of a 1-byte read at a given EEPROM offset. SDA is driven either by the controller (blue) or the responder (yellow) in a cycle. SCL is always driven by the controller. Dashed levels indicate more than one cycle.

minated by a STOP symbol, or another START – known as a repeated START – in which case the controller keeps the bus busy without releasing it.

The top layer is specific to a responding device class. I use the **`EepDriver` layer** as a running example here, modeling a byte-addressable EEPROM, the Microchip 24AA512 [1]. The controller `EepDriver` issues transactions to perform EEPROM *operations*. To write data to the EEPROM, a write transaction is issued with a two-byte data offset followed by the payload bytes to write starting from the offset. To read data from the EEPROM, `EepDriver` first issues a write transaction carrying the data offset, followed immediately by a read request to stream out data starting from the offset. Figure 6.2 shows the timing diagram of reading 1 byte from an offset.

6.2 Efeu design and implementation

The workflow of Efeu is shown in Figure 6.3: a developer writes the implementation specification (devices and topology) of the platform, and Efeu translates it into a Promela model for model checking and iterative refinement. When they are satisfied with the specification, Efeu generates implementations in C and Verilog. While currently focused on I²C, Efeu is generic enough to be extended to other bus-based protocols.

6.2.1 Specifying the driver stack

The structure of Efeu specifications follows the one we developed in previous work [86]: developers write specifications top-down, and then defining each of them as an indefinitely-running finite state machine (FSM). The layered structure makes components reusable across specifications (see Section 6.3).

Some design decisions in our previous approach were however unsuitable for generating real-world drivers. A major one was to fix the direction of communication between layers at specification time: the layer that `calls` another layer initiates communication and the other layer responds by `yield`ing. As I detail in Section 6.2.3, this lacks the flexibility needed in real world settings. Communication primitives in Efeu specifications are therefore symmetric, and

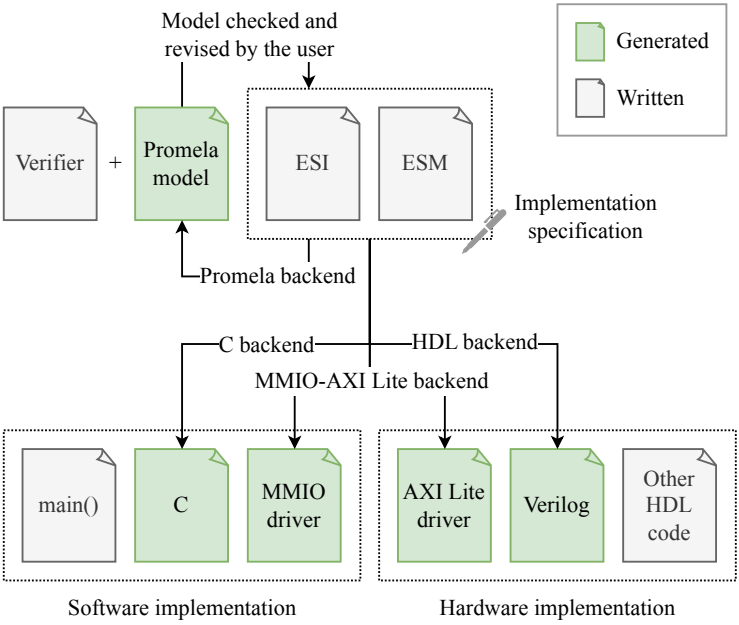


Figure 6.3: Efeu workflow.

```
1  layer CTransaction;
2  layer CEepDriver;
3
4  enum CTAction {
5    CT_ACT_WRITE,
6    CT_ACT_READ,
7    CT_ACT_STOP,
8    CT_ACT_IDLE,
9  };
10
11 enum CTResult {
12   CT_RES_OK,
13   CT_RES_FAIL,
14   CT_RES_NACK,
15 };
16 interface <CTransaction, CEepDriver> {
17   <= {
18     CTAction action;
19     u8 addr;
20     u8 length;
21     u8 data[16];
22   },
23   => {
24     CTResult res;
25     u8 length;
26     u8 data[16];
27   }
28 };
```

Figure 6.4: ESI for controller `Transaction` and `EepDriver` layers and their interface. "<=" and "=>" define the channels from `CEepDriver` to `CTransaction` and vice versa.

the compiler chooses a suitable implementation for the desired scenario. To enable this, the layer declarations also need to include interface declarations. We developed a new lightweight DSL for this called ESI (Efeu System Information).

Figure 6.4 shows an ESI example controller `Transaction` and `EepDriver` layers and the interface connecting them. Interfaces consist of a channel in each direction. In a channel, each data field has a type and a name. Supported types include `bit`/`bool`, unsigned byte (`u8`), 16-bit and 32-bit integers (`i16` and `i32`), enumerations, and 1-dimensional arrays.

Layers are then specified as FSMs in another DSL. In our previous approach we focused on verification and designed the specification language to be easy to translate to Promela. As the new communication model in Efeu required us to change the specification language for layers anyway, we decided to instead make it resemble a subset of C. This improves its usability for writing specifications of real systems by allowing existing tools like syntax highlighting, formatting and static checking to be reused. The new DSL is called ESM (Efeu State Machine) and differs from C as follows:

- The only built-in types are `bit` and `bool` (one-bit), `byte` (or `unsigned char`), `short` and `int`.

- ESI Interface definitions become `structs`; no other `struct` definition is allowed.
- ESI enumerations become C `enum`s; other `enum`s are allowed, but unlike C, corresponding integer values may not be specified.
- Only the unary operators plus (+), negate (-), bitwise not (~), and boolean not (!) are supported.
- The only control flow statements supported are `if`, `while`, and `goto`.
- Each layer is an indefinitely-running function without `return`. No other function definitions are allowed.
- Promela [176] reserved words like `len` and `timeout` are also reserved in ESM.
- ESM supports no pointers, global variables, functions, or variable initialization at declaration time.

Within each layer function, two language primitives, `talk` and `read`, are used to communicate with adjacent layers. Their function stubs are generated by the Efeu compiler. Given two adjacent layers `A` and `B`, `A talk B` is a blocking *round-trip* communication over the interface between `A` and `B` resembling two coroutine switches [101, 129]: values are sent from `A` to `B`, and the operation continues in `A` when `B` issues a corresponding `B talk A` with return values. `A read B` only differs in that no initial values are passed to `B` but `A` waiting for synchronization.

6.2.2 Efeu compiler overview

The Efeu compiler ESMC compiles ESI and ESM code to various targets: Promela code for model checking, C code for software drivers, and/or Verilog code for hardware drivers. Previous work pioneers translating C-like languages to Promela [82, 92] and Verilog [153, 125, 87, 112]. Efeu adopts the idea but uses a single unified specification language.

ESMC is built on Clang/LLVM [107, 123, 35] and leverages existing components. ESMC adds 7823 lines of C++ code (excluding blank lines and comments) to Clang/LLVM, together with 136 system tests to cover this code. ESI files use a custom lexer and parser to parse them an internal representation. ESM code is processed by the Clang frontend, which performs type checking and constructs an abstract syntax tree (AST). Any errors, warnings and/or comments are reported to the user in a readable format through the Clang diagnosis engine [36]. By reusing Clang, ESMC inherits formatted diagnostic messages and C preprocessor support, enabling conditional compilation, compile-time polymorphism, and modular design. The backends operate on the Clang AST.

6.2.3 C Backend

Efeu generates C that can then be compiled into executables or libraries. Recall that in ESM, layers are indefinitely-running FSMs written as functions without return. A straightforward implementation option would transform these functions into threads and the `talk`/`read` operations into inter-thread communication, but this introduces scheduling overhead and dependence on the OS-specific thread implementation.

Instead, we implement layers as stack-based coroutines purely in C with minimal runtime support required, ensuring portability across systems and highly efficient switching between layers.

In principle, to implement two connected layers as stack-based coroutines, either one can be the callee. However, in real applications, as generated drivers are integrated with the rest of the OS, the choice of which layers becoming callees affects the usability of the generated code. We therefore introduce the concept of a *call graph* in the C backend and allow the developer to specify an *entry point* to this graph at compile time, which the compiler provides a function interface to.

Figure 6.5 shows three examples. In the leftmost one, when the generated code is intended to be used as a driver library, it is naturally invoked with the entry point as a top-level function. Efeu performs a depth-first search (DFS) on an undirected graph where nodes are the layers and edges are the connections. The `talk`/`read` operations on the forward edges become function calls, and the reversed ones become continuations. Code generated in this way can be directly

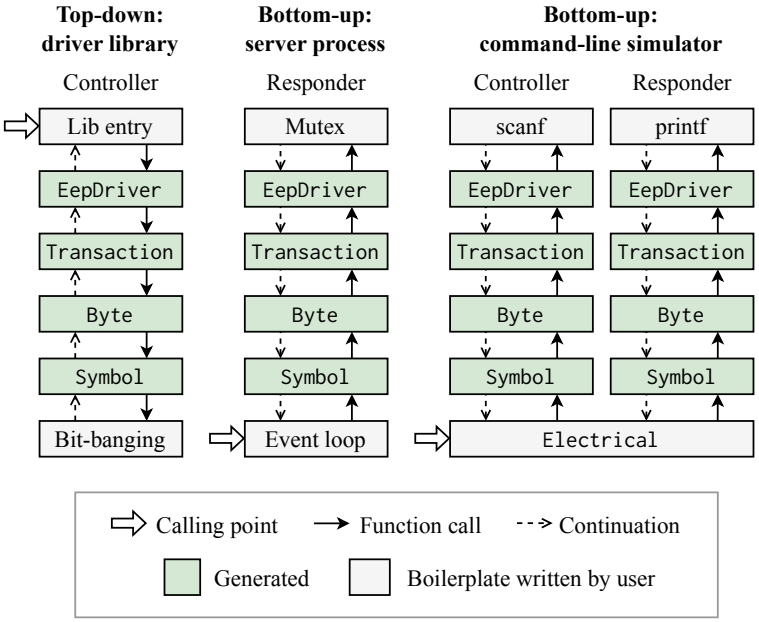


Figure 6.5: Examples of call graphs.

compiled into a usable library. The second example shows the ideal graph when the generated code is used as a part of a server process in the OS: An event loop ("callee" of the OS scheduler) reads values from the bus driver, invokes the stack from the bottom, reads the next electrical levels to write to the bus, and sends them to the physical bus driver. The third example is a command-line simulator of one controller and one responder. The `Electrical` layer is called by an infinite loop. The top layer of the controller reads inputs from the user, which go through the whole stack, and the results are printed at the top layer of the responder.

Given a call graph, `talk` and `read` operations become function calls and continuation calls. Figure 6.6 shows a `talk` operation. To pass values between

ESI

```
layer This;
layer Other;
interface <This, Other> {
     => { i32 x;
          i32 y; },
     <= { i32 z;
          i32 w; }
};
```

ESM

```
v = ThisTalkOther(x, y);
```

Transformed Direct Call

```
Other(x, y, &v.z, &v.w);
```

Transformed Continuation

```
*_ThisToOther_x = x;
*_ThisToOther_y = y;
_continuation_pos = <N>;
return;
_continuation_<N>:
v.z = _OtherToThis_z;
v.w = _OtherToThis_w;
```

Figure 6.6: Transforming a `talk` into a function call or a continuation. `<N>` is a placeholder for a newly allocated continuation index.

layers, function (layer) signatures are also transformed. When a layer is a callee in one connected pair, it passes input values from the other layer by value, and output values by reference (as pointers). The transformations use the Clang Rewriter [37]. Other parts of code remain unchanged as they are already valid C.

6.2.4 Verilog backend

Efeu also generates Verilog for programmable logic like FPGAs. The backend reuses more of the Clang/LLVM pipeline. The Clang AST is further lowered to LLVM IR [41], and thence transformed to Verilog. LLVM IR uses static single-assignment (SSA) form, which maps well to combinatorial logic in Verilog. Each function (layer) becomes a Verilog module, and basic blocks are converted to states. IR instructions are translated into blocking assignments in Verilog to preserve data dependencies between instructions. They will be analyzed by the electronic design automation (EDA) tool used to implement the circuit to extract parallelism. Arithmetic instructions are translated to the corresponding ones in Verilog. Branch instructions (conditional, unconditional, switch, and ϕ nodes [41]) become state transfers. Instructions that involve pointers (such

as stack allocation, load and store) are converted to operations on registers. As ESM disallows pointers and global variables, all pointers appearing in IR can be located with static analysis. The detailed translation rules can be found in a separate report [121].

`talk` and `read` require special handling, since they involve communications with other layers and require more than one cycle. Efeu uses ready/valid handshaking, a flexible and lightweight design widely adopted in designs like the AMBA AXI4 protocol [14]. On a unidirectional channel, the sender outputs data signals and a *valid* signal. The receiver outputs *ready* when it can accept more data.

A `talk` results in the following four states (`read` results in states 2 to 4), encoded as additional basic blocks.

1. Output data. Assert valid. Wait until peer asserts ready.
2. De-assert valid. Assert ready. Wait until peer asserts valid.
3. Save data from peer. Assert ready.
4. De-assert ready.

6.2.5 Generating hybrid hardware/software drivers

Efeu can also generate *hybrid* drivers, with multiple hardware/software layer boundaries defined at compile time. The hardware/software interface is based on AXI Lite [14]: between layers that straddle the boundary, data fields and valid and ready signals from the handshaking protocol are memory-mapped at different offsets. Figure 6.7 shows the case with `EepDriver` in hardware and `Transaction` in software, corresponding to the ESI interface in Figure 6.4.

The hardware handshaking protocol assumes sender and receiver to be in the same clock domain [14]: after a cycle where both valid and ready are raised, the sender needs to lower the valid signal immediately on the next clock cycle if there is no more data to send. Otherwise, the receiver treats data on the bus as the next valid packet. Similarly, the receiver needs to lower the ready signal unless it can immediately accept more data on the next cycle.

This is not the case when one side is in software. If the valid port uses a simple register, the software side might not be able to reset it in time, resulting in the same data being transmitted multiple times. Similarly, if the software does not reset its ready port in time, the hardware may send multiple packets that overwrite one another, causing data loss.

We solve this by performing automatic resets on the hardware side in the AXI Lite driver. Software writing a non-zero value to its output valid port means the data in the data registers is valid *once*. If the data is consumed, the valid signal is lowered in the hardware on the next cycle. Similarly, writing non-zero to the output ready port means the software side is ready to accept *one* packet. Once a packet is in place, the ready signal is lowered by the hardware on the next cycle.

On the software side, as with any device, waiting for the valid signal can be done either by polling or using interrupts. We implement both: as I show later in Section 6.4, they have different impacts on performance and CPU usage.

The software and hardware stub code are generated based on a minimal OS-specific library. We currently have library implementations for Linux and seL4 [99] in less than 100 lines of code each. For the Linux implementation, a small kernel module (less than 150 lines of code including blank lines and comments) creates userspace I/O (UIO) [102] device files for Efeu hardware based on device tree entries. The generated drivers then run in userspace. The library only `mmap`s the device file and provides functions for directing reads and writes to the virtual base address obtained from `mmap`, plus a function to wait for an interrupt which uses a blocking read from the UIO device file. On seL4 we rely on the Microkit SDK [80] to map the device registers into the virtual address space of the driver's protection domain. The read and write functions are then implemented the same as in Linux. Waiting for an interrupt is implemented using the blocking `seL4_Recv` syscall.

Efeu can generate drivers with more than one hardware/software boundary, as in Figure 6.7. These configurations are not optimal for performance, but can be useful for debugging (e.g., temporarily replacing a hardware layer with software).

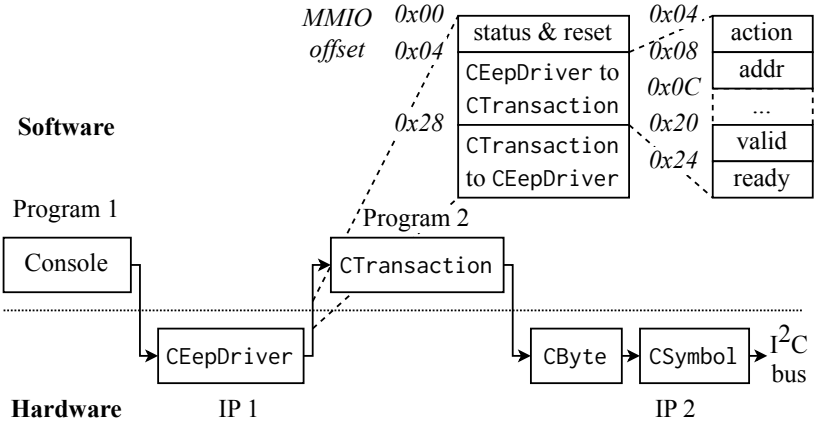


Figure 6.7: Multiple hardware/software boundaries. MMIO-AXI Lite interface between `CEepDriver` and `CTransaction` (corresponding to the ESI definition in Figure 6.4).

6.2.6 Promela backend

The Promela backend transforms the AST into input for the SPIN model checker, preserving syntactic information like variable names and control flow from the AST and allowing the developer to make easy correspondence between the ESM code and the generated Promela. Most ESM constructs have straightforward analogs in Promela, including variable declarations, operators, and control flows. The notable translation rules are listed as follows.

- `bit` and `byte` are not built-in types in C. Stub code generated by ESMC `typedef`s them as `unsigned char` to make the ESM code syntactically correct, but they are translated to exactly `bit` and `byte` in Promela.
- Enumerations translate to `mtype` [176].
- ESM channels translate to synchronous channels in Promela [176].
- Layer functions translate to Promela processes [176] with channels passed as parameters. This allows users to write parameterized verifiers (see Section 6.3.4).
- In Promela, and `if` statement encodes non-deterministic choices [176]. If no option is executable, the expression is blocking. However, in ESM when the condition does not hold, the `if` block is skipped. We encode such behavior by generating an `else -> skip` block if there is no `else` branch in ESM.

The generated Promela models the system, but must be combined with verifier code for input to the model checker. Note that Efeu does not provide fully formally-verified end-to-end properties: we trust ESMC to generate correct code (with extensive tests) and the downstream toolchains to correctly compile it. Removing this gap in the proof could be attempted using a range of techniques for compilers [111, 170] and EDA tools [126, 105, 112] but is beyond the scope of this dissertation.

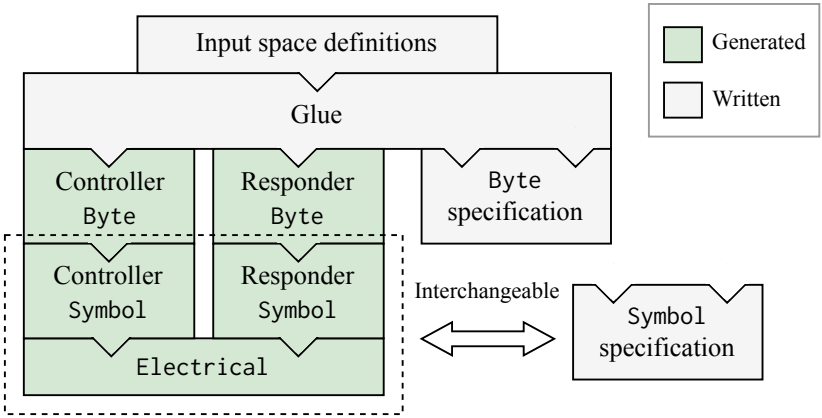


Figure 6.8: Architecture of the `Byte` verifier.

6.3 Verification

In the following, I describe how Efeu is used to verify a generated I^2C stack following the approach from our previous work [86], starting with the simplest case of one controller and one responder. I then discuss how we extend the technique to support multiple devices using parameterized verifiers in Section 6.3.4. I also explain how to model non-standard devices and quirks in Section 6.3.5.

6.3.1 Approach

For each layer except `Electrical`, we verify that the stack conforms to the behavior specification and there is no live- or deadlock in the system. Functional correctness is checked by assertions and the absence of live- and deadlocks is verified automatically by the model checker. Figure 6.8 shows the architecture of the `Byte` verifier as an example. The unit-under-test is the `Byte` layer, which is connected to the layer below. An input space specification defines the valid input to the system. Inputs are fed to both the stack and the behavior specification and the outputs are compared.

The state search space of the whole stack is non-trivial. To mitigate the state explosion issue, we apply the technique we proposed previously [86], which substitutes lower layers with the corresponding behavior specification. This significantly reduces the model checking runtime (see Section 6.3.3) and consequently allows larger input spaces and/or larger systems (Section 6.3.4). Each different class of layer has a different type of behavior specification, as follows:

The **`Symbol`** behavior specification specifies how symbols are combined on the bus. For example, a START symbol and an IDLE symbol (passively listening) are combined into a START operation received by both devices. BIT0 plus BIT1 results in BIT0 due to the pull-down characteristic of the I²C bus. The corresponding input space specification specifies valid control sequences from the `Byte` layers above.

The **`Byte`** behavior specification specifies how the controller and the responder should interact at the byte level. For example, when the controller writes a byte, the responder should be listening and the byte should be seen by both devices ultimately.

The **`Transaction`** behavior specification raises the abstraction level further up. The controller issues read and/or write transactions and the responder observes them. The input space specifies valid control sequences from above as a mixture of read transactions, write transactions, and/or stop operations. It is at this level that model checking scalability becomes an issue. Exploring the whole search space, including the variable payload length and content is infeasible. We therefore currently limit the input space specification for the `Transaction` verifier to a variable payload length of up to 4 bytes and a fixed payload content.

The example **`EepDriver`** behavior specification raises the abstraction level to EEPROM read and write operations. As with the `Transaction` verifier we limit the input space specification to a fixed EEPROM offset for reads and writes and a payload of between 1 and 4 bytes of fixed content.

In our previous work, we chose SPIN as the model checker for its ease of use and maturity. We also used it for this work to reuse the behavior specifications. The verification scalability experiment in Section 6.3.4 shows some limitations of the tool: increasing the number of devices and/or the payload length leads to state space explosion. We expect that more recent model checkers, especially

symbolic SAT-based ones [196] would be able to explore the search space more efficiently. Exploring this and alternative verification strategies, like pairwise verification of devices, are however beyond the scope of this dissertation.

6.3.2 Verification Code Size

While Efeu generates the complete stack in Promela from ESI/ESM, the other components of the verifiers (input space definitions, behavior specifications, and glue) require manual effort. The cost of verification will thus vary from platform to platform. As an indication, I report here the code size of both the generated and the handwritten files.

The ESM and C code is formatted with ClangFormat [38] and counted with `cloc` [52]. For Promela code, to my best knowledge, there is no canonical Promela formatter. Therefore, we build an in-house formatter and consistently apply it on both generated and handwritten Promela code. Comments and empty lines are excluded in all code.

Table 6.1 shows the results; those for generated code are underlined. At the `Symbol` and `Byte` levels, the controller and responder share most of the ESM code using preprocessor macros to include the same files, so I report combined lines. Shared generated Promela code defines common data structures and channels, which cannot be attributed to specific layers. Shared glue code is written once and included in all verifiers.

We see that the generated Promela is roughly the same size as the ESM specification, which is expected due to the close semantics of the languages. The ratio of additional handwritten Promela code to generated code is between 0.96 and 1.49 (excluding shared code). These numbers are only a rough approximation of the cost of verification, but do show how Efeu reduces the verification cost and avoids human errors by automating the translation from the specification to Promela.

6.3.3 Verification Runtime

To demonstrate the practicality and effectiveness of abstraction levels, we measure the runtime of SPIN executing these verifiers on an AMD Ryzen 9 7950X

	Layer				
	`Symbol`	`Byte`	`Transaction`	`EepDriver`	Shared
ESM					
Controller	139	114	106	62	
Responder			177	85	
Promela					
Generated controller	96	143	126	81	111
Generated responder	95	143	116	119	
Behavior spec	65	85	114	55	
Input space and glue	119	203	184	243	121
C					
Generated controller	159	174	125	62	
Verilog					
Generated controller	613	465	571	374	

Table 6.1: Source code lines of layers

16-Core machine with 64GB RAM and 128GB swap memory. The SPIN version is 6.5.2. Each verifier is executed 5 times. SPIN can check for either deadlocks or assertion failures (invalid end states) or livelocks (non-progress cycles) in one execution, but not both. Therefore, each verifier is compiled and executed in each configuration, and the runtime is summed up.

Table 6.2 show average runtime, and also the effect of abstracting lower layers with the corresponding behavior specifications. All verifiers pass. We observe no significant deviation across runs. The maximum coefficient of variation of all measurements is under 2.7%, and so I omit it for brevity.

Verifying `Symbol` is fast. Moving up the stack, the runtime increases dramatically, but limiting the input space allows the I²C stacks to be verified in reasonable time.

Layer	Abstraction Level			
	None	`Symbol`	`Byte`	`Transaction`
`Symbol`	0.24			
`Byte`	11.33	4.01		
`Transaction`	104.53	34.79	6.11	
`EepDriver`	584.78	196.31	38.92	9.15

Table 6.2: Average verification runtime in seconds

6.3.4 Scalability

So far I have shown verification for a single controller and responder. I now show the scalability of our approach modeling and verifying parameterized systems with multiple EEPROMs. The verifier uses channel arrays in Promela [176]. Multiple responders are instantiated and connected to an `Electrical` layer.

We vary the number of EEPROMs and the maximum length of reads and writes. The EEPROM offset and payload content are fixed. I also show a "variable payload" with one EEPROM where the first payload byte is chosen from two options non-deterministically. As in Section 6.3.3, we replace lower levels with behavior specifications to reduce the verification runtime, which is shown in Figure 6.9. Again, I omit the insignificant standard deviations.

This shows that systems with multiple responders can be verified in reasonable time. However, with SPIN the state space still explodes as the input space is enlarged or the number of responders increases. I discussed some strategies to further scale the verification at the end of Section 6.3.1.

6.3.5 Non-Standard Devices

I now show how to model devices that violate the I^2C standard, taking as examples the KS0127 video decoder and the I^2C controller on the Raspberry Pi.

The KS0127 video decoder (unlike the successor KS0127B [104]) has a quirk [103, 118]: in an I^2C read transaction, it attempts to sample a stop condition at the place where the acknowledgment bit should be. Otherwise, the

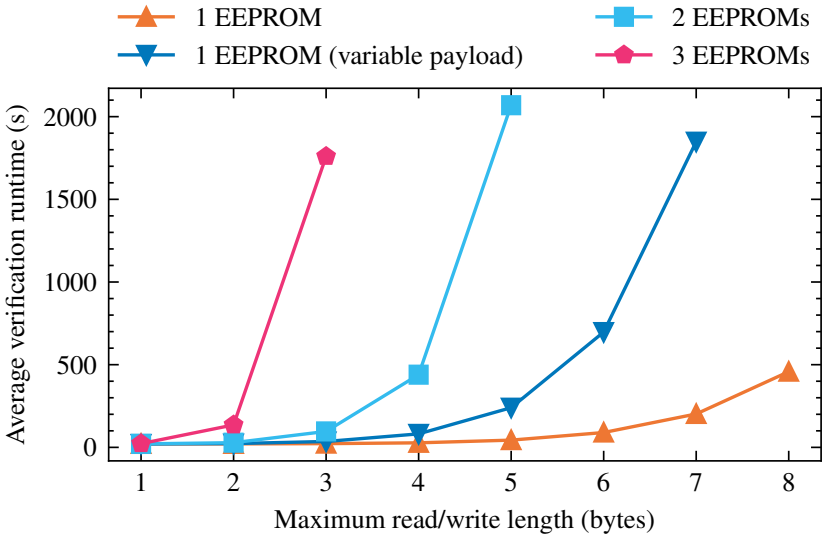


Figure 6.9: Verification runtime of multiple EEPROMs with different maximum payload lengths.

stop condition is not recognized. Linux introduced a flag (`I2C_M_NO_RD_ACK`) to handle this behavior *solely* for this device [118].

In Efeu, we model this quirk by changing only the `Byte` layer for the KS0127 responder to skip reading the acknowledgment bit in a read transaction. This involves 13 lines of additional ESM code. We also modify the maximum read length in the input space specification to 1 as both the KS0127 datasheet [103] and the Linux driver code [118] only specify reading one byte.

When we combine the modified KS0127 `Byte` and the standard controller `Byte`, SPIN reports the system can enter an invalid end state, showing that the standard controller is not interoperable with the responder.

Next, we modify the controller `Byte` layer to make it compatible with KS0127, involving 10 lines of additional code. With this modified controller, the verifier passes. Note that above these modified `Byte` layers, the `Transaction` layer can be used unmodified and the stack fully verified, showing that quirks can be handled within a single layer.

Similarly, using Efeu, we can efficiently model the hardware I^2C controller on the Raspberry Pi that does not handle clock stretching [130], by removing the clock stretching handling code from the controller `Symbol` layer in our specification, requiring 3 additional lines of code (essentially a preprocessor macro). The standard `Symbol` verifier detects problems in the modified stack. If we also remove clock stretching from the input space, essentially modeling a responder that never stretches the clock, the verifier passes.

6.4 Evaluation on real hardware

We evaluate the hybrid hardware/software I^2C controllers generated by Efeu on real hardware, varying the split between hardware and software. In this section, I report the results and show that it is feasible to efficiently explore the trade-offs between achievable bus speed, CPU usage, and FPGA resource utilization. We compare the generated drivers with two baselines: the Linux "bit-banging" kernel driver and the Xilinx I^2C IP [124].

The generated drivers run on a Zynq UltraScale+ MPSoC [210]. The MPSoC features a quad-core ARM Cortex-A53 and a Xilinx 16 nm FPGA. A modified OpenBMC [142] distribution (kernel version 5.15) runs on the ARM cores.

Efeu-generated C code is cross-compiled using GCC 13.2.0 with `-O3` optimization. Generated hardware parts are placed in an FPGA design, implemented using Xilinx Vivado 2022.1 with the default settings, and loaded on the FPGA. All components in the FPGA design are driven by a 100 MHz clock. Two GPIO pins serve as SCL and SDA, routed to an IO connector and connected to an I²C bus.

For I²C responders, we model Microchip 24AA512 EEPROMs [1] which support I²C Fast Mode (400 kbit/s). However, to demonstrate that the generated drivers work in practice, we use a real 24AA512 EEPROM as the responder, connected to the I²C bus. EEPROMs are slow for write operations: page writes on the 24AA512, page writes can take up to 5 ms [1]. During busy periods, the device stops responding to subsequent I²C requests. We focus on the I²C performance of the generated drivers rather than that of the EEPROM and so only report read operation performance.

A Keysight InfiniiVision MSO-X 3024T oscilloscope [95] is used to inspect signals on the I²C bus. The SCL and SDA lines are captured as analog signals. The oscilloscope is capable of finding rising/falling edges and decoding I²C packets.

To get the best performance from the baseline Linux I²C "bit-banging" driver, the GPIO delay [116] is set to 1 (0 results in a longer default delay). Internally, the driver polls the GPIO pins using a spinlock to wait between operations [117].

The Xilinx I²C baseline consists of two parts: hardware IP in the FPGA design with the target frequency set to 400 kHz, and the Linux driver from Xilinx supporting interrupts. The IP offers a similar abstraction level as our `Transaction`, while offering additional functionality like FIFO queues.

I denote software/hardware splits by the *topmost* hardware layer. For example, configuration `Byte` has `Byte` and below in hardware and the rest in software.

6.4.1 Source code size

To show how effective Efeu is in reducing the effort of writing I²C stack implementations, we measure the generated source code size. C code is formatted with ClangFormat [38] while Verilog code generated by Efeu is already formatted. Code size is measured using cloc [52].

Interface	ESI	Generated	
		C	VHDL
`Electrical-Symbol`	10	73	308
`Symbol-Byte`	16	68	295
`Byte-Transaction`	28	73	308
`Transaction-EepDriver`	24	84	391
`EepDriver-World`	23	82	401

Table 6.3: Source code lines for MMIO-AXI Lite interfaces

For layer implementations, sizes of the generated C and Verilog code are in Table 6.1. The generated C code has roughly the same size as the corresponding ESM specification. The generated Verilog files are a few times larger than the ESM files. Table 6.3 shows the result for the MMIO-AXI Lite interface across the hardware/software boundary. Vivado uses VHDL for AXI Lite interfaces, so does Efeu. Mixing VHDL and Verilog is not a problem as Vivado (and many other EDA tools) trivially supports it. The interface specification (ESI) is highly compact, while the generated code contains significantly more lines of code.

By specifying the stack in ESI/ESM once, developers save the effort of implementing the same thing in C and Verlog. While the generated code is expected to be less compact than code written by human experts, I conclude that Efeu helps reduce the effort required to materialize the verified stack.

6.4.2 Achievable Bus Speeds

The EEPROM supports the I^2C Fast Mode (400 kHz SCL) [88]. However, not all controllers can drive the bus at this speed. We measure the achievable bus speeds of both the baselines and the generated drivers to show how the software/hardware split point and the type of interface (polling versus interrupt-driven) affect the I^2C driver performance.

We measure the bus speed achievable by each controller by issuing 3 EEPROM reads of 14 bytes and inspecting the waveforms captured by the oscilloscope. The oscilloscope is triggered by the first falling edge of SDA, signaling the start condition of an I^2C transaction. It records long enough to cover the

whole operation. We use its built-in search function to find all rising edges of SCL. Experiments consistently show 164 rising edges for each EEPROM read operation. We calculate the effective SCL frequency as the inverse of the time between two consecutive rising edges. I show the average frequency and the standard deviation.

In the top half of Figure 6.10, the average frequencies across the whole operation and the 3 repetitions are shown. The error bars show the standard deviations. The Xilinx I²C IP reaches bus speeds close to the target frequency with little variation. The achievable bus speed is 386.57 kHz and the standard deviation is 23.75 kHz. The Linux bit-banging driver achieves an average frequency of 162.81 kHz, less than half of the target frequency. The standard deviation is 12.85 kHz.

The polling-based `Electrical` driver achieves a slightly lower frequency of 154.44 kHz with a standard deviation of 12.97 kHz. The interrupt-driven `Electrical` driver does not function correctly due to excessive interrupts being issued.

By moving the `Symbol` layer into hardware, the polling-based `Symbol` driver achieves a higher bus frequency of 263.32 kHz with a standard deviation of 12.77 kHz. Due to the reduced traffic across the software/hardware boundary, interrupts work for the `Symbol` driver. However, the interrupts yet introduce non-negligible overhead—the interrupt-driven `Symbol` driver only achieves 108.76 kHz.

By moving the next layer `Byte` into the hardware, the polling-based `Byte` driver achieves an average frequency closer to the target of 359.98 kHz. However, the standard deviation becomes more significant, reaching 89.82 kHz. The interrupt-driven `Byte` driver has less overhead. It shows an average frequency of 342.9 kHz, close to the polling-based driver, and similarly a higher standard deviation of 123.58 kHz.

The `Transaction` drivers move one more layer into the hardware. At this point, the generated drivers have a similar abstraction level as the Xilinx I²C IP. The achievable frequencies of both the polling-based and the interrupt-driven drivers are close to the target frequency, averaging 392.48 kHz and 392.24 kHz respectively. The standard deviations also drop to 33.25 kHz and 36.36 kHz respectively. Compared with the Xilinx IP, `Transaction` drivers have slightly higher bus speeds and standard deviations.

When all layers are in hardware, the `EepDriver` drivers achieve 396.02 kHz (polling) and 396.01 kHz (interrupt-driven). The standard deviations further drop to 10.37 kHz (polling) and 10.34 kHz (interrupt-driven).

To assist interpretation of the differences in achievable bus speeds, Figure 6.11 shows several waveforms of SCL. When the driver has a large portion in software, such as the Linux bit-banging driver or the `Electrical` driver, SCL is driven slowly and with unstable frequency. The software part takes longer time to process and to communicate data across the software-hardware boundary. In contrast, when most of the stack is in hardware, like the Xilinx I^2C IP and the `EepDriver` driver, SCL is driven towards the target frequency stably.

The experiment demonstrates how the splits between software and hardware affect the driver performance. Higher and more stable performance can be achieved by moving layers into hardware. This reduces traffic across the software/hardware boundary, where MMIO operations take time. In addition, when implemented in hardware, layer FSMs transit their states deterministically adhering to the hardware clock.

When the whole stack is implemented in software, the Efeu-generated driver achieves a performance close to the Linux bit-banging driver. However, neither of them reaches the target frequency of 400 kHz. By implementing parts of the stack in hardware (if the platform allows), Efeu-generated drivers can achieve the target frequency, comparable with the Xilinx I^2C IP, an optimized hardware implementation.

6.4.3 CPU Usage

The splits of software and hardware affect not only the achievable bus speed but also the CPU usage. If the platform on which the drivers run has limited computing power, understanding the latter correlation helps decide the optimal implementation. In this experiment, we measure the CPU usage of the Efeu-generated drivers and the baselines.

Similar to the experiment discussed in Section 6.4.2, drivers issue EEPROM operations of reading 14 bytes. However, in this experiment, those operations are issued consecutively and indefinitely until manual termination. In this way, we can read out the CPU usage in a stable running state. No other process consumes significant computing power. Behavioral correctness is asserted by

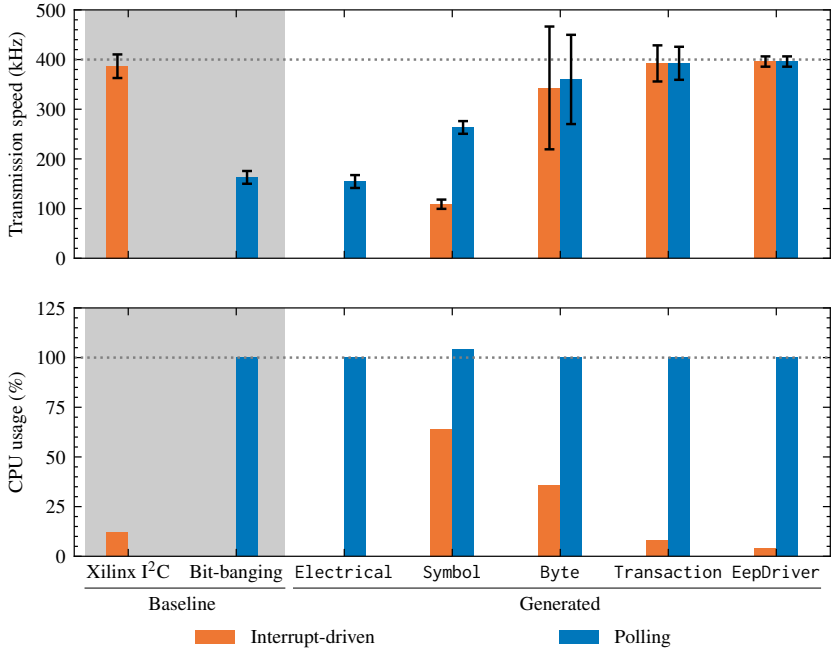


Figure 6.10: Achievable bandwidth (top) and CPU usage (below). 100% CPU means one core (of four) is fully utilized. The shaded area are the baselines and the Efeu configurations are labeled with the highest layer implemented in hardware.

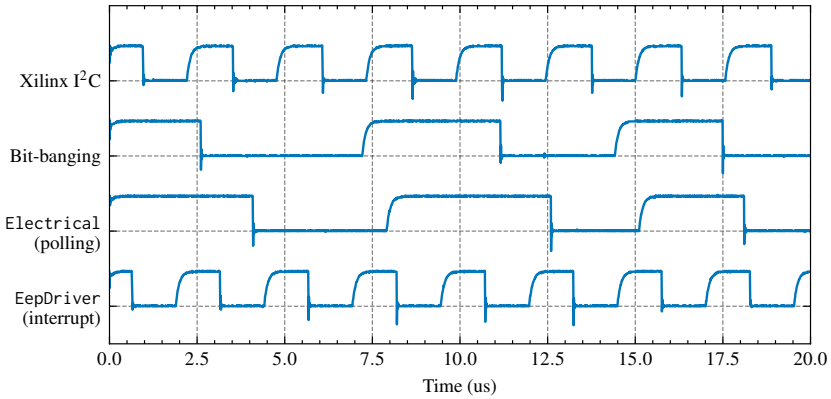


Figure 6.11: Waveforms of the first few SCL cycles, captured by the oscilloscope.

placing software assertions and inspecting I^{2}C transactions decoded by the oscilloscope.

The result is shown in the lower half of Figure 6.10. As expected, all polling-based drivers fully utilize one core. In contrast, using interrupts at the software/hardware boundary reduces the CPU usage. The interrupt-driven `Electrical` driver does not work as explained in Section 6.4.2. The Xilinx I^{2}C IP consumes 12%. The `Symbol` driver consumes 64%. The `Byte` driver consumes 36%. Moving the split point further upwards, the CPU usage drops significantly. The `Transaction` and `EepDriver` drivers takes 8% and 4% respectively.

The experiment demonstrates how the software-hardware split point affects CPU usage of the driver. Naturally, by moving parts of the stack into hardware, the CPU usage decreases for the interrupt-driven drivers but not the polling-based drivers. The interrupt-driven `Transaction` and `EepDriver` drivers generated by Efeu have the lowest CPU usage, lower than the Xilinx IP.

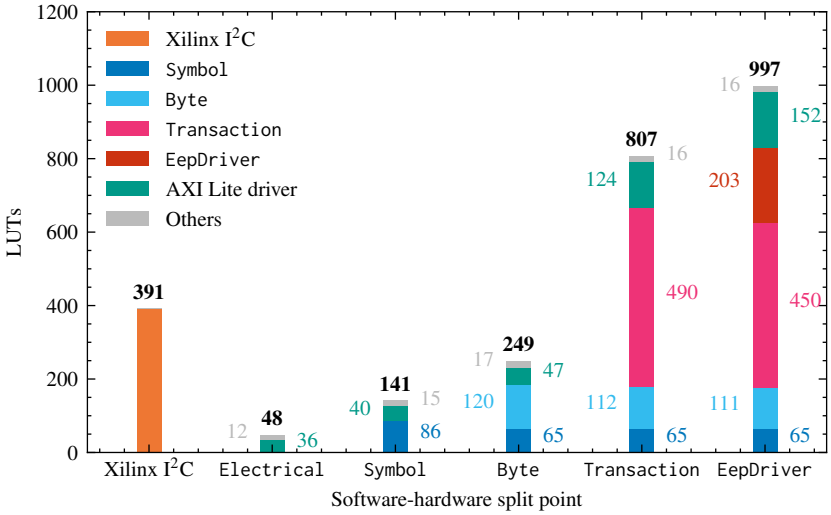


Figure 6.12: LUT utilization.

6.4.4 FPGA resource utilization

In this subsection, I report the FPGA resource utilization of the hybrid hardware/software drivers. We focus on two main resources on FPGAs, look-up tables (LUTs) and flip-flops (FFs). The usage data is extracted from the resource utilization report generated by Xilinx Vivado. Since layers are implemented as Verilog modules, Vivado also reports the resource usage of each layer.

LUT and FF utilization are shown in Figure 6.12 and Figure 6.13 respectively. "Others" is calculated as the total LUTs or FFs minus the sum of all other parts, attributed to the bus adapter and possibly the glue. As the split point moves up along the stack, we see the resource usage of low layers decreases. I believe it is due to Vivado performing cross-boundary optimization among modules.

`Electrical`, `Symbol` and `Byte` use fewer LUTs and FFs than the Xilinx I²C IP. `Transaction`, which has a similar abstraction level as the Xilinx IP, consumes 2.08× of LUTs and 2.11× of FFs. I believe this is a reasonable

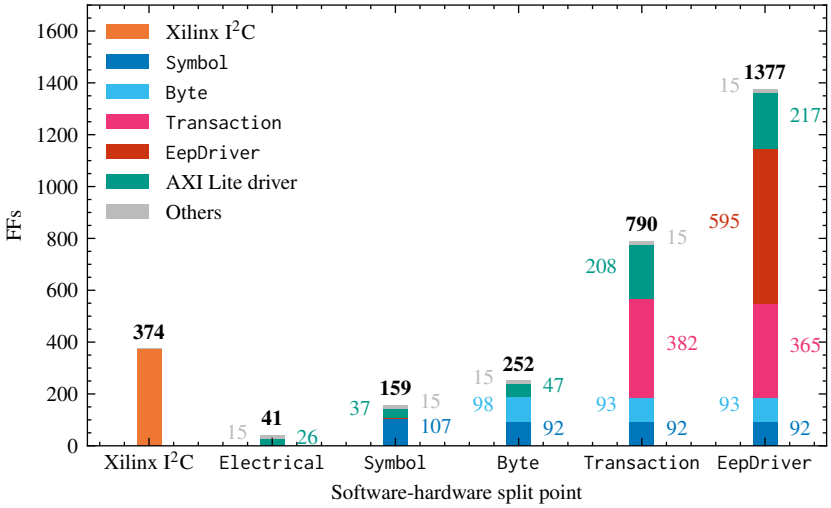


Figure 6.13: FF utilization.

overhead when comparing a generated driver with an IP crafted by human experts, especially when considering that the Efeu-generated drivers possess the assurance gained from the model checking. Furthermore, compared to the available resources on commercial FPGAs, these utilization numbers are very small. The FPGA on the MPSoC has 117120 LUTs and 234240 FFs [3]. In terms of the total available resources, the Xilinx IP consumes 0.33% of LUTs and 0.16% of FFs. The `Transaction` driver consumes 0.70% of LUTs and 0.34% of FFs. Even the entire stack on the FPGA (`EepDriver`) requires only 0.85% of LUTs and 0.59% of FFs.

This experiment demonstrates the Efeu-generated drivers consume very little FPGA resources in terms of the total available resources, comparable to IP crafted by human experts.

6.4.5 Discussion

Combining these evaluations, we demonstrate that Efeu helps to find the optimal implementations depending on different criteria—performance, CPU usage, and FPGA utilization.

On our evaluation platform, by implementing `Byte` and lower layers in hardware and using the interrupt-driven interface, the driver can achieve about 350 kHz bus speed, consume less than 40% CPU, and use less FPGA resources in both LUTs and FFs compared with the Xilinx IP.

If a bus speed higher than 390 kHz is required, at least `Transaction` and all layers below need to be implemented in hardware. Based on that, implementing the `EepDriver` layer in hardware only provides marginal benefits in the bus speed and the CPU usage. However, if extremely stable bus speed is desirable, `EepDriver` is a good option.

If there is no programmable logic available on the platform, `Electrical` is the only option. In this case, the driver cannot achieve the full bus speed (neither does the Linux bit-banging driver). However, it still possesses the assurance provided by the model checking.

6.5 Related Work

Driver reliability is a long-standing issue. There is therefore a large body of work to improve driver quality. There are three main categories of approaches to produce better drivers, and I survey them in turn: hardware/software co-design, synthesis of drivers from specifications, and post-hoc verification of manually written drivers.

6.5.1 Hardware/software co-design

Early approaches like Chinook [32, 33, 34] focused on I/O port allocation, synthesis of multiplexers, arbiters and driver code to share processor interfaces among devices. Chinook also synthesizes low-level hardware interfaces from timing diagrams. The adapter that Efeu uses to ensure the timing on the bus is currently hand-coded in 106 lines of VHDL but could be synthesized with such an approach. CoWare [195] can synthesize more complex hardware to adapt device interfaces to I/O interfaces available on a processor. It also generates code to make these adaptions transparent and maintain the device interface to software. These early systems usually make the assumption that the entire system is synthesized from the specification, and they use ad-hoc protocols for device interaction. In contrast, we demonstrate that we can verify a system built from off-the-shelf components connected with a standard protocol. Ortega et al. [144] extend Chinook to be able to instantiate standard bus protocols like I^2C or CAN and adapt device drivers to be able to use built-in bus controllers. They consider global system analysis for fulfilling real-time constraints but not for functional interference between devices. They also assume a standard I^2C implementation which leaves the many quirks in I^2C hardware out of the picture. Correctness of the co-designed systems is usually tested by full-system simulators that can also be synthesized from the specifications. Efeu gives formal guarantees on the properties it verifies. Later approaches then extend the power of the synthesis: O'Nils and Jantsch [141] present a method to synthesize DMA controllers to proxy off-the shelf devices to offload memory accesses from the CPU. In HINGE [207] a higher level device API can be synthesized and the correct usage of the API is checked at runtime and with BCL [97] the partition between software and hardware can be chosen independent of the specification

and partial implementations (only software, hardware or interface) are possible. None of these approaches focus on interoperability of off-the-shelf components like Efeu does.

6.5.2 Driver synthesis

Earlier work in driver synthesis was mostly concerned with device register specification. Devil [134] and HAIL [179] synthesize low-level register access code from specifications describing the interface to hardware. In NDL [44], one can additionally specify device state transitions and how they relate to the register accesses. The NDL compiler then synthesizes functions for querying and modifying the device state. In Efeu the interfaces are specified with ESI, and we can then not only generate the software to access the registers but also the register interface itself.

While these systems free developers from writing tedious bit manipulation code, this is only a part of modern drivers. Later approaches aim to synthesize full drivers with interface specifications to devices, operating systems and other software. Notably, there is Termite [156, 157] and more recently Ghost Writer [197]. Both systems are limited in the complexity of the device that they can synthesize drivers for. This complexity only gets worse when considering the interactions between devices. With Efeu we therefore chose to go with a verification approach instead. I will review some existing approaches next.

6.5.3 Driver verification

Verifying manually written drivers is pragmatic: the quality of existing drivers can be improved without having to reimplement what is a significant part of the OS [31]. SLAM [19] verifies the correct usage of OS APIs by drivers using model-checking. SafeDrive [209] achieves a similar goal, but they synthesize run time checks to catch driver errors and recover from them. Bošnački et al. [22] use model-checking and static analysis to verify the correct use of Linux APIs by an I^{2}C driver. The biggest difference to Efeu is that we focus on the interaction between the driver and the devices and between devices and not between the driver and the OS. In that sense these approaches are complementary to ours.

An example of post-hoc verification that focuses on the interaction with the device is Kim et al. [96] who verify a flash driver using model checking. There are however limitations to how thoroughly software can be verified if it was not implemented with verification in mind [99]. Many approaches therefore implement the drivers in a highly stylized way. Chen et al. [28] target drivers for interruptible kernels that they verify in Coq. Klomp et al. [100] target an I^2C driver. Pohjola et al. [148] propose Pancake, a DSL targeted at easily writing verifiable device drivers and keeping the verification cost low. We follow a similar approach with Efeu and specify our drivers in a DSL designed for verifying the properties we are interested in. All the above focus however on verifying individual drivers that handle a single device and therefore cannot address interoperability issues that arise in shared bus settings such as I^2C.

6.6 Conclusion

Amid all challenges of producing correct drivers, bus-based protocols like I^2C pose another: interoperability. I have presented Efeu which allows specifying full I^2C subsystems in DSLs, model checking them, and generating hybrid hardware/software drivers. Efeu helps developers to explore trade-offs between throughput, CPU usage and FPGA utilization.

While so far we have only applied the Efeu methodology to I^2C, I believe that it could be extended to other bus-based protocols like SPI or CAN. These protocols share key features with I^2C: multiple agents transmit data by modulating shared wires, and the protocols consist of multiple abstraction levels with potential quirks. The electrical characteristics, like the number of wires that are used, only appear on the lowest layer of an Efeu specification. Furthermore, bus timing is handled by the lowest-level hardware adapter, allowing Efeu to deal only with discrete time steps. This adapter is currently handwritten but could be synthesized using HW/SW co-design techniques for our approach to scale better to other protocols (see Section 6.5.1).

Another item for future work is reducing the trusted compute base (TCB). A large part of it is the Efeu compiler. I outlined some approaches for removing it from the TCB at the end of Section 6.2.6.

The Efeu compiler and all our I²C specifications are available as open source[2].

A correctly functioning I²C stack is critical to an important BMC task: power management. Despite the remaining challenges, Efeu fills an important gap between the correct power sequences we can generate with the approach presented in Chapter 4 and the hardware: By ruling out interoperability issues, we can make sure that the sequences are correctly executed by the regulators. In the next chapter I present a system design that helps preserve this correctness on a real world BMC.

[2] https://gitlab.inf.ethz.ch/project-opensockeye/efeu

7

System Design

BMCs fulfill critical tasks in the system that they manage. The correct operation of any such system depends on the correctness of its BMC. The implicit trust we put in BMCs is however not warranted, as the vulnerability analysis in Chapter 3 shows. All the mainstream BMC systems for which the information is publicly available are Linux-based. This means that any vulnerability in the Linux kernel is a vulnerability at the heart of our computing infrastructure.

The security of BMCs is however only half the story. Arguably one of the most critical functions of a BMC is power management. To ensure the correct functioning of the platform and the safety of the hardware we have to trust the components that implement this and other critical tasks. In the previous chapters we have seen how we can synthesize trustworthy implementations for these components from purpose-built hardware models. These critical services however are not the only processes are running on a BMC. There are BMC components that are less trusted. One reason can be their exposure to the outside world, like a webserver for remote management or a protocol handler that communicates with the firmware on other processing elements on the platform. Another reason can be the lower assurance implementation of a component to save development cost. The escalation of privilege vulnerabilities in Chapter 3 demonstrate the risk: a vulnerability in the webserver can be used to affect a critical function of the system. The vulnerability analysis showed that this type of bug is the most common in today's BMCs.

This makes BMCs mixed-trust environments. It is therefore not enough to create trustworthy implementations for critical components, we also need to

isolate them from less trusted components. Specifically we need to guarantee confidentiality (a components data can not be access by another component), integrity (a components data can not be altered by another component), and availability (a component can not be kept from running and fulfilling its purpose). State-of-the-art systems do not offer these guarantees: on the one hand their OS kernels (usually Linux) are known to have vulnerabilites [171] and on the other hand the way these systems are structured puts a lot of ambient authority in components that can be attacked, usually even remotely. A successful attacker can abuse all this ambient authority to wreak havoc in the system.

In the following I present a proposal for a system design for BMCs that addresses this challenge. It uses seL4 [99] as a separation kernel to provide formally proven isolation guarantees between components. This is however an implementation choice and other solutions exist. I start by exploring different ways of how to provide the necessary isolation and will then present our proposal. We will also see how an existing BMC system – e.g., based on OpenBMC – can be retrofitted with a trustworthy design. In the end I will go back to the classes of BMC vulnerabilities identified in Chapter 3 and see how the system design helps prevents them.

7.1 Providing isolation

There are two major ways of providing isolation between trusted and untrusted components: the first one we are going to look at is physical separation, i.e., running the trusted components on separate hardware from the untrusted ones. This seems to be the trend in industry. Secondly, I am are going to survey options for providing the isolation in software.

7.1.1 Physical separation

In theory, the strongest isolation between critical components and untrusted ones can be achieved by not having them share hardware resources. Indeed, as we will see, there seems to be a trend in industry towards this solution. Hardware separation comes with its own challenges though. I will provide an overview over existing systems and these challenges here.

We have already encountered a simple form of hardware separation in Chapter 4: often the power sequencing is not fully done in software but in a CPLD. This provides very weak isolation though as usually the BMC still has full control over the power sequencer. It is rather a measure to offload complexity from the BMC's power management software.

Systems that employ hardware separation for increased security do so mainly to ensure the integrity of platform firmware. They use a hardware root of trust (RoT), often also called a silicon RoT for this purpose. A RoT is a tamper-proof cryptographic processor that manages keys and offers functions like verifying signatures on firmware files and can provide a cryptographic identity for the machine. Google uses their own design called Titan to ensure the integrity of firmware in their machines. Titan interposes between the machine's BMC and its firmware flash storage. Before it lets the BMC boot, it verifies the cryptographic signatures of the firmware stored on the flash [159] to ensure it has not been tampered with. Titan has been open-sourced as OpenTitan in 2019 [76]. It could potentially not only be used to interpose firmware loading but also for other communication of the BMC with the platform like power sequencing commands. There is evidence that this is planned or even already done as the OpenTitan documentation mentions I^2C as a potential requirement for the Platform Integrity Module use case [143].

A more radical approach is taken by AWS with their Nitro design [17], which powers Amazon Elastic Compute Cloud (EC2). Nitro completely separates the control plane of the machine from the mainboard with the application cores that run customer workloads. The control plane is implemented on Nitro cards. These include cards for I/O like network and storage, but the interesting one for our purposes is the Nitro Controller. It acts as a RoT for the system and like Titan is responsible for firmware integrity: the firmware for both the BMC of the mainboard and the application cores is loaded from storage provided by the Nitro controller. This is facilitated by the Nitro Security Chip that sits on the mainboard. Additionally, the Nitro Security Chip mediates access to any firmware on the board, including access to the BMC from the application cores. If the machine is used as a virtual machine host, the application cores run the Nitro Hypervisor which receives its virtual machine management commands from the Nitro Controller. The Nitro Security Chip than acts as a defense in depth mechanism to double-check accesses to board functions that have been

cleared by the hypervisor. Additionally, this allows running the machine in a mode where the customer gets bare metal access to the application cores: the Nitro Security chip then acts as the secure monitor that prevents a customer from accessing configuration relevant to datacenter security. It is not clear which component handles the low level management of the mainboard like power sequencing. However, the fact that there is still a BMC next to the Nitro Security Chip on the mainboard, leads us to believe that it still fulfills these management functions, probably supervised by the Nitro Security Chip. EC2 offers Intel, AMD and Graviton CPUs (Graviton is an Arm server chip designed by Amazon for its cloud). Having a BMC as part of the mainboard probably enables designing the Nitro control plane more independently of the different mainboards used for the different processor types and their potentially quite different power distribution topology.

The last system that we are going to look at is a rack scale system built by Oxide Computer Company[1]. It takes disaggregating the BMC by physically partitioning the functions traditionally implemented by it to the most extreme (at the time of writing). An Oxide rack consists of up to 32 AMD-Milan-based single socket compute sleds, two network switch sleds and a power shelf [42]. All boards are custom designed and do not have a traditional BMC. Instead they have a *service processor* and a hardware RoT. The service processor takes on the low-level management functions of a BMC, e.g., power and firmware management, and serial console access but does not offer high-level management functions like user or management APIs served by a webserver. The higher-level functions are instead implemented in a distributed program called Nexus that runs on all compute sleds. It exerts control over the rack through the *sled agents* running on each sled. The rack's network is partitioned into a data plane network and a management network. The latter connects all service processors in the rack. The two compute sleds adjacent to the switch sleds are dedicated management gateways that control the switch configuration and forward control plane traffic onto the management network [43]. Organizing the control plane in this way without traditional BMCs on their boards, required Oxide to do completely custom board designs. They also follow an aggressive open source strategy with the promise to make all software and firmware running on the rack publicly

[1] https://oxide.computer/

available [27]. The motivations and implications of this are out of the scope of this chapter, but I will come back to them in Chapter 8.

To summarize, industry has already taken steps to distribute the immense power concentrated in BMCs onto smaller hardware components. The main efforts are about establishing trust in system firmware through cryptographic means implemented in hardware RoTs. Most designs however, especially for off-the-shelf hardware, retain a monolithic BMC. To reduce the required trust in the BMC its actions can be monitored by a physically isolated component. To really increase safety this would mean that this additional component needs an understanding of the tasks it is monitoring, e.g., power sequencing. This then begs the question if it would not be safer and more efficient if that component would perform the task itself instead of just monitoring the partially trusted BMC. Retaining a monolithic BMC most likely also simplifies board design: adding chips that interpose between components on a more traditional design is easier than a full custom design that would truly disaggregate the BMC.

Physical isolation is of course not an option for existing systems as it requires hardware changes. I am going to survey options for software-based isolation next.

7.1.2 Software isolation

There is a large body of work on software isolation, and I will not attempt to give a full survey here. I will rather provide classification criteria with examples of systems and a discussion of which of these criteria are desirable for a BMC system. The criteria are:

- Isolation guarantees: formally verified vs. unverified
- Isolation mechanism: hardware assisted vs. pure software
- System maturity: research prototype vs. commercially deployed
- System applicability: general vs. special purpose
- System partition: static vs. dynamic

Additionally, we will analyze potential paths to incrementally migrate existing BMC systems to a more secure design. Such a migration strategy is desirable: it allows retrofitting existing systems without having to re-implement the entire BMC at once. It also enables early experimentation on real hardware without a fully redesigned BMC stack.

Guarantees. With the advancement in computing power and formal techniques, OS verification has seen significant advancements since the beginning of the millennium. For an excellent introduction and a survey up to 2009 I recommend Klein's work [98]. I list the most notable OS verification projects in the following. seL4 [99] provides a full functional correctness proof and a proof of high-level security properties confidentiality, integrity and availability. CertiKOS [74] proofs functional correctness as well. While seL4 proofs that the semantics of the C code are carried through to the assembly level using translation validation [170], CertiKOS achieves the same with a verified C compiler. SeKVM [114, 115] proofs functional correctness for a hypervisor including protecting virtual machine memory from devices that use direct memory access (DMA). The proofs for these projects are all formalized in interactive theorem provers which offers comparably little proof automation. Other projects like Verve [206], Hyperkernel [140] employ so called "push-button verification" where a solver verifies code automatically based on code annotations. This approach usually needs severe limitations in the complexity of the systems and/or the strength of the properties verified to work [25]. More modern tools show promising advancements in this area as well [108].

Existing BMC systems could be improved by using, e.g., a hardened Linux kernel. However, with the above advancements in producing verifiably correct OS kernels, we are running out of excuses for not introducing such guarantees into what are arguably some of the most critical systems in modern compute infrastructure. While a correct kernel does not automatically mean a correct system [25], work on building fully verified systems warrants optimism that a carefully designed system can be verified [40, 145].

Mechanism. Software can either provide isolation by using hardware features such as a memory management unit (MMU) or purely by software means. Note

that hardware-assisted isolation is not the same as the form of isolation discussed in Section 7.1.1. In the former case, the two components being isolated from each other still run on the same physical core. Software-based mechanisms require the components to be "well-behaved", i.e., they cannot be able to issue arbitrary memory requests. This is usually achieved by using a memory safe language and trusting the compiler to statically enforce isolation. Examples are Modula-3, which was used to implement SPIN, (extended) C#, which was used in Singularity and more recently Rust, which was used to, e.g., implement RedLeaf [139]. The work on RedLeaf also includes a much more thorough survey of language-based isolation. The argument for not using hardware mechanisms is the context switch overhead [139] or to support systems that do not have memory protection hardware. Any system that needs to run untrusted code however, has to use hardware isolation features and all mainstream OSs like Linux or Windows fall in this category. A BMC should arguably not run any untrusted code and so both approaches are applicable. Indeed, RedLeaf targets systems like BMCs [138]. However, with a language-based approach all components need to be implemented in this language which prevents incremental migration. As we have seen in Chapter 2, the most common SoCs for BMCs have cores with MMUs. I therefore favor a hardware-based mechanism.

Maturity. This criterion is about how mature the system is in terms of documentation, hardware support and ecosystem. Many verified systems, are not much more than research prototypes. As the focus is on designing the BMC stack itself and not advancing the state of the art in software-based isolation I favor a system that is easy to use or at least does not pose a major hurdle when implementing a BMC on top of it.

Applicability. This criterion is about whether a system is general purpose or tailored to specific hardware or use cases. I favor flexibility here such that the BMC design could be deployed on a range of systems.

Partition. Software isolation approaches can also be classified by whether the system is partitioned statically or whether new isolated components can be created at runtime. In a BMC the components are usually known statically. For

experimentation, dynamicity would offer more flexibility but all in all this is not a decisive criterion.

In summary, the most important criterion is the strength of the isolation guarantees the system delivers as they strongly influence the overall security of the BMC stack. The isolation mechanism plays a major role in being able to offer a migration strategy where we need to be able to run legacy code next to the reimplemented BMC tasks. Maturity is an important criterion for being able to focus on providing a safer and more secure BMC stack without being impeded by system limitations. Finally, applicability and partition play a less important role for us.

7.1.3 Summary

While physical separation of BMC tasks potentially offers the best isolation between trusted and untrusted components it also comes at a high implementation cost. This is especially true when following a radical approach like Oxide does. Even in their system however, it is hard to tell from public information what the exact trust relationships between different hardware and software components are. A BMC stack cannot be evaluated without running it on real hardware, and it therefore needs to be able to be deploy it on hardware that we as researchers can get access to. The currently only experimentation platform is Enzian. It follows a more conventional platform architecture with a dedicated SoC for the BMC. A design with the same hardware requirements as existing BMC stacks like OpenBMC could also be ported to a commercial system – if one with sufficient documentation ever becomes available. Such a design needs to be software-isolation-based with strong formal guarantees, the ability to run legacy code and enough maturity to not impede design freedom. I describe the proposed design in the next section.

7.2 A trustworthy BMC design

To potentially profit from the strongest isolation guarantees I opted for a system with a full functional correctness proof: seL4. I say potentially here as Enzian's

BMC platform is not in the set of verified configurations [169]. This is a problem with any formally verified system: the proofs need to make fairly specific assumptions about the hardware platform. A platform for which these assumptions have not been checked, therefore does not offer the full assurance of a correctness proof. However, full system verification is out of the scope of this dissertation. I propose a design here that is *amenable* to verification. Even without the full formal verification available for the platform, it still profits from the assurance of the kernel sharing much of its code with verified platforms [79]. This will especially be true once the verification of the 64bit Arm version of the kernel is completed, which is scheduled to happen in the near feature [168].

I chose seL4 over its competitors for its active community with members from both academia and industry [167]. seL4 has been successfully deployed in real world systems, is the best performing microkernel and its capability-based security allows very fine-grained access control [79]. It supports x86, Arm and RISC-V architectures and is a general purpose kernel with a focus on embedded systems, specifically hard real time and mixed criticality pplicaaions. It supports dynamic creation and destruction of isolation domains. However, current research focuses mostly on statically partitioned systems. [80]. While verification of a multicore version of the kernel has not been completed yet, there is ongoing research towards SMP and clustered multikernel support [113, 166].

seL4 also has a story for full-system verification including a migration strategy through *cyber retrofit* [79] (see Section 7.2.1). More recently work on an OS on top of the seL4 microkernel has begun [70]. It is based on a new userspace SDK called Microkit (formerly Core Platform) designed to improve usability [80]. Work on verifying it has begun [145]. By building on this active research, the secure BMC design will be able to profit from new breakthroughs in full-system verification.

The architecture of the design is depicted in Figure 7.1. It uses seL4 as a separation kernel to isolate trusted and untrusted components from each other and ensures a least-privilege communication pattern. It employs the cyber retrofit strategy to gradually migrate the existing OpenBMC stack to a more secure seL4 native stack (Section 7.2.1). The trusted components are synthesized from purpose built hardware models (Section 7.2.2). To extend trust further down, hardware components are also be synthesized where appropriate (Section 7.2.3).

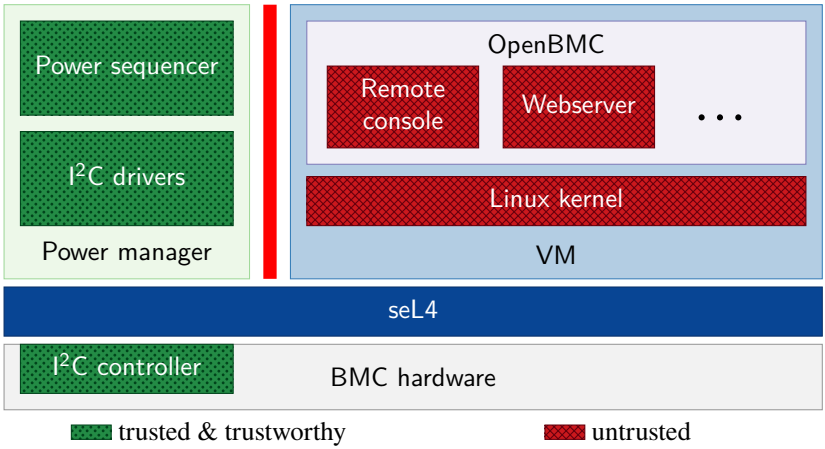


Figure 7.1: Trustworthy BMC architecture with different trust levels for components

7.2.1 BMC cyber retrofit

Being able to reuse components from existing BMC stacks is important for several reasons. For one, trustworthy implementations for some components like a webserver might not add much to the overall assurance of the stack. Furthermore, reusing parts of the existing stack while gradually replacing critical components with trustworthy implementation provides a migration path where the new system can be tested on real hardware without losing functionality. The second part is important as the Enzian cluster is actively used for research on cache coherence [162], hardware architectures for serverless computing [177], hybrid memories [24] and more. Being able to deploy the new design early on will provide validation that the ideas work in practice.

This gradual migration of a legacy system to a trustworthy system is called *cyber retrofit* and has been pioneered in the DARPA HACMS program [40]. They demonstrated the strategy by cyber-retrofitting the mission computer of a quadcopter and an unmanned military helicopter. The resulting systems resisted

a professional penetration testing team. As a first step the existing Linux-based system was moved into a virtual machine with seL4 as a hypervisor. This step does not add assurance but is a stepping stone for the next steps. From here individual components can be reimplemented and moved to native seL4 components without changing the overall system functionality.

We currently have a prototype of the Enzian BMC where we have completed the first step of virtualizing OpenBMC. As mentioned in Chapter 2 the BMC in the original design for Enzian was Armv7 based. seL4 does not support running virtual machines on this architecture, so we had to upgrade the hardware platform to Armv8 (this came with its own set of challenges that I will come back to in Chapter 8). In the next sections I am going to lay out how I envision completing the BMC cyber retrofit.

7.2.2 Creating trusted BMC components

Virtualizing the existing BMC stack itself does not add any assurance to the system. To increase the trustworthiness of a BMC, its critical components need to be isolated from the untrusted components, i.e., moved out of the virtual machine. Crucially however, the components themselves need to be trustworthy. Naturally, many critical components in a BMC interact closely with the hardware. Examples are components for power, clock, and thermal management, components that manage firmware for other processing elements on the platform and components that handle communication with other processing elements e.g., to allow the OS on the application processor to reboot or power off the machine. The correctness of these components naturally heavily depends on their correct understanding of the hardware they interact with. To produce trustworthy implementations for them, we therefore need models of the hardware and if we want to ever be able to give formal guarantees about the components' correctness, these models need to have formal semantics.

The arguably most critical component in a BMC is the power manager: without it, the machine cannot be turned on and, as we have seen in Chapter 4, bugs can permanently damage hardware. I have presented a model for power distribution networks in Chapter 4, and I have demonstrated that we can generate code for a power sequencer from this model. In Chapter 5 I then also laid out an approach for building such a power sequencer. Furthermore, in Chapter 6 I

presented Efeu, a framework for producing provably interoperable I^2C stacks. Together they can be used to produce a trustworthy power manager. This will require scaling the verification of Efeu stacks to 10–20 devices on a bus and more layers to implement SMBus and PMBus functionality. I outlined some potential strategies to achieve this at the end of Section 6.3.1.

Models for other components like thermal management or firmware provisioning remain future work.

7.2.3 Trusted BMC hardware components

Many BMC components are tightly integrated with the hardware that they run on and manage (cf. Chapter 4 and Chapter 6). It is therefore vital to the correctness of the system that the hardware models used to build these components are correct about the hardware's behavior. One way of ensuring this is to use the same hardware model to not just generate software, but also the hardware itself. I have presented such an approach for I^2C controllers in Chapter 6. We have also experimented with pushing higher-level BMC functions into hardware for more deterministic performance [190]. The experimental system implemented a PID controller in hardware that used inputs from temperature sensors to determine fans speeds.

We also need to be able to ensure that trusted hardware can only ever be accessed by trusted components and that untrusted hardware cannot interfere with trusted components. Especially the second part is a hard problem and beyond the scope of this dissertation. There is however work in the larger context of this dissertation that aims to solve this problem, and I will return to this in Chapter 8. Preventing untrusted components from directly accessing trusted hardware can be achieved through memory protection, given that trusted and untrusted devices do not share protection granules, i.e., pages in an MMU-based system. An example for such a violation would be a GPIO bank that has pins for both untrusted and trusted functions. The FPGA on the Enzian BMC allows us to experiment with this and reroute signals to achieve separation between trusted and untrusted functions.

7.3 Preventing vulnerabilities by design

In Chapter 3 I presented a taxonomy for BMC vulnerabilities. This analysis informed the design that I have presented here. In this section I am going to show how the design prevents some classes of these vulnerabilities and helps to prevent others.

7.3.1 Preventing vulnerabilities in critical components

Critical components are by definition trusted, and the design requires making these trust assumptions explicit. For the system to be trustworthy as a whole, these trusted components need to be trustworthy. This means that we need some form of assurance, ideally formal verification. While I have not performed full formal verification of a component in this dissertation, I have laid some important groundwork by developing hardware models with clear semantics that can be used as a basis for formal verification. Namely, in Chapter 4 I presented our model for power and clock distribution topologies from which we can derive correct power sequences. In Chapter 6 I have shown an approach to produce trustworthy drivers for chip-to-chip protocols.

7.3.2 Preventing privilege escalation vulnerabilities

As we have seen in Chapter 3 about half the vulnerabilities in today's BMC systems are privilege escalations. The design prevents these by providing strong isolation between trusted and untrusted components. The security proofs of the seL4 kernel guarantee that a compromise stays contained within a component. Additionally, the capability-based access control in seL4 lets us implement the principle of least privilege [79]. This means that a component has exactly the rights it needs to fulfil its task, and there is no ambient authority that can be abused by an attacker. One aspect of this is a strictly least-privilege communication structure: components can only communicate with the components that they need to and the communication graph can be made *explicit*. By ensuring that untrusted components only have the authority to perform actions that are safe, even if executed on behalf of a potentially malicious actor, we can prevent privilege escalation in the system.

7.3.3 Containing non-critical vulnerabilities

The vulnerabilities in the last category are non-critical vulnerabilities: vulnerabilities that already in today's systems stay contained within the vulnerable component. The exception here are client-side vulnerabilities which while not being able to directly spread into the system can be used to trick an authorized user into performing a privileged action on behalf of an unauthorized attacker. The mitigation of these kinds of vulnerabilities are out of the scope of this dissertation. The other two types are code execution and DoS vulnerabilities. The mechanisms described in the last section ensure that no privilege escalation can result from a code execution vulnerability. Recovering from a DoS attack on a vulnerable component is not directly prevented by the system design. However, recovery mechanisms can be built using, e.g., watchdogs. The mixed-criticality support in seL4 can prevent a component from monopolizing the CPU and ensures that critical components still get the required CPU time to fulfil their task [127].

7.4 Conclusion

In this chapter we have seen how a trustworthy BMC system can be designed. I have laid out how this design can prevent entire classes of BMC vulnerabilities instead of fixing vulnerabilities when they get discovered as state-of-the-art systems do. As the design is not implemented yet, there are challenges remaining. I will present some of them in the next chapter before concluding this dissertation in Chapter 9.

8 Future Work

Building BMC for the Enzian research computer taught me many things about how modern servers work. While I believe that the research in this dissertation is an important contribution to more trustworthy board management, it also surfaced many interesting problems that could not be pursued yet. In the following I describe some of the most intriguing ones.

8.1 Cyber-retrofitting BMCs

In Chapter 7 I proposed a design for a trustworthy BMC system. I also described a migration strategy from a current system like OpenBMC using cyber retrofit [40]. We have started to retrofit the BMC in Enzian and completed the first step of virtualizing the Linux-based OpenBMC distribution with seL4 as a hypervisor. The prototype is based on CAmkES [106], the previous seL4 component framework. It has since been superseded by the seL4 Microkit [80]. The first step towards completing the cyber retrofit would be to port this prototype to a Microkit-based system. I do not expect any major challenges in this endeavor. The next step is to produce trustworthy implementations of critical components and move them out of the virtual machine. One such component is the power manager and I have already outlined how the work presented in Chapter 4, Chapter 5 and Chapter 6 can be combined to produce a trustworthy power manager (cf. Section 7.2.2). I already mentioned the challenges I anticipate (scaling Efeu verification). However, there are more areas where further research is needed.

8.1.1 More trustworthy components

The power manager is not the only critical component. There are obvious other ones like the component that manages firmware for all the devices on the platform (including the BMC itself) or a component that handles authorization. However, in general what constitutes a critical component also depends on the threat model for the platform and the BMC. If, e.g., the firmware and OS on the application CPU are trusted, a trustworthy component that offers a virtual console to the application CPU might add to the trustworthiness of the system. However, if the OS on the CPU is untrusted it might not, as the OS could tamper with the input and output itself. An analysis to determine where to focus attention is needed.

Once the critical components have been identified, we then need domain specific models for them that they can be verified against.

8.1.2 Component interfaces

Another open question is what interfaces these components should offer. For most components I anticipate there to be a trade-off between usability and safety and/or security. In the case of the power manager, the safest interface would be a very high-level one that only offers operations to turn the machine on or off. On the other hand, a more low-level interface would allow for more fine-grained control. Being able to only power the CPU or the FPGA on Enzian is such an example. The larger the interface however, the more permutations of operations exist and the harder it is to check whether unsafe permutations exist. Then there is also the question of trust in the user. In the extreme case the question is "should a privileged enough user be able to perform operations that could damage the hardware?". An interesting question is, how we can use models (like the one for the power topology) to give informed answers to such questions or even synthesize safe interfaces.

8.1.3 BMC interfaces

There are two broad classes of interfaces to the BMC: external interfaces that are used for remote management and intra-machine interfaces through which

the BMC can communicate with other components on the board and offer services like turning power on and off on the request of the application OS. There are several industry standards for protocols. IPMI[90] is used both in external and intra-machine interfaces. It is however infamous for its security issues [21]. Newer standards include Redfish [58], Management Component Transport Protocol (MCTP) [56] and Platform Level Data Model (PLDM) [57]. Redfish is a REST-based protocol and data model for remote management. MCTP [56] and PLDM [57] are a data model and protocol for intra-machine interfaces. Analyzing these protocols and their implementations for security flaws is another interesting research direction. These protocols are also built around current industry standards, and it is unclear how well they are suited for a trustworthy design like the one proposed in this dissertation. As part of the Enzian project we are experimenting with a clean-slate design called Enzian Firmware Resource Interface (EFRI) [205]. EFRI is designed as a general intra-machine interface protocol between a BMC and device firmware but also between different levels of the software stack like the firmware and OS on an application CPU. With EFRI we hope to gain insights into the requirements for such a protocol.

8.2 Hardware topology and schematics

8.2.1 Extracting topology information

In Section 7.2.2 I explained how to build trustworthy components for a BMC from formal hardware models. To be able to do so, we need instances of these models for the platform at hand. While the models are designed to ease the collection of the needed information (cf. Section 4.5.4), it is still a manual effort to create the hardware descriptions. The formal models are also not the only place where such hardware information is needed. Any systems software needs information about the hardware it is running on. Some hardware is discoverable through mechanisms like ACPI [192] or via PCIe's Enhanced Configuration Access Mechanism (ECAM) [146]. These mechanisms however need to be configured by firmware to work, which just relegates the problem to a lower level. Linux, e.g., encodes static hardware information in Devicetrees [55]. A

lot of this information is hardware topology, like what devices are attached to a certain bus or, in the case of our power topology model, which rails supply a certain device and which regulator drives the rail. This topology information is contained in the schematics for a platform. For large platforms, reading these schematics can be tedious – the schematics for the Enzian mainboard are 121 pages long. Extracting the information manually is an error-prone process. However, this information can be extracted in machine-readable format from the CAD tool that was used to design the platform. These so called *netlists* can then be fed to a tool that automates at least parts of the process. In preliminary experiments we were able to extract I^2C bus topologies from the Enzian netlists [200].

8.2.2 Generating netlists from specifications

When we upgraded the BMC module on Enzian to support running a virtualized Linux over seL4 (cf. Section 7.2.1) we faced the opposite challenge: the new BMC module was not pin-compatible with the original one, so I had to design a shim PCB that slots between the Enzian mainboard and the BMC daughter board. Not only did this board have to reroute the majority of signals, it also needed to level-shift a few as the new board offers fewer 3.3 V pins and more 1.8 V pins instead. The search space for finding a working pin allocation with a minimum amount of level-shifters needed is increased by the fact that some banks on these SoMs have configurable I/O voltages. These are configured by applying the desired voltage to a set of reference voltage pins. The manufacturer of the modules offers spreadsheets with the pinouts that then have to be cross-referenced with the reference manual to determine which reference pins determine the I/O voltages of the various banks. To the best of my knowledge this is the current state of the art.

I solved the problem by manually defining the rerouting done by the shim board and then building a relational model of the new BMC module the shim board and the Enzian mainboard. I could then check whether my solution correctly rerouted all the signals to pins with matching reference voltages by joining the tables on the pin number.

This not very confidence-inspiring process led us to experiment with using an satisfiability modulo theories (SMT) solver to automatically come up with a

solution for such problems. We defined a SMT encoding for PCB components and netlists and the solver was able to come up with solutions for the toy examples we fed it. I think that this is a promising direction that needs further investigation.

8.3 Opening BMCs for research

So far I have described research directions that benefit a trustworthy BMC design. Throughout this dissertation we have mostly looked at BMCs through the lens of security and safety. From this perspective the immense power BMCs wield over the platforms they manage is a challenge. However, their central position in modern computer systems is also an opportunity. Research into the power-related behavior (e.g., power consumption or effects of marginal voltage and clock supply on processors) requires instrumentation [71, 189, 158]. A BMC has access to power consumption information about the entire platform and can adjust voltage levels and clock frequencies freely. Open access to the BMC can therefore offer some of this instrumentation for free. What's more, contrary to using mechanisms like Intel's RAPL [53], the BMC enables out-of-band access to these features. The power consumption data can therefore be gathered without interfering with the analyzed workload at all. The Enzian BMC can query each sensor on the platform in about 5 ms. Detailed power measurements coupled with application and CPU performance metrics can be used to develop models for power-aware resource scheduling and allocation.

We demonstrate the Enzian BMC's capabilities in this regard by monitoring the primary power regulators for the CPU and FPGA cores and the CPU-side DRAM channels, sampling each every 20 ms. Figure 8.1 shows a time series of this power data as Enzian boots (with a power spike as the CPU is powered on), checks DRAM, runs a series of memory tests on the CPU, and then initiates an FPGA stress test by switching blocks of flip-flops on every clock cycle. As can be seen, the Enzian BMC can both sample at high rates and offers detailed per-domain measurements, allowing researchers to examine, in real time, the energy performance of hybrid applications and systems software. In practice, all the power and clock regulators in the system, together with a dozen temperature sensors, can be monitored in this way. The smaller FPGA on the BMC can

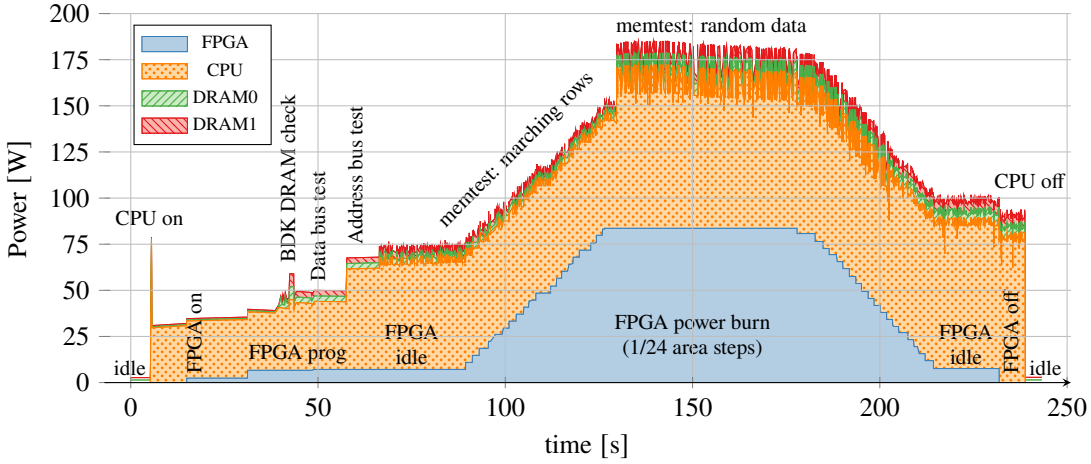


Figure 8.1: Using the Enzian BMC to measure the power consumption of primary components during a boot, diagnostic, and stress test

also provide hardware support for such measurements, facilitating targeted and precise power and performance research.

The potential of offering a very fine-grained but safe power management interface (cf. Section 8.1.3) also opens up opportunities for research into, e.g., performance-aware power managers [72, 73].

8.4 BMCs and the *de-facto* OS

While I focused on BMCs in this dissertation, platform management is a larger problem. Almost every single chip on a PCB runs some sort of firmware or OS kernel. This body of software – even if some of it is called firmware – *as a whole* manages the hardware in the system and therefore fulfills the function of an OS. In other work we have termed this collection of software the *de-facto* OS of a computer system [65]. This is to stress the fact that while it was not *designed* as an OS it effectively *is*. While BMCs take an important role in this *de-facto* OS, we need more than trustworthy BMCs to truly achieve trustworthy platform management. We need *holistically designed* systems as advocated by one of the founders of Oxide Computer [27].

One symptom of the lack of design in the *de-facto* OS are so-called cross-SoC attacks [65]. In these exploits an attacker compromises the firmware of one of the many devices in a system. Further devices can then be compromised by abusing misconfigured memory protection units. There is at least one such vulnerability that affects BMCs [49]. In affected systems the BMC's memory can be compromised from the application processor through a DMA-capable host-to-BMC bridge. More research is needed to systematically prevent these types of bugs.

9

Conclusion

Modern computer systems are so complex that they need a computer inside the computer to function. BMCs are responsible for everything from turning the system on, over provisioning firmware for components on the board, to monitoring and remote management. This makes them one of the most critical systems in modern compute infrastructure. However, the state of the art for them does not live up to this role.

Because of their traditionally proprietary and closed nature, BMCs have been neglected by the research community. Inspired by the very instructive experience of building a BMC for a heterogeneous research system, this dissertation changes that. In Chapter 3, I identified the major problems with the current state of the art. In the remainder of the dissertation I then proposed several pieces to a solution.

The experience with developing the power management stack for Enzian made me aware of how complex the power and clock distribution networks in modern computers are. At the same time there is a lack of publicly available literature on the topic. In Chapter 4, presented a model that facilitates describing these systems and formally captures the complex sequencing requirements. Furthermore, I have demonstrated that we can generate working power sequencing code that can be used on a BMC. These generated sequences, backed by a formal model, provide both higher assurance and better automation than traditionally hand-crafted ones.

To ensure the semantics of the generated sequences are carried through, we also need reliable drivers for the power regulators. They are connected to the

BMC over chip-to-chip protocols like I^2C. Efeu, the framework I introduced in Chapter 6, produces drivers for such devices from specifications. The specifications are model-checked to rule out inconsistencies between devices that could result in driver bugs and system failures. We can generate both hardware and software to provide on-par performance with stacks based on off-the shelf I^2C controllers. We demonstrated that the generated stacks work on real hardware. While we have not yet built a system of the size of a typical server's power distribution network, I am confident that this can be achieved.

Finally, in Chapter 7 I proposed a system design for a trustworthy BMC. It uses a state-of-the-art formally verified microkernel to provide isolation between components. This allows for trusted and untrusted components to securely coexist. The proposal also includes a migration strategy to retrofit existing BMCs with this secure design.

Despite the remaining challenges and open questions (cf. Chapter 8) I believe that the research in this dissertation is an important step to a future of trustworthy BMCs. Apart from contributing concrete solutions for some of the major problems with BMCs, it hopefully also helps raise researchers' interest in the problem.

Its closed nature does not just prevent assessing the quality of firmware but impedes *understanding* of how hardware works at the lowest level visible by software. The experience with Enzian shows that very little of the hardware is documented at this level. Efforts to create open source firmware are therefore crucial to make this knowledge available outside of chip and board manufacturing companies.

Power sequencing is just one example of an area where very little concrete knowledge exists outside these silos. Another one is DRAM initialization. There are several timing parameters and reference voltages that need to be correctly set for each individual memory module. Standards like DDR 4 schematically describe how to find the correct values for these parameters [91]. In reality however, what software interacts with are memory controllers with an enormously complex and badly documented register interface. This is again something learned the hard way when trying to re-implemented DRAM training for Enzian [109]. The ThunderX-1's memory controllers have about 100 registers and anecdotal evidence from private conversations suggest that this is dwarfed by the memory controller interfaces in other CPUs. While the ThunderX-1 manual

describes a 15-step initialization sequence for the memory controllers, the code supplied by the manufacturer inserts several undocumented steps that seem to be crucial for reliable operation. This code however also supports several other chips and DRAM standards and isolating which parts are important for the CPU on Enzian is not an easy task – the full code consist of about 10 thousand lines.

We need more efforts that make such knowledge available. As OS researchers, we need to understand how computers work at this level if we ever want a chance at building truly trustworthy platforms.

List of Figures

2.1 Picture of the Enzian board with the BMC 8
2.2 The testbed used to prototype the power sequencing software . . 12
2.3 Debugging power sequencing code 13
2.4 Picture of the setup for the first bringup of an Enzian board. . . . 16

3.1 Number of vulnerabilities in BMCs over time 19
3.2 Percentage of BMC vulnerabilities per category. 21
3.3 Breakdown of non-critical BMC vulnerabilities 23

4.1 Power tree of a modern two socket server (Enzian) 28
4.2 Detail view of a power tree 32
4.3 IC output voltage range as a function of inputs 33
4.4 Illustrative example of a sequencing graph for an IC 35
4.5 Finding a path through the state table. 39
4.6 First lines of the generated power sequence 41
4.7 Histograms of solving times for problems P1, P2, and P3. 43

5.1 Update regulator model. 54
5.2 Scaling behavior of sequence generation time 59

6.1 The I^2C stack 66
6.2 Timing diagram of a 1-byte read at a given EEPROM offset . . . 69
6.3 Efeu workflow. 71
6.4 ESI for controller `Transaction` and `EepDriver` layers 72
6.5 Examples of call graphs. 75
6.6 Transforming a `talk` into a function call or a continuation . . . 76
6.7 Multiple hardware/software boundaries 79
6.8 Architecture of the `Byte` verifier. 81
6.9 Verification runtime of multiple EEPROMs 86
6.10 Achievable bandwidth and CPU usage 92

6.11 Waveforms of the first few SCL cycles 93
6.12 LUT utilization. 94
6.13 FF utilization. 95

7.1 Trustworthy BMC architecture 110

8.1 Using the Enzian BMC to measure power consumption 120

List of Tables

4.1 An overview of problem instances P1 to P3 44
4.2 Overview of problem instances P1 to P6 46
4.3 Measurements for P1 to P6 46

6.1 Source code lines of layers 84
6.2 Average verification runtime in seconds 85
6.3 Source code lines for MMIO-AXI Lite interfaces 89

List of Acronyms

ACPI	Advanced Configuration and Power Interface
AGESA	AMD Generic Encapsulated Software Architecture
AST	abstract syntax tree
ATF	Arm Trusted Firmware
BMC	Baseboard Management Controller
CAD	computer-aided design
CPLD	complex programmable logic device
CPU	central processing unit
DDR	Double Data Rate
DFS	depth-first search
DMA	direct memory access
DoS	denial of service
DRAC	Dell Remote Access Controller
DRAM	Dynamic Random Access Memory
DSL	domain specific language
DVFS	dynamic voltage and frequency scaling
ECAM	Enhanced Configuration Access Mechanism
EDA	electronic design automation
EFRI	Enzian Firmware Resource Interface
FF	flip-flop
FPGA	Field Programmable Gate Array
FSM	finite state machine

GPU	graphics processing unit
I²C	Inter-Integrated Circuit
iLO	Integrated Lights-Out
IoT	internet of things
IPMI	Intelligent Platform Management Interface
LUT	look-up table
MCTP	Management Component Transport Protocol
MMU	memory management unit
NIST	National Institute of Standards and Technology
ODM	original design manufacturer
OS	operating system
PCB	printed circuit board
PCIe	PCI Express
PLDM	Platform Level Data Model
PMBus	Power Management Bus
QoS	quality of service
REST	representational state transfer
RoT	root of trust
SCL	serial clock line
SDA	serial data line
SMT	satisfiability modulo theories
SoC	system-on-chip
SoM	system-on-module
SSA	static single-assignment

TCB trusted compute base
TDP thermal design power

UIO userspace I/O
USB Universal Serial Bus

XCC xClarity Controller
XSS cross-site scripting

Bibliography

[1] *24AA512/24LC512/24FC512 512K I2C Serial EEPROM*. English. Version 1.0. Microchip. 2021. 45 pp. (cit. on pp. 70, 88).

[2] Advanced Micro Devices. *AMD Generic Encapsulated Software Architecture (AGESA) Interface Specification for Arch2008*. Version 3.04. Jan. 2017 (cit. on p. 1).

[3] Advanced Micro Devices. *Zynq UltraScale+ MPSoC Data Sheet: Overview (DS891)*. en. v1.10. Nov. 2022. URL: https://www.amd.com/content/dam/xilinx/support/documents/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf (cit. on pp. 9, 96).

[4] Advanced Micro Devices. *Zynq-7000 SoC Data Sheet: Overview (DS190)*. en. v1.11.1. July 2018. URL: https://www.amd.com/content/dam/xilinx/support/documents/data_sheets/ds190-Zynq-7000-Overview.pdf (cit. on p. 7).

[5] Aldo Aguilar-Nadalini, Kuk H Chung, Cecilia Marsicovetere, Juan F Medrano, Emilio Miranda, Víctor Ayerdi, and Luis Zea. "Design and On-Orbit Performance of the Electrical Power System for the Quetzal-1 CubeSat". In: *Journal of Small Satellites* 12.2 (May 2023), pp. 1201–1229 (cit. on p. 66).

[6] Amina Albalooshi, Abdul-Halim M. Jallad, and Prashanth R. Marpu. "Fault Analysis and Mitigation Techniques of the I2C Bus for Nanosatellite Missions". In: *IEEE Access* 11 (2023), pp. 34709–34717. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2023.3262410. URL: https://ieeexplore.ieee.org/document/10082916/ (cit. on p. 67).

[7] Vadim Alimguzhin, Federico Mari, Igor Melatti, Ivano Salvo, and Enrico Tronci. "On Model Based Synthesis of Embedded Control Software". In: *Proceedings of the Tenth ACM International Conference on Embedded Software*. EMSOFT '12. New York, NY, USA: Association for Computing Machinery, Oct. 2012, pp. 227–236. ISBN: 978-1-4503-1425-1. DOI: 10.1145/2380356.2380398. URL: https://doi.org/10.1145/2380356.2380398 (cit. on p. 49).

[8] Thomas Alsop. *Server Vendor Market Share by Quarter 2021*. Nov. 2023. URL: https://www.statista.com/statistics/269396/global-market-share-held-by-server-system-vendors-since-1st-quarter-2009/ (visited on 10/03/2024) (cit. on p. 10).

[9] Altera Corporation. *Enpirion Power Datasheet ES1020QI Power Sequencing Controller*. 2014. URL: https://eu.mouser.com/datasheet/2/612/es1020qi_10041-1299392.pdf (cit. on p. 29).

[10] AMI. *AMI Announces OpenBMC-Based MegaRAC OSP BMC Firmware Solution for Robust, Secure Remote Server Management*. Oct. 2020. URL: https://www.ami.com/blog/2020/10/01/ami-announces-openbmcbased-megarac-osp-bmc-firmware-solution-for-robust-secure-remote-server-management/ (cit. on p. 10).

[11] AMI. *MegaRAC*. URL: https://www.ami.com/megarac/ (visited on 10/22/2024) (cit. on p. 10).

[12] Arm Limited. *Arm Base Boot Requirements*. Version Revision 2.1. Apr. 2024 (cit. on p. 1).

[13] Arm Limited. *Arm Power State Coordination Interface*. Version D. Apr. 2017 (cit. on p. 1).

[14] Arm Limited. *Learn the architecture - An introduction to AMBA AXI*. English. Version 3.0. 2022. 62 pp. (cit. on p. 77).

[15] Arm Limited. *Power Control System Architecture*. Version D. Feb. 2023 (cit. on p. 1).

[16] gRPC Authors. *gRPC*. 2024. URL: https://grpc.io/ (visited on 11/01/2024) (cit. on p. 11).

[17] AWS. *The Security Design of the AWS Nitro System - AWS Whitepaper*. Nov. 2022 (cit. on p. 103).

[18] Vlad Babkin. *Supply Chain Vulnerabilities Put Server Ecosystem At Risk*. Dec. 2022. URL: https://eclypsium.com/blog/supply-chain-vulnerabilities-put-server-ecosystem-at-risk/ (visited on 10/03/2024) (cit. on pp. 10, 17).

[19] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. "A Decade of Software Model Checking with SLAM". In: *Communications of the ACM* 54.7 (July 2011), pp. 68–76. ISSN: 0001-0782. DOI: 10.1145/1965724.1965743. URL: https://dl.acm.org/doi/10.1145/1965724.1965743 (cit. on p. 98).

[20] L. Benini, A. Bogliolo, and G. De Micheli. "A Survey of Design Techniques for System-Level Dynamic Power Management". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 8.3 (June 2000), pp. 299–316. ISSN: 1557-9999. DOI: 10.1109/92.845896 (cit. on p. 49).

[21] Anthony J. Bonkoski, Russ Bielawski, and J. Alex Halderman. "Illuminating the Security Issues Surrounding Lights-out Server Management". In: *Proceedings of the 7th USENIX Conference on Offensive Technologies*. WOOT '13. USA: USENIX Association, Aug. 2013, p. 10 (cit. on pp. 2, 11, 117).

[22] Dragan Bošnački, Aad Mathijssen, and Yaroslav S. Usenko. "Behavioural Analysis of an I2C Linux Driver". In: *Formal Methods for Industrial Critical Systems*. Ed. by María Alpuente, Byron Cook, and Christophe Joubert. Berlin, Heidelberg: Springer, 2009, pp. 205–206. ISBN: 978-3-642-04570-7. DOI: 10.1007/978-3-642-04570-7_18 (cit. on p. 98).

[23] Jasper Bouwmeester, Martin Langer, and Eberhard Gill. “Survey on the Implementation and Reliability of CubeSat Electrical Bus Interfaces”. In: *CEAS Space Journal* 9.2 (June 2017), pp. 163–173. ISSN: 1868-2510. DOI: 10.1007/s12567-016-0138-0. URL: https://doi.org/10.1007/s12567-016-0138-0 (cit. on p. 66).

[24] Richard Braun, Abishek Ramdas, Michal Friedman, and Gustavo Alonso. “PLayer: Expanding Coherence Protocol Stack with a Persistence Layer”. In: *Proceedings of the 1st Workshop on Disruptive Memory Systems*. DIMES ’23. New York, NY, USA: Association for Computing Machinery, Oct. 2023, pp. 8–15. ISBN: 9798400703003. DOI: 10.1145/3609308.3625270. URL: https://dl.acm.org/doi/10.1145/3609308.3625270 (visited on 10/24/2024) (cit. on p. 110).

[25] Matthias Brun, Reto Achermann, Tej Chajed, Jon Howell, Gerd Zellweger, and Andrea Lattuada. “Beyond Isolation: OS Verification as a Foundation for Correct Applications”. In: *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*. HotOS ’23. New York, NY, USA: Association for Computing Machinery, June 2023, pp. 158–165. ISBN: 9798400701955. DOI: 10.1145/3593856.3595899. URL: https://dl.acm.org/doi/10.1145/3593856.3595899 (visited on 10/22/2024) (cit. on p. 106).

[26] Thomas Burd, Noah Beck, Sean White, Milam Paraschou, Nathan Kalyanasundharam, Gregg Donley, Alan Smith, Larry Hewitt, and Samuel Naffziger. ““Zeppelin”: An SoC for Multichip Architectures”. In: *IEEE Journal of Solid-State Circuits* 54.1 (Jan. 2019), pp. 133–143. ISSN: 1558-173X. DOI: 10.1109/JSSC.2018.2873584. URL: https://ieeexplore.ieee.org/document/8510845 (cit. on pp. 1, 27).

[27] Bryan Cantrill. *I Have Come to Bury the BIOS, Not to Open It: The Need for Holistic Systems*. Sept. 2022. URL: https://osfc.io/2022/talks/i-have-come-to-bury-the-bios-not-to-open-it-the-need-for-holistic-systems/ (visited on 10/21/2024) (cit. on pp. 105, 121).

[28] Hao Chen, Xiongnan (Newman) Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. “Toward Compositional Verification of Interruptible OS Kernels and Device Drivers”. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’16. New York, NY, USA: Association for Computing Machinery, June 2016, pp. 431–447. ISBN: 978-1-4503-4261-2. DOI: 10.1145/2908080.2908101. URL: https://dl.acm.org/doi/10.1145/2908080.2908101 (cit. on p. 99).

[29] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. “Linux Kernel Vulnerabilities: State-of-the-Art Defenses and Open Problems”. In: *Proceedings of the Second Asia-Pacific Workshop on Systems*. APSys ’11. Shanghai, China: Association for Computing Machinery, July 2011, pp. 1–5. ISBN: 978-1-4503-1179-3. DOI: 10.1145/2103799.2103805. URL: https://doi.org/10.1145/2103799.2103805 (visited on 07/29/2020) (cit. on p. 18).

[30] Zitai Chen and David Oswald. "PMFault: Faulting and Bricking Server CPUs through Management Interfaces: Or: A Modern Example of Halt and Catch Fire". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2023 (2 Mar. 2023), pp. 1–23. ISSN: 2569-2925. DOI: 10.46586/tches.v2023.i2.1-23. URL: https://tches.iacr.org/index.php/TCHES/article/view/10275 (cit. on pp. 25, 66).

[31] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. "An Empirical Study of Operating Systems Errors". In: *ACM SIGOPS Operating Systems Review* 35.5 (Oct. 2001), pp. 73–88. ISSN: 0163-5980. DOI: 10.1145/502059.502042. URL: https://dl.acm.org/doi/10.1145/502059.502042 (cit. on pp. 48, 65, 98).

[32] Pai Chou, Ross Ortega, and Gaetano Borriello. "Synthesis of the Hardware/Software Interface in Microcontroller-Based Systems". In: *1992 IEEE/ACM International Conference on Computer-Aided Design*. USA: IEEE Computer Society, Nov. 1992, pp. 488–495. DOI: 10.1109/ICCAD.1992.279322. URL: https://ieeexplore.ieee.org/document/279322 (cit. on p. 97).

[33] Pai Chou, Ross B. Ortega, and Gaetano Borriello. "Interface Co-Synthesis Techniques for Embedded Systems". In: *Proceedings of the 1995 IEEE/ACM International Conference on Computer-aided Design*. ICCAD '95. USA: IEEE Computer Society, Dec. 1995, pp. 280–287. ISBN: 978-0-8186-7213-2 (cit. on p. 97).

[34] Pai H. Chou, Ross B. Ortega, and Gaetano Borriello. "The Chinook Hardware/Software Co-Synthesis System". In: *Proceedings of the 8th International Symposium on System Synthesis*. ISSS '95. New York, NY, USA: Association for Computing Machinery, Sept. 1995, pp. 22–27. ISBN: 978-0-89791-771-1. DOI: 10.1145/224486.224491. URL: https://dl.acm.org/doi/10.1145/224486.224491 (cit. on pp. 68, 97).

[35] Clang Team. *Clang: C Language Family Frontend for LLVM*. 2024. URL: https://clang.llvm.org/ (visited on 05/21/2024) (cit. on p. 74).

[36] Clang Team. *Clang: Clang::DiagnosticsEngine Class Reference*. 2024. URL: https://clang.llvm.org/doxygen/classclang%5C_1%5C_1DiagnosticsEngine.html (visited on 05/21/2024) (cit. on p. 74).

[37] Clang Team. *Clang: Clang::Rewriter Class Reference*. 2024. URL: https://clang.llvm.org/doxygen/classclang_1_1Rewriter.html (visited on 05/21/2024) (cit. on p. 76).

[38] Clang Team. *ClangFormat*. 2024. URL: https://clang.llvm.org/docs/ClangFormat.html (visited on 05/06/2024) (cit. on pp. 83, 88).

[39] David Cock, Abishek Ramdas, Daniel Schwyn, Michael Giardino, Adam Turowski, Zhenhao He, Nora Hossle, Dario Korolija, Melissa Licciardello, Kristina Martsenko, Reto Achermann, Gustavo Alonso, and Timothy Roscoe. "Enzian: An Open, General, CPU/FPGA Platform for Systems Software Research". In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS 2022. Lausanne, Switzerland: Association for Computing Machinery, Mar. 2022, pp. 434–451. ISBN: 9781450392051. DOI: 10.1145/3503222.3507742. URL: https://doi.org/10.1145/3503222.3507742 (cit. on pp. 3, 5, 7, 9, 29).

[40] Darren Cofer, Andrew Gacek, John Backes, Michael W. Whalen, Lee Pike, Adam Foltzer, Michal Podhradsky, Gerwin Klein, Ihor Kuz, June Andronick, Gernot Heiser, and Douglas Stuart. "A Formal Approach to Constructing Secure Air Vehicle Software". In: *Computer* 51.11 (Nov. 2018), pp. 14–23. ISSN: 1558-0814. DOI: 10.1109/MC.2018.2876051 (cit. on pp. 106, 110, 115).

[41] LLVM development community. *LLVM Language Reference Manual*. 2024. URL: https://llvm.org/docs/LangRef.html (visited on 05/21/2024) (cit. on p. 76).

[42] Oxide Computer Company. *Introduction / Guides*. Sept. 2024. URL: https://docs.oxide.computer/guides/introduction (visited on 10/21/2024) (cit. on p. 104).

[43] Oxide Computer Company. *Networking / Guides*. Sept. 2024. URL: https://docs.oxide.computer/guides/architecture/networking (visited on 10/21/2024) (cit. on p. 104).

[44] Christopher L. Conway and Stephen A. Edwards. "NDL: A Domain-Specific Language for Device Drivers". In: *ACM SIGPLAN Notices* 39.7 (June 2004), pp. 30–36. ISSN: 0362-1340. DOI: 10.1145/998300.997169. URL: https://dl.acm.org/doi/10.1145/998300.997169 (cit. on p. 98).

[45] Nuvoton Technology Corporation. *iBMC*. 2024. URL: https://www.nuvoton.com/products/cloud-computing/ibmc (visited on 10/22/2024) (cit. on p. 9).

[46] CVE Project. *CVE-2013-4783*. July 2013. URL: https://nvd.nist.gov/vuln/detail/CVE-2013-4783 (visited on 10/29/2024) (cit. on p. 17).

[47] CVE Project. *CVE-2019-4169*. Jan. 2019. URL: https://nvd.nist.gov/vuln/detail/CVE-2019-4621 (visited on 10/08/2020) (cit. on p. 2).

[48] CVE Project. *CVE-2019-4621*. Jan. 2019. URL: https://nvd.nist.gov/vuln/detail/CVE-2019-4621 (visited on 10/08/2020) (cit. on p. 2).

[49] CVE Project. *CVE-2019-6260*. Jan. 2019. URL: https://nvd.nist.gov/vuln/detail/CVE-2019-6260 (visited on 11/01/2024) (cit. on p. 121).

[50] CVE Project. *CVE-2020-14156*. June 2020. URL: https://nvd.nist.gov/vuln/detail/CVE-2020-14156 (visited on 10/08/2020) (cit. on p. 2).

[51] CVE Project. *CVE-2024-26593*. Feb. 2024. URL: https://nvd.nist.gov/vuln/detail/CVE-2024-26593 (visited on 05/05/2024) (cit. on p. 67).

[52] Albert Danial. *cloc: v2.00*. Version v2.00. Feb. 2024 (cit. on pp. 83, 88).

[53] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. "RAPL: Memory Power Estimation and Capping". In: *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*. Aug. 2010, pp. 189–194. DOI: 10.1145/1840845.1840883. URL: https://ieeexplore.ieee.org/document/5599016 (visited on 10/30/2024) (cit. on p. 119).

[54] Dell. *Integrated Dell Remote Access Controller (iDRAC)*. URL: https://www.dell.com/en-us/lp/dt/open-manage-idrac (visited on 10/03/2024) (cit. on p. 10).

[55] devicetree.org. *Devicetree Specification*. Version v0.3. Feb. 2020. URL: https://github.com/devicetree-org/devicetree-specification/releases/download/v0.3/devicetree-specification-changebars-v0.3.pdf (cit. on pp. 50, 117).

[56] DMTF. *Management Component Transport Protocol (MCTP) Base Specification*. Version 1.3.0. Nov. 2016 (cit. on p. 117).

[57] DMTF. *Platform Level Data Model (PLDM) Base Specification*. Version 1.0.0. Apr. 2009 (cit. on p. 117).

[58] DMTF. *Redfish specification*. Aug. 2024. URL: https://www.dmtf.org/sites/default/files/standards/documents/DSP0266_1.21.0.pdf (visited on 10/22/2024) (cit. on pp. 11, 117).

[59] Krzysztof Domanski. "Latch-up in FinFET Technologies". In: *2018 IEEE International Reliability Physics Symposium (IRPS)*. Mar. 2018, pp. 2C.4-1-2C.4–5. DOI: 10.1109/IRPS.2018.8353550. URL: https://ieeexplore.ieee.org/document/8353550 (cit. on p. 27).

[60] Marvin Drees. *u-bmc, The Next Gen BMC Software Stack Born from the u-root Ecosystem*. Nov. 2021. URL: https://talks.osfc.io/osfc2021/talk/MA7KHW/ (cit. on p. 11).

[61] Eclypsium. *Virtual Media Vulnerability in BMC Opens Servers to Remote Attack*. Sept. 2019. URL: https://eclypsium.com/blog/virtual-media-vulnerability-in-bmc-opens-servers-to-remote-attack/ (visited on 08/22/2024) (cit. on p. 17).

[62] Enzian Team. *The Enzian Research Computer; Schematics*. Version 1.5c. Apr. 2022. DOI: 10.5281/zenodo.6465908. URL: https://doi.org/10.5281/zenodo.6465908 (cit. on p. 11).

[63] Shuying Fan and Supriya Velagapudi. *Implementing Next-Generation Data Center Platform Management Using Agilex 3 and Agilex 5 Devices*. Intel Corp., Sept. 2023. URL: https://www.intel.com/content/www/us/en/content-details/787067/implementing-next-generation-data-center-platform-management-using-agilex-5-devices.html (visited on 05/21/2024) (cit. on p. 67).

[64] Tian Fang. *Introducing "OpenBMC": an open software framework for next-generation system management*. Engineering at Meta, 2015. URL: https://engineering.fb.com/open-source/introducing-openbmc-an-open-software-framework-for-next-generation-system-management/ (visited on 10/03/2024) (cit. on pp. 2, 10).

[65] Ben Fiedler, Daniel Schwyn, Constantin Gierczak–Galle, David Cock, and Timothy Roscoe. "Putting out the hardware dumpster fire". In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. HotOS '23. Providence, Rhode Island: Association for Computing Machinery, June 2023, pp. 159–166. ISBN: 9798400701955. DOI: 10.1145/3593856.3595903. URL: https://doi.org/10.1145/3593856.3595903 (cit. on pp. 1, 65, 121).

[66] Linux Foundation. *The Yocto Project*. URL: https://www.yoctoproject.org/ (visited on 10/08/2024) (cit. on p. 11).

[67] Jessie Frazelle. “Open Source Firmware”. In: *Commun. ACM* 62.10 (Sept. 2019), pp. 34–38. ISSN: 0001-0782. DOI: 10.1145/3343042. URL: https://doi.org/10.1145/3343042 (cit. on p. 2).

[68] Jessie Frazelle. “Opening up the Baseboard Management Controller”. In: *Commun. ACM* 63.2 (Jan. 2020), pp. 38–40. ISSN: 0001-0782. DOI: 10.1145/3369758. URL: https://doi.org/10.1145/3369758 (cit. on pp. 2, 9, 10, 17).

[69] A. Ganapathi, Viji Ganapathi, and D. Patterson. “Windows XP Kernel Crash Analysis”. In: *LiSA*. Berkeley, CA, USA: USENIX Association, Dec. 2006, pp. 149–159. URL: https://www.usenix.org/legacy/events/lisa06/tech/ganapathi.html (cit. on p. 65).

[70] Gernot Heiser. *Lions OS: Secure, Fast, Adaptable!* Oct. 2024. URL: https://www.youtube.com/watch?v=W8Ka_8kHTj4 (visited on 10/24/2024) (cit. on p. 109).

[71] Saugata Ghose, Abdullah Giray Yaglikçi, Raghav Gupta, Donghyuk Lee, Kais Kudrolli, William X. Liu, Hasan Hassan, Kevin K. Chang, Niladrish Chatterjee, Aditya Agrawal, Mike O’Connor, and Onur Mutlu. “What Your DRAM Power Models Are Not Telling You: Lessons from a Detailed Experimental Study”. In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2.3 (Dec. 2018), 38:1–38:41. DOI: 10.1145/3224419. URL: https://doi.org/10.1145/3224419 (visited on 05/20/2021) (cit. on p. 119).

[72] Michael Giardino, Eric Klawitter, Bonnie Ferri, and Aldo Ferri. “A Power- and Performance-Aware Software Framework for Control System Applications”. In: *IEEE Transactions on Computers* 69.10 (Oct. 2020), pp. 1544–1555. ISSN: 1557-9956. DOI: 10.1109/TC.2020.2978468. URL: https://ieeexplore.ieee.org/document/9025173 (cit. on pp. 49, 121).

[73] Michael Giardino, Daniel Schwyn, Bonnie Ferri, and Aldo Ferri. “Low-Overhead Reinforcement Learning-Based Power Management Using 2QoSM”. In: *Journal of Low Power Electronics and Applications* 12.2 (May 2022). ISSN: 2079-9268. DOI: 10.3390/jlpea12020029. URL: https://www.mdpi.com/2079-9268/12/2/29 (cit. on p. 121).

[74] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. “CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 653–669. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu (visited on 10/15/2024) (cit. on p. 106).

[75] R.K. Gupta, S. Irani, and S.K. Shukla. “Formal Methods for Dynamic Power Management”. In: *ICCAD-2003. International Conference on Computer Aided Design (IEEE Cat. No. 03CH37486)*. Nov. 2003, pp. 874–881. DOI: 10.1109/ICCAD.2003.159778. URL: https://ieeexplore.ieee.org/document/1257911 (cit. on p. 49).

[76] Royal Hansen and Dominic Rizzo. *OpenTitan - Open Sourcing Transparent, Trustworthy, and Secure Silicon*. Nov. 2019. URL: https://security.googleblog.com/2019/11/opentitan-open-sourcing-transparent.html (visited on 10/18/2024) (cit. on p. 103).

[77] Jeff Heath and Akin Kestelli. *Flexible Power Supply Sequencing and Monitoring*. Analog Devices, 2005. URL: https://www.analog.com/en/technical-articles/flexible-power-supply-sequencing-monitoring.html (visited on 10/03/2024) (cit. on p. 29).

[78] Cedric Heimhofer. "Towards high-assurance Board Management Controller software". Master's Thesis. ETH Zurich, Mar. 2021. DOI: 10.3929/ethz-b-000490635. URL: https://doi.org/10.3929/ethz-b-000490635 (cit. on p. 6).

[79] Gernot Heiser. *The seL4 Microkernel – An Introduction*. May 2024. URL: https://sel4.systems/About/seL4-whitepaper.pdf (visited on 10/24/2024) (cit. on pp. 109, 113).

[80] Gernot Heiser, Lucy Parker, Peter Chubb, Ivan Velickovic, and Ben Leslie. "Can We Put the "S" Into IoT?" In: *2022 IEEE 8th World Forum on Internet of Things (WF-IoT)*. New York, NY, USA: IEEE, Oct. 2022, pp. 1–6. DOI: 10.1109/WF-IoT54382.2022.10152198. URL: https://ieeexplore.ieee.org/document/10152198 (cit. on pp. 78, 109, 115).

[81] G.J. Holzmann. "The Model Checker SPIN". In: *IEEE Transactions on Software Engineering* 23.5 (May 1997), pp. 279–295. ISSN: 1939-3520. DOI: 10.1109/32.588521. URL: https://ieeexplore.ieee.org/document/588521 (cit. on p. 64).

[82] Gerard J. Holzmann. "Logic Verification of ANSI-C Code with SPIN". en. In: *SPIN Model Checking and Software Verification*. Ed. by Klaus Havelund, John Penix, and Willem Visser. Berlin, Heidelberg: Springer, 2000, pp. 131–147. ISBN: 978-3-540-45297-3. DOI: 10.1007/10722468_8 (cit. on p. 73).

[83] HPE. *Integrated Lights-Out (iLO)*. URL: https://www.hpe.com/us/en/hpe-integrated-lights-out-ilo.html (visited on 10/30/2024) (cit. on p. 10).

[84] HPE. *OpenBMC Enablement on HPE ProLiant Servers*. 2024. URL: https://www.hpe.com/us/en/compute/openbmc-proliant-servers.html (cit. on p. 10).

[85] Jingmei Hu, Eric Lu, David A. Holland, Ming Kawaguchi, Stephen Chong, and Margo I. Seltzer. "Trials and Tribulations in Synthesizing Operating Systems". In: *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*. PLOS'19. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 67–73. ISBN: 978-1-4503-7017-2. DOI: 10.1145/3365137.3365401. URL: https://doi.org/10.1145/3365137.3365401 (cit. on p. 49).

[86] Lukas Humbel, Daniel Schwyn, Nora Hossle, Roni Haecki, Melissa Licciardello, Jan Schaer, David Cock, Michael Giardino, and Timothy Roscoe. "A Model-Checked I^2C Specification". In: *27th International Symposium on Model Checking Software (SPIN 2021)*. Ed. by Alfons Laarman and Ana Sokolova. Cham: Springer International Publishing, Aug. 2021, pp. 177–193. ISBN: 978-3-030-84629-9. DOI: 10.1007/978-3-030-84629-9_10. URL: https://doi.org/10.1007/978-3-030-84629-9_10 (cit. on pp. 64, 70, 81, 82).

[87] Giang Nguyen Thi Huong. "GCC2Verilog Compiler Toolset for Complete Translation of C Programming Language into Verilog HDL". en. In: *ETRI Journal* 33.5 (Oct. 2011), pp. 731–740. ISSN: 1225-6463. DOI: 10.4218/etrij.11.0110.0654. (Visited on 05/21/2024) (cit. on p. 73).

[88] *I2C-bus specification and user manual*. English. Version 7.0. NXP Semiconductors. Oct. 2021. 62 pp. (cit. on pp. 66, 68, 89).

[89] ASPEED Technology Inc. *ASPEED - Server Management*. 2024. URL: https://www.aspeedtech.com/server (visited on 10/22/2024) (cit. on p. 9).

[90] Intel, Hewlett-Packard, NEC, and Dell. *Intelligent Platform Management Interface Specification 2nd Generation*. Version 1.1. Oct. 2013. URL: https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/ipmi-intelligent-platform-mgt-interface-spec-2nd-gen-v2-0-spec-update.pdf (visited on 10/22/2024) (cit. on pp. 11, 117).

[91] JEDEC Solid State Technology Association. *DDR4 SDRAM*. Sept. 2012 (cit. on p. 124).

[92] Ke Jiang. "Model Checking C Programs by Translating C to Promela". en. MA thesis. Uppsala, Sweden: Uppsala Universitet, Sept. 2009. URL: http://www.diva-portal.org/smash/get/diva2:235718/FULLTEXT01.pdf (cit. on p. 73).

[93] Asim Kadav and Michael M. Swift. "Understanding Modern Device Drivers". In: *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVII. New York, NY, USA: Association for Computing Machinery, Mar. 2012, pp. 87–98. ISBN: 978-1-4503-0759-8. DOI: 10.1145/2150976.2150987. URL: https://doi.org/10.1145/2150976.2150987 (cit. on p. 48).

[94] kaedros. *Raspberry Pi I2C clock-stretching bug GitHub Issue*. Feb. 2022. URL: https://github.com/raspberrypi/linux/issues/4884 (visited on 09/05/2024) (cit. on p. 68).

[95] Keysight Technologies, Inc. *Keysight InfiniiVision 3000T X-Series Oscilloscopes User's Guide*. en. June 2020. (Visited on 08/08/2023) (cit. on p. 88).

[96] Moonzoo Kim, Yunja Choi, Yunho Kim, and Hotae Kim. "Formal Verification of a Flash Memory Device Driver – An Experience Report". In: *Model Checking Software*. Ed. by Klaus Havelund, Rupak Majumdar, and Jens Palsberg. Berlin, Heidelberg: Springer, 2008, pp. 144–159. ISBN: 978-3-540-85114-1. DOI: 10.1007/978-3-540-85114-1_12 (cit. on p. 99).

[97] Myron King, Nirav Dave, and Arvind. “Automatic Generation of Hardware/Software Interfaces”. In: *ACM SIGARCH Computer Architecture News* 40.1 (Mar. 2012), pp. 325–336. ISSN: 0163-5964. DOI: 10.1145/2189750.2151011. URL: https://dl.acm.org/doi/10.1145/2189750.2151011 (cit. on p. 97).

[98] Gerwin Klein. “Operating System Verification—An Overview”. In: *Sadhana* 34.1 (Feb. 2009), pp. 27–69. ISSN: 0973-7677. DOI: 10.1007/s12046-009-0002-4. URL: https://doi.org/10.1007/s12046-009-0002-4 (visited on 10/16/2024) (cit. on p. 106).

[99] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. “seL4: Formal Verification of an OS Kernel”. In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. SOSP ’09. New York, NY, USA: ACM, 2009, pp. 207–220. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596. URL: http://doi.acm.org/10.1145/1629575.1629596 (cit. on pp. 78, 99, 102, 106).

[100] Arjen Klomp, Herman Roebbers, Ruud Derwig, and Leon Bouwmeester. “Designing a Mathematically Verified I2C Device Driver Using ASD”. In: *Communicating Process Architectures 2009*. IOS Press, 2009, pp. 105–116. DOI: 10.3233/978-1-60750-065-0-105. URL: https://ebooks.iospress.nl/doi/10.3233/978-1-60750-065-0-105 (cit. on p. 99).

[101] Donald E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*. 3rd. Vol. 1. USA: Addison Wesley Longman Publishing Co., Inc., 1997. ISBN: 0201896834 (cit. on p. 73).

[102] Hans-Jürgen Koch. Dec. 2006. URL: https://www.kernel.org/doc/html/latest/driver-api/uio-howto.html (visited on 05/21/2014) (cit. on p. 78).

[103] *KS0127 Data Sheet*. en. Samsung Electronics. Feb. 1998, p. 86. URL: https://alltransistors.com/superdatasheets/_pdf/09/ks0127.pdf (visited on 05/11/2024) (cit. on pp. 67, 85, 87).

[104] *KS0127B Data Sheet*. en. Samsung Electronics. May 2000, p. 96. URL: https://alltransistors.com/superdatasheets/_pdf/09/ks0127b.pdf (visited on 05/11/2024) (cit. on p. 85).

[105] Sudipta Kundu, Sorin Lerner, and Rajesh Gupta. “Validating High-Level Synthesis”. en. In: *Computer Aided Verification*. Ed. by Aarti Gupta and Sharad Malik. Berlin, Heidelberg: Springer, 2008, pp. 459–472. DOI: 10.1007/978-3-540-70545-1_44 (cit. on p. 80).

[106] Ihor Kuz, Yan Liu, Ian Gorton, and Gernot Heiser. “CAmkES: A Component Model for Secure Microkernel-Based Embedded Systems”. In: *Journal of Systems and Software*. Component-Based Software Engineering of Trustworthy Embedded Systems 80.5 (May 2007), pp. 687–699. ISSN: 0164-1212. DOI: 10.1016/j.jss.2006.08.039. URL: https://www.sciencedirect.com/science/article/pii/S016412120600224X (visited on 10/30/2024) (cit. on p. 115).

[107] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. CGO ’04. USA: IEEE Computer Society, Mar. 2004, p. 75. ISBN: 978-0-7695-2102-2 (cit. on p. 74).

[108] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. “Verus: Verifying Rust Programs Using Linear Ghost Types”. In: *Software Artifact (virtual machine, pre-built distributions) for "Verus: Verifying Rust Programs using Linear Ghost Types"* 7.OOPSLA1 (Apr. 2023), 85:286–85:315. DOI: 10.1145/3586037. URL: https://dl.acm.org/doi/10.1145/3586037 (visited on 10/22/2024) (cit. on p. 106).

[109] Alessandro Legnani. “Trusted Firmware for a Research Computer”. Bachelor’s Thesis. ETH Zurich, Aug. 2023. DOI: 10.3929/ethz-b-000634201. URL: https://doi.org/10.3929/ethz-b-000634201 (cit. on p. 124).

[110] Lenovo. *XClarity Controller*. URL: https://pubs.lenovo.com/lxcc-overview/ (visited on 10/03/2024) (cit. on p. 10).

[111] Xavier Leroy. “Formal Verification of a Realistic Compiler”. In: *Communications of the ACM* 52.7 (July 2009), pp. 107–115. ISSN: 0001-0782. DOI: 10.1145/1538788.1538814. URL: https://dl.acm.org/doi/10.1145/1538788.1538814 (cit. on p. 80).

[112] Alan Leung, Dimitar Bounov, and Sorin Lerner. “C-to-Verilog Translation Validation”. In: *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. Sept. 2015, pp. 42–47. DOI: 10.1109/MEMCOD.2015.7340466. URL: https://ieeexplore.ieee.org/document/7340466 (cit. on pp. 73, 80).

[113] Corey Lewis. *seL4 Multikernel Roadmap and Concurrency Verification*. Oct. 2024. URL: https://www.youtube.com/watch?v=8JLKtpB1KPM (visited on 10/24/2024) (cit. on p. 109).

[114] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. “Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor”. In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 3953–3970. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/li-shih-wei (visited on 10/15/2024) (cit. on p. 106).

[115] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. “A Secure and Formally Verified Linux KVM Hypervisor”. In: *2021 IEEE Symposium on Security and Privacy (SP)*. May 2021, pp. 1782–1799. DOI: 10.1109/SP40001.2021.00049 (cit. on p. 106).

[116] Linux development community. *Device-Tree bindings for I2C GPIO driver*. 2024. URL: https://www.kernel.org/doc/Documentation/devicetree/bindings/i2c/i2c-gpio.txt (visited on 05/12/2024) (cit. on p. 88).

[117] Linux development community. *linux/drivers/i2c/busses/i2c-gpio.c at Linux v5.15*. 2024. URL: https://github.com/torvalds/linux/blob/v5.15/drivers/i2c/busses/i2c-gpio.c (visited on 05/12/2024) (cit. on p. 88).

[118] Linux development community. *linux/drivers/media/i2c/ks0127.c at Linux v5.15*. 2024. URL: https://github.com/torvalds/linux/blob/v5.15/drivers/media/i2c/ks0127.c (visited on 05/08/2024) (cit. on pp. 67, 85, 87).

[119] Linux development community. *linux/drivers/pci/quirks.c at Linux v6.9*. en. May 2024. URL: https://github.com/torvalds/linux/blob/v6.9/drivers/pci/quirks.c (visited on 05/08/2024) (cit. on p. 67).

[120] Linux Foundation. *OpenBMC*. URL: https://www.openbmc.org/ (visited on 10/03/2024) (cit. on p. 10).

[121] Zikai Liu. "Generating Trustworthy I^{2}C Stacks Across Software and Hardware". Master's Thesis. ETH Zurich, Sept. 2023. DOI: 10.3929/ethz-b-000632755. URL: https://doi.org/10.3929/ethz-b-000632755 (cit. on pp. 5, 77).

[122] Zikai Liu. "Towards Trustworthy BMC Software with Virtualization on seL4". Semester Project. ETH Zurich, Feb. 2023 (cit. on p. 6).

[123] LLVM development community. *The LLVM Compiler Infrastructure Project*. 2024. URL: https://llvm.org/ (visited on 05/21/2024) (cit. on p. 74).

[124] *LogiCORE IP AXI IIC bus interface data sheet*. Version v1.02a. Advanced Micro Devices. July 2012. 38 pp. (cit. on p. 87).

[125] Jiang Long and Robert Brayton. *A Simple C to Verilog Compilation Procedure for Hardware/Software Verification*. en. 2016 (cit. on p. 73).

[126] Andreas Lööw. "Lutsig: A Verified Verilog Compiler for Verified Circuit Development". In: *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2021. New York, NY, USA: Association for Computing Machinery, Jan. 2021, pp. 46–60. ISBN: 978-1-4503-8299-1. DOI: 10.1145/3437992.3439916. URL: https://dl.acm.org/doi/10.1145/3437992.3439916 (cit. on p. 80).

[127] Anna Lyons, Kent McLeod, Hesham Almatary, and Gernot Heiser. "Scheduling-Context Capabilities: A Principled, Light-Weight Operating-System Mechanism for Managing Time". In: *Proceedings of the Thirteenth EuroSys Conference*. EuroSys '18. New York, NY, USA: Association for Computing Machinery, Apr. 2018, pp. 1–16. ISBN: 978-1-4503-5584-1. DOI: 10.1145/3190508.3190539. URL: https://doi.org/10.1145/3190508.3190539 (visited on 10/29/2024) (cit. on p. 114).

[128] Federico Mari, Igor Melatti, Ivano Salvo, and Enrico Tronci. "Model-Based Synthesis of Control Software from System-Level Formal Specifications". In: *ACM Trans. Softw. Eng. Methodol.* 23.1 (Feb. 2014), 6:1–6:42. ISSN: 1049-331X. DOI: 10.1145/2559934. URL: https://doi.org/10.1145/2559934 (cit. on p. 49).

[129] Christopher D. Marlin. *Coroutines*. Vol. 95. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1980. ISBN: 978-3-540-10256-4. DOI: 10.1007/3-540-10256-6. URL: http://link.springer.com/10.1007/3-540-10256-6 (cit. on p. 73).

[130] Advamation mechatronic. *Raspberry Pi I2C clock-stretching bug*. Aug. 2013. URL: http://www.advamation.com/knowhow/raspberrypi/rpi-i2c-bug.html (visited on 11/30/2023) (cit. on pp. 67, 87).

[131] Roman Meier. “Declarative Dynamic Power Management”. Master’s Thesis. ETH Zurich, Oct. 2022. DOI: 10.3929/ethz-b-000583405. URL: https://doi.org/10.3929/ethz-b-000583405 (cit. on p. 5).

[132] *Mercury XU5 SoC Module User Manual*. English. Version 08. Enclustra. Feb. 2021. 65 pp. (cit. on p. 9).

[133] *Mercury ZX5 SoC Module User Manual*. English. Version 06. Enclustra. Feb. 2021. 53 pp. (cit. on p. 7).

[134] Fabrice Mérillon, Laurent Réveillère, Charles Consel, Renaud Marlet, and Gilles Muller. “Devil: An IDL for Hardware Programming”. In: *Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*. San Diego, CA: USENIX Association, 2000. URL: https://www.usenix.org/conference/osdi-2000/devil-idl-hardware-programming (cit. on p. 98).

[135] W. Morris. “Latchup in CMOS”. In: *2003 IEEE International Reliability Physics Symposium Proceedings, 2003. 41st Annual.* Mar. 2003, pp. 76–84. DOI: 10.1109/RELPHY.2003.1197724. URL: https://ieeexplore.ieee.org/document/1197724 (cit. on p. 27).

[136] Janani Mukundan, Hillery Hunter, Kyu-hyoun Kim, Jeffrey Stuecheli, and José F. Martínez. “Understanding and Mitigating Refresh Overheads in High-Density DDR4 DRAM Systems”. In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ISCA ’13. New York, NY, USA: Association for Computing Machinery, June 2013, pp. 48–59. ISBN: 978-1-4503-2079-5. DOI: 10.1145/2485922.2485927. URL: https://doi.org/10.1145/2485922.2485927 (cit. on p. 27).

[137] Sanjai Narain, Gary Levin, Sharad Malik, and Vikram Kaul. “Declarative Infrastructure Configuration Synthesis and Debugging”. In: *Journal of Network and Systems Management* 16.3 (Sept. 2008), pp. 235–258. ISSN: 1573-7705. DOI: 10.1007/s10922-008-9108-y. URL: https://doi.org/10.1007/s10922-008-9108-y (cit. on p. 49).

[138] Vikram Narayanan, Marek S. Baranowski, Leonid Ryzhyk, Zvonimir Rakamarić, and Anton Burtsev. “RedLeaf: Towards An Operating System for Safe and Verified Firmware”. In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. HotOS ’19. Bertinoro, Italy: Association for Computing Machinery, May 2019, pp. 37–44. ISBN: 978-1-4503-6727-1. DOI: 10.1145/3317550.3321449. URL: https://doi.org/10.1145/3317550.3321449 (cit. on pp. 2, 107).

[139] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. “RedLeaf: Isolation and Communication in a Safe Operating System”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 21–39. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/narayanan-vikram (cit. on p. 107).

[140] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. "Hyperkernel: Push-Button Verification of an OS Kernel". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 252–269. ISBN: 978-1-4503-5085-3. DOI: 10.1145/3132747.3132748. URL: https://dl.acm.org/doi/10.1145/3132747.3132748 (visited on 10/15/2024) (cit. on p. 106).

[141] Mattias O'Nils and Axel Jantsch. "Device Driver and DMA Controller Synthesis from HW/SW Communication Protocol Specifications". In: *Design Automation for Embedded Systems* 6.2 (Apr. 2001), pp. 177–205. ISSN: 1572-8080. DOI: 10.1023/A:1011246731756. URL: https://doi.org/10.1023/A:1011246731756 (cit. on p. 97).

[142] OpenBMC project community. *OpenBMC*. 2024. URL: https://github.com/openbmc/openbmc (visited on 09/08/2022) (cit. on p. 87).

[143] OpenTitan. *Platform Integrity Module*. Mar. 2023. URL: https://opentitan.org/book/doc/use_cases/platform_integrity_module/index.html (visited on 10/18/2024) (cit. on p. 103).

[144] Ross B. Ortega and Gaetano Borriello. "Communication Synthesis for Distributed Embedded Systems". In: *Proceedings of the 1998 IEEE/ACM International Conference on Computer-aided Design*. ICCAD '98. New York, NY, USA: Association for Computing Machinery, Nov. 1998, pp. 437–444. ISBN: 978-1-58113-008-9. DOI: 10.1145/288548.289067. URL: https://dl.acm.org/doi/10.1145/288548.289067 (cit. on p. 97).

[145] Mathieu Paturel, Isitha Subasinghe, and Gernot Heiser. "First Steps in Verifying the seL4 Core Platform". In: *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems*. APSys '23. New York, NY, USA: Association for Computing Machinery, Aug. 2023, pp. 9–15. ISBN: 9798400703058. DOI: 10.1145/3609510.3609821. URL: https://dl.acm.org/doi/10.1145/3609510.3609821 (visited on 10/14/2024) (cit. on pp. 106, 109).

[146] PCI-SIG. *PCI Express Base Specification*. Version Revision 6.0. Dec. 2021 (cit. on p. 117).

[147] Havoc Pennington, Anders Carlsson, Alexander Larsson, Sven Herzberg, Simon McVittie, and David Zeuthen. *D-Bus*. Version 0.42. Aug. 2023. URL: https://dbus.freedesktop.org/doc/dbus-specification.html (visited on 10/22/2024) (cit. on pp. 2, 11).

[148] Johannes Åman Pohjola, Hira Taqdees Syeda, Miki Tanaka, Krishnan Winter, Tsun Wang Sau, Benjamin Nott, Tiana Tsang Ung, Craig McLaughlin, Remy Seassau, Magnus O. Myreen, Michael Norrish, and Gernot Heiser. "Pancake: Verified Systems Programming Made Sweeter". In: *Proceedings of the 12th Workshop on Programming Languages and Operating Systems*. PLOS '23. New York, NY, USA: Association for Computing Machinery, Oct. 2023, pp. 1–9. ISBN: 9798400704048. DOI: 10.1145/3623759.3624544. URL: https://dl.acm.org/doi/10.1145/3623759.3624544 (cit. on pp. 65, 99).

[149] CVE Project. *CVE Project History*. URL: https://www.cve.org/About/History (visited on 10/29/2024) (cit. on p. 18).

[150] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. "VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-core Frequencies". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. New York, NY, USA: Association for Computing Machinery, Nov. 2019, pp. 195–209. ISBN: 978-1-4503-6747-9. DOI: 10.1145/3319535.3354201. URL: https://doi.org/10.1145/3319535.3354201 (cit. on pp. 24, 25, 29).

[151] Andrew Regenscheid. *Platform Firmware Resiliency Guidelines*. Tech. rep. NIST SP 800-193. Gaithersburg, MD: National Institute of Standards and Technology, May 2018, NIST SP 800–193. DOI: 10.6028/NIST.SP.800-193. URL: http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-193.pdf (cit. on p. 17).

[152] Ariella Robison. *The Missing Security Primer for Bare Metal Cloud Services - Eclypsium*. Feb. 2019. URL: https://eclypsium.com/blog/the-missing-security-primer-for-bare-metal-cloud-services/ (visited on 10/29/2024) (cit. on p. 17).

[153] Nadav Rotem. *C-to-Verilog.Com: High-Level Synthesis Using LLVM*. Nov. 2010. URL: https://llvm.org/devmtg/2010-11/Rotem-CToVerilog.pdf (visited on 05/21/2024) (cit. on p. 73).

[154] Leonid Ryzhyk, Nikolaj Bjørner, Marco Canini, Jean-Baptiste Jeannin, Cole Schlesinger, Douglas B. Terry, and George Varghese. "Correct by Construction Networks Using Stepwise Refinement". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 2017, pp. 683–698. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/ryzhyk (cit. on p. 49).

[155] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. "Dingo: Taming Device Drivers". In: *Proceedings of the 4th ACM European Conference on Computer Systems*. EuroSys '09. New York, NY, USA: Association for Computing Machinery, Apr. 2009, pp. 275–288. ISBN: 978-1-60558-482-9. DOI: 10.1145/1519065.1519095. URL: https://dl.acm.org/doi/10.1145/1519065.1519095 (cit. on p. 48).

[156] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. "Automatic Device Driver Synthesis with Termite". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles - SOSP '09*. Big Sky, Montana, USA: ACM Press, 2009, p. 73. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629583. URL: http://portal.acm.org/citation.cfm?doid=1629575.1629583 (cit. on pp. 48, 98).

[157] Leonid Ryzhyk, Adam Walker, John Keys, Alexander Legg, Arun Raghunath, Michael Stumm, and Mona Vij. "User-Guided Device Driver Synthesis". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 661–676. ISBN: 978-1-931971-16-4. URL: http://dl.acm.org/citation.cfm?id=2685048.2685101 (cit. on pp. 48, 98).

[158] Behzad Salami, Osman Unsal, and Adrian Cristal. "Fault Characterization Through FPGA Undervolting". In: *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. Aug. 2018, pp. 85–853. DOI: 10.1109/FPL.2018.00023. URL: https://ieeexplore.ieee.org/document/8533473 (visited on 10/30/2024) (cit. on p. 119).

[159] Uday Savagaonkar and Nelly Porter. *Titan in Depth: Security in Plaintext*. Aug. 2017. URL: https://cloud.google.com/blog/products/identity-security/titan-in-depth-security-in-plaintext (visited on 10/18/2024) (cit. on p. 103).

[160] SBS Implementers Forum. *System Management Bus (SMBus) Specification*. Aug. 2000. URL: http://smbus.org/specs/index.html (cit. on p. 65).

[161] Jasmin Schult. "A model-based approach to platform-level power and clock management". Bachelor's Thesis. ETH Zurich, Aug. 2020. DOI: 10.3929/ethz-b-000490632. URL: https://doi.org/10.3929/ethz-b-000490632 (cit. on p. 5).

[162] Jasmin Schult, Ben Fiedler, David Cock, and Timothy Roscoe. "Semi-Open-State Testing for in-Silicon Coherent Interconnects". In: *Proceedings of the 24th Conference on Formal Methods in Computer-Aided Design – FMCAD 2024*. TU Wien Academic Press, Oct. 2024, pp. 153–162. ISBN: 978-3-85448-065-5. DOI: 10.34727/2024/isbn.978-3-85448-065-5_21. URL: https://repositum.tuwien.at/handle/20.500.12708/200788 (visited on 10/24/2024) (cit. on p. 110).

[163] Jasmin Schult, Daniel Schwyn, Michael Giardino, David Cock, Reto Achermann, and Timothy Roscoe. "Declarative Power Sequencing". In: *ACM Transactions on Embedded Computing Systems* 20.5s (Sept. 2021). ISSN: 1539-9087. DOI: 10.1145/3477039. URL: https://doi.org/10.1145/3477039 (cit. on p. 5).

[164] Adrian Schüpbach, Andrew Baumann, Timothy Roscoe, and Simon Peter. "A Declarative Language Approach to Device Configuration". In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. New York, NY, USA: Association for Computing Machinery, Mar. 2011, pp. 119–132. ISBN: 978-1-4503-0266-1. DOI: 10.1145/1950365.1950382. URL: https://dl.acm.org/doi/10.1145/1950365.1950382 (cit. on pp. 49, 67).

[165] Daniel Schwyn, Zikai Liu, and Timothy Roscoe. "Efeu: generating efficient, verified, hybrid hardware/software drivers for I^2C devices". In: *Proceedings of the Twentieth European Conference on Computer Systems*. EuroSys '25. Rotterdam, Netherlands: Association for Computing Machinery, Mar. 2025, pp. 76–93. ISBN: 9798400711961. DOI: 10.1145/3689031.3696093. URL: https://doi.org/10.1145/3689031.3696093 (cit. on p. 5).

[166] seL4 Foundation. *Frequently Asked Questions on seL4*. Oct. 2024. URL: https://docs.sel4.systems/projects/sel4/frequently-asked-questions.html (visited on 10/24/2024) (cit. on p. 109).

[167] seL4 Foundation. *seL4 Foundation Membership*. 2024. URL: https://sel4.systems/Foundation/Membership/ (visited on 10/24/2024) (cit. on p. 109).

[168] seL4 Foundation. *seL4 Project Roadmap*. Oct. 2024. URL: https://docs.sel4.systems/projects/roadmap.html (visited on 10/24/2024) (cit. on p. 109).

[169] seL4 Foundation. *Verified Configurations*. Oct. 2024. URL: https://docs.sel4.systems/projects/sel4/verified-configurations.html (visited on 10/24/2024) (cit. on p. 109).

[170] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. "Translation Validation for a Verified OS Kernel". In: *ACM SIGPLAN Notices* 48.6 (June 2013), pp. 471–482. ISSN: 0362-1340. DOI: 10.1145/2499370.2462183 (cit. on pp. 80, 106).

[171] Alireza Shameli-Sendi. "Understanding Linux Kernel Vulnerabilities". In: *Journal of Computer Virology and Hacking Techniques* 17.4 (Dec. 2021), pp. 265–278. ISSN: 2263-8733. DOI: 10.1007/s11416-021-00379-x. URL: https://doi.org/10.1007/s11416-021-00379-x (visited on 11/01/2024) (cit. on pp. 18, 102).

[172] Mirela Simonović, Vojin Živojnović, and Lazar Saranovac. "Formal Model for System-Level Power Management Design". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. Mar. 2017, pp. 1599–1602. DOI: 10.23919/DATE.2017.7927245. URL: https://ieeexplore.ieee.org/document/7927245 (cit. on p. 50).

[173] Stewart Smith. *CVE-2019-6260: Gaining Control of BMC from the Host Processor*. Jan. 2019. URL: https://www.flamingspork.com/blog/2019/01/23/cve-2019-6260-gaining-control-of-bmc-from-the-host-processor/ (visited on 08/22/2024) (cit. on p. 17).

[174] Amnapardaz Soft. *Take the Lights-out - Implant.ARM.iLOBleed.a*. Dec. 2021. URL: https://threats.amnpardaz.com/en/wp-content/uploads/sites/5/2021/12/Implant.ARM_.iLOBleed.a-en.pdf (visited on 08/22/2024) (cit. on p. 17).

[175] Antmicro Open Source. *ARTIX DC-SCM*. 2020. URL: https://opensource.antmicro.com/projects/artix-dc-scm/ (visited on 05/16/2024) (cit. on pp. 9, 67).

[176] SPIN development community. *Promela Manual page*. 2012. URL: http://spinroot.com/spin/Man/promela.html (visited on 08/02/2023) (cit. on pp. 64, 73, 80, 85).

[177] Roberto Starc, Tom Kuchler, Michael Giardino, and Ana Klimovic. "Serverless? RISC More!" In: *Proceedings of the 2nd Workshop on SErverless Systems, Applications and MEthodologies*. SESAME '24. New York, NY, USA: Association for Computing Machinery, Apr. 2024, pp. 15–24. ISBN: 9798400705458. DOI: 10.1145/3642977.3652095. URL: https://dl.acm.org/doi/10.1145/3642977.3652095 (visited on 10/24/2024) (cit. on p. 110).

[178] Michael Sugden. *The Missing Middle: Addressing the Absence of Firmware Security*. Jan. 2024. URL: https://www.fdd.org/analysis/2024/01/25/the-missing-middle/ (cit. on p. 17).

[179] Jun Sun, Wanghong Yuan, Mahesh Kallahalla, and Nayeem Islam. “HAIL: A Language for Easy and Correct Device Access”. In: *Proceedings of the 5th ACM International Conference on Embedded Software*. EMSOFT ’05. New York, NY, USA: Association for Computing Machinery, Sept. 2005, pp. 1–9. ISBN: 978-1-59593-091-0. DOI: 10.1145/1086228.1086230. URL: https://dl.acm.org/doi/10.1145/1086228.1086230 (cit. on p. 98).

[180] Supermicro. *Supermicro Intelligent Management*. 2024. URL: https://www.supermicro.com/en/solutions/management-software/bmc-resources (visited on 11/02/2024) (cit. on p. 10).

[181] Lalith Suresh, João Loff, Nina Narodytska, Leonid Ryzhyk, Mooly Sagiv, and Brian Oki. “Synthesizing Cluster Management Code for Distributed Systems”. In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. HotOS ’19. New York, NY, USA: Association for Computing Machinery, May 2019, pp. 45–50. ISBN: 978-1-4503-6727-1. DOI: 10.1145/3317550.3321444. URL: https://doi.org/10.1145/3317550.3321444 (cit. on p. 49).

[182] System Management Interface Forum, Inc. *PMBus Power System Management Protocol Specification Part I – General Requirements, Transport And Electrical Interface*. Mar. 2015. URL: https://pmbusprod.wpenginepowered.com/wp-content/uploads/2022/01/PMBus-Specification-Rev-1-3-1-Part-I-20150313.pdf (cit. on pp. 30, 65).

[183] System Management Interface Forum, Inc. *PMBus Power System Management Protocol Specification Part II – Command Language*. Mar. 2015. URL: https://pmbusprod.wpenginepowered.com/wp-content/uploads/2022/01/PMBus-Specification-Rev-1-3-1-Part-II-20150313.pdf (cit. on pp. 30, 65).

[184] systemd. *systemd*. 2024. URL: https://systemd.io (visited on 10/22/2024) (cit. on p. 11).

[185] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. “CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management”. In: *26th USENIX Security Symposium (USENIX Security 17)*. 2017, pp. 1057–1074. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang (cit. on pp. 24, 25, 29).

[186] Enzian Team. *Enzian*. July 2021. URL: http://enzian.systems (cit. on pp. 3, 7).

[187] TechInsights. *ODM Sales Soar as Hyperscalers and Cloud Providers Go Direct*. URL: https://www.techinsights.com/blog/odm-sales-soar-hyperscalers-and-cloud-providers-go-direct-1 (visited on 10/03/2024) (cit. on p. 10).

[188] Positive Technologies. *Dell EMC Fixes iDRAC Vulnerability Found by Positive Technologies*. July 2020. URL: https://www.ptsecurity.com/ww-en/about/news/dell-emc-fixes-idrac-vulnerability-found-by-positive-technologies/ (visited on 08/22/2024) (cit. on p. 17).

[189] Konstantinos Tovletoglou, Lev Mukhanov, Georgios Karakonstantis, Athanasios Chatzidimitriou, George Papadimitriou, Manolis Kaliorakis, Dimitris Gizopoulos, Zacharias Hadjilambrou, Yiannakis Sazeides, Alejandro Lampropulos, Shidhartha Das, and Phong Vo. "Measuring and Exploiting Guardbands of Server-Grade ARMv8 CPU Cores and DRAMs". In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. June 2018, pp. 6–9. DOI: 10.1109/DSN-W.2018.00013. URL: https://ieeexplore.ieee.org/document/8416198 (visited on 10/30/2024) (cit. on p. 119).

[190] Sarah Tröndle. "Real-time Board Management using an FPGA". Bachelor's Thesis. ETH Zurich, Apr. 2021. DOI: 10.3929/ethz-b-000533010. URL: https://doi.org/10.3929/ethz-b-000533010 (cit. on pp. 6, 112).

[191] u-bmc. *u-bmc*. 2015. URL: https://u-bmc.org/ (visited on 07/22/2024) (cit. on p. 10).

[192] UEFI Forum Inc. *Advanced Configuration and Power Interface (ACPI) Specification, Version 6.3*. Jan. 2020. URL: https://uefi.org/sites/default/files/resources/ACPI_6_3_final_Jan30.pdf (cit. on pp. 50, 117).

[193] UEFI Forum Inc. *Unified Extensible Firmware Interface Specification*. Version 2.6. Jan. 2016 (cit. on p. 1).

[194] Steven J. Vaughan-Nichols. *MINIX: Intel's hidden in-chip operating system*. Nov. 2017. URL: https://www.zdnet.com/article/minix-intels-hidden-in-chip-operating-system/ (visited on 10/10/2020) (cit. on p. 2).

[195] D. Verkest, K. Van Rompaey, I. Bolsens, and H. De Man. "CoWare—A Design Environment for Heterogeneous Hardware/Software Systems". In: *Design Automation for Embedded Systems* 1.4 (Oct. 1996), pp. 357–386. ISSN: 1572-8080. DOI: 10.1007/BF00209910. URL: https://doi.org/10.1007/BF00209910 (cit. on p. 97).

[196] Yakir Vizel, Georg Weissenbacher, and Sharad Malik. "Boolean Satisfiability Solvers and Their Applications in Model Checking". In: *Proceedings of the IEEE* 103.11 (Nov. 2015), pp. 2021–2035. ISSN: 1558-2256. DOI: 10.1109/JPROC.2015.2455034. URL: https://ieeexplore.ieee.org/document/7225110 (cit. on p. 83).

[197] Bingyao Wang, Sepehr Noorafshan, Reto Achermann, and Margo Seltzer. "Synthesizing Device Drivers with Ghost Writer". In: *Proceedings of the 12th Workshop on Programming Languages and Operating Systems*. PLOS '23. New York, NY, USA: Association for Computing Machinery, Oct. 2023, pp. 10–17. ISBN: 9798400704048. DOI: 10.1145/3623759.3624545. URL: https://dl.acm.org/doi/10.1145/3623759.3624545 (cit. on p. 98).

[198] Shaojie Wang and Sharad Malik. "Synthesizing Operating System Based Device Drivers in Embedded Systems". In: *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS '03. New York, NY, USA: Association for Computing Machinery, Oct. 2003, pp. 37–44. ISBN: 978-1-58113-742-2. DOI: 10.1145/944645.944655. URL: https://doi.org/10.1145/944645.944655 (cit. on p. 48).

[199] Nate Warfield, Scott Scheferman, and Vlad Babkin. *BMC&C: Lights Out Forever*. July 2023. URL: https://eclypsium.com/research/bmcc-lights-out-forever/ (visited on 08/22/2024) (cit. on p. 17).

[200] Georg Wehrli. "Generating Platform Configuration from Netlists". Bachelor's Thesis. ETH Zurich, May 2024. DOI: 10.3929/ethz-b-000684943. URL: https://doi.org/10.3929/ethz-b-000684943 (cit. on pp. 6, 118).

[201] Robert V. White. "PMBus: A Decade of Growth: An Open-Standards Success". In: *IEEE Power Electronics Magazine* 1.3 (Sept. 2014), pp. 33–39. ISSN: 2329-9215. DOI: 10.1109/MPEL.2014.2330492. URL: https://ieeexplore.ieee.org/abstract/document/6891449 (cit. on p. 65).

[202] Rob Wilbert. *Enabling Open Embedded Systems Management on PowerEdge Servers*. Nov. 2022. URL: https://www.dell.com/en-us/blog/enabling-open-embedded-systems-management-on-poweredge-servers/ (cit. on p. 10).

[203] Chao Xu, Felix Xiaozhu Lin, Yuyang Wang, and Lin Zhong. "Automated OS-level Device Runtime Power Management". In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '15. New York, NY, USA: Association for Computing Machinery, Mar. 2015, pp. 239–252. ISBN: 978-1-4503-2835-7. DOI: 10.1145/2694344.2694360. URL: https://doi.org/10.1145/2694344.2694360 (cit. on p. 49).

[204] Chao Xu, Felix Xiaozhu Lin, and Lin Zhong. "Device Drivers Should Not Do Power Management". In: *Proceedings of 5th Asia-Pacific Workshop on Systems*. APSys '14. New York, NY, USA: Association for Computing Machinery, June 2014, pp. 1–7. ISBN: 978-1-4503-3024-4. DOI: 10.1145/2637166.2637233. URL: https://doi.org/10.1145/2637166.2637233 (cit. on p. 49).

[205] Pengcheng Xu. "Enzian Firmware Resource Interface". Semester Project. ETH Zurich, Feb. 2023. DOI: 10.3929/ethz-b-000603460. URL: https://doi.org/10.3929/ethz-b-000603460 (cit. on pp. 6, 117).

[206] Jean Yang and Chris Hawblitzel. "Safe to the Last Instruction: Automated Verification of a Type-Safe Operating System". In: *SIGPLAN Not.* 45.6 (June 2010), pp. 99–110. ISSN: 0362-1340. DOI: 10.1145/1809028.1806610. URL: https://dl.acm.org/doi/10.1145/1809028.1806610 (visited on 10/15/2024) (cit. on p. 106).

[207] Jeong-Han Yun, Gunwoo Kim, Choonho Son, and Taisook Han. "Automatic Generation of Hardware/Software Interface with Product-Specific Debugging Tools". In: *Embedded and Ubiquitous Computing*. Ed. by Edwin Sha, Sung-Kook Han, Cheng-Zhong Xu, Moon-Hae Kim, Laurence T. Yang, and Bin Xiao. Berlin, Heidelberg: Springer, 2006, pp. 742–753. ISBN: 978-3-540-36681-2. DOI: 10.1007/11802167_75 (cit. on p. 97).

[208] Jim Zemlin. *OpenBMC Project Community Comes Together at The Linux Foundation to Define Open Source Implementation of BMC Firmware Stack*. Mar. 2018. URL: https://www.linuxfoundation.org/blog/blog/openbmc-project-community-comes-together-at-the-linux-foundation-to-define-open-source-implementation-of-bmc-firmware-stack (visited on 10/08/2024) (cit. on p. 10).

[209] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. "SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques". In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. OSDI '06. USA: USENIX Association, Nov. 2006, pp. 45–60. ISBN: 978-1-931971-47-8 (cit. on p. 98).

[210] *Zynq UltraScale+ Device Technical Reference Manual*. English. Version v2.2. Xilinx. Dec. 2020. 1214 pp. (cit. on p. 87).