OSS Architecture for Mixed-Criticality Systems

A Dual View from a Software and System Engineering Perspective

---

Ralf Stefan Ramsauer, M. Sc.

OSS Architecture for Mixed-Criticality Systems

A Dual View from a Software and System Engineering Perspective

---

Von der Fakultät für Elektrotechnik und Informatik

der Gottfried Wilhelm Leibniz Universität Hannover

zur Erlangung des akademischen Grades

DOKTOR-INGENIEUR

(abgekürzt: Dr.-Ing.)

genehmigte Dissertation

von Herrn

Ralf Stefan Ramsauer, M. Sc.

2021

Referent:	Prof. Dr.-Ing. habil. Daniel Lohmann
Korreferent:	Prof. Dr. rer. nat. Wolfgang Mauerer
Tag der Promotion:	21. Dezember 2021

Ralf Stefan Ramsauer, M. Sc.
OSS Architecture for Mixed-Criticality Systems
Dissertation, 8. Oktober 2021

Abstract

Computer-based automation in industrial appliances led to a growing number of logically dependent, but physically separated embedded control units per appliance. Many of those components are safety-critical systems, and require adherence to safety standards, which is inconsonant with the relentless demand for features in those appliances. Features lead to a growing amount of control units per appliance, and to a increasing complexity of the overall software stack, being unfavourable for safety certifications. Modern CPUs provide means to revise traditional *separation of concerns* design primitives: the consolidation of systems, which yields new engineering challenges that concern the entire software and system stack.

Multi-core CPUs favour economic consolidation of formerly separated systems with one efficient single hardware unit. Nonetheless, the system architecture must provide means to guarantee the freedom from interference between domains of different criticality. System consolidation demands for architectural and engineering strategies to fulfil requirements (*e.g.*, real-time or certifiability criteria) in safety-critical environments.

In parallel, there is an ongoing trend to substitute ordinary proprietary base platform software components by mature OSS variants for economic and engineering reasons. There are fundamental differences of processual properties in development processes of OSS and proprietary software. OSS in safety-critical systems requires development process assessment techniques to build an evidence-based fundament for certification efforts that is based upon empirical software engineering methods.

In this thesis, I will approach from both sides: the software and system engineering perspective. In the first part of this thesis, I focus on the assessment of OSS components: I develop software engineering techniques that allow to quantify characteristics of distributed OSS development processes. I show that ex-post analyses of software development processes can be used to serve as a foundation for certification efforts, as it is required for safety-critical systems.

In the second part of this thesis, I present a system architecture based on OSS components that allows for consolidation of mixed-criticality systems on a single platform. Therefore, I exploit virtualisation extensions of modern CPUs to strictly isolate domains of different criticality. The proposed architecture shall eradicate any remaining hypervisor activity in order to preserve real-time capabilities of the hardware *by design*, while guaranteeing strict isolation across domains.

Keywords—real-time operating system, mixed-criticality, static hardware partitioning, development processes, development process reconstruction, quantitative software engineering

Kurzfassung

Computergestützte Automatisierung industrieller Systeme führt zu einer wachsenden Anzahl an logisch abhängigen, aber physisch voneinander getrennten Steuergeräten pro System. Viele der Einzelgeräte sind sicherheitskritische Systeme, welche die Einhaltung von Sicherheitsstandards erfordern, was durch die unermüdliche Nachfrage an Funktionalitäten erschwert wird. Diese führt zu einer wachsenden Gesamtzahl an Steuergeräten, einhergehend mit wachsender Komplexität des gesamten Softwarekorpus, wodurch Zertifizierungsvorhaben erschwert werden. Moderne Prozessoren stellen Mittel zur Verfügung, welche es ermöglichen, das traditionelle *Trennung von Belangen* Designprinzip zu erneuern: die Systemkonsolidierung. Sie stellt neue ingenieurstechnische Herausforderungen, die den gesamten Software und Systemstapel betreffen.

Mehrkernprozessoren begünstigen die ökonomische und effiziente Konsolidierung vormals getrennter Systemen zu einer effizienten Hardwareeinheit. Geeignete Systemarchitekturen müssen jedoch die Rückwirkungsfreiheit zwischen Domänen unterschiedlicher Kritikalität sicherstellen. Die Konsolidierung erfordert architektonische, als auch ingenieurstechnische Strategien um die Anforderungen (etwa Echtzeit- oder Zertifizierbarkeitskriterien) in sicherheitskritischen Umgebungen erfüllen zu können.

Zunehmend werden herkömmliche proprietär entwickelte Basisplattformkomponenten aus ökonomischen und technischen Gründen vermehrt durch ausgereifte open source software (OSS) Alternativen ersetzt. Jedoch hindern fundamentale Unterschiede bei prozessualen Eigenschaften des Entwicklungsprozesses bei OSS den Einsatz in sicherheitskritischen Systemen. Dieser erfordert Techniken, welche es erlauben die Entwicklungsprozesse zu bewerten um ein evidenzbasiertes Fundament für Zertifizierungsvorhaben basierend auf empirischen Methoden des Software Engineerings zur Verfügung zu stellen.

In dieser Arbeit nähere ich mich von beiden Seiten: der Softwaretechnik, und der Systemarchitektur. Im ersten Teil befasse ich mich mit der Beurteilung von OSS Komponenten: Ich entwickle Softwareanalysetechniken, welche es ermöglichen, prozessuale Charakteristika von verteilten OSS Entwicklungsvorhaben zu quantifizieren. Ich zeige, dass rückschauende Analysen des Entwicklungsprozess als Grundlage für Softwarezertifizierungsvorhaben genutzt werden können.

Im zweiten Teil dieser Arbeit widme ich mich der Systemarchitektur. Ich stelle eine OSS-basierte Systemarchitektur vor, welche die Konsolidierung von Systemen gemischter Kritikalität auf einer alleinstehenden Plattform ermöglicht. Dazu nutze ich Virtualisierungserweiterungen moderner Prozessoren aus, um die Hardware in strikt voneinander isolierten Rechendomänen unterschiedlicher Kritikalität unterteilen zu können. Die vorgeschlagene Architektur soll jegliche Betriebsstörungen des Hypervisors beseitigen, um die Echtzeitfähigkeiten der Hardware bauartbedingt aufrecht zu erhalten, während strikte Isolierung zwischen Domänen stets sicher gestellt ist.

Schlüsselwörter—Echtzeitbetriebssysteme, mixed-criticality, statische Hardwarepartitionierung, Entwicklungsprozesse, Rekonstruktion von Entwicklungsprozessen, quantitatives Software Engineering

Danksagungen

Ich möchte meine Danksagung mit Personen außerhalb meines Arbeitsumfeldes beginnen. Ohne Euch wäre diese Arbeit schlicht nicht möglich gewesen wäre.

Allen voran danke ich meinen Eltern *Hildegard* und *Manfred* für ihre unermütliche Unterstützung. Euch beiden gilt mein größter Dank und Respekt, und Euch möchte ich diese Arbeit widmen.

Ein besonderer Dank gilt Dr. rer. nat. *Annette* Schnettelker. Vielen Dank für Deine ständige Unterstützung und Motivation!

Weiterhin danke ich all meinen Freunden aus der *Binary Kitchen*, unserem Hackspace. Es sind unzählige Abende mit ertragreichen fachlichen Diskussionen, welche mir stets neue Inspiration gebracht haben. Unmöglich kann ich Euch alle aufzählen. In diesem Sinne—Danke, dass ihr alle da seid!

Diese kooperative Arbeit entstand unter der wissenschaftlichen Betreuung von Prof. Dr.-ing. habil. *Daniel* Lohmann und Prof. Dr. rer. nat. *Wolfgang* Mauerer. Lieber Daniel, lieber Wolfgang, ich danke Euch beiden für Eure exzellente Betreuung und der ausgezeichneten wissenschaftlichen Führung! Es war und ist mir eine Freude mit Euch zusammen zu arbeiten. Ich durfte Teil Eurer beiden Arbeitsgruppen sein. Daher danke ich allen Mitgliedern beider Gruppen für die vielen ergebnisreichen Diskussionen, einer exzellenten Arbeitsatmosphäre, ausgezeichneter Betreuung und Rückhalt.

Teile dieser Arbeit entstanden im Rahmen einer Industriekooperation mit Siemens Corporate Technology, München. Mein besonderer Dank gilt *Jan* Kiszka. Jan, Du hast mir mit viel Geduld *Systementwicklung mit System* gelehrt. Nirgendwo sonst konnte ich in einer solchen Tiefe die Kunst der anwendungsnahen Systementwicklung lernen!

Regensburg, Oktober 2021

Contents

1. Introduction 1
   - 1.1. Consolidation of Systems . . . 5
   - 1.2. Safety-Critical Systems and Open Source . . . 8
     - 1.2.1. Requirements . . . 8
     - 1.2.2. Related Approaches . . . 12
   - 1.3. Research Context of this Thesis . . . 14
     - 1.3.1. Safety-Critical Systems, OSS and Certification . . . 14
     - 1.3.2. Mixed-Criticality Systems, OSS and System Architecture . . . 17
   - 1.4. Structure . . . 20

I. Reconstruction and Analysis of Software Development Processes 23

2. Reconstruction 25
   - 2.1. Overview . . . 26
   - 2.2. Fundamentals . . . 29
   - 2.3. Clustering Similar Patches . . . 34
     - 2.3.1. Rating Similarity of Two Patches . . . 35
     - 2.3.2. Parameters . . . 38
     - 2.3.3. Reduction of problem space and clustering patches . . . 40
     - 2.3.4. Working with Mailing List Data . . . 41
   - 2.4. Evaluation . . . 42
     - 2.4.1. External Evaluation . . . 43
     - 2.4.2. Example: Duration of patch integration . . . 46
     - 2.4.3. Comparison to Other Approaches . . . 47
   - 2.5. Discussion . . . 48
     - 2.5.1. The Algorithm . . . 48
     - 2.5.2. Plus-Minus-based approach . . . 49
     - 2.5.3. Performance . . . 49
   - 2.6. Threats to Validity . . . 50
     - 2.6.1. Internal Validity . . . 50
     - 2.6.2. External Validity . . . 50
     - 2.6.3. Construct Validity . . . 51

2.7. Related Work 51
2.7.1. Reconstruction of Development Processes 51
2.7.2. Distinction from Code Clone Detection 52
2.8. Summary and Conclusion 53

3. Analysis 55
3.1. Structure 55
3.2. Linux Kernel Development Process 57
3.2.1. Core Characteristics 57
3.2.2. Organigram and Areas of Responsibility 59
3.2.3. Lifecycle Management 60
3.2.4. Exceptional Vulnerability Handling 62
3.2.5. Formalisation 62
3.3. Extraction of Development Characteristics 64
3.3.1. Ignored Patches 65
3.3.2. Conform Integration of Patches 73
3.4. Violation of Development Processes 77
3.4.1. Secret Integration Channels 78
3.4.2. Analysis 79
3.4.3. Related Work 85
3.4.4. Acknowledgements 86
3.5. Discussion 87
3.5.1. Validity 87
3.5.2. Consequences 90
3.6. Summary 92

End of Part I

II. System Consolidation of Safety- and Mixed-Critical Systems 97

4. Ideal Hardware Partitioning 99
4.1. Requirements on Ideal Hardware Partitioning 103
4.1.1. Efficiency of VMMs 104
4.1.2. Architectural System Limitations 105
4.1.3. Device Specific Requirements 106
4.1.4. Platform Specific Requirements 108
4.2. The Jailhouse Hypervisor: Philosophy and Architecture 111
4.2.1. Overview 111
4.2.2. Hardware and Software Support 117

4.3. Cross-domain Protection Against Speculative Execution Exploits 117
4.3.1. Attacks and Mitigations 119
4.3.2. Jailhouse and Speculative Execution Attacks 121

5. Evaluation and Discussion 125
5.1. Hypervisor Activity 125
5.1.1. Common Hypervisor Activity 125
5.1.2. Hypervisor Activity on x86 Platforms 126
5.1.3. Hypervisor Activity on ARM Platforms 128
5.2. Evaluation 129
5.2.1. Hypervisor Overhead 130
5.2.2. ARM: The Cost of Interrupt Reinjection 132
5.2.3. x86: The Cost of the Moderation of accesses to MSR 133
5.2.4. The Cost of Spectre Mitigations 137
5.3. Discussion 145
5.3.1. The Jailhouse Approach 145
5.3.2. Hardware Limitations 146
5.3.3. Speculative Execution and Static Hardware Partitioning 147
5.4. Smoke Test 148
5.5. Summary 150

End of Part II

6. Summary, Conclusions and Further Ideas 155
6.1. Summary of the Thesis and Conclusion 155
6.1.1. Software Engineering 155
6.1.2. System Engineering 156
6.2. Further Ideas 157

A. Appendix 159
A.1. Quantification of Mainlining Efforts 159
A.1.1. Approach 160
A.1.2. Discussion 162
A.1.3. Conclusion 164

Lists **165**
Acronyms 165
References 171
List of Figures 187
List of Tables 188

List of Listings 188
List of Algorithms 188
Lebenslauf 189

1 Introduction

> *Our life, work, and society have become highly dependent on software—in fact, we live today in a software world!*
>
> **— Frank J. Furrer**
> Future-Proof Software Systems [Fur19]

The broad availability of microprocessors started the digital revolution in the beginning of the 1970s that, since then, fundamentally changed almost all areas of life. Their flexibility allows to dynamically customise and adapt the same enabling technological building block to a widespread range of applications. Machine-readable transformations of algorithms—»*Software*«—is the basis of their flexibility: Software allows for using general-purpose components for special-purpose appliances.

Software-based systems became omnipresent. Modern electronics that range from highly-integrated consumer electronics, such as smartphones, to highly-specialised industrial control systems, such as industrial assembly lines, can hardly be imagined without being controlled by software [Vya13]. Yet, independent of the specific case of application, the main objective remains the same: process improvement by optimisation and automation. In industry, automation is the main driver of efficient manufacturing processes. Microchips assist to automise formerly manual or semiautomatic fabrication processes, in order to improve the efficiency of process sequences.

Besides manufacturing industry, microchips are increasingly used in means of transportation, such as in the automotive or avionic industry, and for a wide range of medical devices, for example dosing pumps or pacemakers. Those areas of application are examples where, in contrast to consumer electronics, software failures can have direct and severe consequences on the integrity of human life [Kni02].

In such *safety-critical environments*, human life depend on an accurate, deterministic, precise and correct operation of the system in any situation [Kni02]. This prerequisite had already existed before the mass usage of microchips and software. However, the use of microchips and software in such systems make both part of the safety-critical aspects of the appliance.

Consider the evolution of brake systems in cars as an example. Automotive industry is one industrial sector that was and still is massively influenced by microprocessor-based technologies. Automotive brake systems are one partial aspect that illustrate the technological evolution of a system that was strongly influenced by the technological advancement of microchips and software.

Since their invention, the goal of any brake system is to cause negative acceleration to moving objects. One traditional braking mechanism is based on hydraulic components: Hydraulic pressure is generated by applying force to a brake pedal. The pressure is forwarded to cylinders that shift pistons towards brake pads that, in turn, generate friction with brake disks. Brake disks are connected to the rotating wheels; eventually, the brake system converts kinetic energy to thermal energy in order to decelerate the car.

Microchips and software offer the possibility to refine the goal of braking the car: they enable to brake the car in an efficient and optimal way. Therefore, physical models are implemented as software algorithms that receive fine-granular sensor values as input. A software-based *real-time control system*, which is the failure-intolerant deterministic mind of the hardware, determines control parameters for actuators of the system (*e.g.*, the brakes) within a deadline that must never be exceeded: if the brake force that must be applied to a wheel is determined too late, the car may loose friction.

Such software-based control loops enrich the entire appliance by additional safety features, like the anti-lock braking system (ABS) that detects and eliminates wheels from locking up, or the electronic stability control (ESC) that stabilises the traction of cars by targeted brake control of specific wheels. Functional feature units are typically controlled by their own dedicated software-controlled electronic control unit (ECU): a hardware module that contains communication interfaces to sensors and actuators, controlled by a software-driven microchip.

Yet, brake systems are only one functional segment of a car where computer-controlled systems enable the enhancement and implementation of safety-critical assistance systems. The brake system is only one representative member of an amalgamation of *embedded systems* of different criticality that, in total, form a superior appliance. The criticality of elements or functions of a system is an ordinal number that is determined by a hazard and risk analysis (for automotive, Ref. [ISO26262] Part 9: Automotive Safety Integrity Level (ASIL)-oriented and safety-oriented analyses). Depending on the particular assessment metrics, the result of hazard and risk analysis can, for example, classify the engine control module (ECM) with the same criticality as the airbag control unit (ACU). Still, both have a higher criticality than the rear view camera system, which in turn has higher criticality than entertainment systems. Severity, exposure and controllability

are the base criteria for the classification: A failure of the entertainment system (*e.g.*, miss of an alert due to failure of loudspeakers) is bearable, while a failure of the ECM (*e.g.*, unwanted acceleration) or the ACU (*e.g.*, inadvertent deploy) can lead to the loss of human life.

However, subsystems depend on each other, despite differences in criticality. Close interaction and communication with each other is a precondition to fulfil functional requirements: For example, the entertainment system with low criticality shall adjust the loudspeaker volume level with respect to the vehicle's speed, but it must immediately silence the radio in case of acoustic warning signals that are indicated by one of the safety-critical ECUs. At the same time, the acoustic report of a critical event of the collission avoidance system (CAS) has a higher priority than the acoustic report of, for instance, a coincident detection of a broken brake light. Flow and prioritisation of information between subsystems requires careful definition and consideration.

Those non-trivial engineering challenges are complicated by modern cars that contain up to 100 different ECUs [HH08] that span 100 million lines of code (LOC) [Wen+15]. Nonetheless, the guarantee of safe operation of the appliance requires a failure-tolerant, deterministic, reliable and robust functional interaction of all components in any situation. Such aforementioned »non-functional requirements (NFRs)« [Rom85].[1] They specify project global requirements on *how* software implements a specific requirement rather that *what* exactly is implemented [TD90]. NFRs can have a direct influence on the system's hardware- and software architecture [Chu+12].

Other NFRs demand for the certification or qualification of the systems: safety-critical components must conform with international standards. In the automotive industry, for example, systems *shall* conform with the de-facto standard [ISO26262] »Road vehicles - Functional safety«. Parts 4, 5, and 6 of the international standard demand for well-defined development processes for the development of the system for both, the hardware, and the software level. A safety assessment requires arguing that the underlying development processes are effective and fulfil particular safety requirements, and that the development is traceable: it must adhere to those associated processes.

Certified development processes need to implement high standards regarding traceability and auditability of all development decisions, including hardware and software architecture. Without the loss of generality, this is not limited to the automotive industry: in alignment with the peculiarities of the specific application environment, the industrial norm [IEC61508] is, among others, a superior norm to application-specific norms like [IEC60601] and [IEC62304] for medical electrical equipment, [ARINC653] and [DO-178B] for avionics, or [IEC60880] for nuclear power stations. The interpretation of

[1] NFRs are also referred to as quality attributes [BBL76] or soft goals [Chu+12].

directives of those norms on development processes can reflect on decisions on the system's architecture. However, they provide a certain *scope of action*.

Hence, system engineering approaches, hardware design decisions, and software architecture, as well as engineering principles became inseparable disciplines as hardware and software components influence each other:

> Hardware and software must be designed together to make sure that the implementation not only functions properly but also meets performance, cost, and reliability goals. [Wol94]

Hardware needs to be chosen to fulfil the requirements of the system's operational scenario (*e.g.*, environmental or electrotechnical aspects). Simultaneously, software and hardware put mutual demands. Hardware needs to be able to serve certain system properties, for example providing a physical Controller Area Network (CAN) bus, or to provide enough computational performance for maintaining a certain cycle time or real-time properties. At the same time, the system's software architecture needs to be designed with due regard to those exact properties of the underlying hardware.

Decisions on system architecture are closely tied to the capabilities or *features* of the underlying processor architecture. A capability can for example be the existence of hardware memory protection mechanisms, as they are implemented by memory management units (MMUs) or memory protection units (MPUs). Such capabilities can be used to accomplish non-functional requirements (*e.g.*, robustness), by exploiting guarantees that are given by (certified) low-level hardware mechanisms in software.

In many *traditionally-engineered appliances*, a functional task is assigned to a subsystem of the appliance that corresponds with its self-contained, dedicated embedded system (*i.e.*, a ECU). These decentralised *units construction principle* allows for high modularity (*i.e.*, high variety of product lines) of an appliance.

Traditionally-engineered appliances—underlines the matureness of a tried and tested engineering strategy, but also emphasises the existence of attempts to renovate approaches [Bro06; BD13]. In this thesis, I investigate challenges of two, at first glance independent, ongoing tendencies of paradigm shifts in engineering approaches: the increasing request to use OSS components in safety-critical systems, and the consolidation of systems of mixed-criticality to one centralised appliance, and the close relation of both topics.

1.1 Consolidation of Systems

Manufacturers of safety-critical and uncritical products still tend to split components with different levels of criticality to separate hardware units. In such traditional mixed-criticality (MC) environments, single logical control tasks are strongly bound to dedicated physical control units. Appliances that implement this traditional architectural approach range from automotive industry, where it is not uncommon that a single car contains dozens to a hundred of separate control units [HH08], to industry automation, where Programmable Logic Controllers (PLCs) and Human Machine Interfaces (HMIs) are physically separated: *critical* logical tasks compute on different physical computing platform than the *uncritical* HMIs. Historically, the separation of components stems from a time, where components were autonomous subsystems. In [Bro06], Manfred Broy summarises:

> The first software-based solutions were very local, isolated and unrelated. The hardware/software systems were growing bottom up. This determined the basic architecture in cars with their dedicated controllers (ECUs) for the different tasks as well as dedicated sensors and actuators. Over the time to optimise wiring, bus systems (see [CAN91]) were deployed into the cars by which the ECUs became connected with the sensors, and actuators.

Additionally, industrial real-time control systems are often built by extending general purpose commercial off-the-shelf (COTS) hardware components to reduce development effort in time and cost by maximising the re-use of existing solutions. The approach is commonly taken in many industrial domains, for instance automation and control systems [KG19], civil infrastructure projects [Fou19], medical appliances [Kis09] or robotics [Qui+09].

The re-use and extension of components is beneficial if flexibility in system capabilities is more important than potential reductions in cost that can be achieved by mass-producing tailored devices that precisely satisfy requirements, but usually never exceed them. Such scenarios often appear, for instance, in the automotive industry, but are rarely applicable to low-volume domains like medical appliances, industrial control, or even home automation.

Traditional architectural approaches are being revised since multi-core (MC) CPUs became broadly available. Figure 1.1 shows the evolution of CPUs during the last decades: Simultaneously with the flattening of the increase of the per-core frequency since ≈2005, the number of cores per CPU increases. Today, powerful multi-core CPUs are omnipresent, and de-facto standard components in COTS hardware, in enterprise hardware, in the high-performance computing (HPC) domain and in server markets. Additionally,

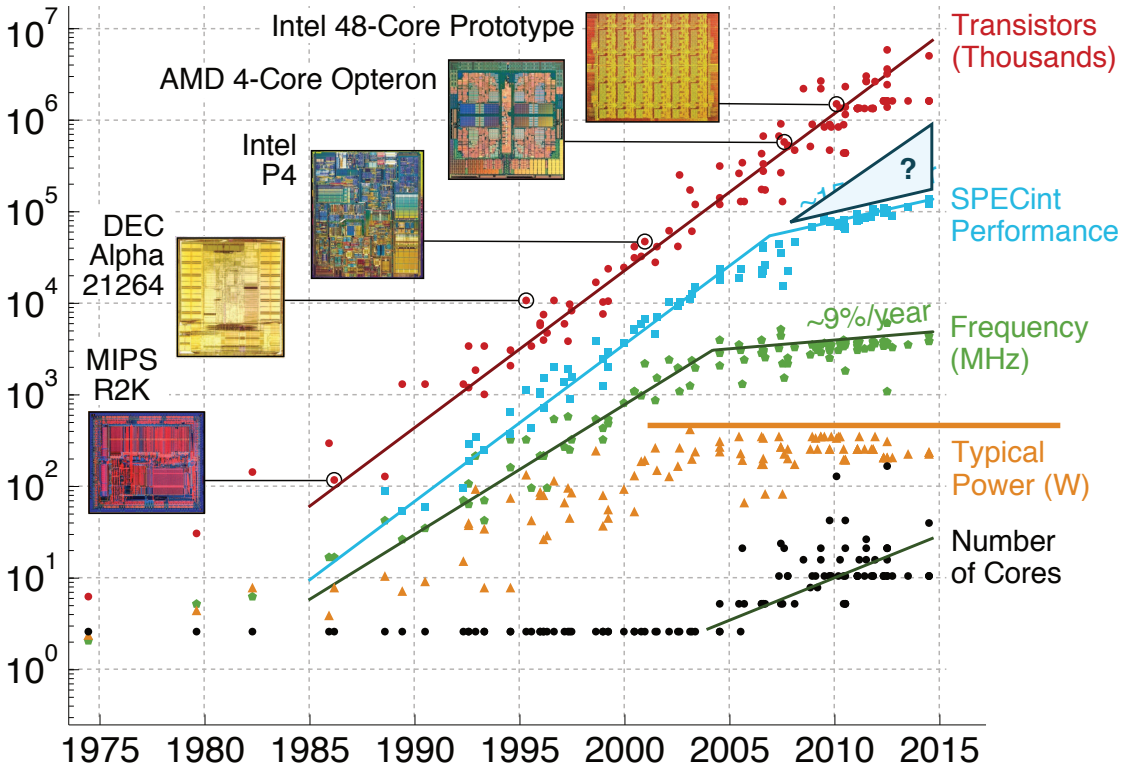

Figure 1.1.: Evolution of processors since 1975. Since ≈2005, the number of cores per central processing unit (CPU) steadily increases. Simultaneously, the increase of processor frequency declined. Taken from [Bat20].

MC CPUs gain in importance in embedded markets [Moy13]. Yet, component-based hardware architectures forego the capabilities and the resources of multi-core CPUs: Their resources allow for running tasks of previously dedicated hardware units on one single hardware unit [Moy13]. The consolidation of traditional systems to single hardware units is still an ongoing architectural trend [Bro06]. The reasons for this architectural evolution are:

1. **Reduction of hardware costs.** The consolidation of components removes duplicated or redundant hardware components, which eventually lowers expenses for hardware. Every formerly logically separated component required, for example, its own redundant power supply. Further on, physically distanced components need communication links. Robust bus systems are required for reliable data transmission. However, every component shares, for example, the same duplicated hardware driver stages for accessing shared communication buses. These and similar duplicated components can be efficiently replaced by on-chip software communication channels [Moy13] in consolidated systems.

2. **Simplification of system architectures.** The abovementioned components are superfluous in a consolidated architecture, as communication between components happens within the same computing unit: Besides complex bus systems and their hardware components, higher layer protocols can be replaced by lightweight on-chip communication primitives. As a side effect, architectural simplifications can also be beneficial for software certification–that is expensive formal attestation efforts, as typically required for software for safety-critical systems [KZ09].

3. **Improvement of maintainability.** Industrial systems need to be maintained during their whole life cycle. Product life cycles range from decades (*e.g.*, in the automotive industry) up to centuries (*e.g.*, in avionics or for civil infrastructure). As the software is tightly coupled to the hardware architecture, long-term maintenance is simplified by the reduction of the variability of different hardware components of an appliance.

   Formerly distributed hardware units require coordinated software update strategies in order to prevent software incompatibilities between different units. In consolidated architectures, there is only one single instance that requires maintenance and updates. This simplifies large-scale roll-outs of software updates in the broad field.

4. **Reduction of development costs.** "50-70% of the development costs of the software/hardware systems are software costs" [Bro06]. The consolidation of systems can save software development costs, as the overall complexity of the systems is significantly reduced, and concentrated to one single unit. Furthermore, consolidation eases testing and simulation of systems. The simulation of formerly complex interwoven distributed hardware units is substituted by the simulation of one single platform, which contains less software that is object to testing.

System consolidation is a straight forward and well-understood engineering challenge with various architectural implementation possibilities in case of the consolidation of uncritical systems (*e.g.*, refer to the techniques that are widely applied in cloud computing). However, it becomes a non-trivial challenge if one or more safety-critical systems (SCSs) are involved: despite the considerable advantages of system consolidation, there remain strict requirements on SCSs in general, and on mixed-criticality systems in particular that inevitably enforce decent and well-considered design of the system's architecture. I will elaborate those requirements in the next section.

1.2 Safety-Critical Systems and Open Source

According to [Kni02], a system is a SCS, if a failure of the system leads to consequences that are considered *unacceptable*. Unacceptable consequences can be the loss of human life, property damage, or environmental damage [Kni02]. Industries that are concerned with safety-critical systems are, for example, the automotive industry, aircraft industry, energy industry (*e.g.*, power plants), and industrial automation (*e.g.*, robot or process control).

As mentioned before, depending on the field of application and the application environment, different standards and norms put functional as well as non functional requirements on those systems. However, there are common basic requirements that are put on any SCS.

1.2.1 Requirements

The existence of unacceptable consequences implies that there *must* exist measures to either mitigate or prevent them. The archetype of SCS are airbags in cars. In sum, in case of unpreventable accidents, the unacceptable consequence is severe injuries of occupants, or, even worse, the loss of human life. The countermeasure (resp., the mitigation) is the ignition of an explosive charge to inflate the *airbag*.

Req. 1 **Real-Time and OSS.** The precise ignition of the explosive charge is the software-controlled key element for the success of the safety measure; the airbag must be inflated at the right time—it *must not* be inflated too early or too late. Precise ignition implies both, on time ignition in critical situations, and no ignition in all other situations—misdetermination leads to unacceptable consequences. Hence, airbags are safety-critical systems, and typically have the highest classification of initial hazard (*i.e.*, ASIL D [ISO26262; Alj+09]).

The example illustrates the necessity of the time-critical evaluation of measurement data, and the on-time execution of corresponding countermeasures that must be guaranteed under all circumstances: A countermeasure that is not executed, executed too early or executed too late leads to severe consequences. This forms one fundamental requirement that can be found in any SCS: the industrial robot must stop movement in case of approaching humans—on time, the control systems of the autopilot of an aircraft must take the right decisions—on time, the infusion pump must administer the drug according to a defined gradient—on time.

From the non-functional perspective, every SCS is a time-critical system. In the lingua of computer science, time-critical systems are hard real-time systems [Kop11]. I want to emphasise that *any* safety-critical system *is* a hard real-time system: Interventions **must** be executed within a strictly predefined time-window, and there must not exist any outliers beyond that time-window. Software systems without hard real-time requirements are not safety-critical.

Commonly, proprietary system software components are used to implement real-time systems. Such special purpose real-time operating systems are, for example, Nucleus RTOS, PikeOS [KW07], QNX or VxWorks. The downside of many of those operating systems (OSs) is the lack of available features, especially when those systems are being used in complex appliances that demand heavy-weight features. Yet, the demand for features corresponds with the level of criticality: the infotainment system of a car obviously has lower criticality than the ACU, yet it must provide a wider range of features, such as to support various audio/video codecs, to support wireless interconnectivity, etc.

For such feature-rich applications, the OSS ecosystem provides a wide range of diverse solutions that range from highly flexible and adaptable OSs to high-level userspace libraries. Meanwhile, Linux is *the* de-facto universal standard OSS OS component in various non-critical industrial appliances. It comes with an extensive support for many hardware devices and implements numerous standardised protocols while it remains highly customisable [Die+12] for specific use-cases: Linux supports hardware configurations that range from supercomputers to deeply embedded systems. Linux as base platform component in combination with other components of the OSS ecosystem form the base platform components of various systems.

By default, Linux is not a real-time operating system (RTOS). To serve real-time payloads, there exist several approaches to extend the Linux kernel by real-time (RT) capabilities: Real Time Application Interface (RTAI) [Man+00], Xenomai [Kis09], Preempt-RT [PRT20], and RTLinux [Yod99]. Xenomai and Preempt-RT are the most popular extensions [RMF19]. However, none of the aforementioned approaches has yet been fully integrated in the Linux kernel, some of them will never be integrated. Hence, maintenance and enhancements of those projects happens in parallel to regular ongoing development activities of Linux, which leads to enormous maintenance efforts [RLM16]. Nonetheless, Linux and the OSS ecosystem remain attractive opportunities to be used in safety-critical environments.

The amount of industrial consortia that is hosted by the Linux Foundation (LF)[2] underlines the strong interest on OSS in that area. Refer to whitepapers of different stakeholders: The Civil Infrastructure Platform (CIP) [CIP17], the Automotive Grade Linux (AGL) [AGL18] or Enabling Linux in Safety Applications (ELISA) [LF18].

However, real-time capabilities of the software system are not the unique criterion to qualify a system for being used in safety-critical applications. Irrespectively of the origin of the software, regulatory authorities demand for software qualification. This is the second requirement on safety-critical systems.

Req. 2 **Software Certification.** Safety-critical systems typically require system certification. As explained before, certifications shall ensure the correctness of the behaviour of a system in any situation. Therefore, systems must fulfil several NFRs, such as reliability, robustness, failure-tolerance, and others. An important criterion to succeed certifications is to provide evidence that those NFRs are fulfilled.

Formal verification methods can be used to qualify for certain certification criteria [Jää+12]. While it is possible to prove the formal correctness of aspects of systems [Kle+14], formal verification is a highly complex endeavour that is hard to transfer to real-world systems that comprise tens of millions of lines of code. The size of a software system is a limiting factor for any certification effort.

Safety standards and norms set great store on development processes, process hierarchies and traceability of development as a whole, and its decisions. The norms expect that strict adherence to preset processes shall implement high software quality. As evidence, the adherence to the process must be verifiable. The verification can be justified with quantifiable aspects of the development process. Usually, non-functional *safety* aspects of a project are concomitant with the development of a project. As this is one of the central aspects of this thesis, I will later explain (Ref. Section 1.3.1) in detail why those requirements are in conflict with the nature of OSS projects. While standards and norms recommend certain development models (*e.g.*, the V-Model in case of [IEC61508]), they generally do not forbid to incorporate other models, if they are structured and managed [PMB18]. Nevertheless, a clear definition of the process is prerequisite and requires means to measure the adherence of the process.

A carefully considered choice of software components for different levels of criticality can beneficially contribute to the solution of this issue. However, especially

[2] The LF is a non-profit organisation that hosts different consortia of different groups of interest with the goal to promote the growth of Linux.

in MC scenarios, where multiple systems of different criticality execute on the same platform, system architecture must implement means to guarantee freedom from interference of different executing domains.

Req. 3 **Strict Isolation.** Because of the increasing computational power of single systems, formerly separated hardware/software components are increasingly consolidated on single systems. Such mixed-criticality systems are a specialisation of SCSs. In MC scenarios, multiple applications or systems that may have different safety integrity levels (SILs) run on the same computational unit (*i.e.*, platform). Different computing domains must be strictly isolated to ensure the dependability of the appliance. Additionally, the hardware and the system software stack must provide means to guarantee the freedom from interference, that is, the "absence of cascading failures between two or more elements that could lead to the violation of a safety requirement" [ISO26262]. Having the software certification requirement in mind, strict isolation of computing domains, simplicity and minimality of critical aspects of the system architecture and the minimisation of the amount of critical code are encouraging factors for safety certification.

In many cases, the individual software components that are object of consolidation stem from formerly dedicated hardware units and, hence, already received certification as they already underwent the certification process. Thus, it is important to run legacy payloads in those isolated domains without major modification.

With respect to those requirements, the context of this thesis is as follows:

1. I consider consolidated mixed-criticality systems, where Linux serves as feature-rich OS for aspects of the system with less criticality.

2. Domains with high criticality shall execute in dedicated and strictly isolated domains. Those domains shall run (potentially pre-certified and preexisting) RT payloads.

3. The underlying system architecture must maintain real-time capabilities, as domains may run safety-critical payloads.

4. The system architecture must suite the certification requirements of the whole system.

1.2.2 Related Approaches

Since 2014 [OSADL14], the SIL2LinuxMP project is an ongoing industrial research project that aims to address and fulfil the requirements of Section 1.2.1. Authors investigate certification possibilities of safety-related products that use an OSS software stack that builds upon Linux as OS. The goal of the project is to provide a certification template that can be reused for a product-specific implementation of their proposed architecture up to a safety level of SIL 2 (Ref. [IEC61508]).

With respect to Req. 3, the certification template shall provide qualification arguments for *pre-existing software components* that include, besides Linux as the foundation of their architecture, base platform components, such as the standard C library (*i.e.*, glibc) and userland tools (*e.g.*, BusyBox) [PMB18]. By using isolation mechanisms that are provided by Linux, SIL2LinuxMP shall allow to run *applications* of mixed-criticality, that is, to run isolated *SIL 0 containers* in parallel to *SIL 2 applications* (*cf.* Fig. 1.2).

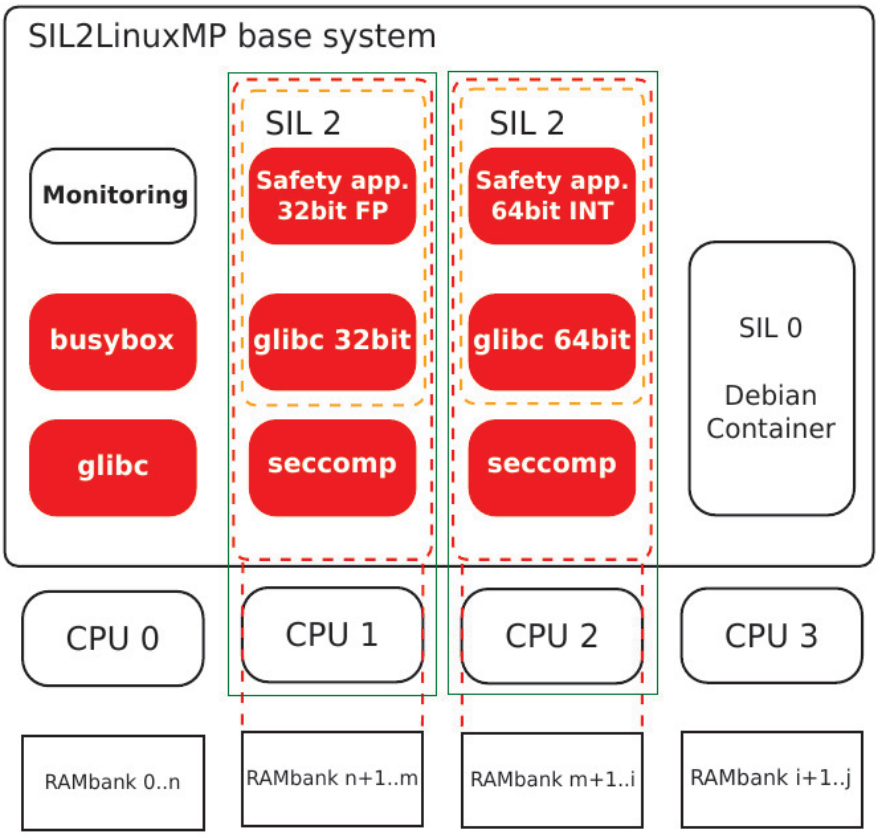

Figure 1.2.: SIL2LinuxMP architecture proposal. Proposed in and taken from [PMB18].

To address Req. 1, a real-time capable Preempt-RT-patched Linux is at the core of their platform [Gui14]. They execute on single or MC platform that uses COTS components. To

ease Req. 2, they minimise the amount of code, the Linux kernel configuration must be tailored to only fulfil functional and non-functional requirements that relate to the specific product. While their architecture is not bound to a specific implementation, authors of SIL2LinuxMP suggest one possible implementation. In their proposal, SIL2LinuxMP uses PALLOC [Yun+14], a DRAM bank-aware memory allocator for performance isolation on MC platforms [Gui14]. Besides Preempt-RT, PALLOC is the second external extension for Linux that is not officially supported by the Linux kernel community, which complicates the maintenance of the system (Ref. Appendix A.1). It allows for dynamic and flexible partitioning of DRAM banks of common COTS systems, and assigns partitions to cores of the system. "The private banking strategy reduces performance variations (up to 4.4x), and offers better real-time performance on COTS multi-core platforms" [Yun+14].

Safety-critical and uncritical applications are pinned to and only execute on dedicated CPUs. To achieve isolation between different domains, SIL2LinuxMP leverages Linux Namespaces [BN06] and Control Groups [Heo+15]. For domain management, they propose to handle cgroups manually (*i.e.*, without comprehensive standard tooling), and to implement a minimal, tailored launcher [Gui14] that better suits for certification. The software utility and tool suite BusyBox shall serve as a minimalist runtime environment, backed by the heavyweight glibc as standard C library for safety-critical parts of the system. The rationale behind this choice is to "minimise safety-related runtime overhead" [Gui14]. To avoid illicit system calls during operation, they propose to use seccomp. Seccomp whitelists system calls that are permitted in safety-critical domains. System calls that are not used shall not be allowed to be executed in the operative phase.

Uncritical *SIL 0* domains shall use standard container (*i.e.*, sandboxing) mechanisms that are provided by Linux. SIL2LinuxMP uses a separate domain for system monitoring and admission (*cf.* Fig. 1.2).

To further address Req. 2, authors argue that statistical modelling of development process characteristics, for example bug rates of a system, or quantification of development process activities, supports certification efforts. They further argue that "the Linux kernel developers define a quite rigorous development process [LKP18] which, in principle, can address most of the requirements for a structured and managed process set forth in [IEC61508] Ed. 2 part 3." [PMB18]

Authors of SIL2LinuxMP identify the lack of certifiable MC CPUs as one of the most striking issues for their certification efforts, while they see the certification of a software stack that contains several millions of lines of code even in the tiniest configuration, surprisingly, as a manageable issue. However, the Linux kernel of SIL2LinuxMP incorporates two invasive out-of-tree developments, which may result in major maintenance

issues: In Appendix A.1, I will present development characteristics of the Preempt-RT and quantify maintenance efforts. However, the PALLOC extensions did, at the time of writing, not receive any updates since three years. The latest supported kernel Version is v4.4 from January 2016.

As SIL2LinuxMP purely relies on Linux-based isolation mechanisms, all computing domains share the same kernel: a critical system failure in any computing domain of the system leads to an overall system failure, which endangers Req. 3. As the shared monolithic Linux kernel operates across all levels of criticality, isolation and freedom from interference requirements can not be fulfilled under all circumstances.

1.3 Research Context of this Thesis

In this section, I define the contextual frame of this thesis, and derive my research questions.

1.3.1 Safety-Critical Systems, OSS and Certification

As an intellectual property of individuals or legal entities, many industrial software components are based on the results of different, highly specialised proprietary–closed source–development undertakings. Over the last decades, general purpose OSS components gained great interest for being used in industrial special purpose systems, as they are highly adaptable, customisable and tailorable, which makes them suitable for special purpose applications. OSS components are a high-quality economic alternative to former proprietary components in a wide range of applications.

The fundamental difference of OSS and closed source software is the licensing model. In proprietary software development undertakings, a customer achieves their objectives (for example, the implementation of new features, or the adherence to certain development processes) by commercial transactions. In OSS projects, there is no necessity of such customers in the classic sense. Decision makers and customers can further be disjoint parties.

While, in the first place, such dynamics in OSS projects may appear incompatible with economic principles of the industry, there is an increasing interest for the use of OSS components for uncritical as well as for safety-critical purposes. Especially base platform components that implement indistinguishable characteristics, like firmware, bootloaders, or operating systems are increasingly being used in industrial appliances.

However, the increasing usage of OSS components even in critical domains demands for certification strategies. Their comprehensive support of hardware, their wide range of application, and their universality are the drivers for efforts to utilise OSS components in safety-critical environments–environments, where software components formerly mainly stem from proprietary products, which have controlled development environments with respect to non-functional processual aspects.

As explained earlier in Section 1.2, regulatory norms put such requirements on the development process of software that is being used in safety-critical applications. They require, for example, the strict adherence to a formally defined development process as well as implementing mechanisms to verify the adherence. Typically, in safety-relevant proprietary development processes, the adherence to development processes is measured by traceability means that are implemented in the process. Any decision that affects the code needs to be traceable.

However, in OSS projects, "the process is defined and enforced by social contract, but not by legal working contracts" [Bul17a]. Linux, for example, is an OSS general purpose operating system (GPOS) that has a wide area of application–safety-critical applications represent only *one* small field of application. The existing development process that evolves within the community optimally fits for this wide range of applications. Consequently, regulations that are given by safety norms are not relevant for the vast majority of stakeholders of the project. The conformance with safety regulations would lead to enormous extra effort and expense during the development for the whole community, while only a small amount of stakeholders would benefit from them. Hence, communities are understandably not willing to replace their existing, proven in use *social contracts* by external requirements.

As the development process of OSS projects can not be adjusted ad libitum, it raises the question if (a) existing development processes already fulfil safety-critical requirements to some degree, and (b) if the real ongoing development actually is in alignment with its own development process. While the first question (a) needs to be addressed by safety experts and regulatory authorities, the second question (b) can be split up into fine-granular research questions that can be answered by ex-post analyses of the development process of OSS projects by using quantitative software engineering methods. The key enabling factor lies in the nature of OSS projects: their high public transparency. Virtually all development artefacts, such as code changes, developer discussions, code review or bug trackers are available to the public. However, to answer questions on the development process requires a *traceable* representation of a software project, which is not given out of the box. Due to the used tools (*e.g.*, mailing lists), development artefacts are represented in an unorganised fashion, when being compared with the development

result, traceability against the software repository. Even in the mature Linux community, traceability of software artefacts is a frequent topic for discussions,[3] and not given in most cases. In the context of using OSS components in SCS, this leads to the research questions that concern safety certification efforts:

Research Question 1: *Can complex OSS development processes be reconstructed in ex-post analyses?*

Research Question 2: *What are reasonable metrics to quantify the adherence to or violations of OSS development processes?*

These research questions define both, the contributions as well as the contentual boundaries of this thesis:

Contributions I show that it is possible to reconstruct complex OSS development processes by exploiting their key characteristic: transparency. In Part I, I present a highly accurate methodology to reconstruct software development processes of projects that incorporate *noisy* mailing lists (MLs) as their main communication platform. Especially low-level system software, which I target in this thesis, heavily make use of MLs. I present the Patch Stack Analysis (`PaStA`), an industry-grade, fully published and extensible framework that allows for further in-depth analyses and scales with the world's largest software development projects.

On an empirical basis, I demonstrate that the approach can be used to extract specific indicators that support certification efforts of OSS components for the use in safety-critical systems. To underline the capabilities of the approach, I define quintessential indicators that can be used to qualify development processes of OSS projects in safety-critical contexts.

Boundaries The measurements are used to demonstrate the practicability of the presented approach. While they give us answers to real-world issues, they do not implicitly enable a safety assessment: The quantification of effects does not provide qualitative judgements on processual reasonableness, yet they form one important component in a superior certification endeavour. Particular questions need to be defined by safety experts, and can be answered by leveraging the methodical framework. The incorporation of such answers to safety certification efforts remains future work.

[3] Refer to a discussion on a Linux kernel mailing list: `https://lists.linuxfoundation.org/pipermail/ksummit-discuss/2019-August/006739.html`

Software engineering and the analysis of development processes is an essential requirement for safety certifications, yet meta-discussions on incidents in the development chain will not implement fundamental functional properties of the *system architecture* of SCSs. Functional properties, for example real-time capabilities (*cf.* Req. 1), are enabled by the system's architecture which is the result of the system engineering process.

The challenge is to design a system architecture, which is beneficial to safety certifications: with respect to non-functional requirements, a sound system architecture can significantly minimise the amount of software that is relevant to safety certifications.

1.3.2 Mixed-Criticality Systems, OSS and System Architecture

The architecture presented in Section 1.2.2 has the strong disadvantage that it only relies on isolation mechanisms that are given by the OS, Linux. A more dependable alternative is to rely on stronger isolation guarantees that are given by hardware-based mechanisms: virtualisation extensions. Embedded virtualisation substantially differs from common enterprise, desktop or mainframe virtualisation [Hei08], where the technology has its roots. Many of these traditional users of virtualisation consider the consolidation of services as major motivation.

While hypervisors are often optimised for high throughput and optimal performance in the desktop and enterprise segment, virtualisation solutions for real-time constrained embedded systems significantly differ: the architecture needs to target low latencies, deterministic computation cycles and maintaining real-time capabilities. The resulting scenarios have received substantial attention during the last decade [BD13], and the *conceptual* advantages and disadvantages of the many possible approaches to build such systems are well researched [Hei08; KW07; MRC05; CRM10; SK10a; Xi+11; Pin+14; Pin+17; Li+19].

Nevertheless, many embedded hypervisors adopt established practices from *classical* virtualisation: overcommitting of hardware, paravirtualisation [BDF+03; CRM10] or emulation of devices, and guest scheduling [KW07]. Those techniques lead to software runtime overheads, as the system software stack (the hypervisor as well its guests) must implement significant amounts of code to implement those techniques. They are a major source of hard to control indeterminism [Rie16; Xi+11; Kis11] and endanger Req. 1, hard real-time requirements.

Static hardware partitioning is a special case of embedded virtualisation that exclusively assigns hardware resources to isolated computing domains that execute on the same logical platform. It makes the assumption that available resources are greater or equal

than the required computational power: there is no need for sharing any resource of the system. As mentioned before, this is already the case in numerous industrial appliances.

Static hardware partitioning means that the assignment of physical resources to computing domains is *static*, that is, the assignment does not dynamically change over time. *Partitioning* implies the isolation across partitions or domains. Virtualisation extensions of modern CPUs [AMD05; UNR+05; Int18b; VH11] can be exploited to create such static and distinct execution environments. However, static hardware partitioning does not yet give guarantees on implementing a real-time capable system architecture.

If it is possible to implement static hardware partitioning upon virtualisation extensions of modern architectures with **no** (software-induced) hypervisor resp. virtual machine monitor (VMM) overhead during system operation, then the underlying real-time guarantees of the execution platform (that *must* exist in any case) are inherited by execution domains (*i.e.*, the guests) by design and without any further software interaction. If, in addition to that, the system guarantees freedom for interference across domains, I will call it an *ideally* partitionable system. Yet, it remains an open question if modern COTS are *ideally* partitionable systems.

The technique is not only beneficial for maintaining RT guarantees, but also for Req. 2: One can argue that software that does not run during operation does not need to be certified. Under ideal conditions, the hardware partitioning only sets up the partitioning and never gets active during the operational phase. Only the state of the architecture *after* partitioning needs to be considered for certification. This drastically reduces the amount of code that is relevant for certification. Only hardware components require certification, which, again, is required in any case.

However, the outlined architecture raises numerous engineering challenges *especially* when being evaluated against real-world hardware. In the second part of this thesis, I will provide answers to the following research questions:

Research Question 3: *What are hardware requirements to implement ideal hardware partitioning?*

As I will conclude that **RQ 3** is already possible under specific circumstances, but, in general, not yet possible on current COTS hardware components, I raise the last research question:

Research Question 4: *What are the limitations on current COTS hardware to implement ideal hardware partitioning? What are unavoidable overheads?*

Contributions First, I will give a clear definition of the term *ideal hardware partitioning*, and systematically evaluate associated hardware requirements. I present contributions to architecture and implementation of a partitioning industrial open source hypervisor that allows to build real-world systems that offer effective interference guarantees derived from hardware partitioning.

While contemporary CPUs provide sophisticated virtualisation extensions, they still cannot provide suitable interfaces to implement a fully zero-trap (*i.e.*, no VMM interaction) *ideal* approach. Consequently, I elaborate on widely underestimated hardware requirements that are necessary to implement hardware partitioning without hypervisor interception. The presented approach is a solution to enable safe coexistence of workloads of mixed criticality on a single system for many general purpose use cases.

On real-world systems, static hardware partitioning is not possible without hypervisor activity due to hardware limitations. Nevertheless, evaluations and microbenchmarks underline the suitability of static hardware partitioning for numerous real-world use cases. In particular, the microbenchmarks include an evaluation of mitigations against Spectre-like attacks [Sch+19a; Min+19; Van+18; Wei+18; Lip+18; Koc+19; Can+18], as some mitigations require intervention of the hypervisor, if the guest decides to protect its own domain. I explain how architectural decisions reduce the attack surface for *cross*-domain low-level hardware attacks, such as Spectre [Koc+19].

I will conclude that static hardware partitioning offers a promising and safe solution for system consolidation—even on low-cost contemporary COTS hardware components.

Boundaries In contrast to existing solutions, only the payload of critical domains remain relevant for safety certification, and, possibly, the thin hypervisor code base that is used for partitioning. However, because of this, and as the case-specific deployment of the solution depends on the requirements of the specific use-case, the suitability of the approach needs to be addressed by safety experts.

I will intentionally not elaborate on concrete strategies *how* the approach can be certified in particular as this is beyond the scope of this thesis. Nevertheless, the approach successfully minimises the amount of code that is relevant for safety certification without the loss of any functionality.

Furthermore, I will focus on the software stack and its minimisation by leveraging hardware extension. Those extensions are relevant for certification. This thesis does not cover aspects of the certifiability of hardware.

1.4 Structure

This thesis is structured into two major parts. I begin with Part I that addresses the topic *Safety-critical systems, OSS and certification* (investigating Section 1.3.1 and **RQs 1-2**) with respect to Req. 2. Part II addresses *Mixed-Criticality Systems, OSS and System Architecture* (investigating Section 1.3.2 and **RQs 3-4**) with respect to Req. 1 and Req. 3. The structure of this thesis is outlined in Fig. 1.3. The rest of this thesis is structured as follows:

Part I *Reconstruction and Analysis of Software Development Processes* (pp. 25–95)
Part I of this thesis centres around quantitative software engineering. I develop a methodology to reconstruct and quantify development processes in OSS projects. I will use the framework to provide answers to questions on the process. In this part, I lay the foundation to give answers to research questions **RQ 1** and **RQ 2** (Ref. Section 1.3.1).

Chapter 2 *Reconstruction* (pp. 25–53)
In this chapter, I address **RQ 1**. I present the methodology of `PaStA`, a software engineering framework that reconstructs development processes of OSS projects, by linking otherwise unorganised development artefacts (*i.e.*, proceedings in the process before integration) to development results (*i.e.*, software repositories, the result *after* the actual development process). This connection is precondition to quantitatively answer further in-depth questions on development processes.

Chapter 3 *Analysis* (pp. 55–93) This chapter builds the fundament to answer **RQ 2**, the adherence to or violations of development processes. Without the loss of generality, I will exemplarily answer the question by conducting analyses on the Linux kernel.

Part II *System Consolidation of Safety- and Mixed-Critical Systems* (pp. 99–153)
Part II of this thesis focuses on the system architecture for mixed-criticality systems. I present and evaluate Jailhouse, a Linux-based static partitioning hypervisor that aims to address requirements in Section 1.2.1.

Chapter 4 *Ideal Hardware Partitioning* (pp. 99–123)
In Chapter 4, I present the architecture and philosophy of Jailhouse. Given the architecture, I define the concept of *ideal hardware partitioning*, and derive associated requirements on hardware to answer **RQ 3**. Finally, I evaluate those requirements against COTS hardware and conclude that ideal hardware partitioning on current COTS is realisable, but only under certain conditions.

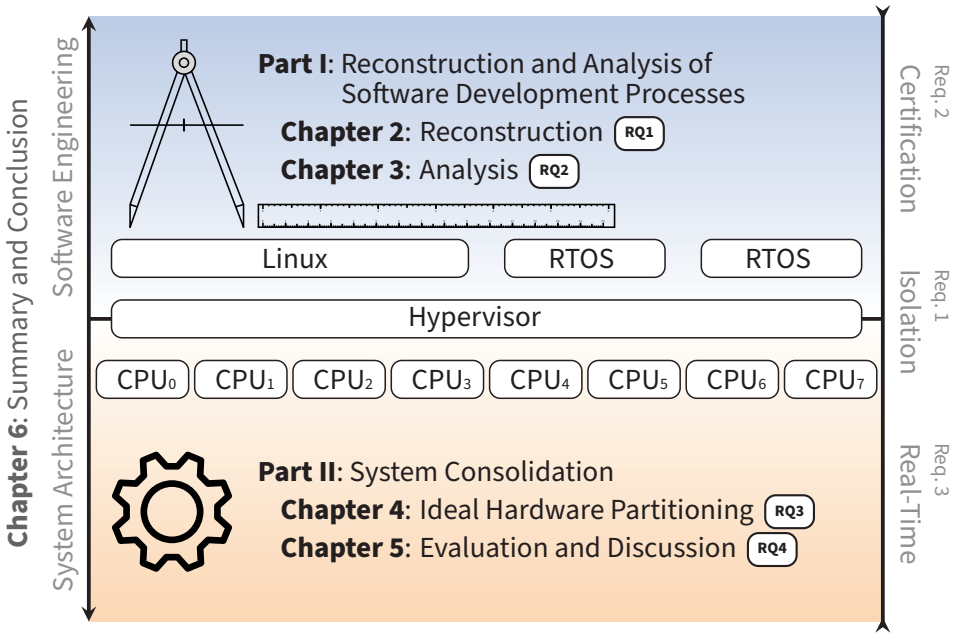

Figure 1.3.: Structure of this thesis. The software engineering and system engineering shares of this work are split up into two major parts. Four chapters are used to answer **RQs 1-4**. Answers to RQs will be given in the last Chapter 6, the Summary and Conclusion of the thesis.

Chapter 5 *Evaluation and Discussion* (pp. 125–151)

As there arise limitations on current COTS, I inspect those limitations and investigate, if they constitute as barriers for real-world application of the proposed approach. To answer **RQ 4**, I present a set of methodologies to quantify the impact of the limitations on real-time systems. Given a specific use-case on a specific platform, those measurements serve as a basis for decision making if the approach is able to fulfil the requirements of a specific application.

Chapter 6 *Summary, Conclusions and Further Ideas* (pp. 155–159)

In the last chapter of this thesis, I summarise the results of Part I and Part II in a higher-level view, and draw conclusion on the consequences. I give answers to my research questions, and complete this thesis with a concluding summary.

Part I

Reconstruction and Analysis of Software Development Processes

2 Reconstruction

> *In the practical world of computing, it is rather uncommon that a program, once it performs correctly and satisfactorily, remains unchanged forever.*
>
> — **Niklaus Wirth**
> Program Development by Stepwise Refinement [Wir71]

This chapter shares material with the OpenSym '16 paper "Observing Custom Software Modifications: A Quantitative Approach of Tracking the Evolution of Patch Stacks" [RLM16] and the ICSE '19 paper "The List is the Process: Reliable Pre-integration Tracking of Commits on Mailing Lists" [RLM19].

Software for safety-critical systems requires certification (*cf.* Req. 2). Corresponding regulatory norms and standards put high demands on various aspects of the development process. Hence, development processes in orthodox proprietary *Enginner-to-Order* environments comply with those processual requirements.

Compared to conventional, proprietary industrial software, OSS exhibits different dynamics [MJ13], and often requires fundamentally different development processes [Cor11] because of project size and a high number of massively geo-dispersed stakeholders. Because of this nature of OSS, projects do not necessarily meet certification criteria [Bir+08]. While OSS communities may have their own (more or less formally defined, and more or less strictly followed) development process or *guidelines*, the process is defined by and evolves within the community [Bul17a]. For most projects, processual requirements that stem from safety-critical environments only play a subordinate role.

Nevertheless, OSS components are routinely deployed in industrial fields, and their use is increasingly explored in safety-critical or mixed-criticality appliances [Fou18a; Fou18b; LF18; BD13], such as medical devices or in automotive products. Especially for core components of a system that implement business-wise non-differentiating features

such as the system-software stack or middleware, OSS provides adequate solutions that have already proved to be reliable in other non-critical application domains (*cf.* Req. 1). Yet, most OSS projects are, of course, not willing to adjust their development process(es) to fulfil requirements for safety-critical systems for reasons explained in Section 1.3.1.

The question that remains is, if current effectively practised development processes in OSS projects already fulfil these requirements. To provide evidence, it is mandatory to be able to quantitatively *capture and trace the process* to answer this question.

In this chapter, I lay the foundation to answer **RQ 1.** I present the `PaStA` methodology and tool that is able to reconstruct the otherwise inaccessible development processes of OSS projects in ex-post analyses. The results of `PaStA` can be used to answer further questions on properties and characteristics of the underlying process. Later, in Chapter 3, I will use the methodology to give insights in OSS development processes.

2.1 Overview

Nowadays, the source code of projects is usually organised in version control systems (VCSs), such as SVN or git. They allow for tracking the changes of the software on a fine-granular level: *commits*. Commits in repositories typically consist of three components: the actual changes to the code, an informal description that motivates and explains the code change, and metadata, such as information on the author of the code. VCSs keep track of all changes that were made to the project, and allow for retrospective inspections of the evolution of a project at any point in time.

While repositories keep track of all changes to the project, they miss information on *how* the change materialised. What happened *before* integration to the repository? Was the change controversially discussed? How long did integration take, and how many revisions of the patch were required until final integration?

Furthermore, repositories only keep track of changes that made it into the project. Naturally, repositories miss patches that were never, or at least not yet, accepted for integration. While this may sound futile, as those patches do not directly affect the code, they *do* affect architectural decisions of the project. *Why* was a change rejected? Was it rejected after careful review, or was it simply ignored?

Given a formally defined development process, it is possible to provide answers to these questions. Such performance indicators can support argumentations for safety certification efforts, as they prove adherence to, or violations of the development process by providing evidence on a quantitative basis.

These hypothetical questions–some of them will be addressed in Chapter 3–shall illustrate that the software repository is the result of a preceding process. While a considerable corpus of research on software evolution focuses on mining changes (*i.e.*, commits) in software repositories, it omits their pre-integration history.

Software patches may have come a long way before their final integration into the official branch (known as *mainline* or *trunk*) of a project. There are many possible ways of integration. Among others, the origin of a patch can be a merge from other developers' repositories (*i.e.*, integration of branches or patches from foreign repositories), pull requests on web-based repository managers such as GitHub or GitLab, vendor specific patch stacks, or mailing lists (MLs).

In particular, MLs have widely been in use for software development processes for decades [Ere03]. They have a well-known interface (plain text emails), and come with an absolute minimum of tool requirements (*i.e.*, a mail user agent). Because of their minimality, simplicity, scalability, reliability and interface robustness, they are still widely used in many OSS projects. In particular, MLs are a core infrastructure component of long-lasting OSS projects such as low-level systems software (*e.g.*, QEMU, U-Boot, GRUB, etc.), operating systems (*e.g.*, the Linux kernel) or foundations (*e.g.*, Apache, GNU). Mailing lists form the backbone of the development processes [HNH03] in projects that are potential candidates for base platform components for safety-critical systems.

MLs are not only used to ask questions, file bug reports or discuss general topics, but implement a patch submit-review-improve strategy for stepwise refinement [Wir71] that is typically iterated multiple times before a patch is finally integrated to the repository (*cf.* Fig. 2.1).

Therefore, MLs contain a huge amount of information on the pre-integration history of patches. A commit in a repository may be the outcome of that process, while all intermediate steps leave no direct traces in the repository. Mailing lists allow for analysing development history and code evolution, but also enable to inspect reviewing and maintenance processes. They further allow for inferring organisational [Job+17] and socio-technical [Bir+06; Her07; Val+07] aspects of software development. This all is possible because MLs contain information on interactions between developers.

To assess non-formal OSS development processes, mapping patches on mailing lists to repositories is a key requirement, because the mails contain the facts: They are the artefacts of the development process. Together with the outcome of the process—the repository—, this forms a solid base for further analysis. Patches that appear on mailing lists are manually selected (*cherry-picked*) by the maintainer before integration into the

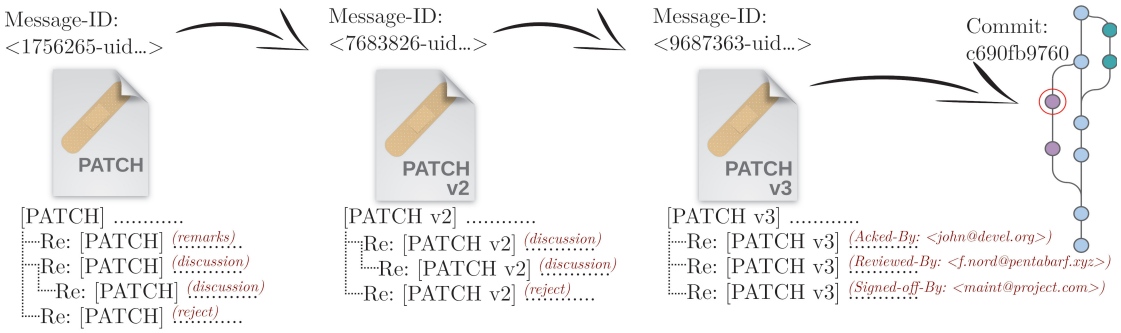

Figure 2.1.: Typical workflow: A patch gets resubmitted and improved for two times, before its final integration into the repository.

repository. They are also routinely combined (*squashed*) and modified (*amended*) on-the-fly, which is convenient for developers, but complicates tracking. Either way, a direct connection between the history on the mailing list and the repository commit is lost in the process [BGD07]. While this issue can be solved with appropriate technical means for the future, it does not solve the lack of traceability in the past, which in particular is important for certification.

I present a methodology accompanied by comprehensive automated tool support.[1] The methodology of the toolkit `PaStA` (**Pa**tch **St**ack **A**nalysis) allows for:

1. Reconstruction of Software Development Processes (this Chapter)
   a) tracking several revisions of a patch on mailing lists.
   b) mapping those patches on lists to upstream commit hashes, if the patch was integrated.
2. Analysis of Software Development Processes (Ref. Chapter 3).
   a) extracting process characteristics, that is, performance indicators of the development process.
   b) measuring the adherence to and violations of the process.

I identify and formalise **1.** as cluster analysis, and provide an algorithm to track the evolution of code change, that is, to reconstruct the development process. I will provide an in-depth evaluation of my, as well as of other approaches, together with a quantification of the accuracy of the approach. The traceability of code will be reduced to finding similar patches.

[1] Published under the GPLv2 license at `https://github.com/lfd/PaStA`.

2.2 Fundamentals

From an analytical standpoint, the downside of patch submission on mailing lists is asynchronicity, as there is no direct connection between the mailing list and the software repository. Maintainers manually integrate patches from the list and commit them to the repository. This process is typically assisted by tools provided by the version control system.[2] During this process, the connection of the email with the patch (identified by the unique Message-ID header of the mail) and the commit in the repository (usually identified by a commit hash) is lost.

Other difficulties are contextual divergences and textual differences [BGD07]. The commit in the repository may significantly vary from the patch on the mailing list, as other patches between submission and integration might have affected the patch. Additionally, maintainers may introduce additional changes to the patch.

There is also no connection between several revisions of a patch within the mailing list. A patch undergoes a certain evolutionary process between revisions, hence patches of different revisions may significantly differ as well, while they still introduce the same logical change.

Structure of Code Changes Independent of the type of submission, I formally define a patch p as a 2-tuple that consists of a commit message and a diff. This applies to patches that are posted on mailing lists as well as to commits that can be found in repositories. While the commit message informally describes the changes, the diff annotates the actual modifications (insertions and deletions) surrounded by a few lines of context. Context lines ease the understandability of the patch for human review. Patches can also include meta information, such as the author of a patch or the timestamp of its creation (Author Date). Not all types of patches contain the same set of metadata. Emails with patches contain several mail headers, while those headers are removed when the patch is applied to the repository. Repositories, in contrast, contain information on the exact spatial location of the patch.

A diff of a file consists of a sequence of *hunks* that describe the changes at a textual level. Every hunk h is introduced by a range information that determines the location of the changes within a file and contains a section heading h_{head}. Section headings display "the nearest unchanged line that precedes each hunk" [MES13] and are determined by a regular expression. Range information is followed by the actual changes: lines h^+ that are added to the new resulting file are preceded by '+', lines h^- that are removed from

[2] For example, `git am` (apply mail from mailbox) or git cherry-pick (apply the changes introduced by some existing commits).

the original file are preceded by '−' and lines h° that did not change are preceded by a whitespace '␣' (*cf.* Listing 2.1).

Metadata may also change over time [Bir+09; GAH16]; even the author of a patch may change. Therefore, metadata is intentionally not considered in the similarity analysis.

Mapping patches on MLs to commits in repositories requires to understand common workflows in projects [Ere03]: When the author of a patch wants his or her patches to be integrated in the project, they need to send their patch or *patch series* to the mailing list of the project. In the vast majority of cases, the patches are created with tools of the VCS. Typically, the VCS provides functionality to directly send the patches as emails.[3]

Patch Series A patch series is a cohesive set of mails that contain several logically connected patches that, in the big picture, introduce one logical change that is split up in fine granular steps. Listing 2.1a and Listing 2.1b show two successive mails in a patch series. Again, the submission of a patch or patch series is typically tool-assisted by the VCS.

Code Integration Process After patches are submitted, reviewers or any subscriber of the list may comment on them. This is done by starting a free-form textual discussion by replying to a mail that contains a patch. Inline comments refer to the related code lines. As replies on emails reference the Message-ID of the original email, it is possible to reconstruct the whole mail thread (*i.e.*, the discussion) of the thread. I will exploit this important property later in the analysis in Chapter 3.

Concerning change integration, the reviewing process may end up in the following scenarios:

1. The maintainer decides to integrate (*i.e.*, to commit) the patch(es).
2. The maintainer decides to reject the patch(es).
3. The patch(es) need further improvement and need to be resubmitted to the list.

It is not unusual that **3.** is repeated several times. In this case, further revisions of the patch are typically tagged in the email subject header with `[PATCH v<N>]` prefix, where `<N>` denotes the revision round. This iterative process of resubmitting further revisions of changes is a fundamental aspect of the development process and makes it necessary that a patch on a mailing lists must not only be linked to the repository, but also against other revisions of the patch in order to track its evolution. Fig. 2.1 illustrates

[3] For example, the `format-patch` in combination with the `send-email` tools of git.

a typical workflow: a patch was resent two times (v2 and v3), before being integrated to the repository.

Once maintainers decide to accept a patch, they may still amend the commit message or the code. Depending on the submission process of the project, maintainers or other persons working on the patch add additional *tags* to the commit message, such as `Acked-by: <mail>`, `Tested-by: <mail>`, `Signed-off-by: <mail>` among others. The exact usage of those tags depend on the project, and are part of the code review process, and hence, part of the development process.

Reviewers that vote for inclusion of the patch reply to it with a mail that adds an `Acked-by`, where `<mail>` contains the email address of the person who acknowledged the patch. Anyone who successfully tested a patch may send their `Tested-by`. The `Signed-off-by` tag indicates that the patch conforms with the Developer's Certificate of Origin.[4] Maintainers pick up mails with such tags (*i.e.*, mails `In-Reply-To` the initial patch) and append them to the commit message before integration. It is the decision of the maintainer whether to integrate the patch or not.

A patch on a list may significantly differ from its final version in the repository, which makes it hard to link them. Listing 2.1 demonstrates the complexity of finding similar patches. This examples contains two patches that appeared on the mailing list of BusyBox [BB18] and the eventual commit in the repository. In this case, the maintainer (Denys Vlasenko) heavily changed the original patches (authored by Tias Guns) that were sent to the project's mailing list: He picked up both mails, consolidated them to one commit (known as *squashing patches*) and additionally changed the commit message. During this process, metadata changed as well: the author date of the commit message is neither related to `[PATCH 2/6]` nor to `[PATCH 3/6]`. Still, both emails are related to the commit in the repository. This example was automatically found and linked by `PaStA`.

The complexity of finding similar patches is aggravated by the fact that patches are relative to a specific state of the code base, determined by the commit where the patches base on. When the latter changes between the time a patch was submitted and it was integrated, as other patches had been applied meanwhile, the version control system tools try to (semi-)automatically adopt the changes, which leads to different context information despite identical changes. If automatic methods fail, merge conflicts must manually be solved by humans.

Multiple maintainers may commit the same patch to their own branch. In this case, a patch occurs multiple times on the master branch of the repository, once those branches

[4] See Linux's Documentation/process/submitting-patches.rst. Other projects use similar tags.

```
Message-ID: <1338734589-11512-3-git-send-email-tias@ulyssis.org>
Date: Sun,  3 Jun 2012 16:43:04 +0200
To: Discussion and development of BusyBox <busybox.busybox.net>
From: Tias Guns <tias@ulyssis.org>
Subject: [PATCH 2/6] android: use BB_ADDITIONAL_PATH

Signed-off-by: Tias Guns <tias@ulyssis.org>
---
 include/platform.h |    4 ++++
 1 file changed, 4 insertions(+)

diff --git a/include/platform.h b/include/platform.h
index d79cc97..f250624 100644
--- a/include/platform.h
+++ b/include/platform.h
@@ -334,6 +334,10 @@ typedef unsigned smalluint;
 # define MAXSYMLINKS SYMLOOP_MAX
 #endif

+#if defined(ANDROID) || defined(__ANDROID__)
+# define BB_ADDITIONAL_PATH ":/system/sbin:/system/bin:/system/xbin"
+#endif
+

 /* ---- Who misses what? ------------------------------------ */

--
1.7.10
```

(a) [PATCH 2/6] in a series: the author adds some conditional preprocessor definitions.

are merged. In practice, it means that I can not shortcut the analysis once a patch on is mapped to a commit in the repository.

Those and other facts [Bir+09; Jia+14] underline that similar patches can not be simply linked against each other by examining their textual equality.

```
Message-ID: <1338734589-11512-4-git-send-email-tias@ulyssis.org>
Date: Sun,  3 Jun 2012 16:43:05 +0200
To: Discussion and development of BusyBox <busybox.busybox.net>
From: Tias Guns <tias@ulyssis.org>
Subject: [PATCH 3/6] android: fix 'ionice', add ioprio defines

patch inspired by 'BusyBox Patch V1.0 (Vitaly Greck)'
https://code.google.com/p/busybox-android/downloads/detail?name=pa[...]

Signed-off-by: Tias Guns <tias@ulyssis.org>
---
 include/platform.h |    2 ++
 1 file changed, 2 insertions(+)

diff --git a/include/platform.h b/include/platform.h
index f250624..ba534b2 100644
--- a/include/platform.h
+++ b/include/platform.h
@@ -336,6 +336,8 @@ typedef unsigned smalluint;

 #if defined(ANDROID) || defined(__ANDROID__)
 # define BB_ADDITIONAL_PATH ":/system/sbin:/system/bin:/system/xbin"
+# define SYS_ioprio_set __NR_ioprio_set
+# define SYS_ioprio_get __NR_ioprio_get
 #endif


--
1.7.10
```

(b) `[PATCH 3/6]` in a series: the author adds further definitions under the same condition.

```
commit 3645195377b73bc4265868c26c123e443aaa71c6
Author: Tias Guns <tias@ulyssis.org>
Date:   Sun Jun 10 14:26:32 2012 +0200

    platform.h: Android tweaks: ioprio defines, BB_ADDITIONAL_PATH

    Signed-off-by: Tias Guns <tias@ulyssis.org>
    Signed-off-by: Denys Vlasenko <vda.linux@googlemail.com>

diff --git a/include/platform.h b/include/platform.h
index d79cc97..ba534b2 100644
--- a/include/platform.h
+++ b/include/platform.h
@@ -334,6 +334,12 @@ typedef unsigned smalluint;
 # define MAXSYMLINKS SYMLOOP_MAX
 #endif

+#if defined(ANDROID) || defined(__ANDROID__)
+# define BB_ADDITIONAL_PATH ":/system/sbin:/system/bin:/system/xbin"
+# define SYS_ioprio_set __NR_ioprio_set
+# define SYS_ioprio_get __NR_ioprio_get
+#endif
+

 /* ---- Who misses what? ----------------------------------- */
```

(c) Maintainer squashed both mails to one commit and amended the commit message.

Listing 2.1: Example of two mails and one commit that were automatically found and linked by PaStA.

2.3 Clustering Similar Patches

Let $\mathcal{C}$ be the set of all patches (commits) in a software repository, and $\mathcal{M}$ be the set of all patches of their potential origin. Note that $\mathcal{M}$ can be both, patches that were submitted to mailing lists (*i.e.*, mails containing patches) or commits in other branches of repositories (*e.g.*, patches on other branches of the repository[5]). The universe $\mathcal{U} = \mathcal{M} \cup \mathcal{C}$ forms the set of all patches.

In is most general form, the informal equivalence relation S: *patches are semantically similar* can be defined as $S \subseteq \mathcal{U} \times \mathcal{U}$. This covers all eventualities, including situations like *patch committed twice in the repository*, *patch went through several rounds of review before integration*, or *patch was not integrated*.

The foundation of the analysis is the algorithm sim that is able to quantify the similarity of two patches within the universe $\mathcal{U}$:

$$\text{sim}_{\text{tf,th,dlr,w}} \colon \mathcal{U} \times \mathcal{U} \to [0, 1] \tag{2.1}$$

The algorithm's sensitivity is controlled by four parameters. The principles of sim are explained in Section 2.3.1; the influence of the parameters is explained in Section 2.3.2. It measures the similarity of two patches where 0 denotes complete dissimilarity (*i.e.*, no commonalities) and 1 denotes complete equivalence on a textual level. Note that symmetry

$$\forall a, b \in \mathcal{U} : \text{sim}_{\text{tf,th,dlr,w}}(a, b) = \text{sim}_{\text{tf,th,dlr,w}}(b, a) \tag{2.2}$$

and reflexivity

$$\forall a \in \mathcal{U} : \text{sim}_{\text{tf,th,dlr,w}}(a, a) = 1 \tag{2.3}$$

hold.

Let $V = \mathcal{U}$ be the set of all vertices of the undirected graph $G = (V, E)$. Every edge in E connects two patches that exceed the *acceptance threshold* ta:

$$E = \{\{a, b\} \subseteq U | \text{sim}_{\text{tf,th,dlr,w}}(a, b) > \text{ta}\} \tag{2.4}$$

The connected components of G form subgraphs of similar patches that divide $\mathcal{U}$ into disjoint partitions. Those partitions induce equivalence classes

$$[x]_S = \{y \in V | x \rightsquigarrow_G y\} \tag{2.5}$$

[5] In Appendix A.1, I will use this property to quantify mainlining efforts of *out-of-tree* developments by comparing the content of different repositories.

where $\rightsquigarrow_G$ denotes reachability. The corresponding equivalence relation $\sim_S$ can be used to determine all equivalence classes by pairwise patch comparison in a process that iteratively merges equivalence classes where the similarity of two patches exceeds a certain threshold ta (*cf.* Fig. 2.2).

Ordinary pairwise comparison of n patches against each other requires $\mathcal{O}(n^2)$ comparison operations. As the necessary string operations are computationally intensive, I employ a coarse-grained pre-evaluation that serves as a filter. Strategies to mitigate combinatorial explosions will be presented in Section 2.3.3.

From another perspective, the partition of the equivalence relation S can also be seen as an unsupervised threshold-based flat clustering of $\mathcal{U}$ [SMR08]. In Section 2.4, I will use this fact to evaluate the accuracy of the approach with external evaluation methods for clusterings. With this, the problem of finding clusters of similar patches can be reduced to a function sim, which rates the similarity of two patches. In the following, I introduce sim, the function that scores the similarity of two patches, and its set of parameters that control the sensitivity of the function.

2.3.1 Rating Similarity of Two Patches

As mentioned before, in order to group patches into equivalence classes and find them in the base project, it is necessary to detect similar patches and commits. Generally, a patch consists of a unique identifier (*i.e.,* the commit hash in case of a commit in the repository, or Message-ID in case of a patch on a ML), a descriptive message that informally summarises the modifications, and so called *diffs* [MES13] that describe the actual changes of the code.

Existing work on detecting similar code fragments primarily targets on detecting code duplicates [DRD99] or on revealing code plagiarism. Possible approaches include language-dependent lexical analysis, code fingerprinting [SH09], or the comparison of abstract syntax trees [Jia+07]. However, all these approaches concentrate on the comparison of code fragments and not on the comparison of *similar diffs* or commits, as required in my case.

In contrast to the detection of code plagiarism or the detection of code duplicates, the content (on a textual level) between successive revisions of the same patch tends to stay very close. For this case, string or edit distances provide a straight forward and powerful language independent method for detecting similar code fragments.

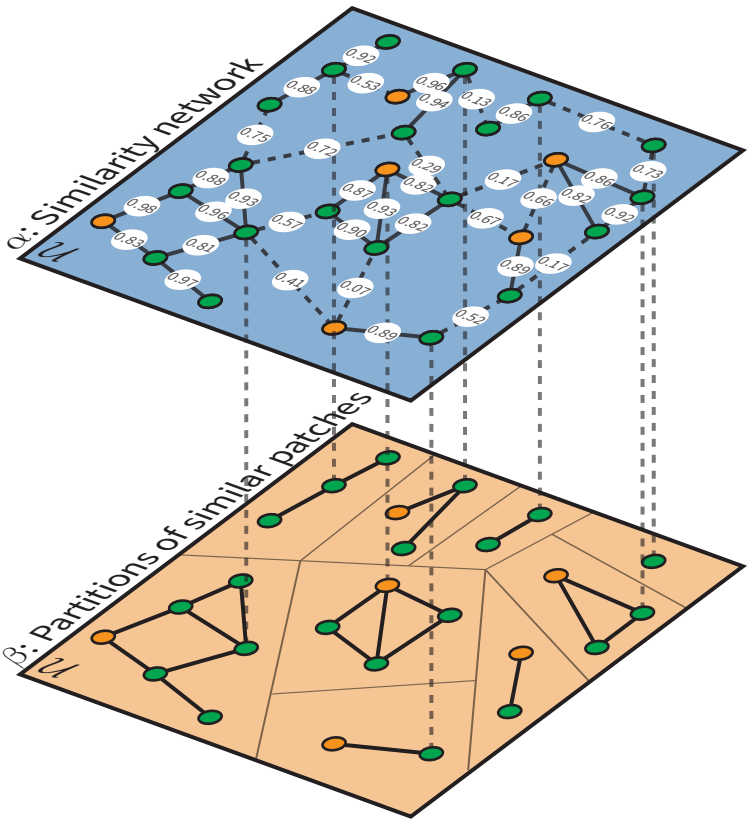

Figure 2.2.: α: sim determines the similarity (edge weights) of patches. Dashed edges remain below the threshold ta $= 0.80$. β: Connected components above the threshold form equivalence classes of similar patches. Green and orange vertices exemplarily denote patches on MLs ($\mathcal{M}$) and commits ($\mathcal{C}$) respectively.

The following properties can be observed in projects that use MLs:

- Commit messages of upstream patches tend to be more verbose, but still are similar to those on patch stacks.
  Rationale: Maintainers tend to amend or extend commit messages for better understandability.

- Variable and identifier names do not significantly change between different revisions of the same logical change.
  Rationale: If an author changes code of at a specific location, they will usually use the identifier names of the surrounding context. Even in case of major refactorings, at least the lines that have to be removed will stay the same.

- Range information (*i.e.*, the location of a hunk within a file) of similar hunks changes between different revisions of a patch.
  Rationale: The code base of active projects are in motion. Multiple authors may change the same locations in parallel. If an author sends a patch that modifies, for

example, a function within file, it is not unusual that another patch was integrated in the meanwhile and changed surrounding context of the patch, which displaces the patch. While range information within the file changes, it is still very likely that the original patch still addresses the same routine.

Until final integration or final rejection, patches evolve over time. While the commit message and the code may change, they still introduce the same logical change. As the commit message and diff may evolve independently, I calculate two independent scores that quantify the similarity of the two commit messages and the similarity of the two diffs ($r_{\text{msg}}, r_{\text{diff}} \in [0, 1]$). Again, 0 means no commonalities while 1 means equivalence on a textual level.

Similarity of commit messages Maintainers may amend or reword commit messages before they integrate the patch. They can also rearrange or reformat the patch to make it easier to understand, or to avoid ambiguities. Nevertheless, keywords that are used in those messages tend to remain the same. All tags (Ref. Section 2.2) that were added by maintainers, will be removed before comparing the commit messages, as they do not appear in the initial patch. The next step is to tokenise and sort all words in a commit message. The tokens are separated by whitespaces, and then pairwise compared against each other by using the Levenshtein string distance [Lev66]. The closest match for each token is selected for further processing. The arithmetic mean over all matches forms the score r_{msg}. I chose the Levenshtein string distance together with tokenisation, as it respects restructured messages as well as minor changes in wording, such as typo fixes.

Similarity of diffs Even if code changes or evolves over time, different versions of a patch still tend to affect the same code paths and files and use similar keywords or variable names. Diffs are compared in an iterative process. A single patch may modify several files. Hence, when comparing the diff component of two patches, I only consider to compare changes of files within *similar* filenames.[6] The threshold of the Levenshtein similarity for filenames is determined by the parameter tf, which must be exceeded if the diff of two files is considered for actual comparison.

A diff of a given file may consist of several *hunks*, which describe changes to a certain section within the file. Hunks are annotated with the line number within the file and a *hunk header* that describes the context of the change (*cf.* Listing 2.1). They display "the nearest unchanged line that precedes each hunk" [MES13]. The hunks of the two diffs are pairwise compared against each other, if two adjacent hunk headers exceed a certain

[6] I consider similar filenames, as filenames may change between submission and integration of a patch.

similarity threshold th. Hunks for which a mapping can not be established are ignored, as the hunk might have been added or removed in one of the patches. To compare those hunks, I disregard context lines as they might have changed in the meanwhile, compare insertions only against insertions, and deletions only against deletions. Therefore, I again tokenise deletions resp. insertions and use the Levenshtein string distance to compute a score for the hunk. The arithmetic mean of scores of all hunks provides the similarity score for the diff, r_{diff}.

The algorithm calculates a rating for the similarity of the commit message and a rating for the similarity of the diff. When comparing diffs, only similar hunks of commonly changes files are compared. Insertions and deletions are compared independently.

Algorithm 1 describes the evaluation of two patches. The algorithm calculates two ratings, a message rating $r_{\text{msg}} \in [0, 1]$ and a diff rating $r_{\text{diff}} \in [0, 1]$. r is the weighted arithmetic mean of r_{msg} and r_{diff}, weighted by a heuristic factor $w \in [0, 1]$. If the resulting rating exceeds the threshold ta, $r \geq$ ta, the two patches are classified similar. Given a patch, GETCOMMIT returns the corresponding message m and diff d. STRIPTAGS removes all *tags* (CC:, Signed-off-by:, Acked-by:, ...) as they are not relevant for comparing the content of commit messages. Given the diff of a patch, CHANGEDFILES returns all touched files of the diff. GETHUNKS returns all hunks of the diff of a file while HUNKBYHEADING searches for the closest hunk which heading matches x with a rating of at least th given a section heading x and the diff of a file. DIST takes either two strings or two lists of strings and returns a rating between 0 and 1, where 0 denotes no commonalities and 1 denotes absolute similarity.

2.3.2 Parameters

The extensive use of string metrics for measuring the similarity of different parts of a patch opens a wide spectrum for different thresholds of similarity. Additional parameters (tf, th, dlr, w, ta) investigate the structure of the patch and control the sensitivity of the comparison.

tf: filename threshold A file might have been renamed in the time window between the submission and acceptance of a patch. As mentioned above, the algorithm only considers the pairwise comparison of files with a similar filename. The filename threshold (tf $\in [0, 1]$) denotes a similarity threshold for filenames that must be exceeded if two files shall be considered for comparison.

Algorithm 1 Measure the similarity of two patches.

```
1:  function SIM(a, b, tf, th, dlr, w)
2:      if not PREEVAL(a, b) then
3:          return 0
4:
5:      (m_a, d_a) ←GETPATCH(a)
6:      (m_b, d_b) ←GETPATCH(b)
7:      d_min ← min(len(d_a), len(d_b))
8:      d_max ← max(len(d_a), len(d_b))
9:      if d_min/d_max < dlr then
10:         return 0
11:
12:     r_msg ←DIST(STRIPTAGS(m_a), STRIPTAGS(m_b))
13:     r_diff ← []
14:     for each file_a ←CHANGEDFILES(d_a) do
15:         for each file_b ←CHANGEDFILES(d_b) do
16:             if DIST(file_a, file_b) < tf then
17:                 continue
18:             hunks_a ←GETHUNKS(d_a, file_a)
19:             hunks_b ←GETHUNKS(d_b, file_b)
20:             r_f ← []
21:             for each hunk_a ← hunks_a do
22:                 for each hunk_b ← hunks_b do
23:                     if DIST(hunk_a, hunk_b) < th then
24:                         continue
25:                     r_f.append(DIST(hunk_a^+, hunk_b^+))
26:                     r_f.append(DIST(hunk_a^−, hunk_b^−))
27:             r_diff.append(MEAN(r_f))
28:     r_diff ←MEAN(r_diff)
29:     return w · r_msg + (1 − w) · r_diff
```

th: hunk header threshold Within a file, the location of a hunk might have moved in the time window between submission and acceptance of a patch. Either the author moved the location of the hunk, the upstream location changed or a maintainer moved the code. Hunk headings try to ease the readability of the patch. Regular expressions backward-search for anchor lines that will appear in the hunk heading, such as, for example, function names. The hunk heading threshold (th $\in [0, 1]$) denotes the similarity of two hunk headings of hunks that must be exceeded if two hunks shall be considered for comparison.

dlr: diff-length ratio Similar patches only slightly differ in size. It is unlikely that a patch that a simple bug fix that modifies one single line of code is related to a patch that

introduces a new feature with hundreds of lines of code. Because of this, patches are considered dissimilar if the diff-length ratio (dlr $\in [0,1]$), which is the fraction of the number of changed lines of the smaller patch by the number of lines patched by the bigger patch, is not exceeded.

w: commit-diff weight Different maintainers in different projects often have different strategies of how they handle patches. In some projects, maintainers heavily modify commit messages (see Listing 2.1), while in other projects maintainers attach less importance on the commit message and leave it as it is. However, instead of asking for a further revision of a patch, they might modify the code on their own. Since I calculate two independent scores for the commit message and for the diff, `PaStA` can be adjusted to the peculiarities of a project. A heuristic factor $w \in [0,1]$ weights the relative importance of r_{diff} to r_{msg} and denotes the overall similarity:

$$\text{sim}_{\text{tf,th,dlr,w}}(a,b) = \begin{cases} 0 \text{ if } \min(a,b)/\max(a,b) < \text{dlr} \\ w \cdot r_{\text{msg}}(a,b) + (1-w) \cdot r_{\text{diff}}(a,b) \text{ else} \end{cases} \tag{2.6}$$

ta: auto accept threshold The auto accept threshold ta denotes the required score for patches to be considered similar. Patches are only considered similar, if

$$\text{sim}_{\text{tf,th,dlr,w}}(a,b) \geq \text{ta} \tag{2.7}$$

Section 2.4 investigates the significance of the chosen set of parameters.

2.3.3 Reduction of problem space and clustering patches

Scalability is a concern of my approach. Consider a huge project like the Linux kernel. The mailing list archive of the major mailing list of the Linux kernel, the Linux Kernel Mailing List (LKML) (`linux-kernel@vger.kernel.org`), contains $\approx 2.8 \times 10^6$ mails where from 2002-01 – 2018-07, and $|\mathcal{M}| \approx 8.5 \times 10^5$ mails contain patches. The corresponding range in the repository (≈v2.6.12–v4.18) contains $|\mathcal{C}| \approx 7.6 \times 10^5$ commits. This leads to a patch universe of $|\mathcal{U}| \approx 1.6 \times 10^6$ entries, with a total number of $\binom{|\mathcal{U}|}{2} \approx 1.3 \times 10^{12}$ pairwise comparisons. A *significant* reduction of this number is required for practical application of the approach.

Hence, I introduced a preevaluation phase (PreEval in Algorithm 1) that drastically reduces the impractical number of pairwise comparisons.

Time Window First and foremost, I only consider pairs of patches for comparison within a certain time window. Two patches will only be considered for similarity rating, if they were submitted within a time window of one year. In the evaluation, I show that this covers the vast majority (*i.e.*, 99.5%) of all patches.

Pre-filter Heuristics Secondly, two patches can not be similar if they do not modify at least one common file. This fact can be used for further optimisation: only select pairs of patches that modify at least one *similar* file. In addition to that, the diff-length-ratio is also a computationally inexpensive measure to exclude patches of disproportional size from further cost-intensive evaluation.

In addition to that, I first determine clusters of similar patches for emails ($\mathcal{M} \times \mathcal{M}$). At the beginning of the evaluation, every email is assigned to its own single-element cluster. I successively merge clusters in an iterative process by comparing representatives of clusters against each other. A representative of a cluster is the patch with the youngest submission date. This patch is chosen as representative, as it will have the closest similarity with further revisions, or with the commit in the repository, if it was integrated.

After the creation of the clusters for emails, representatives of those clusters are compared against the commits in the repository ($\mathcal{M} \times \mathcal{C}$).

2.3.4 Working with Mailing List Data

The first step of the process is the acquisition of mailing list data. Naturally, this can be done by subscribing to mailing lists. However, this will only capture incoming emails since subscription and misses historic mailing list data.

The Linux Kernel community (as well as many other communities) officially provides mailing list archives.[7] As archiving method, they use the public-inbox storage format.[8] The public-inbox approach stores mails in git repositories and provides a convenient data exchange format as standard tooling can be used to search for or to extract mails from the repository. Different mailing lists are stored in separate repositories.

However, reliable analyses with resilient results require full coverage of all mailing list data for the time frame of interest. While official resources reach back to early days of Linux, archives do not cover all mailing lists. At the time of writing this thesis, only a

[7] For the Linux kernel, refer to `http://lore.kernel.org`.
[8] See `https://public-inbox.org/README.html`.

subset of ≈100 lists of over 200 referenced lists of the project are provided by the Linux Foundation.

Therefore, I subscribed to all >200 publicly available lists and collect mailing list data since May 2019. Our group's archives receive regular updates and are publicly available.[9]

The second step is to filter relevant emails containing patches and to convert them to a unified format that can be used for further processing [Bir+06]. There are plenty of methods how an user may send a patch, or how the mail user agent (MUA) may treat (and mistreat) the message. PaStA's parser is able to identify the most commonly used methods. It respects patches in attachments, multiple patches in attachments, inlined patches, and is able to repair several erroneous encodings that frequently occur on mailing lists.

2.4 Evaluation

The results of a heuristic method depend on the chosen set of parameters. In the following, I identify significant predictors from the available set of tuneables, and further evaluate the algorithms accuracy for the optimal choice.

To establish a ground truth, I chose a one-month time window (May 2012, a typical month of Linux kernel development without any exceptional events) of the high-volume LKML. Assisted by tool support, I extracted mails with patches and manually compared them against a three month time window in the repository in an elaborate and time-consuming task using interactive support of PaStA. The creation of a sound ground truth requires domain-specific knowledge to judge the relationship of patches, which is available by my active involvement in the respective communities.

The same data was reanalysed with PaStA, under permutation of parameters in a reasonable range, as shown in Table 2.1. Prior to choosing the exact parameter ranges, I performed a coarse-grained analysis to roughly estimate the influence of parameters. The chosen domains result in 803,682 different analysis runs.

In the observed time frame, the list received 16,431 emails. Among these, 5,470 were recognised as patches (33.3%). Assisted by PaStA (and supported by an interactive interface that ensures a swift workflow), the patches were compared against all commits between Linux kernel versions v3.3 and v3.6 (34,732 commits). Those commits are

[9] Available at https://github.com/orgs/linux-mailinglist-archives/

within the time window 2012-03-18 – 2012-09-30 (see Section 2.4.1 for a justification of this choice).

The ground truth consists of 3,852 clusters of patches, where 2,525 clusters are linked to at least one commit in the repository. 990 clusters contain more than one email (*e.g.*, multiple revisions of a patch), 394 clusters more than two emails, and 154 more than three emails. 1,712 clusters contain exactly one email, which means the changes were immediately accepted after their initial submission without further refinements.

The ground truth is then compared against all clusters from the permutation of parameters as shown in Table 2.1. In other words, the ground truth is compared against the 803,682 results of `PaStA`.

2.4.1 External Evaluation

External evaluation methods quantify the similarity of two clusterings [SMR08]. While there are many standard evaluation methods available, the correct choice relies on the structure of the clustering [Ami+09]. In contrast to typical clustering problems where a large number of elements (*e.g.*, *documents*) is distributed to a small number of clusters (*e.g.*, *document types*), clustering similar patches entails a large number of clusters (similar patches) with only few elements (patch revisions and commits in repositories) per cluster. This inherently implies a considerable number of "true negatives" (TN), since two randomly chosen elements are assigned to two distinct clusters with high probability. For a sufficiently large number of clusters, any random clustering will exhibit a high number of TNs.

Several external evaluation methods were tested for their suitability: mutual information score [SMR08], purity [SMR08], V-measure [RH07], and the Fowlkes-Mallows index [FM83]. Purity is not suitable for the problem because it intrinsically produces good results for large cluster count. A high number of clusters always implies good

Table 2.1.: Set of parameters used for evaluation.

Parameter	Description	Interval	Step
tf	threshold filename	[0.60, 1.00]	0.05
th	threshold heading	[0.15, 1.00]	0.05
dlr	diff-length ratio	[0.00, 1.00]	0.10
w	message-diff weight	[0.00, 1.00]	0.10
ta	threshold auto-accept	[0.60, 1.00]	0.01

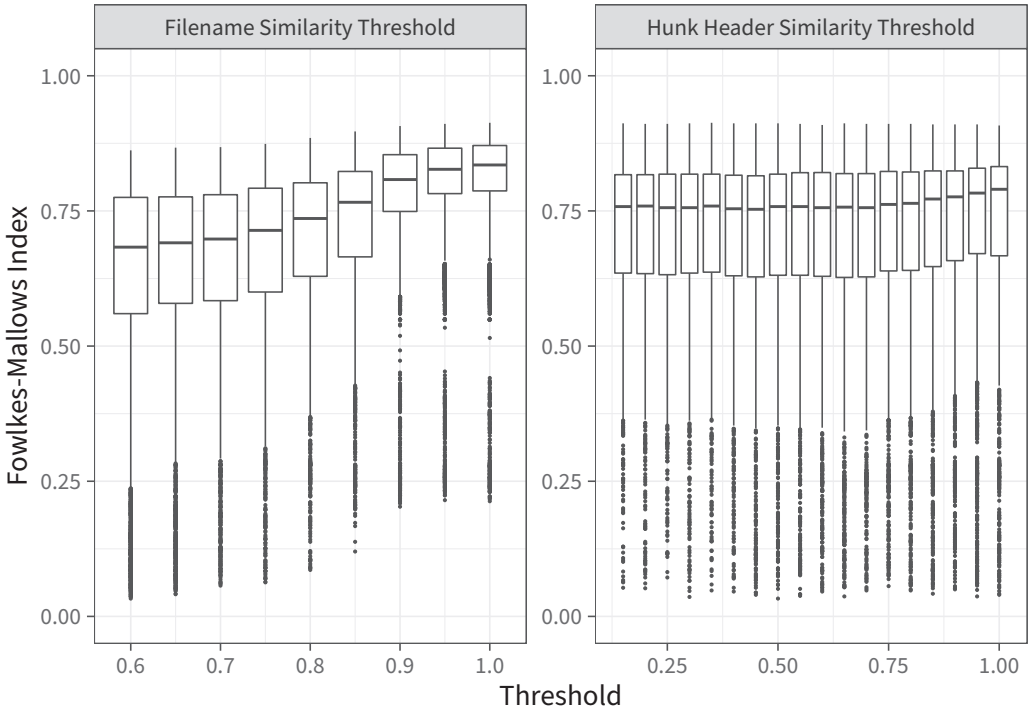

Figure 2.3.: Boxplot of irrelevant parameters: filename and hunk header threshold have no substantial influence.

purity [SMR08]. The V-measure is the harmonic mean of two other measures, completeness and homogeneity, and also produces good results when many clusters are present. I consequently choose the Fowlkes-Mallows index, since it is not sensitive to the number of TN, and shows robust results for clusterings with a high number of clusters. The Fowlkes-Mallows FM index is defined as

$$\mathrm{FM} = \sqrt{\frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FP}} \cdot \frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FN}}} \tag{2.8}$$

where TP denotes the number of true positives, and FP and FN provide the number of false positives and negatives, respectively.

A way to confirm the validity and suitability of an index is to compare it against an unrelated clustering [SMR08]. Therefore, I compared the ground truth against a random clustering, while maintaining the structure of the clustering, that is, the number of clusters and the number of elements per cluster. Compared against the ground truth, this reveals a bad Fowlkes-Mallows index of 0.05. Since the results for the analyses lie within the interval $[0.231, 0.911]$, this indicates a high validity of the chosen index.

To identify parameters with a relevant influence on the result, I compute the Fowlkes-Mallows index for each of the $803{,}682$ clusterings against the ground truth. This provides

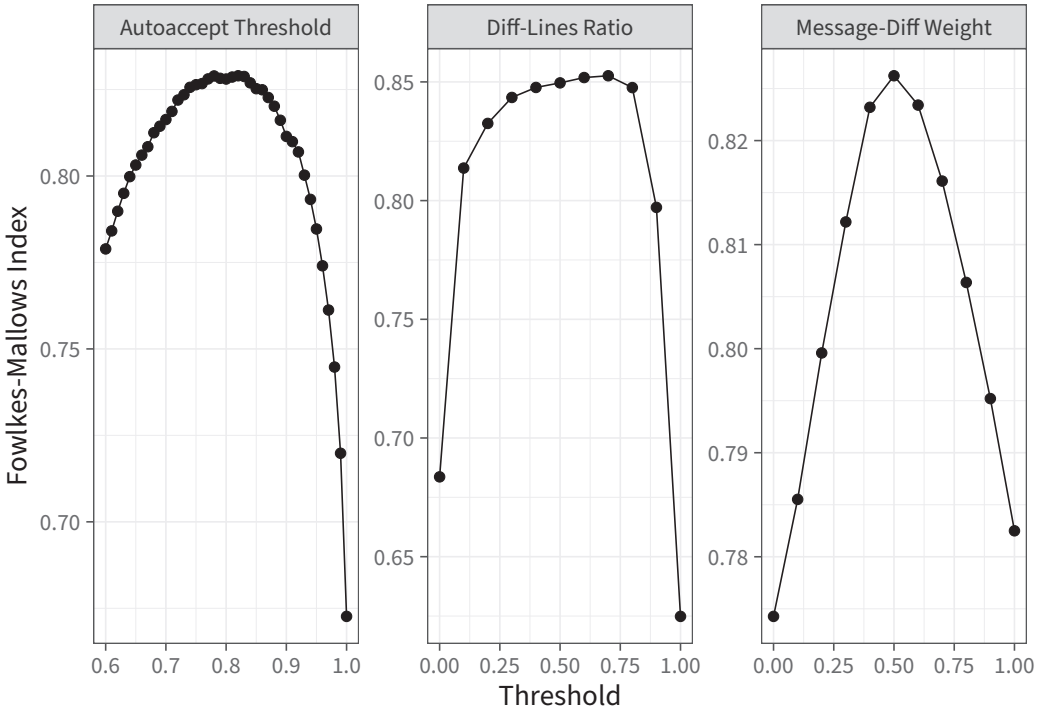

Figure 2.4.: Illustration of the influence of auto-accept threshold, diff-length ratio and the message-diff weight (connecting lines in all figures are used to guide the eye).

a similarity score for clusterings for each combination of parameters. To draw conclusions on the significance of a parameter, I selectively observe the distribution of the Fowlkes-Mallows index for each parameter. Fig. 2.3 illustrates the Fowlkes-Mallows index for different values of the filename threshold resp. the hunk header threshold. It can be seen that the settings for tf and th have little influence on the results. Instead, best results are achieved for the boundary setting 1 in both cases (I analyse the reason for the behaviour Section 2.5). For the further analysis, I only regard the subset of results with $\mathrm{tf} = 1$ and $\mathrm{th} = 1$ due to their lack of significance. This requires to consider $2{,}662$ clusterings.

Fig. 2.4 shows the plot of the mean of the Fowlkes-Mallows index for auto-accept threshold, diff-length ratio and message-diff weight. Having the filename and hunk header threshold set to 1, the approach performs best with a auto-accept threshold of 0.82, a diff-length ratio of 0.4 and a message-diff weight of 0.3. With this combination, it achieves a Fowlkes-Mallows index of 0.911 on the selected time window.

To confirm the universal validity of those parameters for the whole project, I cross checked the parameters with another mailing list: the linux-commits-tip mailing list. Every patch that is committed to the Linux tip repository is automatically sent to the linux-commits-tip mailing list [JAG13] by the tip-bot. In contrast to standard emails, they

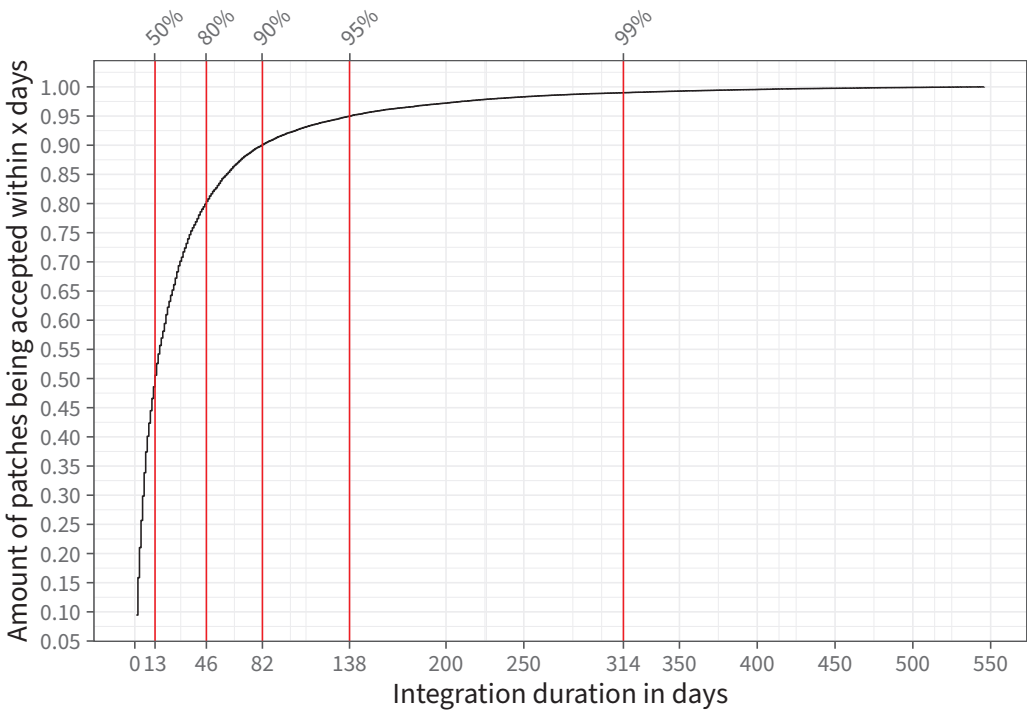

Figure 2.5.: Empirical distribution function of the integration duration of patches on the LKML.

contain the commit hash in the corresponding repository in their header. This allows for simple cross-validation of the best parameter set. The list can be used to prove the general functioning of the approach, as the analysis should lead to an exact match of all patches.

Using a sample of $1{,}047$ emails from linux-tip-commits ML compared to the linux-tip-commits repository results in a Fowlkes-Mallows index of 0.988. Some minor mismatches are caused by very close, but still dissimilar patches that are erroneously considered similar, and induced by technical corner cases where the diff for a patch being sent to the mailing list produces different output as the diff in the repository (*e.g.*, mode-changes of files or moved files). In sum, there were $1{,}086$ TPs, 18 FPs, and 9 FNs. Note that there are more TPs than actual emails, because some clusters correctly contain more than one email or more than one commit; a correct cluster with n elements contains $\binom{n}{2}$ TPs. Once more, these numbers underline the high accuracy of `PaStA`.

2.4.2 Example: Duration of patch integration

Comparing patches is a computationally intensive task. The number of comparisons can be reduced if potential comparison candidates are restricted to patches within a certain

time window, as less patches are considered for the eventual cost-intensive comparison. PaStA already provides a set of qualitative analyses, such as the integration duration of a patch: *Given a patch in the repository, how long did integration take, from first appearance on the mailing list until final integration to the repository?*

From an analytical point of view, this question supports to reduce the computationally intensive task of comparing patches. From a practical point of view, this question also gives interesting qualitative insights on how long it typically takes for integrating a patch into a project.

To determine the size of this window, I re-run the analysis on the whole LKML and the whole repository with the determined optimal set of parameters. I define the time interval between the date of the latest revision of a patch (*i.e.*, email submission date) and the date of integration in the repository (*i.e.*, the commit date) as integration duration.

Fig. 2.5 shows the empirical distribution function of the integration duration of all patches of the 99.9% quantile of all patches. Interestingly, within the outliers beyond that quantile I found patches that took indeed five years for integration. 99% of all patches were integrated within one year, 80% of all patches within 46 days, 50% of all patches within two weeks.

2.4.3 Comparison to Other Approaches

In [Jia+14], Jiang and colleagues also present a method for mapping patches on mailing lists to repositories. Their Plus-Minus-based approach assigns each tuple of changed line and filename to a set of ids, where the id can either be a message ID or a commit hash. They then search for patches that contain sufficient identical changes. A threshold between [0, 1] determines the fraction of the number of identical changes that needs to be exceeded if patches are considered similar.

I used their original implementation to evaluate it against the time window of the aforementioned ground truth, and vary their threshold setting in the range [0, 1]. Fig. 2.6 shows the results of the analysis. The threshold has no significant impact on the accuracy within the range ≈[0.25, 0.75]. The best Fowlkes-Mallows index of 0.743 that I could reach with their method is observed at threshold 0.26.

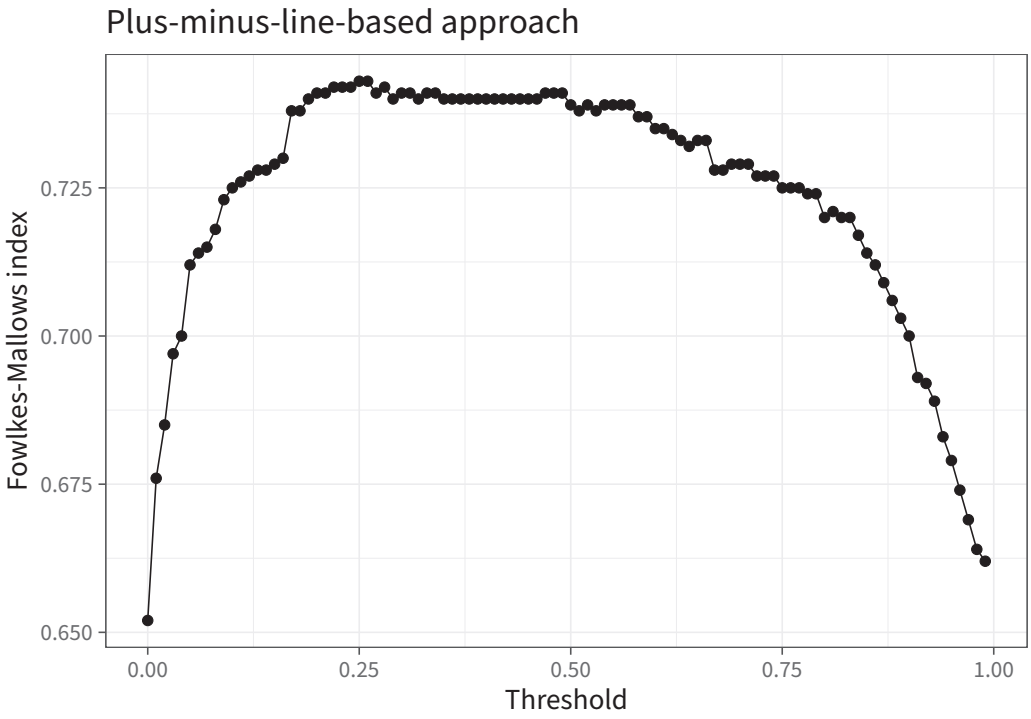

Figure 2.6.: Evaluation of the Plus-Minus-based approach: highest FM index at 0.26, while the threshold only has little influence between [0.25, 0.74].

2.5 Discussion

I previously showed the high accuracy of PaStA's method, and quantitatively compared it with other existing methods. I will now turn the attention to interpreting the meaning of the optimal set of tuneable parameters, further discuss other methods, and examine the performance (and, thus, practical applicability) of the approach.

2.5.1 The Algorithm

In Section 2.4 I found that both, filename and hunk header threshold, produce best results for the boundary value 1.00. A filename threshold of 1.00 implies that patches on the list will not be associated with a commit in the repository if affected files were renamed between submission and integration of the patch, and the hunk header threshold of 1.00 disregards relocations of a hunk within a file. The rationale for these extreme settings is that both, file moves and relocations within a file, do not occur frequently in real-world development. It is unlikely that a patch hits this exact window. While a lower threshold improves recall, it disproportionally decreases precision since more patches are erroneously considered similar when relocations occur.

In contrast to filename and hunk header threshold, other parameters significantly influence the results: auto accept threshold, diff-length ratio and message diff weight. As expected, too strong or too weak thresholds lead to over- and underfitting. The diff-length ratio of 0.4 is reasonable because it allows, for instance, an initial two-line patch to expand into five-line patch in a future revision, but filters for strongly imbalanced sizes of patches. It is, for instance, unlikely that a one-line patch will evolve into a 20-line patch in a future revision. A message-diff weight of 0.3 underlines the importance to consider both, commit message and diff, with a slight bias towards the code. It also stresses that involving actual code for analyses is vital for such analyses.

2.5.2 Plus-Minus-based approach

While not explicitly mentioned in their paper, the authors of [Jia+14] chose a threshold of 0.5 for their algorithm, based on their experience and intuition[Ada18]. The evaluation of the Plus-Minus-based approach shows evidence that this threshold is indeed within a range where the algorithm performs best.

The authors determine the accuracy of their approach based on the F-Score, defined as $F = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$. It requires knowledge of precision *and* recall. While calculating precision is straightforward (*i.e.*, counting the number of true and false positives), a solid ground truth is required to determine the exact recall of an algorithm, as the recall requires to know the number of false negatives. They argue that it is hard to determine such a ground truth (a statement that I fully agree with), and therefore employ the concept of "relative recall". The relative recall incorporates results of the checksum–based technique and the clone-detection–based technique. The accuracy of these approaches is not known and therefore relative recall only forms an approximation with unknown quality. An evaluation with a ground truth leads to more reliable results.

2.5.3 Performance

Performance is an important factor for real world practicability. In particular, a well-performing implementation is required for the evaluation of the optimum parameter set, as it requires to run several analyses. Therefore, `PaStA` massively parallelises steps of the analysis.

The full analysis of the Linux kernel (v2.6.12 – v4.18 against the whole ML) with my method requires 13 hours on a machine equipped with two Intel® Xeon® E5-2650 processors (20 cores / 40 threads) using the optimal thresholds derived in Section 2.4. This

includes run-once preparation steps like converting mailing list data to a suitable format, parsing mailing lists for patches or creating intermediate caches.

It was not possible to run the full analysis of the Linux kernel with the plus-minus-line–based approach, because of limitations of their implementation.

Nonetheless, the plus-minus-line–based approach is considerably more performant than my approach. For the one-month test set, the approach takes 80 seconds on the same machine as mentioned before, and only consumes one single CPU core. `PaStA` takes between two and eight minutes to analyse the same set, depending on selected thresholds. The comparison of textual equivalence used by the plus-minus-line–based technique is less computation-intensive than the use of Levenshtein string distances.

Yet, `PaStA` is applicable for real world use cases and its best Fowlkes-Mallows index is 22% higher than the best score achieved by the plus-minus-line–based approach.

2.6 Threats to Validity

2.6.1 Internal Validity

Other than a perfect gold standard, a manually created ground truth underlies some uncertainties. The creator may be biased or misjudge decisions, and there is always a certain degree of subjectivity. The creation of the ground truth (judging similarity of patches) was carefully done by an experienced developer with domain-specific knowledge and a track record of active participation in several open source communities, including the Linux kernel, and I am confident that the ground truth contains negligible faults.

2.6.2 External Validity

I focus on the Linux kernel for the evaluation, which has strict submission guidelines, such as requiring detailed commit messages. Patches must be structured in a fine-grained fashion and must only introduce one small change. Other projects established different strategies, such as less-verbose commit messages or larger patches.

Because of this fact, the set of parameters that I found in the evaluation are therefore thresholds that *suit* Linux, but are not necessarily applicable to other projects. As a

consequence, this demands to repeat the evaluation, when analysing other projects that the Linux kernel, in order to determine its proper set of thresholds.

However, numerous other low-level systems that are object of my analyses adopted the submission guidelines of the Linux kernel that are known as best practises in the communities. The same set of parameters lead to high accuracy in other such projects (*e.g.*, QEMU, BusyBox, U-Boot, ...).

2.6.3 Construct Validity

Working with mailing lists requires handling noisy data. Bird et al. [Bir+06] found that 1.3% of the Apache HTTP Server Developer mailing list contains malformed headers.

`PaStA` implements appropriate filters, and consequently uses a custom best-effort parser adapted to handle these difficulties. Since authors may submit their patches in many ways, finding all patches cannot be guaranteed, though. Based on the knowledge in the ground truth, the amount of patches that are not captured is insignificant. Additionally, the revision control system git that is widely used for Linux kernel development provides tool support to prevents common mistakes in email-based patch flows, which reduces the number of unparseable emails. Following op. cit., I deem this threat minor.

2.7 Related Work

2.7.1 Reconstruction of Development Processes

A patch consists of an informal commit message that describes the changes of the patch in natural language, and annotations of the modifications to files of a project. First and foremost, patches modify source code, but also documentation, build system, tools and any other artefacts of a project. A single patch may modify several files. Within the context of a file, hunks are segments that describe changes to a certain area within a file. Listing 2.1 illustrates the typical structure of patches on the ML (a, b) and in the repository (c). The issue is to find similar patches to track patch evolution.

Jiang, Adams and German [JAG13] present a coarse-grained checksum-based technique for mapping emails that contain patches to commits. After trimming whitespaces they calculate MD5 hashes over chunks of the patch. Two patches are considered similar if they have at least one checksum in common (*i.e.*, share one equivalent hunk).

In another work [Jia+14], the authors refine their technique and present further approaches: A plus-minus-line-based technique and a clone-detection-based technique. The plus-minus-line–based technique weights the fraction of equivalent lines of two patches. This includes insertions (+) and deletions (-). The clone-detection–based technique incorporates CCFinderX [BTH12], a code-clone detector. They evaluate their three techniques, and conclude that the plus-minus-line–based technique is performing best. This evaluation is based on the F-Score that depends on the precision and recall of the actual algorithm. In contrast to measuring the precision, the F-Score requires a ground truth for determining the recall. As a ground truth is hard to obtain, authors use the concept of *relative recall* that provides a qualitative approximation.

2.7.2 Distinction from Code Clone Detection

Finding similar patches needs to be distinguished from detecting similar code. Code Clone Detection (CCD) is a well-researched topic mainly driven by revealing code plagiarism [CJ12] or redundancy reduction [Bax+98]. The underlying problems of detecting similar patches and detecting similar code are related, but differ in one decisive property: code clone detection analyses a certain *snapshot* of the code, while detecting similar patches requires analysing a *diff*, which comprises only fragments detached from the code base. Additionally, a patch also contains an informal commit message that is not considered by CCD.

Many CCD techniques use language-dependent lexical analysis and analyse similarities of abstract syntax trees [Jia+07; Bax+98]. Since patches only provide differences between syntactically incomplete fragments of code, and may also modify non-code artefacts, CCD techniques are typically inapplicable when working with patches.

Another approach uses locality sensitive hash functions for quantifying code similarity [Jia+07; Sæb+09]. Such hash functions produce similar output for similar input. Arwin et al. proposed a *language independent* approach [AT06] that analyses intermediate code produced by the compiler. This is not applicable to my problem since the aforementioned analysis of documentation, scripts, build-system artefacts etc. needs to be independent of any language restrictions.

Bacchelli et al. [Bac+09; BLD11; BLR10] link emails to source code artefacts in a repository. In contrast to `PaStA`, they focus on discussions and conversations instead of analysing mails with patches. Naturally, informal conversations have a different structures than patches. However, my approach of linking patches on mailing lists to repositories allows for transitively linking follow-up discussions of a patch, since the Message-ID of the initial patch remains in the “reference header” of responses.

2.8 Summary and Conclusion

I presented a method that is able to reliably link emails with patches to commits in repositories with high accuracy. Additionally, I formalised the mathematical background of the problem and identified it as a clustering problem. Based on this, an elaborate evaluation built upon a solid ground truth quantifies the high accuracy of my approach. The ground truth and the `PaStA` framework can be used to evaluate the accuracy of other approaches, and the fully published framework allows for independent (industrial) evaluation required in certification efforts.

The evaluation verified that the presented approach performs better than existing work. For Linux and the LKML, `PaStA` achieves a 22% larger Fowlkes-Mallows index of 0.911 than the best score achieved by the (previously best) plus-minus-line–based approach.

In the first chapter of this part, I presented the methodology of `PaStA`, a methodology to reconstruct development processes of various OSS projects. These data build the fundament for various further qualitative analyses on properties and characteristics of OSS projects.

The next chapter focuses on the practical applications of the approach. I will focus on the assessment non-formal OSS development processes. My methodology provides the basis for such analyses, as it systematically makes the history of the process accessible. Its accuracy makes it suitable for further qualitative software analyses.

3 Analysis

> *Über den Gemeinspruch: Das mag in der Theorie richtig sein, taugt aber nicht für die Praxis*
>
> — **Immanuel Kant**
> 1793

This chapter shares material with the OpenSym '16 paper "Observing Custom Software Modifications: A Quantitative Approach of Tracking the Evolution of Patch Stacks" [RLM16] and the CCSW '20 paper "The Sound of Silence: Mining Security Vulnerabilities from Secret Integration Channels in Open-Source Projects" [Ram+20].

3.1 Structure

In the previous chapter, I presented the methodology of `PaStA`, a methodology to reconstruct the pre-integration history of OSS projects. In this chapter, I will use the approach and apply it to real-world projects, in order to provide evidence on the (dys-)functioning and performance of development processes. Analysis results can be used to support safety-certification efforts, as they provide deep insights in the development process.

Without the loss of generality I will conduct analyses on the Linux kernel. I chose the Linux kernel as test object for *reference analysis* for three reasons. First and foremost, Linux is a building block of the architecture presented in Section 4.2. As mentioned in Section 1.2.1, there is an increasing interest on Linux for being used in safety-critical environments, which makes safety certifications of Linux mandatory. The following analysis can be used as a model for such efforts. Secondly, the Linux kernel is one of the world's largest software development undertakings that uses more than 200 different mailing lists that receive approximately one mail every 20 seconds. Conducting the analyses on the Linux kernel stresses the scalability of the approach, and underlines the suitability for being used in real-world scenarios. Finally, with slight variations, the Linux Kernel Development Process (LKDP) is adapted by various OSS communities, such

as GCC, QEMU, U-Boot, LLVM, BusyBox, and many others. This allows for transferring the approach to other projects.

Note that the sole analysis of development processes is not the enabling factor for safety-certification, it is *one* component in a complex superior certification endeavour. For the particular certification of a product, specific questions on the development process need to be questioned. Those case-specific questions have to be carefully addressed. This chapter shall exhibit the opportunities of quantitative software engineering techniques, and underline their ability to provide profound answers on issues on processual characteristics and dynamics in OSS projects. Given a formal definition of an issue, I will show that it is possible to derive quantitative performance indicators that assist to provide answers on a formal basis. Of course, concrete issues need to be defined by assessors.

In the following analyses, the focus is on adherence of development processes. First and foremost, I will give detailed overview of the LKDP in Section 3.2. I will then ask informal questions on the adherence of the development process, and show how quantitative indicators can be derived to answer these questions. In particular, I will present an analysis that measures the adherence to development processes. As a consequence, this analysis also uncovers violations of the development process, which includes unexpected aspects of development processes. With my approach, it is possible to early uncover violations of the development process, which includes *intentional* violations of regular processes, such as processual exception for handling fixes for security vulnerabilities. This endangers responsible disclosure processes that are implemented by many OSS projects.

Extraction of Development Characteristics In Section 3.3, I will show how the methodology can be used to extract key characteristics of the development process. These data can be used to quantify the adherence to formally defined processes. I investigate dynamics on mailing lists, and define and quantify two performance indicators in a time series: the amount of patches that have been ignored on Linux kernel mailing lists, and how the dynamics evolved over almost a decade. Secondly, I investigate the adherence to development processes, by measuring the ratio of correctly integrated patches.

Violations of Development Processes Public development processes are a key characteristic of open source projects. In Section 3.4, I systematically mine for violations of the otherwise regular development process. Contrary to regular development activities, fixes for security vulnerabilities, for example, are usually discussed privately among a small group of trusted maintainers, and integrated without prior public involvement

This is supposed to prevent early disclosure, and cope with embargo and non-disclosure agreement (NDA) rules. While regular development activities leave publicly available traces, fixes for vulnerabilities that bypass the standard process do not.

Based on the methodology presented in Chapter 2, I present a data-mining based approach to detect code fragments that arise from such infringements of the standard process. By systematically mapping public development artefacts to source code repositories, I can exclude regular process activities, and infer irregularities that stem from non-public integration channels. For the Linux kernel, the most crucial component of many systems, I apply my method to a period of seven months before the release of Linux 5.4. I find 29 commits that address 12 vulnerabilities. For these vulnerabilities, my approach provides a temporal advantage of 2 to 179 days to design exploits before public disclosure takes place, and fixes are rolled out.

Established responsible disclosure approaches in open development processes are supposed to limit premature visibility of security vulnerabilities. However, my approach shows that, instead, they open *additional* possibilities to uncover such changes that thwart the very premise. I conclude by discussing implications and partial countermeasures.

In the following, I will use the term *PaStA* interchangeably for both, the methodology of reconstructing the development process, and the extension of the approach, that is, the quantification of characteristics of a development process.

3.2 Linux Kernel Development Process

This section gives a brief overview of the LKDP. One peculiarity of the LKDP is the large number of contributors (thousands per year) and participants, which lead to the well-known hypothesised connection given above between the decreasing difficulty of detecting bugs with an increasing number of reviewers. Since I abuse the principle in Section 3.4 to detect patches that have seemingly *not* receive sufficient public attention, it is pertinent to recapture key characteristics of the development process that are relevant for my approach.

3.2.1 Core Characteristics

Development of the Linux kernel proceeds in two-phase cycles: New code and features are merged during a two-week long *merge window*, which is followed by a two-month

long *stabilisation window* [Lin20]. This leads to development cycles of approximately 2.5 months between two major releases. More than 10,000 patches are integrated in each cycle into Linus Torvalds' (the project owner's) git tree, which is commonly called *Linux mainline*. Before code changes (*patches*) are integrated into mainline, they must have been discussed on a public mailing list [Kro16]. This is demanded by the submission guidelines of Linux, and is intended to ascertain good code quality [MW09]. Submission guidelines are part of the official LKDP.[1]

Similar to a commit in a repository, an email encapsulates a patch that contains a commit message, an informal description of the changes, and a *diff* that specifies insertions and deletions of code—relative to a specific code base. Typically, larger logical changes are split into multiple small patches. This gives a *patch series* whose elements are tied together by a *cover letter*. Cover letters give an informal, higher-level overview of the series. Together with the proper patches, it is sent as a mail thread to maintainers and the corresponding list(s) of the affected subsystem(s) of the project.

Everyone can join the discussion of patches as lists are usually unmoderated. Maintainers who receive the patch and who are responsible[2] for the area or subsystem that the patch addresses eventually

(a) refuse the patch,

(b) ask for further refinement of the patch,

(c) pick up the patch and commit it to their maintainer tree.

Maintainer trees are staging points before code changes are finally integrated mainline. It is not unusual that **(b)** is repeated over several iterations until the patch series is deemed acceptable for merging.

Because of the project's size and the massive number of emails and patches, the Linux kernel currently utilises over 200 different mailing lists that are logically partitioned by topic or subsystem. On average, an email is received by one of those lists every 20 seconds.

[1] Ref. `https://www.kernel.org/doc/html/latest/process/`
[2] I will later clarify how *areas of responsibility* are defined and determined

3.2.2 Organigram and Areas of Responsibility

To get a patch integrated into mainline Linux, the developer that writes a patch for Linux needs to determine the recipients of the patch. According to patch submission guidelines of the Linux kernel,[3] the patch should at least be sent to one mailing list:

> You should also normally choose at least one mailing list to receive a copy of your patch set. linux-kernel@vger.kernel.org functions as a list of last resort, but the volume on that list has caused a number of developers to tune it out. Look in the MAINTAINERS file for a subsystem-specific list; your patch will probably get more attention there.

The abovementioned MAINTAINERS file allocates areas of responsibility (*i.e.*, *sections* of responsibility). Such a section maps portions of source code to maintainers and mailing lists. Listing 3.1 exemplarily shows the entry of two sections in MAINTAINERS, APPLETALK NETWORK LAYER and NETWORKING DRIVERS.

If a patch modifies any file inside `drivers/net`, the patch will be assigned to the NETWORKING DRIVERS subsystem. Note that a patch can affect multiple sections as once, as it can touch multiple areas of responsibility at once. Furthermore, sections can overlap: The file drivers/net/appletalk/ltpc.c will be assigned to both sections, APPLETALK and NETWORKING. In sum, the MAINTAINERS of the Linux Kernel, as of v5.10, consists of 2,236 different sections.

The tool `get_maintainers.pl` supports developers to find recipients for their patch: `get_mainainters.pl` automatically assigns a patch to one or more of those sections and proposes appropriate recipients of the patch. Note that the contents of MAINTAINERS, sections and areas of responsibility evolves and changes over time: Sections are removed or added, and maintainers change their area of responsibility. Additionally, sections in MAINTAINERS contain a status that indicates whether the area is being actively maintained or not.

```
1  APPLETALK NETWORK LAYER
2  L:      netdev@vger.kernel.org
3  S:      Odd fixes
4  F:      drivers/net/appletalk/
5  F:      include/linux/atalk.h
6  [...]
7
8
9
10 [...]
```

[3] Ref. `https://www.kernel.org/doc/html/v5.10/process/submitting-patches.html`.

```
NETWORKING DRIVERS
M:      "David S. Miller" <davem@davemloft.net>
M:      Jakub Kicinski <kuba@kernel.org>
L:      netdev@vger.kernel.org
S:      Maintained
W:      http://www.linuxfoundation.org/en/Net
Q:      https://patchwork.kernel.org/project/netdevbpf/list/
T:      git git://git.kernel.org/pub/scm/linux/kernel/git/netdev/net.git
F:      Documentation/devicetree/bindings/net/
F:      drivers/connector/
F:      drivers/net/
F:      include/linux/if_*
[...]
```

Listing 3.1: The APPLETALK NETWORK LAYER and NETWORKING DRIVERS section in MAINTAINERS. M: assigns maintainers, L: mailing lists. S: denotes the current state of the section.

Maintainers themselves are organised in a semi-formal hierarchy [Mau10]. During a *merge window*, maintainers ask hierarchically higher-level maintainers to *pull their changes*, which is possible in two ways: Either by picking up and integrating patch data from mailing lists, or by *pulling* code from repositories. Once the top-level maintainer Linus Torvalds pulls and publishes changes, they become part of Linux mainline.

With slight variations, many other projects (*e.g.*, QEMU, U-Boot, or Xen) use either the same or similar format to define the areas of responsibility. `PaStA` supports the semantics and concepts of MAINTAINERS of those projects.

3.2.3 Lifecycle Management

The latest release of Linux is called the *stable tree*, and is actively supported with bug fixes until the next mainline release is cut, and becomes the new stable tree. Additionally, the Linux kernel community supports several versions of the kernel in parallel [Kro07] that are referred to as Long Term Support (LTS) versions. They are based on selected stable trees, and receive official support for up to six years. Fig. 3.1 illustrates the parallel development of mainline Linux and the maintenance of LTS versions.

Linux distributions and vendors usually choose LTS versions as the basis of their kernel (which may additionally contain a substantial amount of added drivers, domain-specific features, and many other additional elements), since they provide a stable and reliable base that will not be subjected to invasive changes (*e.g.*, API changes) during their lifetime. New features are only accepted mainline. Stable and LTS trees may only receive stabilisation patches, bug fixes, or fixes for vulnerabilities.

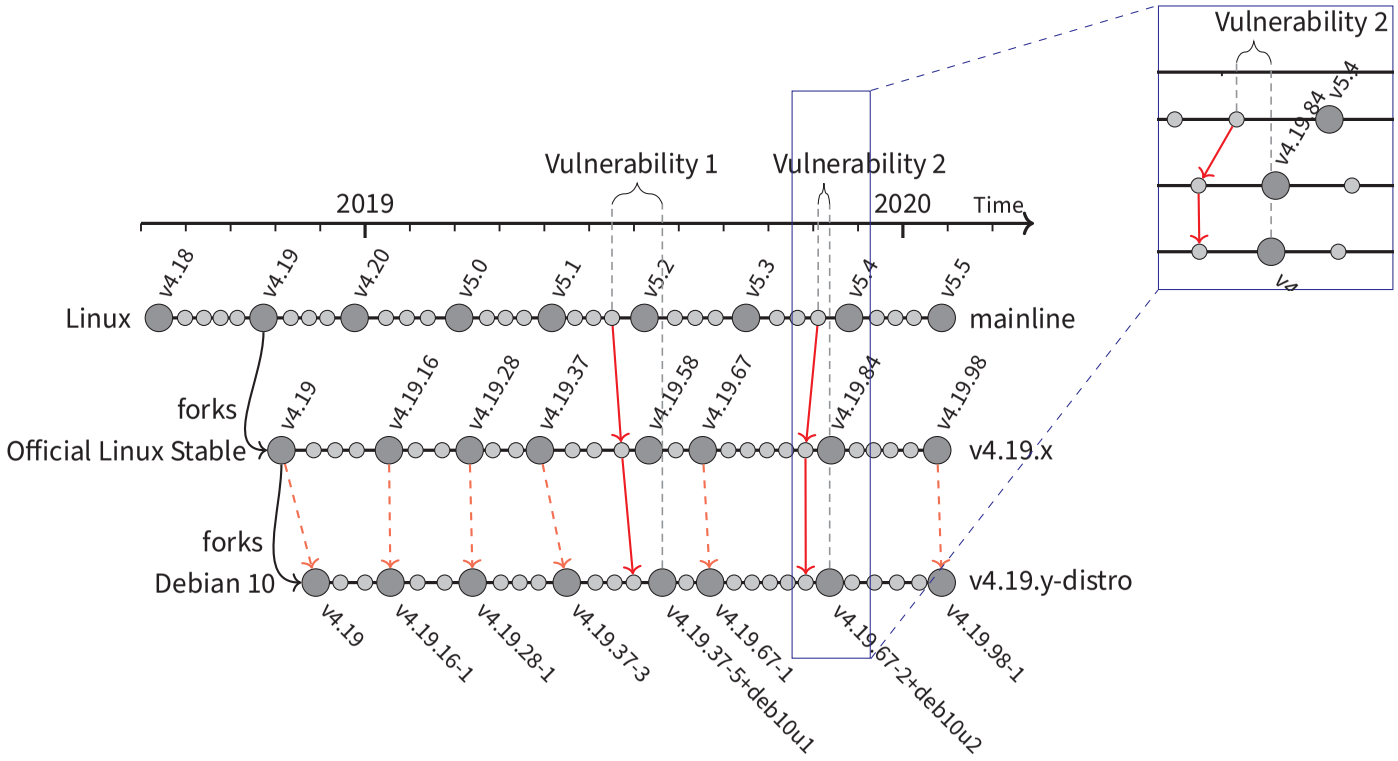

Figure 3.1.: Linux development timeline: Mainline, stable trees and distribution trees are supported in parallel. Typically, fixes for vulnerabilities are first fixed in mainline (*cf.* Vulnerability 1) and on a stable tree, before they are ported back by distributions. In rare cases (*cf.* Vulnerability 2), patches appear in distributions before they are published in mainline.

In case patches to LTS versions are also relevant for mainline, they must be, by the *upstream first* convention, integrated in mainline before they are ported back to stable releases.[4] After their release, distributions pick patches from stable versions and apply them to their own kernel repository.

From a temporal perspective, the typical pathway of a bug fix is mainline→stable→distribution, while exceptions apply.

3.2.4 Exceptional Vulnerability Handling

The aforementioned public review and integration process allows for an exception when fixes for security vulnerabilities must be handled. The Linux kernel is a key software component of a large class of machines from embedded industrial control appliances to cloud computing servers. Consequently, the Linux kernel community has established standard procedures for responsible disclosure [CW02; Fre+10].

Linux submission guidelines encourage developers to report exploitable security bugs to the non-public security team mailing list security@kernel.org: "For severe bugs, a short embargo may be considered to allow distributors to get the patch out to users; in such cases, obviously, the patch should not be sent to any public lists." [The20]

Similar to the regular public development process, patches for vulnerabilities are iteratively discussed, reviewed and refined – but all related conversations take either place in private email conversations or on closed lists. Once participants agree on a fix [Kro20], or after embargoes are expired, the majority of fixes follow the same procedures as bugs: Patches for mainline and affected stable versions are published at the same time, before they are integrated into distribution repositories. Fig. 3.1 (Vulnerability 1) illustrates the temporal process of a typical vulnerability. There is a second type of coordinated disclosure for severe vulnerabilities that I discuss in Section 3.5.

3.2.5 Formalisation

Let $\mathcal{M}$ be again the set of patches on mailing lists and $\mathcal{C}$ be the set of commits in the repository. Let further $\mathcal{U}$ be *universe of patches* $\mathcal{U} = \mathcal{M} \cup \mathcal{C}$. The `PaStA` approach turns assigning patches in mails to commits in repositories to a problem in graph theory: The universe $\mathcal{U}$ forms the vertices of an undirected and weighted graph $G = (\mathcal{U}, E)$.

[4] Ref. `https://www.kernel.org/doc/html/v5.10/process/stable-kernel-rules.html`

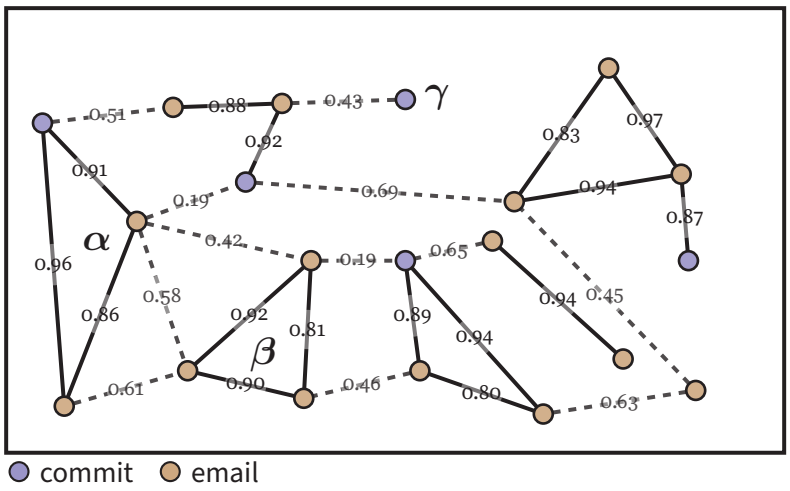

Figure 3.2.: Creating clusters of similar patches. Patches that exceed the threshold $t = 0.8$ form subgraphs of similar patches. Cluster α: contains patches on mailing lists as well as commits, cluster β only contains patches on mailing lists. Cluster γ is not mapped to any mail artefact and a potential off-list patch.

The graph G is created as explained in Section 2.3 and used to derive an undirected and unweighted subgraph $G' = (\mathcal{U}, E')$ that only contains edges exceeding a certain threshold ts for the edge weight. G' consists of connected components that divide $\mathcal{U}$ into partitions of similar patches, that is, equivalence classes. I identify those equivalence classes as $\sim_S$:

$$[x]_S = \{y \in \mathcal{U} \mid x \rightsquigarrow_{G'} y\}, \tag{3.1}$$

where $\rightsquigarrow_{G'}$ denotes reachability. Note that $|[x]_S| > 0$. Figure 3.2 illustrates the creation of clusters of similar patches. Given G', three different types of clusters can be distinguished:

α Integrated Patches: $\exists x_1, x_2 \in [x]_S : x_1 \in \mathcal{M} \wedge x_2 \in \mathcal{C}$
This category describes the finished integration process, as members of the equivalence class can be found on both: the mailing list and the repository. Again, $[x]_S \cap \mathcal{M}$ describes several revisions of the patch, $[x]_S \cap \mathcal{C}$ denotes the assigned commits in the repository.

Note that several commits in the repository may be assigned: While backports also match to mainline commit, there may even been multiple mainline commits, as a patch can be picked up by multiple maintainers and appear as multiple commits.

β Unintegrated Patches: $[x]_S \subset \mathcal{M}$
All members of this equivalence class can only be found on mailing lists. Members are, for example, different revisions of the same patch.

No commit in repositories can be found in this category. This means that the patch has either not been integrated yet (as the discussion is, for example, still ongoing), or, that the patch has been rejected and is not object to integration.

γ **Off-list Patches:** $[x]_S \subset \mathcal{C}$
No public development artefact can be assigned to the commit(s) in $[x]_S$. Besides false positives results of the heuristic, this category contains commits that arise from non-public integration channels.

The automatic detection of patches in group γ offer the possibility to identify commits that come from secret integration channels. Those commits will be subject of my analysis in Section 3.4.2.

3.3 Extraction of Development Characteristics

In this section, I ask two question that examine processual characteristics of the current LKDP on a quantitative basis:

1. What amount of patches is ignored on mailing lists?
2. Were patches integrated in conformance with development process guidelines?

The first question **1.** specifically examines type β patches that were never integrated. Among these patches, the question is which portion has never been addressed by reviewers or maintainers. Hence, it can be used as a proxy variable to draw conclusions the scalability of the process, and the *utilisation* of maintainers. By using `PaStA`'s methodology, I can automatically filter such patches and investigate the corresponding email thread.

The second question **2.** addresses patches of type α that have been integrated. Again, `PaStA` can automatically filter those patches. I will define formal criteria that allow for quantifying *conform integration*, according to development process guidelines. The amount of correctly resp. incorrectly integrated patches gives an overview of the overall adherence to self-imposed development processes.

Due to their exceptional role, patches of type γ will be examined in Section 3.4.

With *development process*, I refer to the LKDP process as it was described in Section 3.2. Linux, as well as other projects, is separated in different areas of responsibility. Those areas are the foundation of this analysis, as they allow for verifying if integrators (*i.e.* maintainers) of patches are effectively responsible for the integration.

I find both, a decreasing amount of ignored patches over time, as well as an decreasing amount of unconformingly integrated patches. I find that in the time window of the analysis (2011-05-01–2021-02-01), 26% of all patches were not integrated according to those *sections* of responsibility.

3.3.1 Ignored Patches

To answer the first question "*What amount of patches is ignored on mailing lists?*" requires a clear definition of the term *ignored patch*. I define a patch as ignored, if the following conditions are met:

1. The patch was sent to a public mailing list.
2. The mail with the patch received no responses from entities (*i.e.*, humans or bots) other than the original author.
3. The patch was not integrated in the official repository.
4. All *related patches*[5] meet the abovementioned conditions.

While the first criteria **1.** might sound straight forward, it is noteworthy that the Linux kernel community receives patches on over 200 mailing lists (Ref. Section 3.2). Not all of those MLs are exclusively used for patches for the Linux kernel project; some lists are mixed-use. They are, for example, used for receiving patches for userland tools (*e.g.*, tools for filesystems). Hence, `PaStA` filters those patches, by applying a heuristic: An email is a patch for Linux, if it patches files within the directory structure of the project.

Furthermore, I only want to respect patches that were sent by particular entities (*i.e.*, humans or bots). In order to select such patches, `PaStA` filters for patches that were sent by bots. The Linux kernel community uses several bots that are, for example, used to automatically send patches that are selected for backports or bots that resend patches that were picked up by maintainers for integration. `PaStA` detects those bots by keywords in the email body (*e.g.*, *Deet-doot-dot, I am a bot.*), special email-headers (*e.g.*, `X-Stable:`) and known mail addresses (*e.g.*, `pr-tracker-bot@kernel.org`).

Next, `PaStA` filters for development process-related emails with patches, such as pull requests or automated reports that contain patches or patches that are proposed for stable review.[6] Those emails are detected by searching for keywords in the subject or

[5] That is, other revisions of the patch in other mail threads.
[6] That is, patches that shall be ported back to older kernel versions.

body of the mail. For example, pull requests contain the keyword `[GIT PULL]` in their subject.

Finally, PaStA excludes patches that were sent *in reply to*[7] a previous patch. Developers often answer with proposals of alternative approaches of patches in replies. For the analysis, only the first patch(es) of an email thread are of interest. Therefore, a patch must fulfil one of the following conditions:

(a) The email that contains the patch must be the root of the email thread.

(b) The email that contains the patch must be the child of the root, and the root must not contain a patch.

The first criterion **(a)** matches for regular single patches. The second criterion **(b)** covers patch series. As explained in Section 3.2, patch series consist of a *cover letter*, an informal email that describes the patch series, and one or more patches, that are sent *in reply to* the cover letter.

For the time window of the following analysis, I chose 2011-07-21 – 2020-12-31, which covers almost a decade of development between the corresponding kernel versions v2.6.39 and v5.13-rc3. In that time window, I find 11,088,826 mails on 237 MLs. Frequently, mails are sent to multiple lists at once. After filtering for duplicates, I identify 6,609,392 unique emails. Within those unique emails, 2,499,510 mails contain patches (a ratio of 37.8%).

Table 3.1.: Ratio of ignored patches per year.

Year	Ratio of ignored patches
2011	3.78%
2012	3.21%
2013	2.65%
2014	2.35%
2015	2.06%
2016	1.84%
2017	2.03%
2018	1.88%
2019	1.80%
2020	1.73%
Average	2.14%

Table 3.2 shows the *composition* of all patches within the time window of interest. A fine granular illustration of the composition of patches on mailing lists of the Linux kernel can be found in Fig. 3.3. The graph also visualises dynamics in development cycles of the project: at every release of a new kernel version, I observe an increase of patches being sent to the lists. These are the patches that are sent during the two-week *merge window* (Ref. Section 3.2). After the merge window, I observe a decrease of incoming patches per week, the number of patches per week being sent to the lists is at a local minimum before the release of the next version.

[7] Email headers contain the necessary information.

Table 3.2.: Composition of all unique patches on all mailing lists in the time window of the analysis.

Type	Absolute amount	Relative amount
Regular Patch	1,417,081	56.7%
Not Linux	414,997	16.6%
Stable review	300,137	12.0%
Not first	164,335	6.6%
Process-related	103,061	4.1%
Bot	96,076	3.8%
Linux-next	3,823	0.2%
Σ	2,499,510	100%

It is noteworthy that a significant decrease of patches being sent to mailing lists can be observed at the end of every year, which obviously correlates with Christmas holidays. Note that only every second release is listed in the secondary x-axis for reasons of clarity.

For addressing the second criterion **2.** (*The mail with the patch received no responses from entities other than the original author*), I reconstruct the thread of the patch (across all available lists) by analysing the headers of the email. If the email was sent with a different identity than the original author, it is considered *not ignored*.

The third and fourth criterion **3.** and **4.** (*The patch was not integrated in the official repository* and *all related patches meet the abovementioned conditions*) are verified by `PaStA`'s methodology explained in Section 2.3. Therefore, I establish patch clusters for all emails in the time window and try to map them against 709,909 commits in the repository. Of course, all emails in type α clusters (*i.e.*, patches that are mapped to a commit in the repository) are considered not ignored. Obviously, off-list patches (type γ) can not be judged, as they are not observed. What remains are type β patches, patches that are not integrated. If any patch in a type β cluster received an answer by an email address other than the original author, patches of the cluster are not ignored. The cluster is counted as ignored, if no thread of all patches received an foreign answer. This strict criteria results in a lower bound of ignored patches.

Table 3.1 shows the accumulative fraction of ignored patches for all mailing lists per year. While the average amount of regular patches per week tripled from 2012 until 2021 (≈1,300 patches per week in 2012 vs. ≈4,000 patches per week at the end of 2020, *cf.* regular patches in Fig. 3.3 and total patches in Fig. 3.4), the absolute amount of ignored patches stayed at a almost constant level (*cf.* the red graph in Fig. 3.4).

Between 2012 and 2019, the amount of ignored patches almost halved in size. Note that I do not consider the fraction of ignored patches in 2020, as there can be patches that

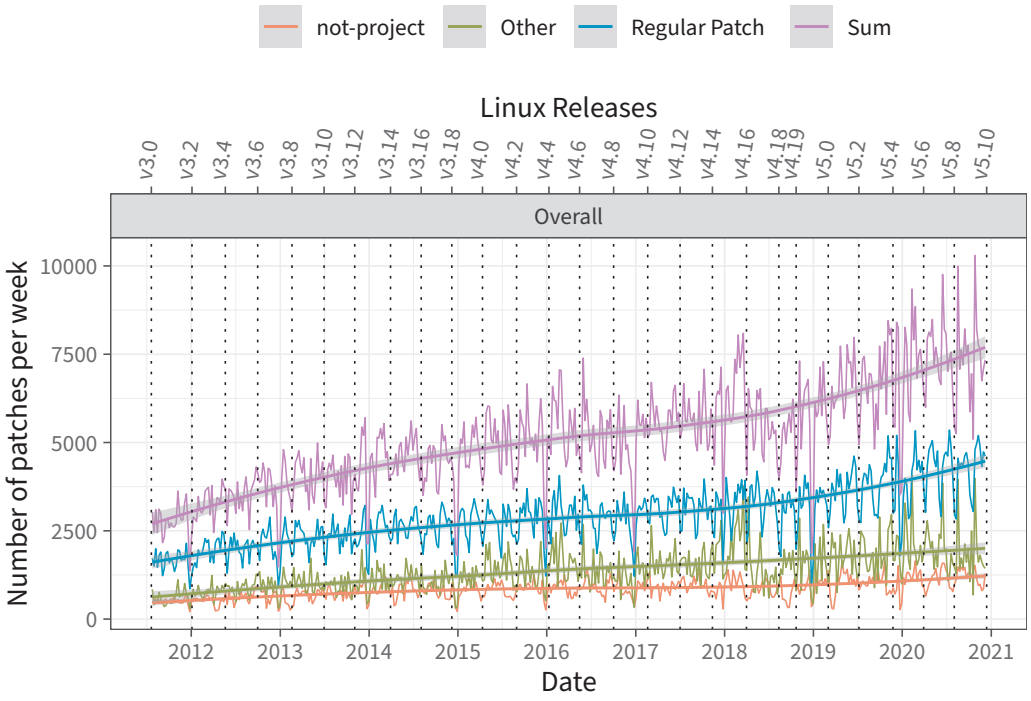

Figure 3.3.: Composition of *type* of patches on the 227 Linux kernel mailing lists. The type *Other* accumulates bots, process-related, linux-next, and not-first patches. Solid lines denote the smoothed conditional mean.

were classified as ignored, while they just have not yet been addressed at the time of writing this thesis. This argument is underlined by the integration duration in days as it is shown in Fig. 2.5. The 99%-quantile of patches is integrated within 314 days, which overlaps with the time window of the analysis. A fine-granular ratio of ignored patches on a weekly basis can be found in Fig. 3.5.

These plots give an overall overview of the amount of ignored patches on all MLs of the Linux kernel. However, as mentioned before, the Linux kernel community incorporate more than 200 different mailing lists. Different mailing lists are used for different *topics* of the kernel. Different subsystems have different communities that behave differently. Faceting the graph by mailing lists gives a better overview of per-list behaviour. This allows, for example, to identify *hot spots* of ignored patches. Exemplarily, Fig. 3.6 and Fig. 3.7 illustrate the dynamics of ignored patches on the top four lists with highest patch traffic: the *main* Linux Kernel Mailing List LKML, Network Development (netdev), hardware component description (devicetree), which is closely related to the ARM port of Linux (linux-arm-kernel).

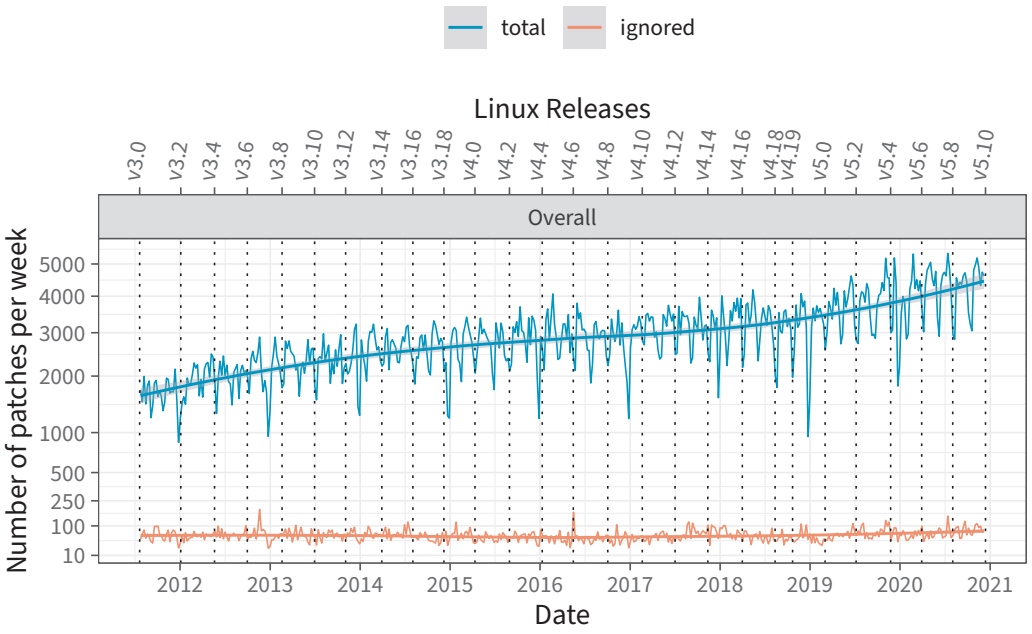

Figure 3.4.: Regular and ignored patches between v3.0 and v5.10. The green graph denotes the total amount of regular patches, the red line denotes the amount of ignored patches. Note the square root scale for the guidance of the eye. Solid lines denote the smoothed conditional mean.

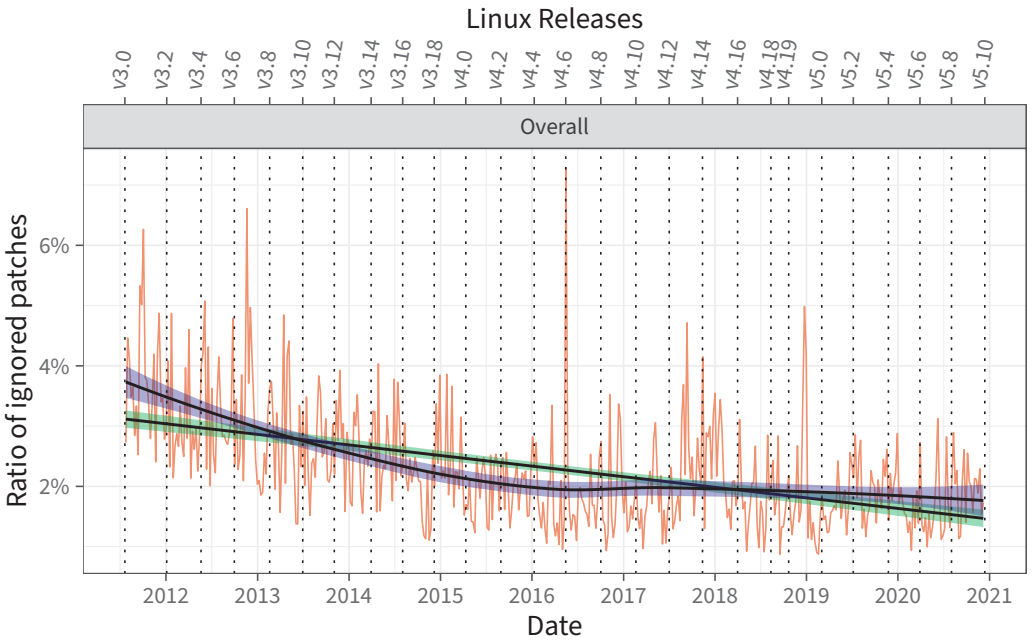

Figure 3.5.: Ratio of ignored patches per week between v3.0 and v5.10. The violet curve denotes the smoothed conditional mean, the green curve fits a linear model.

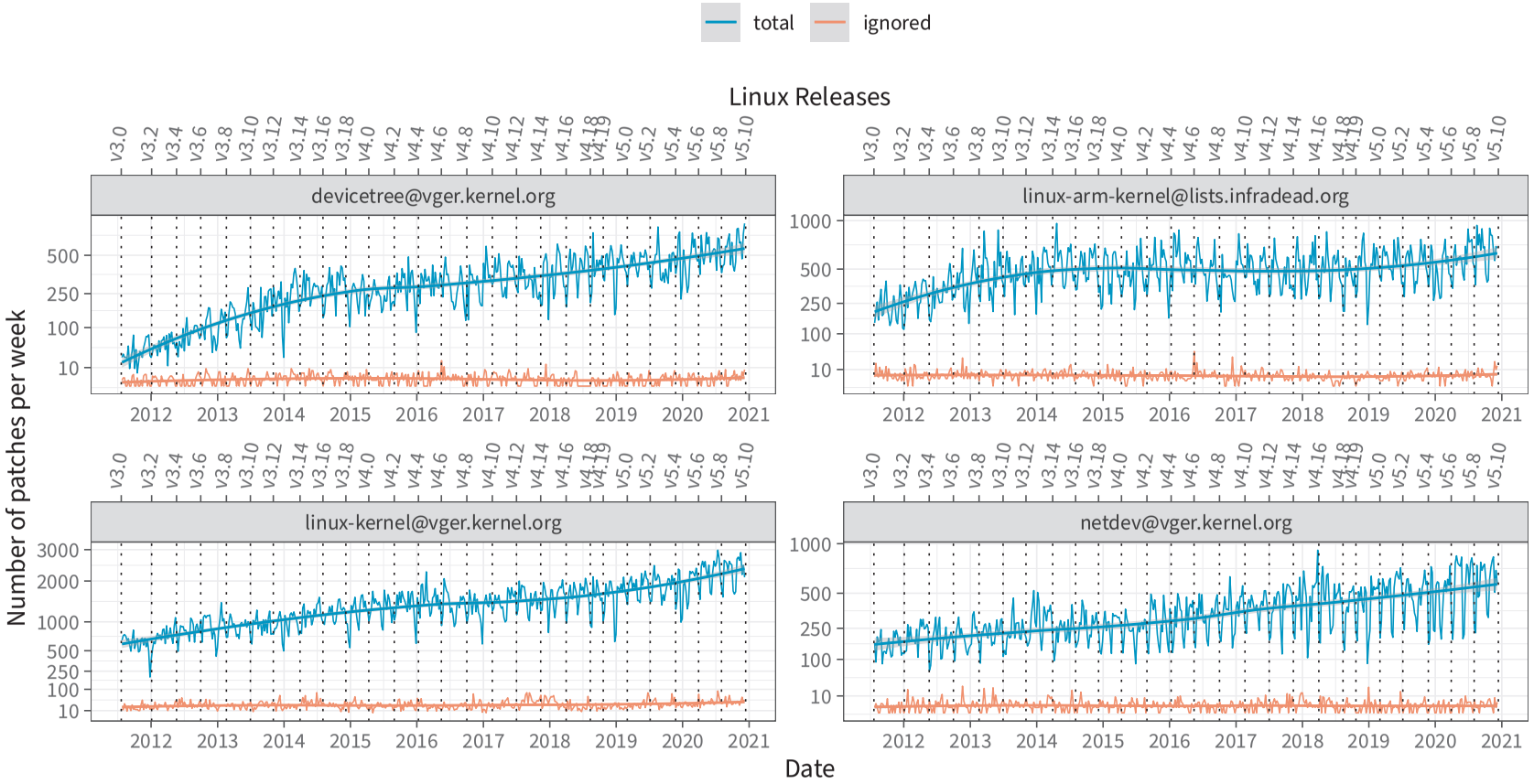

Figure 3.6.: Regular and ignored patches between v3.0 and v5.10 on the top four high patch traffic lists.

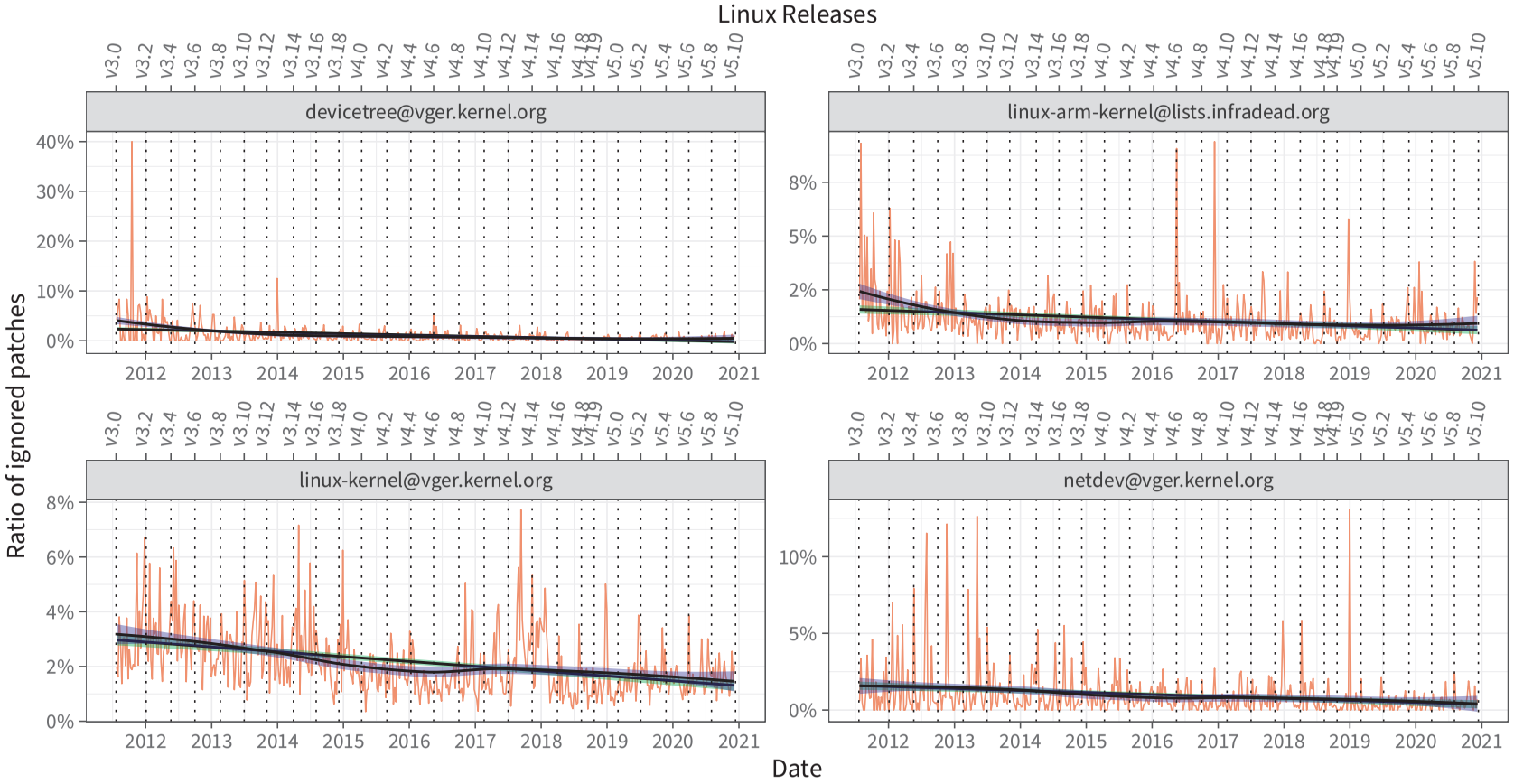

Figure 3.7.: Ratio of ignored patches between v3.0 and v5.10 on the top four high patch traffic lists.

3.3.2 Conform Integration of Patches

To answer the second question (*Were patches integrated in conformance with the development process guidelines?*), I first define the term *conformally integrated patch*.

With respect to the core characteristics of the LKDP as defined in Section 3.2, I further define a patch as correctly integrated, if the patch was integrated (*i.e.*, committed) by a maintainer that is, at the time of the integration of the patch, responsible for at least one section that is affected by the patch.

For the following analysis, I use the same time frame as mentioned in Section 3.3.1. For each regular patch for the Linux kernel that was sent to a list and written by a human, I first identify if the patch was integrated, otherwise it is counted as *not integrated*. If the patch was integrated, I then verify if the integrator (*i.e.*, committer) of the patch is responsible for the area of patch, according to the definition in the MAINTAINERS file at the time of integration. Therefore, `PaStA` reimplements `get_maintainers.pl`, as the original implementation is unsuitable for massive batch processing of millions of patches.

Figure 3.8 shows the fraction of correctly, incorrectly, and unintegrated patches per week, fitted with a smoothed conditional mean. As mentioned in Section 3.3.1, an increasing amount of unintegrated patches at the end of the analysis can be observed. This can be explained by a high amount of patches, that have *not yet* been integrated at the time of the analysis. Nevertheless, across all lists, I can observe an almost constant fraction of unintegrated patches between 2012 and 2019, while amount of correctly integrated patches slightly increases.

Analogously to the previous analysis of ignored patches, I facet the analysis by mailing list. Figure 3.9 shows the fraction of unintegrated an (in-)correctly integrated patches across the top 4 high patch traffic list. The plot discloses the probability if a patch was integrated correctly, if it was initially sent to a specific ML. Across all top four high-volume patch traffic lists, I observe an increasing amount of correctly integrated patches.

The results of this analysis can be interpreted as an increasing awareness of the importance to the adherence to development processes. Projects at the size of the Linux kernel require clearly defined development processes: results show, that the amount of correctly integrated patches increases over time, while the absolute amount of patches that is sent to mailing lists increases well. The scalability of the project accompanied by the required maintenance effort of maintainers is enabled by the conformance with development guidelines.

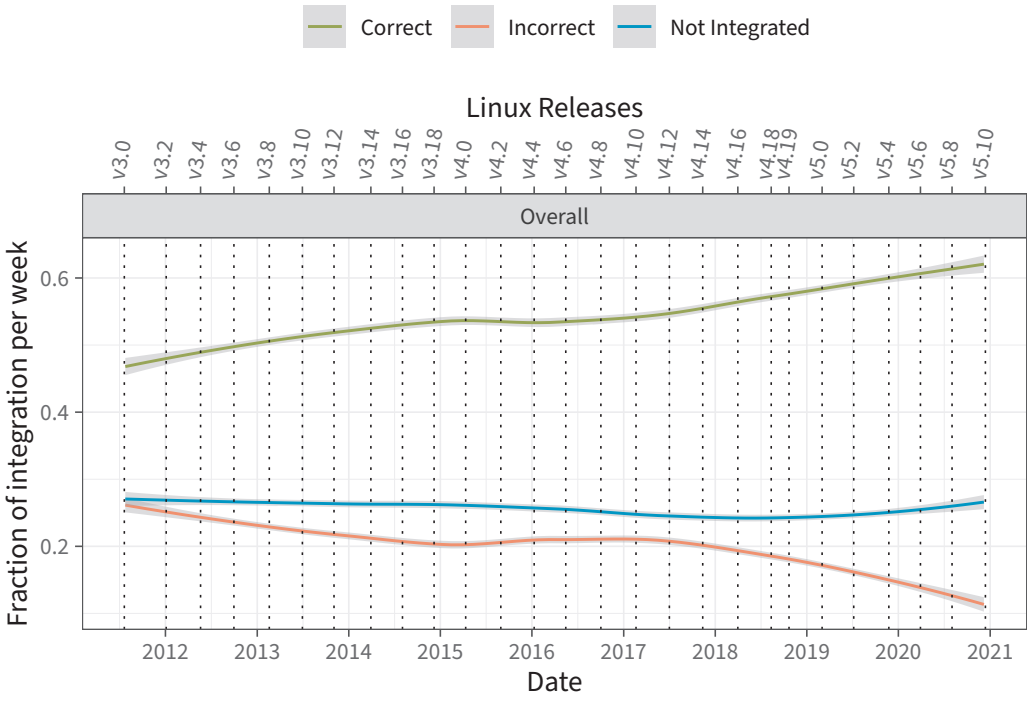

Figure 3.8.: All mailing lists: Fraction of (in-)correctly and unintegrated patches.

For safety certifications efforts, these data can be used to quantify the robustness of the development process, to identify hot spots of process violations, or to derive qualitative metrics on the development process.

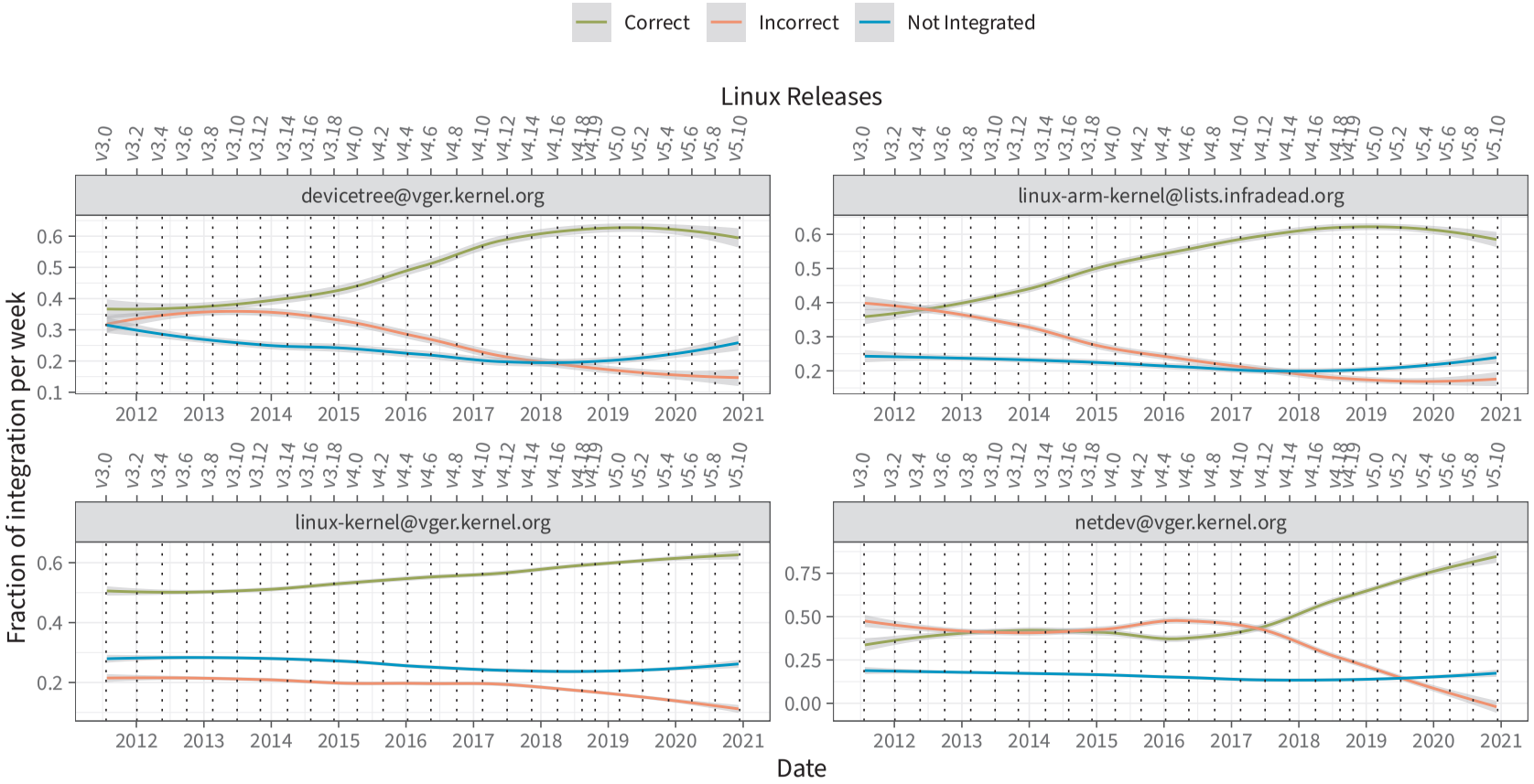

Figure 3.9.: Top four high-volume mailing lists: Fraction of (in-)correctly and unintegrated patches.

3.4 Violation of Development Processes

> *Given enough eyeballs, all bugs are shallow*
>
> **— Eric S. Raymond**
> Linus' law [Ray99]

In the last chapter, I focused on examining characteristics of patches and commits that were integrated resp. not integrated, that is, patches and commits in categories α and β. What remains are patches of type γ, patches that can be found in the repository, but never have been seen a publicly accessible resource before integration.

On 14 August 2018, a series of patches was integrated in Linux to provide mitigations for the Level 1 Terminal Fault (L1TF) [Van+18; Wei+18] vulnerability[8] – a speculative execution attack with severe consequences that enable large scale data leakage across virtual machines on Intel-based cloud appliances. While associated Common Vulnerabilities and Exposures (CVE) entries were already filed in December 2017 [CVE17], the vulnerability was embargoed until 14 August 2018 [Int18c] – the same day of the disclosure and integration of the critical patches for Linux. Unlike ordinary patches, these patches were—for obvious reasons—not discussed and developed on one of Linux's public communication channels (*i.e.*, MLs) beforehand.

However, the fact that a patch was *not* publicly discussed betrays it: I will show that it is possible to detect such patches as soon as they enter a public repository. This gives attackers valuable information advantage to design exploits. For the aforementioned attack, it took another five days until the patches were integrated and rolled out by Debian 9,[9] a popular and wide-spread Linux distribution.

By identifying commits of type γ, it us possible to uncover commits from non-public secret integration channels with high probability using the `PaStA` methodology. It is possible to systematically obtain *off-list patches*: Code changes that were developed outside the official public lists. Besides fixes for security vulnerabilities, I also find that there exist systematic channels to inject code into the Linux kernel while bypassing public discussion.

In addition to the possibility to quantify and detect violations of the development process, the `PaStA` methodology provides two advantages for malicious attackers:

(a) it significantly reduces search efforts for fixes of security vulnerabilities, compared to fully manual investigation, and

[8] See Linux commit 958f338e96.
[9] See the announcement of Debian kernel 4.9.110-3+deb9u3.

(b) it provides temporal advantage for the design of attacks.

I will show that it is possibly to systematically detect development process infringements in OSS projects that works as soon as commits arrive in repositories. By analysing type γ commits, I categorise different types of secret integration channels of the Linux kernel, such as *bypass of development processes* or non-publicly discussed fixes for *security vulnerabilities*.

3.4.1 Secret Integration Channels

The openness of the development processes is a key aspect of any OSS project: Almost all development activities happen in public. Since development artefacts (*i.e.*, discussions or patch data on public mailing lists) are observable, this allows for analysing the process in detail.

However, especially the development of fixes for critical security vulnerabilities intentionally happens behind closed scenes [The20]. After their disclosure, fixes silently appear as commits in the repository. Nonetheless, those commits can not be assigned to any prior artefact that relates to its public pre-integration history. Unless the vulnerability is explicitly announced or attracts medial attention, I disprove the common belief that patches typically drown in the noise of other commits in the repository.

Nevertheless, a full coverage of all public available development resources allows for systematically excluding *regular development noise* in order to separate it from irregularities: I mine for commits in repositories that come from secret integration channels—and detect them just-in-time to design exploits for vulnerabilities. For the Linux kernel, I *do* have full coverage of all MLs. Figure 3.10 illustrates the chase for missing links: I deduce that commits that can not be assigned to publicly observable artefacts must arise from secret integration channels.

A rapidly changing code base, size and complexity inherently results in software defects that can lead to severe software vulnerabilities. In 2019, 170 CVE entries were filed for all different versions and flavours of the Linux kernel, and many more potential security vulnerabilities have been fixed without CVE analysis and assignment [Cor19]. Unavoidably, the *kernel community* has processes on managing critical vulnerabilities.

In contrast to regular development activities, vulnerabilities shall be reported to and discussed on private mailing lists [The20]. The rationale behind private discussions is the *responsible disclosure* vulnerability disclosure model: Software producers get the chance to provide fixes for vulnerabilities before they are publicly disclosed [CCR04]. Therefore,

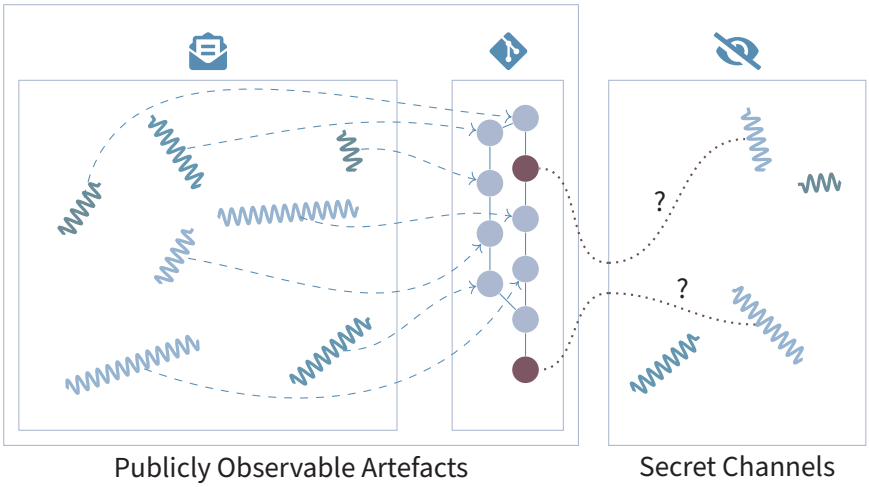

Figure 3.10.: Disclosing secret integration channels. On the left: artefacts on public channels (*e.g.*, patches on mailing lists) are assigned to commits in the repository. On the right: Commits that lack assignable public artefacts arise from secret integration channels.

security mailing lists are closed-recipients lists to avoid early public attention. Only carefully selected and trusted individuals have permission to join those lists. Security lists are used for coordination, and to setup private communication between reporters and affected subsystems. They can also be used to develop the actual fixes for the issues [Kro20].

Eventually, when the fix is in its final state, it is released for all affected version of the kernel that are supported by the community: This leaves the first publicly visible footprint of the vulnerability: the patch(es) in the repository. Yet, it misses a link to a publicly observable artefact.

For detecting type γ commits in repository, I reverse the question, and find commits in repositories that do *not* enjoy any traceable pre-integration history.

Temporally, a patch should first appear on a public mailing list before it can be found in the repository. Hence, any new commits in the repository that can not be assigned to emails that were received before integration were integrated through non-public integration channels.

3.4.2 Analysis

In contrast to a *just-in-time* online analysis that constantly monitors new incoming mails and commits on a regular basis, I perform the detection of off-list patches as an ex-post analysis of a predefined time window. From a retrospective view, I can examine if a

commit would have been detected as an off-list patch if a just-in-time online analysis would have been performed.

Overview

I am naturally limited by the availability of artefacts for the choice of time window for the analysis. For the time window of emails, I consider the date since creation of our lab's collection (2019-May-01) until I performed the analysis (2019-Dec-01). I did intentionally not use the official repositories, as they miss MLs (Ref. Section 2.3.4).

Patches typically take weeks to months until they are integrated to the repository (Ref. [JAG13] and Section 2.4.2). Note that the VCS of the Linux kernel, git, distinguishes between two temporal events: the author date and the commit date. The commit date is the date when the commit has been applied to the developer's (local) repository. Rewriting a repository's history can affect commit dates. The author date is the date when the commit was originally made (*e.g.*, the date when the code was committed by the original author to their repository) or, in case of an email-based workflow, the timestamp when the email was sent (*i.e.*, the `Date:` header of a mail). Hence, I integrate all commits with an author date within the same time window as chosen for emails. I respect all commits that meet the abovementioned criterion up to Linux version 5.4 (released 2019-Nov-24).

In that time window, $516{,}197$ different messages can be found, $\approx 40\%$ of them contain actual patches. However, not all mails that contain patches are relevant for the analysis. Messages contain mails from bots, pull requests, backports and other noise. The tool `PaStA` filters those messages by applying appropriate heuristics. $30{,}396$ commits can be found in the corresponding time window of the repository.

In the time window of the analysis, `PaStA` was able to map $\approx$96% of all commits against patches from mailing list and therefore regular development noise, while $1{,}240$ commits were not assigned to any Message-Id.

A commit with a missing mapping to a message can fall into one of the following categories:

1. The heuristic failed to detect the patch (false negative).
2. The original patch was sent to the list before I started recording mailing list data (miss of discussions).
3. *Off-list patches* – patches that were integrated through a non-public channel.

Off-list Patches

With a manual investigation of the remaining 1,240 commits, I was able to find different categories for off-list patches in the Linux kernel repository. Figure 3.11 illustrates different types of off-list integration channels.

Revert commits A revert commit is a commit that reverts a previous commit in the repository's history. They are used, for instance, to eliminate new features or enhancements if they cause undesired side effects or if they are in a defective or an incomplete state. It is often the preferred choice to revert the commit, as it is more efficient and less error prone to simply revert corresponding changes rather than to provide expensive or complex fixes, especially at the end of a development cycle. A refined version of the commit can later be integrated during the next development cycle.[10]

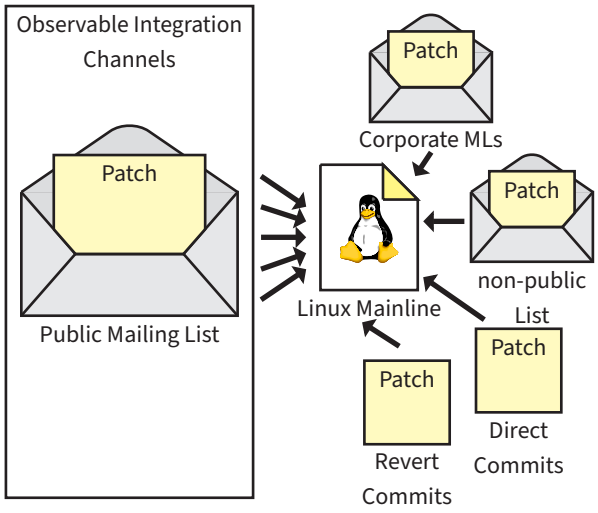

Figure 3.11.: Public observable (left) and non-public integration channels (right).

Many maintainers do not send reverting patches to mailing lists. They either integrate the reverting patch directly, or they send a response to the original thread of introducing patch that it will be reverted while omitting the actual reverting patch. Hence, the reverting patch can not be found on the mailing list.

Such reverting patches can automatically be detected, as the subject line of the commit message contains the keyword `Revert` by convention. For the time window of the analysis, I detected 64 off-list revert commits in the repository.

Commits by Repository Owners Repository owners have a special role in projects: They have the permission to push code to official resources.

In case of Linux, Linus Torvalds is the owner of the official repository and the last approving authority. It is his final decision to judge if a patch or pull request is integrated mainline.

[10] Example: Linux Commit 69bf4b6b54fb: *[...] and it's [the bug] not immediately obvious why it happens. It's too late in the rc cycle to do anything but revert for now.*

This, in turn, allows him for integrating or reverting patches ad libitum. It is not unusual that he reverts patches without discussion or short before the release of a new version.[11]

Torvalds sees himself as the manager of the Linux kernel – and no longer as active developer. Nevertheless, he sometimes integrates code or fixes without any prior public discussion. Those commits can automatically be detected, as project owners are known. For the time window of the analysis, I detected 48 off-list patches from Linus Torvalds. None of them contained security-related fixes.

However, the phenomenon of bypassing public review processes can also be observed at other maintainers.

Bypass of public review processes During the analysis, I found several regular patches that have never been sent to any public mailing list. To exclude false negatives of the heuristic, I contacted 18 different authors and collected affirmative answers from 14 authors – four did not answer.

All of them confirmed my finding that their patch(es) have never been posted on a public mailing list. For example, I found that one maintainer committed 40 patches to their repository in the time window of my analysis. The author confirmed my assumption and commented that they did not expect it to be that many. Most patches were only minor stylistic fixes, but I also found invasive patches. The author agreed that those patches would have required a public review process. I questioned maintainers why they skipped the official review process. Their typical answer was that they *accidentally forgot to send the patch*.

While many of those commits contain uncontroversial changes like documentation, style or typographical fixes, other commits contain in-depth fixes for subsystems. One maintainer explained that they picked up a fix from another subsystem that is also valuable for their area of responsibility. However, all responding maintainers agreed that those patches should have been publicly discussed.

Established non-public integration channels Besides maintainers that directly commit patches without discussion, I also found subsystems that tend to bypass public review processes.

My observations give evidence that some subsystems deliberately bypass public review processes. For example, there are whole architectures and subsystems that are in the

[11] Example: Release of Linux 5.3: Linux Commit 72dbcf7215.

responsibility of certain companies.[12] A corporate representative has the role as an official mainline subsystem maintainer, which gives them the possibility to send pull requests to Linus—by trust. Within those subsystems, off-list patches can be found from authors other than the maintainer. Still, those patches can not be found on any public mailing lists. At the same, the author's and maintainer's email address show that both work for the same company.

From such artefacts in commits, I conclude the existence of non-public company internal review and integration processes. However, those patches do intentionally bypass public review process.

One maintainer confirmed my assumption and underlined that they *forgot to add the public list*, and that *normally all patches are discussed on the public mailing list before they land*.

Security Vulnerabilities The remaining commits contain fixes for security vulnerabilities. According to Linux's security process (explained in Section 3.2), patches for security vulnerabilities should be discussed on private non-public communication channels.

Typically, the majority of those patches drown in the noise of thousands of other commits. To prevent simple keyword-based search heuristics, commit messages are worded neutrally, links to CVE entries are only sometimes mentioned in the commit message [Kro19].

To confirm the assumption that I hit security vulnerabilities through non-public integration channels, I contacted 12 authors. A list of the related and confirmed vulnerabilities can be found in Table 3.3. All of them confirmed that those patches are security related and that they have either been discussed on the non-public security mailing list, or been sent directly to the maintainer.

I calculated, in days, how long it takes for Debian 10 (Buster) and Ubuntu 18.04 (Bionic Beaver Hardware Enablement Kernel) to apply the patch to the distribution's fork of the Linux kernel. Positive numbers denote a potential temporal advantage for an attacker, negative numbers mean that the distribution applied the patch before it was disclosed to public. The categories of vulnerabilities contain denial of service attacks, buffer overflows, privilege escalation, and buffer over-reads.

In the analysis, I found, among others, fixes for the spectre-like attacks CVE-2019-11135 [Sch+19b] and CVE-2019-1125 [CVE19]. Ubuntu integrated both fixes before they

[12] I do not want publicly point to those subsystems.

Table 3.3.: A list of vulnerabilities that were detected by my approach and confirmed by corresponding authors. A negative period means that patches were integrated by distributions before the vulnerabilities were disclosed. SecML means if the patch was routed through the security mailing list, or privately discussed with maintainers.

CVE-2019	Description	Patches	SecML	Ubuntu 18.04	Debian 10
NA	DoS vulnerability for Cavium systems	4	no	59d	n/a
NA	smack: use after free	1	no	14d	n/a
13233	x86/insn-eval: use after free	1	yes	54d	61d
13272	pot. privilege escalation	1	yes	25d	12d
12817	ppc: inter-process memory leak	2	yes	-5d	44d
1125	x86/speculation: spectre v1 swapgs	4	yes	-5d	2d
14283	floppy: out-of-bounds read	2	yes	71d	18d
14283	floppy: DoS / div by zero	2	yes	71d	18d
11833	ext4: leak of sensitive data	1	yes	71d	29d
NA	s390: pot. leak of sensitive data	1	yes	45d	n/a
NA	apparmor: out of bounds by user-controlled data	1	no	179d	60d
11135	x86/tsx/speculation: TSX async abort side channel	9	yes	-1d	-1d

were publicly disclosed, while Debian only integrated fixes for CVE-2019-11135 before they were publicly disclosed.

I also found patches for an *easy to exploit*[13] denial-of-service attack for ARM64-based Cavium systems. The vulnerability has no assigned CVE entry. It took almost two months for Ubuntu Bionic to integrate the patch. At the time of writing, Debian Buster, as well as the affected 4.19 Linux LTS tree, still lack appropriate fixes.

For the majority of vulnerabilities, my approach gives an attacker a temporal advantage from 2 to 179 days. While most patches for vulnerabilities are included on the stable Linux LTS trees, some distributions still lack patches for the corresponding vulnerabilities.

3.4.3 Related Work

Software vulnerability life cycle analysis is related to this area work, and a well-researched topic [Hua+16; SSL12; Aro+10].

Huang et al. [Hua+16] find a considerable delay between disclosure of vulnerabilities and the availability of fixes. Based on a case study of six different projects, they found an average time of 52 days from vulnerability disclosure to releasing an actual fix. However, they also find that almost half of the vulnerabilities are fixed within one week.

In 2010, Arora et al. [Aro+10] argue that instant disclosure of a vulnerability forces vendors to speed up the release of a fix by 35 days.

Shahzhad et al. [SSL12] analyse the life cycle of vulnerabilities that are filed in software vulnerability data sets. In their large-scale analysis that includes a big variety of different projects, they find that the amount of time required to fix vulnerabilities decreased from 1998 to 2011: Since 2008, 80% of all vulnerabilities are fixed by vendors before their disclosure. Yet, their study does not consider that providing a patches is only a first step, but necessitates integration in software distributions, and actual deployment by users.

In a large-scale empirical study, Li and Paxson [LP17] investigate bug-fixes for security vulnerabilities in open-source projects. For their comprehensive analysis, they assign 3,094 CVE entries in the National Vulnerability Database (NVD) to 4,080 commits in 682 unique git repositories. Mining for links to commits in the CVE description establishes the approximate connection between CVE entry and commit hash. Later, they extract characteristics of security-related commits. They find that security fixes are less complex

[13] According to an assessment by the author of the fixes [Zyn19].

and more localised than non-security fixes. Furthermore, they find that 70% of security-related patches were committed before public disclosure and conclude that development and deployment processes provide a window of opportunity for exploitation. However, for a responsible disclosure process, it is necessary that patches must be developed (and committed) before disclosure. Yet, the date of a commit is not necessarily the date of its public visibility. In this work, I showed that developers intentionally distribute and release patches on secret channels before they finally publicly publish the repositories. Attackers do not have the opportunity for prior exploitation in those cases. In this analysis, I respect this fact and use the time difference of the public availability of a binary software release and the date of the public disclosure as the basis for my analysis.

Kroah-Hartman argues that only a small fraction of Linux kernel security fixes are assigned to CVE entries [Kro19]. From 2006-2018, 1005 CVEs were assigned to the kernel. He argues that, on average, bugs with CVE entries are 100 days fixed in mainline before they get a CVE assigned. Furthermore, he argues that the amount of vulnerabilities of vendor distributions can significantly be reduced by choosing LTS versions of Linux.

Insider attacks, such as infiltration, or compromises of organisational structures, are well-known in literature [Bis+14; KP13]. I showed a practical *outsider attack* that exploits the openness of the development model itself by using its development artefacts to conclude to systematic integration of patches that lack public discussion. In [And02], Anderson argues that the security of a development model should *not* depend whether it is open or closed.

The software engineering community uses artefact mining techniques to draw quantitative conclusions on development processes [Job+17] or to determine various software performance indicators [Hem+13; Chá+17].

3.4.4 Acknowledgements

I thank Greg Kroah-Hartman for giving us the opportunity to discuss the topic with Linux kernel security officers. I also thank authors of off-list patches for their detailed answers, discussions on their patches and fruitful conversation. I do not mention them by name.

3.5 Discussion

This section is a common discussion of Section 3.3 and Section 3.4. I first examine validity and potential weaknesses of my approach, and then discuss how my results affect OSS development processes. I conclude with suggestions how they can be adapted to accept (and deal with) risks that are anyway unavoidable, and concentrate on handling highly critical issues as good as possible.

3.5.1 Validity

Analysis method My approach is an ex-post analysis. In Section 3.3, I consider a time window of almost a decade of development. For this time window, I lack full coverage of all Linux Kernel mailing lists. However, as this analysis investigates patches and commits of type α and β, full coverage is not required. In particular, I focus on the analysis of the Top-4 high frequent MLs of the Linux kernel, which is part of my ML archive of the time window of the analysis.

In contrast, the analysis in Section 3.4 requires full coverage of all MLs. Hence, I consider a time window of seven months, since the beginning of the ML collection of our laboratory. This allows for judging from a *future* perspective if a patch *would have* been detected as an off-list patch at the time it was integrated into the repository.

Nevertheless, while the retrospective position is only required to determine the practicability of the approach in case of the offlist analysis: It is straightforward to extended the methods to apply just-in-time, which is obviously necessary to abuse any undistributed security fixes. Periodic, frequent updates of the repository and mailing list data ascertain valid and current data, and are a mere technical detail. New incoming commits must be compared against the available mailing list data. If a patch is not an off-list patch, then the corresponding mailing list entry must be available at the time of the analysis. As soon as a commit is pushed to a public available repository, my method allows to determine if the commit comes from a private channel.

In a private discussion, Greg Kroah-Hartman, maintainer, among others, of the stable and LTS trees of the Linux kernel, states the undocumented procedures how patches are distributed behind the scenes [Kro20]. The exchange strategies vary depending on the involved maintainer(s) and the issue at hand: One possibility is to exchange patches via private email. Another method is to distribute patches as git bundles, a technique that allows for exchanging elements of a git repository without relying on public remote servers, while it still guarantees stable commit hashes to maintain unique

patch identifiers. In a third method Linus Torvalds pulls patches from a maintainer tree. Since such trees are publicly available, this method opens a further temporal advantage for attackers, as a just-in-time analysis can also monitor patches from maintainers' repositories.

Generalisability The primary concern of this analysis is an in-depth analysis of patch flow into the Linux kernel repository from public and non-public resources by using peculiarities of its mail-based development process. However, the approach is neither limited to Linux as analysis target, nor to mailing lists as means of discussion. Except for handling some technical details and taking minor process differences (*e.g.*, the use of multiple parallel communication channels) into account, the approach can be directly applied to such systems.

Of course, the exact reasons for the existence of non-public integration channels depend on the project. Especially in projects with smaller communities, maintainers often tend to directly commit code changes without public announcement or discussion (*e.g.*, BusyBox), as upfront public discussion is often considered time-consuming and dispensable. However, this limitation is mitigated by the fact that projects with smaller communities only receive a moderate amount of patches. Especially critical system software typically demands adherence to public review processes, regardless of community size.

My idea of development process reverse engineering is also applicable to processes that do not build upon mailing lists: If *any* publicly available development artefacts (*e.g.*, pull requests, entries in issue trackers, …) are available that include relevant data before their integration, then reverse process engineering uncovers any irregularities, in particular, deliberate violations of the development process.

Scalability Concerning Section 3.3, I conducted the analysis on the Linux kernel with a time window of almost a decade. $2{,}499{,}510$ patches were compared against $709{,}909$ commits. To the best of my knowledge, Linux is *the* largest OSS projects that uses ML as the core element of its development process. This underlines the scalability of my approach with Linux, as well as with other projects.

With respect to Section 3.4, I conducted the analysis on a time window of roughly seven months, I found $30{,}396$ relevant commits in the repository (authored after 2019-May-01 and integrated before Linux v5.4, released 2019-Nov-24). Within those commits, I found $1{,}240$ potential off-list patches. By applying heuristics to exclude revert patches and commits by project owners, I was able to exclude further 112 commits. With my approach, I filtered ≈96% of regular development noise.

Nevertheless, 1,128 commits required manual analysis, which may seem to imply a considerable impediment to a fully automatic system at first glance. However, commits span a time window of 207 days. On a daily basis, this accounts to manual investigation of (rounded up) six commits per day. Assuming, in accordance to my personal experience gathered, that an experienced developer can decide within a minute or two if a patch addresses a vulnerability, then the daily time investment would only require a reasonable amount [Mur+19] of around ten minutes.

Not enjoying the benefits of PaStA would require a fully manual inspection of all incoming commits, which is unrealistic: The official repository of the Linux kernel (merge commits are already excluded) received 70,632 commits between release v5.0 and v5.4. The development between those releases took 329 days. On average, 215 patches were integrated per day. Assuming the same amount of time required for manual investigation, an experienced developer would need more than three hours of concentrated reviewing per day. Hence, I argue that my approach is suitable for real-world scenarios, as it significantly reduces the amount of time that is required for manual review.

However, the time to find *some* security-related fixes could be reduced even to zero by employing simple heuristics, such as filtering for well-known author or institution names: For instance, out of the 12 fixes I identified, 3 originated from Jann Horn (GPZ). While this might have been pure coincidence, I argue that learning about the social structure behind Linux could be exploited in this respect.

Internal Validity The approach of PaStA is able to reconstruct the development process. The approach can be used for further quantitative analysis of aspects of the development process.

In particular, the approach systematically uncovers non-public integration channels and identifies commits that are potential fixes for security vulnerabilities. However, the method fails for vulnerabilities that are discussed in public before integration.

Statistical data on how many patches are sent to private security mailing lists, or how many critical vulnerabilities are discussed in public are not available. Hence, it is hard to calculate the accuracy of the approach since the recall is not available. Yet, I found 12 vulnerabilities in my analysis, which underlines the practical utility of the approach.

However, it is worth mentioning that counting or searching for CVE entries for a certain time window is neither an appropriate method of accounting the number of vulnerabilities in a system nor an alternative method to automatically find security vulnerabilities: Only a fraction of kernel security fixes get CVEs [Kro19; Edg19] assigned. CVEs are

also known to be abused as *integration shortcuts* [Kro20],[14] and do on occasion not even address real vulnerabilities [Cor19].

Construct Validity With respect to Section 3.4, I discussed my method with experts of the closed Linux security mailing list. They confirmed validity of my approach to gain information on non-public integration channels.

3.5.2 Consequences

Fixes for Vulnerabilities The primary success criterion for my approach is simple: Can attackers gain temporal advance to design exploits? I argue that this is the case if the patch can be found in public resources before software distributors roll out patches: Reverse engineering of the development process allows for aimed targeting of commits that would otherwise *hide* between thousands of other commits.

As mentioned in Section 3.2, the majority of patches for vulnerabilities first appear in the Linux mainline and stable trees before distributions pick up the relevant patches. From a temporal perspective, patches first appear on mainline and stable trees, and are then integrated by distributions (*cf.* Fig. 3.1, Vulnerability 1). I call this the *mainline first* disclosure model.

However, there is an exception for highly critical vulnerabilities: Before their public disclosure, patches are secretly disclosed to the kernel maintainers of the distributions, which buys them time to prepare their kernel tree to roll out updates (*cf.* Fig. 3.1, Vulnerability 2) as soon as an embargo ends. In this way, a patch can be integrated to the distribution's tree before it is published mainline.

This method ensures that affected systems can receive fixes as soon as the vulnerability is officially disclosed. Yet, this process requires time-consuming and extensive coordination between maintainers of distributions and the kernel community, since a strict temporal publishing coordination is required to make the approach effective. Coordination efforts are even more complex when hardware bugs (such as bugs in speculative execution [Koc+19; Lip+18]) are involved, as multiple operating systems can be affected. This additionally requires cross-community coordination—between different operating systems (variants of BSD, Windows, macOS), commercial and non-commercial vendors,

[14] For instances, processes of commercial companies that must be passed before contributions can be placed in open source projects can contain shortcuts for critical vulnerabilities, and "critical" is equated with "has CVE assigned".

and, under exceptional circumstances [Koc+19], even with compiler manufacturers. This process is therefore only considered in rare cases.

I call this process the *distro first* disclosure model, as patches are integrated by distributions before they are officially published mainline.

According to Kroah-Hartman [Kro20], there is no clear definition of the disclosure process, and no definitive criteria for circumstances when the *distro first* model should be used. As an ad-hoc process, subsystem maintainers decide how to handle a fix: patches can, for example, be routed through maintainer trees to Linus Torvalds, or Linus merges the patch directly, depending on the area of the kernel that was involved.

To give an example, fixes for flaws in the speculative execution model (*cf.* CVE-2019-11135 [Sch+19b] and CVE-2019-1125 [CVE19]) of modern CPUs were entirely developed and rolled out to distributions in private. My approach can still detect that the patches stem from off-list channels as soon as they are available in a repository – but at that point in time, patched binaries are already available for the public. Nevertheless, my method can still provide some valuable temporal advance as the availability of patches does not imply immediate deployment in the field.

However, the *mainline first* disclosure model is used for the majority of fixes for vulnerabilities. As distributions maintain forks of the Linux kernel, and manually select patches that are integrated from mainline, it can take up to months for patches to be integrated (*cf.* Table 3.3). In particular, selecting patches for local forks on a case-by-case basis misses relevant fixes that are available on LTS.

For these cases, the integration process of distribution kernels can be considered as *security by obscurity*, since

(a) the patches do not follow a coordinated disclosure process to distributions to protect affected systems before their official publication, and

(b) the existence of the actual fixes is obfuscated by private discussion and regular development noise.

I hence argue that release strategies of distributions should be reconsidered, as I have demonstrated that distributions are vulnerable for attacks over long periods of time.

Furthermore, I argue that fixes for vulnerabilities should be publicly discussed *after* their disclosure. While preliminary versions for severe vulnerabilities that require *distro first* integration should be developed under the *distro first* model, I recommend using a full disclosure model in all other cases. Early versions of fixes for vulnerabilities can still be discussed on secret lists, but they should be publicly reviewed after their embargo.

A public review process can enhance the software quality of the fix per se—after all, this is the main concern of public discussion—, but can also avoid the inadvertent introduction of *additional* vulnerabilities by fixing one vulnerability, which is unfortunately a real pattern [Cor19]. Public discussion before integration would also defeat my mechanisms, which is eventually desirable.

Code Infiltration and Violations of the Development Process In addition to detecting fixes for vulnerabilities, I also encountered hidden integration channels *besides* security mailing lists, such as maintainers or companies that—systematically or inadvertently—bypass official submission procedures, for instance by direct maintainer commits without external review, or company-internal review. The existence of such channels, shows that trusted individuals can easily infiltrate the project, and secretly introduce malicious artefacts (while this possibility is given, my method allows for finding concrete instances, which is otherwise not possible). The existence of such commits contradicts one of the key promises of an open development model.

I contacted maintainers for subsystems for which I found such patches, and they confirmed the assumption that they integrated code without prior public review. While maintainers are aware of that they sometimes intentionally bypass the process, they were surprised of the magnitude of unreviewed patches—the confirmed "record" is more than 40 per half-year per author, the estimated number for unconfirmed cases is higher.

3.6 Summary

I showed real-world applications that apply the methodology for reconstructing development processes as shown in Chapter 2.

In Section 3.3, I focus on regular development ongoings. With the Linux kernel as a reference for the analysis, I examined the evolution of ignored patches, and the evolution of conformally integrated patches over almost a decade. I showed the decreasing amount of ignored patches, as well as the increasing amount of correctly integrated patches. This underlines an increasing conformance to self-imposed development processes.

These data provide a quantitative basis for argumentation on the evolution and adherence to development processes, as it is, for example, required for the certification of safety-critical systems.

Analyses show, how quantitative software engineering techniques can support safety certification efforts, as they provide solid and evidence-based characteristics of the functioning of development processes. Furthermore, they provide a better understanding of development dynamics in large-scale OSS projects.

In Section 3.4, I showed that reverse engineering of public development processes also allows to detect code that arises from non-public integration channels. My approach removes 96% of regular development noise and points to hot spots that contain fixes for critical security vulnerabilities. With my method, I was able to detect 12 vulnerabilities in Linux in a time window of seven months. I collected responses from all authors that confirm my presumptions. Attackers can use this information to gain temporal advantage, as they can design exploits before affected systems receive patches.

Furthermore, I found evidence that some subsystems and maintainers of the Linux kernel intentionally bypass the regular development process. Therefore we argue that it is possible to systematically infiltrate malicious code to the kernel by bypassing the (mandatory) public review processes. I shared our findings with the Linux kernel community and discussed possibilities of potential mitigations.

Both analyses show the wide range of possibilities that is given by the reconstruction of development processes, and provide a powerful and scalable instrument to support safety certification efforts, as they give evidence-based and quantitative answers to the factual situation of the development process. Quantitative software engineering techniques provide answers to questions that arise from non-functional requirements on a formal basis.

End of Part I

Part II

System Consolidation of Safety- and Mixed-Critical Systems

4 Ideal Hardware Partitioning

> *Simplify, then add lightness*
>
> — **Colin Chapman**
> Automotive Engineer

This chapter shares material with the OSPERT '17 paper "Look Mum, no VM Exits! (Almost)" [Ram+17].

Industrial real-time control systems are often built by extending general purpose COTS hardware components to reduce development effort in time and cost by maximising the re-use of existing solutions. The approach is commonly taken in many industrial domains, for instance automation and control systems [KG19], civil infrastructure projects [Fou19], medical appliances [Kis09] or robotics [Qui+09].

The approach is beneficial if flexibility in system capabilities is more important than potential reductions in cost that can be achieved by mass-producing tailored devices that precisely satisfy requirements, but usually never exceed them. Such scenarios often appear, for instance, in the automotive industry, but are rarely applicable to low-volume domains like medical appliances, industrial control, or even home automation.

Currently, CPUs with multiple physical (and virtual) cores are a de-facto standard in modern COTS hardware for non-microcontroller appliances, and their specifications and capabilities often considerably exceed the least demand for a given set of requirements. Consequently, most systems provide unused, excess hardware resources that can be used to integrate additional tasks.

Systems of increasing complexity and software intensiveness need to deal with workloads that contain tasks at different levels of criticality; the resulting scenarios have received substantial attention during the last decade [BD13], and the *conceptual* advantages and disadvantages of the many possible approaches to build such systems are well researched.

One particular scientific focus of analysis for classically tailored embedded systems is on schedulability of workloads, fault tolerance or optimal work balancing to achieve deterministic and an optimum utilisation of the hardware. In those scenarios, minimum system requirements determine the most cost-effective choices for the hardware. For high volume systems, it is not unusual that special-purpose CPUs or Systems-on-a-chip (SoCs) are designed to satisfy very specific use cases, which then necessitates intensive, software-moderated sharing of resources. With few exceptions [DL17; Mül+14], this resource management is typically implemented by operating systems.

Static Hardware Partitioning A common industrial requirement is to safely run workloads (*i.e.*, self-contained functional aspects of a system) of mixed criticality on such multi-core systems [Bul17b] *aside* Linux (*cf.* Req. 1). As more CPUs than different workloads are available, mixed-critical tasks can be exclusively assigned to dedicated CPUs, and the availability of Linux (and its feature-rich ecosystem) allows for running less critical tasks on the remaining CPUs.

In conventional resource-sharing systems, resource management introduces a significant portion of complexity to the system software stack, and complexity is a well-known source for erroneous behaviour. Furthermore, complex software stacks endanger real-time capabilities of the system, as deterministic behaviour of complex systems is hard to control.

Static hardware partitioning is an approach to *minimise* the amount of software required for resource management. In statically partitioned environments, workloads execute in isolated domains. Workloads have direct access to underlying hardware resources of the system, without the need for sharing them with adjacent domains. To the greatest extend, additional runtime overheads caused by software-based mechanisms to enable overcommitting are obsolete, as workloads directly access hardware resources without any intermediate software layer that abstracts or logically partitions the underlying device or resource. Depending on the workload of a domain, hardware-moderating instances (*e.g.*, OSs that govern multiple workloads) become optional. Static hardware partitioning allows to run digital signal processing (DSP)-like workloads in parallel to a fully-fledged OS.

The superfluity of resource management covers all hardware resources that are typically shared in *conventional* managed systems: CPUs, memory and input / output (I/O) (*i.e.*, peripheral devices). As CPUs are unshared, there is no need time-sharing of CPUs in statically partitioned environments. Hence, there are no software-based scheduling and scheduling-related overheads of a transparent underlying moderator (*i.e.*, OS scheduler or VMM scheduler). As a consequence, systemic overheads that stem from

overcommitting hardware resources and that are typically object to real-time analyses are eliminated.

Of course, the guest software can implement its own scheduling or hardware abstraction mechanisms. Furthermore, workloads can be fully-fledged operating systems (*e.g.*, Linux), for less-critical aspects of the system (*cf.* Req. 1).

Static hardware partitioning requires, besides static assignment of hardware resources to computational domains, strict cross-domain isolation: There must exist means which guarantee that domains can not interfere with neighbouring domains in an unacceptable and unintentional way (*e.g.*, illegitimately access of adjacent resources).

One approach to enable static hardware partitioning and strict isolation is classical hardware-assisted process virtualisation: A hardware component, the MMU, is used to limit the visibility of the physical address space of a execution domain to fine granular segments. Access beyond the scope of view is not possible. In many modern GPOSs, MMUs are the basis for memory-isolated multitasking. However, process virtualisation comes with major limitations. While many hardware devices are accessed via Memory Mapped I/O (MMIO) that can be directly assigned to the virtual address space of the domain, there exist platform devices that can not be partitioned with standard means of process virtualisation. Examples for mechanisms that are not partitionable by the MMU without further extensions are the MMU itself, Programmed Input/Output (PIO)-based port access or access of model-specific registers (MSRs) on x86 systems. Access to such registers must be moderated by an underlying authority, as it might affect the whole state of the system. Additionally, on many architectures and without further measures, interrupt controllers can not be assigned to computing domains, as illegitimate changes could affect other domains. However, especially in real-time environments it is crucial to have fine granular control over those devices.

Furthermore, the concept of memory management or memory virtualisation is often tightly coupled with the concept of hierarchical protection domains, privilege levels, or CPU *rings*. Software that runs in the least privileged level is, for example, not allowed to execute instructions that modify the operating state of the CPU, reconfigure the interrupt controller or access system ports. However, especially domains which execute real-time critical workloads require to execute such reconfigurations without limitations. Running the domain in higher privileges is a non-satisfactory solution as this would give domains permissions to extend its own visibility on hardware, which is contrary to isolation primitives. At this point, solely exploiting the concept of process virtualisation would lead to a massive overhead of moderation of otherwise non-partitionable platform devices.

Ideal Hardware Partitioning Virtualisation extensions of modern COTS CPUs offer opportunities for hardware virtualisation. While process virtualisation makes the strong assumption of the existence of an underlying OS-layer, hardware virtualisation, in contrast, allows for the creation of execution environments that model the full platform of the underlying platform. This includes full access to low-level platform components without logically affecting neighbouring domains. Virtualisation extensions allow for assigning formerly unpartitionable hardware devices to computing domains, while guaranteeing strict isolation. A computing domain sees its own subset of *guest physical* memory, and can take the full advantage of the MMU as it would function on bare-metal, without being virtualised—the *hardware* is *virtualised*. This allows guests for running legacy payloads, payloads that formerly run on bare-metal hardware without the need for major modifications (*cf.* Req. 2).

Those virtualisation extensions have their roots in common enterprise, desktop or mainframe virtualisation [Hei08]. Many of these traditional usages of virtualisation consider the consolidation of services as major motivation, while their focus is on high throughput instead of strict maintenance of determinism and low-latency. Providing functionality for efficient sharing and overcommittment of hardware resources is more important than providing a fully partitionable system without hypervisor intervention. Nevertheless, virtualisation extensions provide promising methods for embedded RT virtualisation.

Maintaining RT capabilities of the platform is the major concern of embedded virtualisation. Hence, I aim at a fully partitionable hardware architecture with zero software overhead due to the intervention of hypervisors or VMMs during runtime: software that is not required during runtime and that can be fully offloaded to hardware components can not cause additional unintended latencies. It only depends on RT guarantees that are given by the hardware. Note that the hardware needs to give RT guarantees in any scenario. A zero-trap hypervisor is the strongest requirement that can be demanded from a virtualised real-time system architecture. No software indeterminisms can stem from the VMM during the operative phase of the system. A zero-trap hypervisor is a requirement for the concept of *ideal hardware partitioning*, which I will define and present in the next section.

With zero traps, I refer to zero hypervisor activity in the operative phase: Of course, there are exceptions for intended hypervisor activity during the startup of the system, that is, the partitioning phase. Furthermore, VMM activity is allowed during bootstrapping of an execution domain, or during (intended, synchronous) maintenance tasks. As a beneficial side effect, zero traps implies less software that otherwise would be required

to handle the traps, as moderation becomes superfluous. This results in a thinner code base, which simplifies certification efforts.

While static hardware partitioning aims to minimise hypervisor overhead, ideal hardware partitioning fully eradicated any unintended hypervisor overhead. With ideal hardware partitioning, I address all requirements on safety-critical resp. mixed-criticality systems (Ref. Section 1.2.1). Yet, it remains an open question if this approach is feasible on modern COTS hardware components. In the next section, I will elaborate fundamental requirements on static hardware partitioning, and introduce the term of and requirements for *ideal hardware partitioning*.

4.1 Requirements on Ideal Hardware Partitioning

In 1974, Popek and Goldberg postulated *Formal Requirements for Virtualizable Third Generation Architectures* in their seminal work [PG74]. They provide a fundamental formal definition of VMMs, and give requirements on their (efficient) implementation. Virtual machines have to satisfy three properties: *equivalence*, *resource control* and *efficiency*.

Equivalence implies that any program must behave the same, whether it is run on a virtual machine or on real hardware – exceptions to this principle are permitted for timing issues, and for the availability of physical resources. This is obviously problematic for the application domains I consider in this work, in particular real-time critical workloads. *Resource control* implies that the virtual machine monitor is responsible for the allocation and moderation of hardware resources. *Efficiency* implies that *most* instructions should be natively executed without the need of hypervisor interception; notably, the definition of "most" is left unspecified.

As the efficiency criteria would exclude, for instance, emulated systems, Smith and Nair [SN05] confine VMM requirements to the equivalence and resource control criterion. In addition, they call VMMs that fulfil the efficiency requirement *efficient VMMs*. Note that the efficiency criterion is satisfied if it only holds for *most* instructions, as by the definition above, which, in turn, necessarily implies that the criterion can only relate to average case efficiency. Citing Popek and Goldberg [PG74]: "Because of the occasional intervention of the control program, certain instruction sequences in K may take longer to execute, so assumptions about the length of time required for execution might lead to incorrect results." Consequently, the definition of efficient VMMs does not inherit timing- and latency guarantees that are otherwise given by the raw hardware. Those guarantees are required by real-time use cases.

4.1.1 Efficiency of VMMs

To provide a more quantitative version of the aforementioned efficiency requirements, consider a measurement M that is performed on a program P which, in turn, defines an observable property of the system. M is, in my case, restricted to measure a temporal duration: The time value t_0 records the starting time of the measurement, and time t_1 records when the measurement is finished (the criterion for "finished" is given by the arrival of some external event, or by a satisfied internal logical condition). The value M_P of the measurement is then given by $M_P = \Delta t = t_1 - t_0$. An ideal system that is not subjected to any other loads than the measurement proper, repeated measurements deliver identical values for all runs: $M_P^{(i)} = m = \text{const.}$, where the superscript (i) indicates the i-th measurement. The criterion does, of course, not hold for systems that provide asynchronously triggered computational services (for instance, performing interrupt service routines, performing scheduling, ...) besides executing the subject program P. Such activities effectively influence the measurement in the form of noise, which I model by a stochastic parameter b, drawn from some probability distribution that must be provided depending on the actual circumstances. M_P^b represents a measurement subject to such noise.

Given a set of measurements $\mathcal{M} = \{M_P^{b,(i)}\}$ of the observable quantity P under noise b, I define that the observable is *transitive* for operation op if $\text{op}(\mathcal{M}_{\text{HW}}) = \text{op}(\mathcal{M}_{\text{VMM}})$ holds ($\mathcal{M}_{\text{HW}}$ and $\mathcal{M}_{\text{VMM}}$ denote that the measure is performed without and under the influence of the VMM). If transitivity for a given operation holds for all observables, I say that the observable itself is transitive.

For a throughput-optimised system, "avg" is arguably the operation of highest interest because the average-case performance is crucial. For real-time systems, "max" is the relevant operation because Worst Case Execution Time (WCET) behaviour is *the* essential characteristic of such systems.

Trivially, transitivity for avg and max is guaranteed by *ideal VMMs* that do not require any traps during the execution of guests. I tighten the definition of *efficient VMMs* and call a VMM an »*ideal VMM*«, if no traps are required during the operational phase:

Definition of ideal VMMs

A VMM is ideal, if all instructions are natively executed during the operational phase. Only maintenance operations may be intervened by the hypervisor. Instructions that cause hypervisor intervention are considered violations.

This means that the additional cost of overhead on ideal VMMs is only limited to the virtualisation overhead (*virtualisation cost*) of the hardware [Dre08].

While the definition of an ideal VMM is hard to satisfy by hypervisors that premise on hardware resource sharing and rely on software intensive hardware overcommitting (implemented by, for instance, device emulation, paravirtualisation [BDF+03] or domain scheduling), it is a realistic goal for partitioned setups. For partitioned systems, I further define:

Definition of Ideally Partitioned Systems

A partitioned system is ideal, if exclusive resource access is granted by an ideal VMM.

Consequently, the *ideal VMM* criterion can only apply to a *subset* of partitions of a partitioned system, for reasons that we discuss in the next section. I call a partition that runs as a guest of an ideal VMM an *ideal partition*.

While ideal partitions can already be achieved with modern virtualisation extensions for constrained environments, complex real-world scenarios still require occasional intervention.

4.1.2 Architectural System Limitations

Modern hypervisors usually try to satisfy the efficiency criterion by using various hardware based virtualisation extensions provided by modern processor architectures (*e.g.*, VT-x [UNR+05], VT-d [Int18b], Secure Virtual Machine (SVM) [AMD05], VT [VH11], ...) that allow for executing *most* instructions natively. MMU enhancements [UNR+05; AMD05]) of those extensions (*e.g.*, page-table virtualisation) assign host physical memory to guests. Address translation of guest addresses to host physical addresses is transparently performed by the MMU and does not require any hypervisor interception—it will only trap in case of access violation. Furthermore, those extensions introduce an OS-superior privilege level in which all hardware resources are accessible. The hypervisor may, for instance, moderate access to shared resources, or directly assign resources to guests.

Other hardware based extensions target the reduction of interrupt overhead [Int18b; AMD05; Int10; ARM16]. *Interrupt remapping* allows to directly route selected interrupts to virtual machines without the need of hypervisor interception. Without interrupt remapping support, interrupts trap the hypervisor, which will dispatch the interrupt, and, if necessary, reinject it to guests. If a device is directly assigned to a guest, or if a platform specific interrupt (*e.g.*, a platform timer interrupt) arrives at the CPU interface,

interrupt remapping will directly send the interrupt to the virtual machine, if running. The aim of those extensions is further reduction of VMM overhead.

Nevertheless, depending on their semantic, a hypervisor may, for instance, be required to moderate the access to sensitive system registers, such as MSRs or different Control Registers (CRs) on x86, or Control Coprocessor (CP) registers on the ARM architecture.

The motivation of any of hardware based virtualisation extension is to reduce the activity of the hypervisor by trap reduction in order to increase the performance of the system – frequently required policy decisions are offloaded to hardware. Nevertheless, the development of those extensions is often driven by throughput-oriented general purpose systems (optimised on the average case): it is sufficient to offload *most* decisions, while for zero-trap hypervisors it is essential that *all* decision can be offloaded to hardware.

During the development of a hypervisor that aims towards zero traps, I elaborated concrete system requirements for ideal partitioned systems. In the next section, I present device specific and platform specific requirements for real-world systems. For any requirement, I present examples that violate the requirement, as well as potential software-based workarounds. Such workarounds are, of course, contrary to the envisioned concept, but required due to hardware limitations as discussed in Section 4.2.2.

4.1.3 Device Specific Requirements

Peripheral devices (*e.g.*, Serial Peripheral Interface (SPI), I^2C, Universal Asynchronous Receiver Transmitter (UART) or ethernet controllers) are essential components of any real-world setup, but they are often ignored and underestimated during systems development under laboratory conditions. Peripheral devices are partitionable entities, if they can be spatially and logically isolated.

Requirement 1: Logical Device Partitioning

The platform must provide means to transparently assign device control to guests.

In their simplest form, a device consists of control structures and a signalling interface. The platform must provide means to assign those interfaces to guests without hypervisor intervention.

On many architectures, device control structures are accessed through MMIO. The MMIO address space of a device is backed by the device's registers. The typical page size of almost all modern architectures is 4 KiB or more, and represents the finest granularity

of memory that can be assigned by the MMU. Hence, devices need to be spatially isolated by the granularity of the page size.

32 bits can be seen as the de-facto lowest limit of physical address space of modern CPUs. While this provides enough space to place different devices on separate pages, hardware manufacturers often place multiple devices on one single page, even different types of devices.

This becomes problematic for hardware partitioning, when those devices need to be assigned to different domains, since only pages can be assigned to guests without the need to trap and dispatch memory access.

A software based workaround to overcome this issue is *subpaging*, a technique where the hypervisor allows for mapping memory areas to guests that are smaller than the page size. The hypervisor traps on any access and only forwards the request if the guest has access permission. Any other access is a violation. This leads to noticeable and undesired slow-downs.

Spatial isolation can be solved by hardware manufactures by assigning different devices to separate pages.

Furthermore, devices need a signalling interface, typically implemented by interrupts. The platform must provide means that interrupts directly arrive at the guests without hypervisor intervention. This technique is called interrupt remapping and is already supported by the virtualisation extensions of an increasing amount of architectures [Int18b; Int10; AMD05; ARM16].

Metafunctions of devices (*e.g.*, device power and reset control, speed, baud rate) are often controlled by secondary devices, such as clock or reset controllers. Such instances must be partitionable on a device scope level. Any modification within a device scope must not affect other devices.

Requirement 2: Hierarchical Autonomy of Devices

Any device metafunctionality must be isolated from other devices and must be logically partitionable.

On many ARM-based platforms, for example, the above mentioned clock and reset controllers are not partitionable without hypervisor interception. They are (a) located on single memory pages, and (b) control all peripheral devices of the system. In addition to this, they (c) may also include clock and reset lines, which are often implemented as complex dependent hierarchical structures.

Currently, guest access to those functions is not possible without complex hypervisor intervention. While guests should be allowed to change device settings during runtime, one workaround (without traps) is to statically set up the device settings during the boot phase of the hypervisor and to forbid any further modification. This can be inconvenient for some[1] scenarios. A complex alternative is to paravirtualise those devices.

Nevertheless, these issues must be addressed by hardware manufacturers by designing device control instances in a partitionable way: One possible implementation would be to place all meta functions of a device to a single page and to reduce inter-device dependencies of hierarchically structured clocks. This provides configurational flexibility, as the page can simply be hidden if a guest shall not be permitted to access these functions. Another approach is to use system specific registers in a standardised manner[2] for device reconfiguration. Whitelists can be used to grant fine-grained permission to functionalities.

However, during the implementation, I observed platforms[3] where access to disabled devices stops the whole platform. This erroneous behaviour was confirmed by hardware manufacturers. Hardware manufacturers must ensure, that erroneous device access from within a domain must not affect the whole platform.

Logical isolation of a device is not limited to clock and reset controllers. Any instance that interacts with a device (*e.g.*, Direct Memory Access (DMA) controllers) must be designed in a partitionable way. This means, usage, access or configuration of the instance must not interfere with any other device or CPU.

4.1.4 Platform Specific Requirements

Requirement 3: Platform Resource Partitioning

System platform resources must be partitionable with respect to their domain affinity.

A CPU interface must not be able to change the macro- and microarchitectural state of a CPU of another computing domain. This includes, for example, power management such as sleep states or frequency scaling, memory management or interrupt delivery.

[1] For example, when reconfiguration of device speed is required during runtime. This can, for example, occur, when multiple SPI devices on the same bus require different speed parameters.
[2] Yet to be defined.
[3] That is, the Nvidia Tegra family – refer to the discussion on the Linux-Tegra mailing list: `https://lore.kernel.org/linux-tegra/f6c3e818-f828-276c-961c-9d61bf4990cd@kapsi.fi/`.

The platform must provide CPU-local control structures, or structures that are restrict to the local computing domain.

Many traps on a platform result from the lack of (full) virtualisability of platform specific resources. Access to sensitive system registers, reconfiguration of CPU power management settings or interactions with interrupt controllers are typical causes for frequent traps that require hypervisor assisted moderation. The hypervisor must ensure that any access must not cause any unintended side effects to other domains. Simple policy-based decisions can be resolved by hardware support.

On x86 platforms, for example, a hypervisor can conditionally trap MSR access, based on permission bitmaps. It allows either unmoderated access to insensitive registers, to either trap on reads or writes, or to trap on both. Platform resource partitioning requires that any interaction with machine specific registers must not leak information of other domains, or affect them.

The x2APIC implements interrupt controller virtualisation support for Intel® x86 platforms. It uses MSR-based register access instead of conventional MMIO-based access. While in a partitioned setup, a hypervisor may allow unmoderated access to insensitive registers, access to sensitive registers, such as the Interrupt Control Register (ICR), must be intercepted. The ICR is used to send inter-processor interrupts (IPIs) to other CPU interfaces. Hence, raw write access must be forbidden, as CPU interfaces of other domains can be addressed. Access must be intercepted by the hypervisor, which will check permissions and forward the request. Other architectures like ARM [ARM13b; ARM16] have similar interfaces that require moderation by the hypervisor.

Interception of platform devices can generally be avoided, if the hypervisor can limit the scope of visibility of CPU local interfaces of its guests. Similar fine-grained conditional register trap that is, for example, based on bitmasks, is possible and already supported for various other CPU CRs on x86 [UNR+05].

Requirement 4: Cross Core Independence

The microarchitectural state of a core must not be affected by neighbouring cores.

Many publications and successful attacks on microarchitectural and speculative attacks underline the risks of shared hardware resources [Koc+19; Van+18; Wei+18; Sch+19a; Min+19; Lip+18]. For real-time performance reasons, and for security reasons [Van+18; Wei+18; Sch+19a], parts of the execution unit must not be shared. Simultaneous multithreading (SMT), for example, violates cross core independence.

Besides SMT, many microarchitectures implement further carriers of potential coverage channels: caches. Last Level Cache (LLC) is often shared across different physical CPUs. Depending on the architecture's cache organisation, this can result in sharing the LLC across different domains. Sharing caches can lead to performance and security issues and should be considered dangerous due to following reasons.

1. On many Intel® CPUs, the LLC is an inclusive cache. This means, the LLC includes all data from lower cache levels. Consequently, the eviction of an entry in the LLC causes the eviction of the entry in all lower levels. Aimed memory traffic generated by a CPU can cause consequent overwrites of the whole shared LLC. As the LLC is inclusive, it will invalidate everything in the L1 cache of all other CPUs [Int15]. With this, a CPU can cause cache misses of another CPU which is assigned to a different domain. This causes unintended and unacceptable slow downs.

2. Furthermore, (shared) caches are a common target for many microarchitectural attacks [Sch+19a; YF14; Wei+18; GBK11]. Yarom et al. have shown that their FLUSH+RELOAD side channel attack can be used to reconstruct the control flow of programs, if two independent processes share the same pages (*e.g.*, shared libraries). In their paper [YF14], they extract cryptographic secrets by the analysis of the control flow. The FLUSH+RELOAD pattern is the foundation of many further microarchitectural attacks [Wei+18; Koc+19; Sch+19a]. Shared caches increase the attack surface.

In partitioned setups, there is no sharing of common physical pages across cores. Therefore, partitioned systems do not benefit from shared caches. This protects them against attacks mentioned in **2.**, but still exposes them to threats mentioned in **1.**.

To overcome the scenario explained in **1.**, Intel® implements the Cache Allocation Technology (CAT) [Int15] as part of their Resource Director Technology (RDT) [Int19a]. CAT allows to partition the LLC by the exclusive assignment of dedicated cache portions to cores. Nevertheless, I believe that their implementation should be considered inconsistent: While a core may only allocate and evict cache lines within its scope," a read or write from a core may still result in a cache hit if the cache line exists anywhere in the LLC." [Int15] This, in turn, opens a new potential (unaudited) attack vector for side channel attacks: an attacker can FLUSH and RELOAD a cache line. The data is then in use by neighbouring cores, if the access time measurement confirms L3 presence right after the flush.

There are too many indicators that shared caches misbehave in certain situations, yet there are no benefits in partitioned scenarios. Platforms should either not support shared caches, or implement cache partitioning in an consequent nonreactive manner.

Shared system resources and traces in the microarchitectural state of a CPU endanger many modern computing systems. It requires careful analysis if and to what degree partitioned systems might be affected.

4.2 The Jailhouse Hypervisor: Philosophy and Architecture

In this section, I present the architecture and philosophy of the Jailhouse[4] Hypervisor, a Linux-based static partitioning hypervisor that aims to implement ideal hardware partitioning. The jailhouse project was initiated by Jan Kiszka [Ram+17], and is subsequently refined as OSS project.

4.2.1 Overview

Jailhouse transforms symmetric multiprocessing (SMP) systems into asymmetric multiprocessing (AMP) systems by inserting "virtual barriers" to the system and the I/O bus. From a hardware point of view, the system bus is still shared, while software is *jailed* in *cells* from where the guest software, so-called *inmates*, can only reach a predefined subset of physical hardware.

Jailhouse, in contrast to all existing solutions, starts with Linux and then uses deferred (or late) hypervisor activation [Rut06] to partition the hardware underneath the already running Linux.[5] Jailhouse piggybacks on Linux, and exploits its capabilities to do the code-intensive heavy lifting of most of the hardware and *then* takes over the system.

Jailhouse is enabled by a kernel module from within a fully booted Linux system, see Fig. 4.1. It takes control over all hardware resources, reassigns them back to Linux according to a configuration of the system, and lifts Linux into the state of a virtual machine (VM). The hypervisor core of Jailhouse acts as VMM. This scheme does not fit into the traditional classification of hypervisors [Gol73] – it can be seen as a mixture of Type-1 and Type-2 hypervisors: At the time of writing this thesis, it runs on raw hardware like a bare-metal hypervisor without an underlying system level, but still

[4] Available at `https://github.com/siemens/jailhouse` under GPLv2.
[5] Rutkowska [Rut06] was the first who used this technique to inject undetectable malware (*i.e.*, a thin hypervisor) into computer systems.

cannot operate without Linux as a system aide to provide initialised hardware. Linux is used as bootloader, but not for operation. However, there exist first prototypes to fully detach Jailhouse from Linux.[6]

Unlike other real-time partitioning approaches (*e.g.*, PikeOS [KW07]) that aim to manage hardware resources and may forbid direct access by guest systems, Jailhouse *only* supports direct hardware access. Instead of using complex and time-consuming (para-)virtualisation [BDF+03] schemes to emulate device drivers and share physical hardware resources, Jailhouse follows an exokernel-like approach [EKO95] in that it only provides isolation (by exploiting virtualisation extensions) but intentionally neither provides a scheduler nor virtual CPUs. As Jailhouse aims for ideal hardware partitioning, only (few) resources that can, depending on the hardware support, not yet be partitioned in that way are virtualised in software (Ref. Section 4.2.2).

No scheduler (and hence, no scheduling overhead) activity is required by the hypervisor, as computing domains are statically assigned to CPUs [LDW11; SK10b]. Nevertheless, operating systems running as guests of the hypervisor may of course implement their own scheduling strategies. As a result of the architectural decisions, the minimality of Jailhouse significantly reduces the code size of the hypervisor—a slim code base is a beneficial for certifiability (*cf.* Req. 2).

For virtualised mixed-criticality environments, it is important to maintain real-time capabilities by design. With static hardware partitioning, it is possible to minimise the hypervisor overhead during operation. Motivated by maintaining real-time capabilities, Jailhouse aims to implement an ideal, *zero-trap partitioning hypervisor*. Under ideal conditions, there are no software interactions of guests with the hypervisor during runtime. Hence, the intended system propagates real-time capabilities that are given by the hardware directly to guests, and completely eliminates further OS/guest-hypervisor interactions. Hence, the software of the VMM can not introduce further software-based indeterminism that endanger real-time capabilities of the system architecture (*cf.* Req. 1). Virtualisation extensions of modern CPUs are used to statically and exclusively assign hardware resources to computing domains to achieve strict and safe isolation of computing domains.

With respect to the cost effectiveness criteria in Section 1.1, many industrial applications cannot give up on the capabilities and feature-richness of Linux in their systems, yet they face increasing demands to simultaneously cope with safety or other certification requirements that are difficult to achieve with Linux [PMB18]. Jailhouse's architectural approach fulfils these needs. However, it can also be considered as an ideal framework

[6] Discussion can be found on `https://groups.google.com/g/jailhouse-dev/c/AYeZHwxGFSc/m/NfNwyxk8BQAJ`.

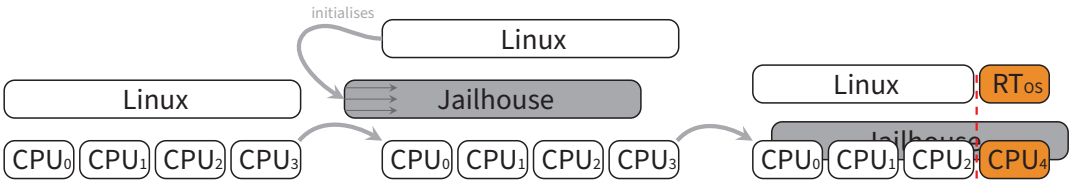

Figure 4.1.: Activation sequence of the Jailhouse hypervisor. After Linux has loaded and started the hypervisor, an additional real-time operating system is started in an isolated critical domain.

to ease the integration of state-of-the art research or experimental systems that solve a specific problem in a novel way with industry-grade solutions based on Linux.

Another advantage of the approach relates to running certified payloads: Many industrial codes are, for historical reasons, designed to run on single-core systems, and would require substantial porting efforts to leverage MC execution environments. Such changes would demand a re-certification of the codes; likewise, a time- and cost-consuming re-certification would be required if workloads are equipped with protection against, for instance, Spectre-type [Koc+19; Can+18] CPU weaknesses. Executing such legacy payloads in a partitioned cell has the advantage that the code does not require protection against said CPU weaknesses, because they are already implicitly required by the partitioning hypervisor. When no code changes are necessary, existing certifications can be retained, which is a clear and substantial commercial advantage.

Jailhouse consists of three main components: a Linux Kernel module that helps to activate, control and operate the hypervisor, userspace tooling that abstracts access to the kernel module, and the hypervisor binary, the actual core of the hypervisor.

When enabling Jailhouse, the kernel module loads the architecture-specific hypervisor binary and a system-specific configuration to a reserved area of memory and calls the hypervisor startup code on each CPU–the point of the handover. Among platform specific information, the system configuration holds information on the system's topology, such as number of CPUs cores, memory regions, Peripheral Component Interconnect (PCI) devices and interrupt controllers. After the hypervisor startup code is executed on each CPU, Linux continues to run as a virtual machine and *guest* of Jailhouse, the so-called *root cell*.

Jailhouse establishes an identity-mapped shadow page tables between the host and Linux: the hypervisor controls the mapping of guest physical pages to host physical pages. Virtualisation technologies (*e.g.*, Second Level Address Translation (SLAT)) allow trap-free access to those pages without hypervisor intervention. The same applies to CPUs: Jailhouse will not overcommit and provide multiple virtual CPUs, Linux will operate on the same physical CPUs as it did before activation. Jailhouse exploits further

virtualisation extensions to minimise the hypervisor's activity. Depending on hardware support (I will later discuss that in detail) Jailhouse will directly re-route interrupts to guests: interrupts are delivered to guests without hypervisor intervention. Support for I/O memory management units (IOMMUs) protects guests against memory violations by DMA-capable hardware devices.

It is hard and code-intensive for hypervisors to support the myriad of existing different hardware in their system. Linux, on the contrary, is an extremely feature-rich operating system concerning hardware support. Jailhouse takes this advantage and hijacks Linux. The untypical *deferred* activation procedure of the VMM has the considerable practical advantage that the majority of hardware initialisation is fully offloaded to Linux, and Jailhouse can entirely concentrate on managing virtualisation extensions. The direct assignment of hardware devices allows Linux for continuing executing as before. Unlike other partitioning approaches (for instance, [LWM14]), Jailhouse does not require any specific device drivers except for minimalist, optional debug helpers, for example, a simple UART driver. Hence, similar to the exo-kernel [EKO95] approach, Jailhouse is an *exo-hypervisor*, with the difference that the skeleton (*i.e.*, the minimalist VMM) is modelled by the corpus (*i.e.*, the operating system, Linux), and not vice versa.

It is a deliberate design decision that Jailhouse does not share physical hardware resources across guests. Especially in mixed-critical environments, Jailhouse makes the realistic assumption that every computing domain is assigned to its own dedicated hardware resources. To create such isolated domains (in Jailhouse lingua called *non-root cells*), Jailhouse removes hardware resources (*e.g.*, CPU(s), memory, PCI or MMIO devices) from Linux and reassigns them to the new domain. Similar to the system configuration, topological information of non-root cells are specified in *cell configurations* that describe the assigned resources.

Linux releases the hardware if it has previously been in use, offlines selected CPUs and calls the hypervisor to create the new cell. This includes physical CPUs that are assigned to the new domain: the configuration of a partition consists at least of one CPU and a certain amount of memory. The *cell inmate*, a secondary operating system or a bare-metal application, can be preloaded by the root cell before the domain is started (*kicked off*). While cells may share memory regions, subsequent memory access by the root cell can, of course, be disallowed by the hypervisor, which prohibits inadvertent modifications. Other resources, like PCI devices, MMIO devices or I/O ports, can be exclusively reassigned to the new guest as well. Non-root cells can dynamically be created, destroyed (*i.e.*, resources are assigned back to the root cell) or relaunched.

Virtualisation extensions (See [ARM13a; AMD05; UNR+05] for the four major architectures ARMv7 with Virtualization Extensions (VE), ARMv8, Intel® 64-bit x86 with VT-x

and VT-d support, as well as amd64 with SVM support) guarantee spatial isolation: any access violation, for instance illegal access across partitions, *traps* [PG74] the hypervisor, which eventually stops the execution of the faulting CPU. Certain instructions executed by guests cause traps and must be handled by the hypervisor.

Since Jailhouse only remaps and reassigns resources, the ideal design conception is that – besides management – it does not need to be active after setting up and starting all guests, and only intercepts in case of access violations: "Look Mum, no VM Exits!" [Ram+17].

However, hardware is not (yet) perfectly suited for this approach, so on current hardware, the following circumstances still require intervention by the VMM:

- Interrupt reinjection (depending on the architecture, interrupts may not directly arrive at guests)
- Interception of non-virtualisable hardware resources (*e.g.*, parts of the Generic Interrupt Controller (GIC) on ARM)
- Access of platform specifics (*e.g.*, accessing Control Coprocessor 15 (CP15) or Power State Coordination Interface (PSCI) on ARM)
- Emulation of certain instructions (*e.g.*, `cpuid` on x86)

The following traps are unavoidable, and not contrary to the concept, as they only occur in case of *jailbreak* or cell management:

- Access violations (memory, I/O ports)
- Cell management (*e.g.*, creating, starting, stopping or destroying cells)

These interceptions introduce overhead and latencies – virtualisation, of course, comes at a cost. Even if there were no exits, a second-level page table walk, for example, introduces additional latencies for memory access [Dre08]. In Section 5.2, I exemplarily present the evaluation of fundamental microbenchmarks: the additional latency of interrupt reinjection on ARM platforms, and the influence of mitigations for Spectre-like attacks [Sch+19a; Min+19; Van+18; Wei+18; Lip+18; Koc+19; Can+18] that partly base on hypervisor intervention.

Despite the strict segregation of resources across guests, Jailhouse still allows cells to share physical pages. Besides enabling inter-cell communication, the mechanism also allows for sharing MMIO pages, which, if desired, allows for accessing hardware resources from within multiple domains. Such concurrent access is, however, not arbitrated by Jailhouse and needs to be addressed appropriately by the guests.

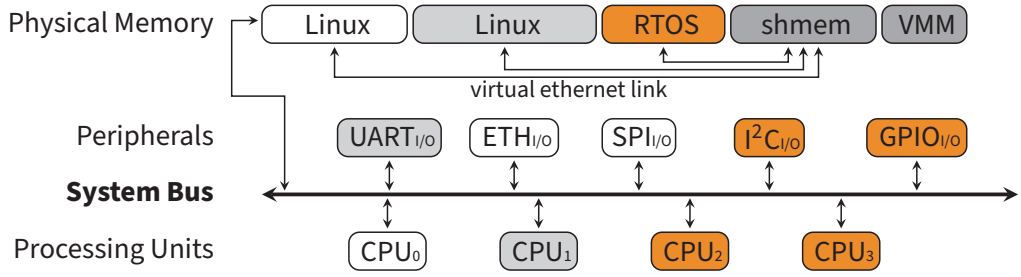

Figure 4.2.: Ideal Hardware Partitioning: Hardware is fully, partitionable; there are no shared resources.

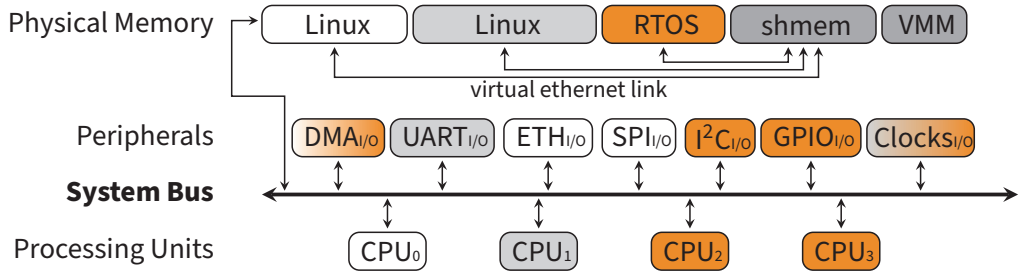

Figure 4.3.: Hardware Partitioning on Real Hardware: Certain devices (*e.g.*, DMA or clock devices) are not fully partitionable; sharing must either be avoided or requires moderation.

Fig. 4.2 and Fig. 4.3 show a possible partitioned system layout for three cells: the Linux root cell (white), an additional Linux non-root cell (light gray) and a minimalist real-time operating system (orange). The left Fig. 4.3 shows an *ideal setup:* devices are not shared across domains. However, hardware limitations prevent ideal conditions under certain circumstances (Ref. Fig. 4.2). Those exceptional cases are discussed in Section 5.1.

Communication between cells is realised by shared memory regions, together with a signalling interface. This minimalist design requires no additional device driver logic in the hypervisor. Depending on the hardware support, it is implemented based on a thin virtual PCI device through Message Signalled Interrupts (MSI-X) or legacy interrupts. A guest may use this device to implement, for example, a virtual ethernet, block, or console device on top of it. On systems without PCI support, Jailhouse emulates a generic and simple PCI host controller. Emulation is chosen in this case, as PCI provides a configuration space: The PCI device identifies itself and its capabilities. This enables, if supported, automatic configuration in guests, and the virtual PCI host controller results in only six lines of code and does not increase the overall code size.[7]

[7] `https://github.com/siemens/jailhouse/commit/7b9f373dcfc14a4951928c43ded9c02b9f1ac02c`

4.2.2 Hardware and Software Support

Architectures Jailhouse supports four different architectures. On x86, Jailhouse supports Intel® x86 64-bit with VT-x and VT-d support [Int19b; UNR+05], as well as amd64 with SVM [AMD05] support. On ARM platforms, Jailhouse supports 32-bit ARMv7 with virtualisation extensions (VE) [ARM13a], and 64-bit ARMv8 [ARM20a] platforms. It is worth mentioning that despite the fact that Jailhouse supports four different CPU architectures, which goes beyond what is provided by many experimental or research systems, its minimalist approach results in only a few thousands lines of code for the core parts. This simplifies certification processes, but allows developers to concentrate on important issues without spending time on providing a never complete number of device drivers that are required to make the system usable in realistic environments. The simplicity of the core is a good basis for a formal verification of the hypervisor, similar to the formal verification of related system software [Kle09].

Operating Systems Several operating systems apart from Linux are already available as Jailhouse guests (L4 Fiasco.OC on x86 [Bar16], FreeRTOS on ARM, Erika Enterprise RTOS v3 on ARM64). The Real-Time Executive for Multiprocessor Systems (RTEMS) real-time operating system for the ARM architecture has already been successfully ported with very limited effort – modifications are mostly required for platform specific board support. The simplicity of porting systems suggests an opportunity to expose feature-incomplete research approaches to realistic industrial use-cases by combining them with an industrial grade base. Besides running existing common operating systems, Jailhouse provides an own *inmate* library that allows for running minimalist applications.

4.3 Cross-domain Protection Against Speculative Execution Exploits

A strong industrial requirement for real-world systems is that unwanted, unintentional interference between domains of different criticality must be absent, that is, the system must guarantee freedom from interference. In many scenarios, freedom from interference must also guarantee the unintended leakage of data from neighbouring domains.

The class of recently discovered attacks on speculative execution [Sch+19a; Min+19; Van+18; Wei+18; Lip+18; Koc+19; Can+18] that have not only received substantial academic consideration, but have even reached the attention of the general public, high-

lights the risks of hardware resource sharing, particularly when workloads of mixed criticality are scheduled on the same physical execution unit. All variants of the above-mentioned speculative execution attacks are, roughly speaking, side-channels on speculative execution of CPUs sided by timing attacks on CPU caches [GBK11; YF14; OST06]. They open a covert channel which can be used to leak confidential data between payloads of different criticality by the exploitation of fundamental CPU primitives. This violates or subverts many guarantees that are given by formerly trusted hardware units on which architectures of mixed-criticality solutions usually rely upon.

Performance, throughput and efficiency of almost all modern CPUs rely on aggressive microarchitectural optimisations. Pipelining, speculative execution and out-of-order execution are prominent and effective optimisation techniques.

Out-of-order execution allows single CPUs to efficiently reorder instructions to achieve an optimal utilisation of the CPU pipeline. CPU pipelines allow parallel execution of different stages of multiple independent instructions. Branch prediction is a speculative execution technique, to achieve optimal utilisation of the CPU pipeline. A CPU that implements branch prediction speculatively executes instructions in advance of conditional branches with yet unknown results. It may execute instructions that, in the end, may not be needed or that are not allowed. High utilisation of all execution units in parallel is one of the elementary reasons of the high performance of modern CPUs. However, those techniques can be counterproductive in real-time environments [Cul+10].

Naturally, speculative execution inherently leads to erroneous decisions. Thus, executed mispredictions are transparent to users as they are rolled back to preserve an accurate external state. However, they leave microarchitectural traces in the internal state of the CPU that open potential covert channels. Misdirection in combination with internal state analysis allows an attacker conclude the external state of the CPU. In 1995, Sibert et al. point to the potential existence of such microarchitectural state dependent covert channels [SPL95].

Two decades later, in the beginning of 2018, researchers [Lip+18; Koc+19], independent of each other, present a whole new class of microarchitectural attacks: the family of Spectre attacks. Since then, many researchers found new methods or variations of attacks on speculative execution of CPUs to leak otherwise protected, unreachable information.

All Spectre attacks and their variations violate fundamental guarantees on the confidentiality of data that is given by (core-local) protection mechanisms of a CPU. Software based solutions in operating systems and system firmware, as well as processor microcode updates are required to mitigate attacks. Many of those numerous mitigations are cost-intensive, and endanger the real-time capabilities of a system.

In the following, I will give an overview of speculative execution attacks (*i.e.*, Spectre, Meltdown, Foreshadow and Microarchitectural Data Sampling (MDS)), and analyse their impact on real-time systems in general, as well as on static and ideal hardware partitioning in particular. Later, in Section 5.2.4, I will evaluate and quantify their impact on selected systems.

4.3.1 Attacks and Mitigations

Spectre One pattern of Spectre attacks is to mislead execution units to perform dependent loads. Transient execution attacks [Koc+19] try to speculatively load memory where the address *depends* on the offset of a secret (dependent loads). This intentional misguidance leads to mistaken speculative execution and the external state is rolled back. While this preserves external consistency, attackers can draw conclusions on the secret by analysing the internal state that was modified by the execution of transient instructions. Many attacks analyse the state of caches to leak information on the internal state: evaluation of memory access time (*e.g.*, FLUSH+RELOAD attacks [YF14]) to adjacent memory cells can be used test if data is present in caches. A valid cache line can be loaded through a transient execution. The number of the warm cache line carries the original secret.

Those attacks are mitigated by CPU microcode and system firmware updates that introduce speculation barriers, by compiler-assisted conversion of indirect branches to return statements, and by OS-based protection against speculation on user-controlled data in kernel space and others.

Meltdown A similar attack is Meltdown [Lip+18] (aka. Spectre v3 or Rogue Data Cache Load). It exploits out-of-order execution to bypass illegal memory access to areas protected by MMUs on many Intel® and ARM processors. While access to protected memory will cause an exception, out-of-order execution bypasses MMU-based protection mechanisms. Again, the secret can indirectly be used to warm up a cache line that remains as an artefact of the internal microarchitectural state. Meltdown is able to leak data from present, but protected privileged pages (*e.g.*, data from kernel space).

For performance reason, many operating systems share the same page table for user and kernel space. Kernel space pages are marked as privileged and not accessible from user space. This saves cost-intensive page table switches on privilege level switches. Meltdown overcomes this security barrier.

It is mitigated by Page Table Isolation (PTI), the isolation of user-space and kernel-space pages. The page tables that are used in user-space only contain a minimal mapping of privileged kernel-space code that is required to hand over to the actual kernel-space mapping: A second mapping that, besides user pages, contains a full visibility of the kernel-space. This requires page table switches on every privilege level switch.

Foreshadow / L1TF Foreshadow [Van+18] and Foreshadow-NG [Wei+18] are attacks on Software Guard Extensions (SGX) and MMUs of modern Intel® CPUs. Foreshadow allows to read secret data from SGX enclaves, and Foreshadow-NG (also known as L1TF or Level 1 Terminal Fault) allows to read any data from the core-local level 1 cache. Foreshadow-NG exploits additional design flaws of MMUs: Intel® MMUs speculatively use physical addresses of invalid page table entries (*i.e.*, entries with cleared 'present'-bit). Intel® is hypothesised to "implement L1 tag comparison in parallel with the address translation process for performance reasons" [Wei+18]. While access to invalid page table entries raises an exception (*i.e.*, Terminal Fault), the data of the L1 cache is already used for transient out-of-order execution of the following instructions. Analogously to other microarchitectural attacks, change of the internal microarchitectural state is used to leak secrets. L1TF is able to leak any data that is present in the core-local L1 Cache.

User-space processes may speculate on previously available pages that are not present (*e.g.*, swapped pages). Operating systems running as virtual machines are able to (a) leak data from the hypervisor and (b) leak data from other virtual machines that are scheduled on the same core and leave data traces in the L1 cache. Secrets can also leak through neighbouring SMT siblings as they share the L1 cache.

To mitigate L1TF, operating systems implement Page Table Entry (PTE) inversion and conditional cache flushes. PTE inversion applies a bitmask to the physical address of unpresent pages with the intention to point to invalid physical addresses. This protects operating systems from users that speculate on unpresent pages. To protect hypervisors against malicious virtual machines, and to protect virtual machines against each other, operating systems implement (expensive) conditional L1 cache flushes on privilege level switches. A full prevention of cross-VM exploits requires to disable SMT.

Microarchitectural Data Sampling Rogue In-Flight Data Load (RIDL) and Fallout present MDS attacks that target CPU-internal buffers (*e.g.*, Store Buffer, Fill-Buffer or Load-Port) of Intel® CPUs. During its execution, a victim process utilises CPU internal buffers with private data. Later, the scheduler of the operating system replaces the victim process with the attacking process. "When the attacker also performs a load, the processors

speculatively uses in-flight data from the Line Fill Buffers (LFBs) (with no addressing restrictions) rather than valid data" [Sch+19a]. Covert channels, for example the FLUSH+RELOAD attack, finally reveal the secret of the victim process.

Recent Intel® CPU microcode versions patch instructions to perform flushes of various exploitable internal CPU buffers. For virtualised environments, an alternative, yet more cost-intensive mitigation are L1D cache flushes. Nevertheless, this is the preferred mitigation for systems that are vulnerable to L1TF, as they need to conditionally flush L1 caches on the same paths in either case. "The mitigation is invoked on kernel/userspace, hypervisor/guest and C-state (idle) transitions." [KRNL20]

While store buffers are partitioned across SMT threads, entering or leaving sleep states repartitions the buffers and data can be exposed between SMT threads. Depending on the workload, full mitigation requires SMT to be disabled as fill buffers are shared between SMT threads. [Kle19]

4.3.2 Jailhouse and Speculative Execution Attacks

All known speculative execution attacks exploit CPU-local interfaces. At the time of writing, there are no known speculative execution attacks across physical CPU boundaries. To attack the victim, it needs to temporarily share the same core with the attacker. Naturally, CPUs can not leak data they do not know or data they never see.

To isolate domains of different criticality, Jailhouse exclusively and statically assigns CPUs to its guests, that is, to different execution domains. By design, Jailhouse does not schedule domains. It has no means built in to share a physical CPU between multiple guests. This differentiates Jailhouse significantly from conventional hypervisors that are used in cloud environments.

This fundamental architectural decision provides a strong cross-domain protection layer against speculative execution attacks. Nevertheless, the following scenarios have to be carefully assessed: (a) Inter-Guest attacks, and (b) Attacks on the hypervisor.

Inter-Guest Attacks While Jailhouse does not schedule guests, a guest (*e.g.*, an operating system) may, of course, schedule different processes. Hence, malicious code can be used to leak secret information of other processes of the same execution domain.

Nevertheless, if a domain needs protect itself against attacks from within the domain, the operating system can implement countermeasures as it would otherwise be required on a regular VMM-less bare-metal configuration as well. On the ARM64 architecture,

speculation barriers are implemented by the secure monitor running in Exception Level 3 (EL3) [ARM20b] (the hypervisor runs in the lesser privileged Exception Level 1 (EL1)): A context switch from EL1 to EL3 acts as speculation barrier. Calls from EL1 (OS / kernel) of a guest to EL3 require interception and moderation by the hypervisor. On affected ARM64 platforms that implement secure monitor-based speculation barriers, Jailhouse calls anti speculation barriers on every trap. If a guest decides to implement mitigations, the hypervisor - besides the secure monitor - causes additional latency for forwarding the mitigation. A precise evaluation of the effects of those mitigations will be given in Section 5.2.4

However, based on the threat model of a specific domain, it is the decision of the guest whether further inter-guest mitigations are required. If a guest does not require mitigation of Spectre attacks, the existence of the hypervisor will not cause any overhead.

Hypervisor Attacks One design goal of Jailhouse is to setup hardware partitioning, which, ideally, requires no further hypervisor interception during regular operation. Conceptually, the hypervisor should only be active during its boot and partitioning phase, and only handle unrecoverable critical exceptions during its operational phase. In an ideal trap-free setup, the hypervisor will never get active in the operational phase.

This means that the architecture of Jailhouse inherently provides strong protection of the hypervisor: speculative execution attacks can only work in cases, where the victim actively triggers higher execution levels to execute code, which leaves traces in the microarchitectural state. Hence, there is no threat for a zero-trap hypervisor, as it will only get active in critical, unrecoverable situations.

While the zero-trap goal is already achieved for some use cases on Intel® x86 systems, it is generally limited by current hardware support (Ref. Section 4.1). However, to implement Jailhouse on common architectures, the hypervisor needs to intercept or moderate certain situations that depend on the target architecture's virtualisation capabilities. Furthermore, Jailhouse implements a slim synchronous hypercall interface for management tasks. This involves hypervisor activity that can potentially be used in speculative execution attacks. This attack surface requires careful assessment.

A CPU can only leak what it can see. In case of Jailhouse, this is includes its binary code, configuration, and sensitive guest state information. As the hypervisor is developed as an Open Source project, hypervisor binary code does not need protection. The system configuration contains partitioning information and information on the platform's topology. This does not contain secret data that needs further protection.

Malicious guests may use or even synchronously control hypervisor activity to prepare for speculative execution attacks. By design, Jailhouse only exposes a minimum attack surface to guests as it only maps a small subset of guest pages into its address space that is required to perform its duties. Jailhouse maintains isolated core-local address spaces and does not share CPU private pages across CPUs. Core-local CPU state is not visible to other CPUs. Only a small set of uncritical management information (*e.g.*, the hypervisor state) is shared across all CPUs. Because of its simplicity, address space isolation was implemented with reasonable effort. At the time of writing this thesis, Linux's Kernel-based Virtual Machine (KVM) undergoes efforts of implementing a similar isolation strategy.[8]

The design of Jailhouse prevents leakage of sensitive hypervisor data across domains.

Attacks on SMT SMT is a further technique to optimally utilise available hardware resources. SMT transparently exposes multiple logical CPU interfaces to users, while parts of the underlying physical units are still shared (*e.g.*, L1 caches). Execution units can be shared or duplicated between logical threads.

Sharing of execution units may lead to mutual contention between different threads. While SMT increases overall performance and throughput of a system, contention causes unintended latencies, which have negative impact on the real-time behaviour of systems. Hence, I share the opinion of [PRT20] and [Rie16] to disable SMT in any case, in order to maintain real-time capabilities.

However, if SMT remains active, logical threads are vulnerable to attacks that exploit shared execution units or shared caches. Hence, I recommend to only allocate threads of physical CPUs to the same execution domain. It is then the decision of the guest if further inter-guest OS-based mitigations are required.

In any case, in the Jailhouse architecture, all secret information remain in guests on isolated CPUs. Under ideal conditions, no secrets remain in the hypervisor.

In Section 5.2.4, I will investigate the cost of mitigations for spectre-like attacks on bare-metal real-time systems in general, and on virtualised environments in particular.

[8] See `https://lkml.kernel.org/lkml/20190514070941.GE2589@hirez.programming.kicks-ass.net/T/`.

5 Evaluation and Discussion

> *Controlling complexity is the essence of computer programming*
>
> — **Brian W. Kernighan**
> Computer Scientist

5.1 Hypervisor Activity

The goal of Jailhouse is to achieve an ideal, zero-trap hypervisor. However, features of current state-of-the-art hardware architectures and platforms do not yet provide full support to achieve this goal in all situations. Depending on the architecture, there exist different circumstances where the hypervisor needs to intervene. In this section, I analyse remaining hypervisor activity.

I will investigate different types of hypervisor activity, and their impact on real-world applications. The measurements serve as a reference for real-world deployments of the hypervisor. They assist to assess the suitability of the approach in real-world appliances.

5.1.1 Common Hypervisor Activity

On all target platforms, it is obvious that VMM activity is inherently required during the startup and the partitioning phase. For domain management, Jailhouse uses a thin hypercall interface. In the following, I will explain situations where hypervisor intervention is required, and differentiate between unique hypervisor intervention during the startup, and hypervisor intervention when the system is in an operational phase.

On all architectures, it is unavoidable that the hypervisor moderates access to certain MMIO regions. On real-world systems, two separate hardware devices are frequently located on the same physical page, and if both devices are assigned to separate domains,

the hypervisor must moderate the access, as the page size is the finest granularity for trap-free memory assignment to a cell. All platforms share this issue to some extent, violating **Requirement 1: Logical Device Partitioning** in Section 4.1.3. To overcome such obstructive platform topologies, Jailhouse implements the proposed workaround—*subpaging*.

5.1.2 Hypervisor Activity on x86 Platforms

While some real-world workload types already achieve zero-traps on x86 platforms, other workloads may lead to hypervisor activity: on x86, the hypervisor needs to trap MMIO-based access to the Advanced Programmable Interrupt Controller (APIC), as the hypervisor must ensure that no other domains will be affected by the access, for example, a domain must not be allowed to send IPIs to neighbouring domains. As the APIC has no built-in means to understand the semantic of partitioned hardware, access needs to be moderated by the hypervisor.

However, some hypervisor activity due to the moderation of access to interrupt controller registers via MMIO can be avoided by the use of the almost trap-free MSR interface if the platform provides the successor of the xAPIC – the x2APIC. The MSR interface of the x2APIC provides register-granular direct guest assignment. The reason for, and an evaluation of remaining hypervisor activity (access to the ICR register of the APIC) will be quantified in Section 5.2.3. The occurrence of those exits depends on the workload type, and is bypassable in domains with one exclusive core.

There are two further MSRs that require moderation by the hypervisor: Writes to Page Attribute Table (PAT) and Memory Type Range Register (MTRR). Those are registers to control caching attributes of memory ranges. The occurrence of access to PAT and MTRR typically only occur once during the boot-phase of secondary guests and hence introduce negligible overhead.

Besides MSR and MMIO, x86 provides yet another PIO interface to control peripheral devices: port-mapped I/O (PMIO)—a relic from a distant past. In early ages of the x86 architecture, address space was limited, and memory was an esteemed resource. PMIO introduced a secondary address space with dedicated use for device I/O in order to save physical address space. x86 is an architecture that is well-known for accumulation of legacy issues for over four decades. In this manner, and in the era of memory plethora, PMIO is still excessively being used for controlling platform-specific peripherals. Among platform UARTs, PS/2 keyboard, and real-time clocks, PMIO is used for accessing the

PCI configuration space. While virtualisation extensions allow for selective direct assignment of ports to guests, access to, for example, the PCI configuration space must be moderated in order to limit the scope of visible PCI devices of a domain.

This applies to all methods of conduction (*e.g.*, purely MMIO-based access to the configuration space on ARM architectures), as access to the PCI configuration space exposes control over all PCI devices on the bus. The hypervisor must ensure that only devices within the domain's scope are accessed, and it must prohibit changes of PCI capabilities that can affect other domains (*e.g.*, power management settings).

The x86 instruction `cpuid` discovers the type and features of a processor. It is used by virtualised guests to discover the type of a hypervisor. Hence, Jailhouse traps execution of this instruction and presents its existence to the guest. `cpuid` instructions typically do not occur in the operational phase of the system. It is worth mentioning that `cpuid` can be used as a debugging instruction, as it is executable in user-space but directly traps to the hypervisor without prior activity of the OS.

Non-maskable interrupts (NMIs) arrive at the hypervisor and cause additional activity of the VMM. The hypervisor will then reinject the NMI to the guest. NMIs only occur under rare conditions and signalise a severe hardware condition (*e.g.* chipset or Error-Correcting Code (ECC) memory errors). A delay of the propagation of a NMI does not additional dangers to the system, as the safety strategy of a system must include the transition to a safe state in case of system failures.

The alignment check (#AC) is an optional feature on x86 systems. An exception is thrown, when the processor detects an unaligned memory operand. While this exception can conditionally be caught by the hypervisor, CVE-2015-5307 enforces hypervisors to trap on the exception. An infinite stream of #AC exceptions caused by a malicious guest causes the microcode to enter an infinite loop. No other interrupts will arrive at the core. Jailhouse implements the same strategy as KVM, to unconditionally trap on #AC.[1] This will only have an effect on cores that execute a malicious guest.

On x86, the CR0 register is used to globally (de-)activate cache. It is obvious, that access to those registers must be moderated. Additionally, the CR4 registers contains information of the existence of a VMM, and Jailhouse will present its existence to guests. Hence, access to those registers is trapped. The `xsetbv` instruction on x86 is used to set extended control registers—registers that control arithmetic extension of the processors, such as Streaming SIMD Extensions (SSE) or Advanced Vector Extensions (AVX). The execution of this command causes a trap to the hypervisor, which will forward the

[1] Ref. `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=54a20552e1eae07aa240fa370a0293e006b5faed`.

request. `xsetbv` instructions typically only occur during the boot phase of the system, and not in the operational state. The abovementioned instructions are examples that violate **Requirement 3: Platform Resource Partitioning**, and **Requirement 4: Cross Core Independence**.

5.1.3 Hypervisor Activity on ARM Platforms

ARM platforms share architectural limitations mentioned in Section 5.1.1 on MMIO-based devices that are located at the same physical page. Additionally, devices on ARM platforms can have inner fragmentation: System often contain unpartitionable platform controllers, such as clock or reset controllers. Clock controllers are used to adjust speed and frequency of a peripheral device (*e.g.*, for setting the frequency of SPI devices). Reset controllers are used to specifically reset single devices within a platform. Those, often hierarchically organised platform management controllers are contiguously mapped to MMIO address space, and suffer from inner fragmentation: Depending on the specific implementation of the controller, peripheral devices are often controlled with single bit operations within the controller. Furthermore, devices or reset lines can be hierarchically organised in groups, and modifications can have cascading effects on the whole platform across domains, violating **Requirement 2: Hierarchical Autonomy of Devices**

However, isolation criteria require that a domain must not have any side effects to other domains. On recent platforms, Jailhouse implements two possibilities to overcome this issue. In the Configure & Freeze-approach, the root-cell will configure all devices of the platform to the state that is later required in the operational phase. After the setup, the state of the controller is frozen: the hypervisor will prevent subsequent access to platform controllers. This approach is suitable for systems, where clocks do not require further adjustment in the operational phase. The alternative is to implement a thin paravirtualised layer in the hypervisor that implements the semantic of the specific underlying platform controller, and moderates access.

In addition to that, the hypervisor needs to address the interrupt system of ARM platforms: besides vendor-specific interrupt controller designs, ARM provides generic specification for interrupt controllers. Jailhouse supports the broadly used GICv2 and GICv3 specification. Both specifications require to reinject external interrupts: Interrupts will first arrive at the hypervisor, and need reinjection to the guest. I will discuss and evaluate this architectural limitation in detail in Section 5.2.2. The specification of future versions of the GIC [ARM16] address this limitation and implement support for interrupt remapping, which is beneficial for the proposed approach.

For IPIs, ARM platforms use software generated interrupts (SGIs) that are issued via the GIC. Similar to x86 platforms, the hypervisor must ensure that the SGI does not address a CPU interface outside the scope of the domain. Hence, access to the interrupt distributor requires interception and moderation.

On ARM platforms, the PSCI firmware runs at highest privilege level. It is used for platform control, for example, to enable or disable secondary CPUs. As a side effect and as later explained in Section 5.2.4, firmware calls can also be exploited to mitigate Spectre v2, as they implicitly act as speculation barriers. The hypervisor will trap on firmware calls that are issued by the guest, as it must ensure that firmware calls do not address resources outside the scope of the domain. However, if the guest does not require mitigations against Spectre (Ref. Section 5.3.3 why mitigations are not required in many scenarios), firmware calls are typically not conducted in the operational phase.

On 32-bit ARM platforms, the hypervisor must intercept access to the platform co-processor (CP15), as access to registers of the CP15 may have effects across domains.

While there are still various reasons for the hypervisor to intervene in certain situations, most trap types do not occur in the operational phase of the system and are especially required during the boot phase of systems. Additionally, Jailhouse is able to achieve zero traps for real-world scenarios under x86 and it can be observed that future hardware extensions are beneficial to achieve zero traps in partitioned setups (*e.g.*, GICv4).

5.2 Evaluation

An important industrial requirement on real-world systems is that it must be possible to guarantee (under a reasonable definition of assurance) that *partitioning* implies the maintenance of determinism within a computing domain and *freedom from interference* between the partitioned domains.

Not only on partitioned or virtualised, but also on conventionally scheduled systems, the surface of potential cross-domain interference is determined by the degree of interaction between different computing domains. This includes interactions between tasks, tasks and operating systems, operating systems and an underlying hypervisor, and interference of the system's firmware (*e.g.*, non-maskable interrupts caused by System Management Interrupts (SMIs) or firmware-assisted mitigations for Spectre-like attacks).

As mentioned before, the aim of Jailhouse is to implement ideal hardware partitioning, and hence to eradicate the activity of the hypervisor during operation. Though this

would be possible in theory, the sole existence of a hypervisor introduces additional latencies [Dre08] that do not exist without a VMM. For example, shadow page tables may introduce additional memory access latencies because of additional steps in the page table walk in case of Translation Lookaside Buffer (TLB) misses.

To evaluate and determine the (real-time) performance of the hypervisor, several environmental conditions must be considered. It is hard or even impossible to quantify the hypervisor overhead with one single scalar. This results in a set of microbenchmarks that serve as a basis for a specific decision on the qualification of the system's architecture.

For all benchmarks, single-shot measurements do not allow to draw any conclusions on the behaviour of the system. Microbenchmarks should be repeated under certain environmental conditions, such as the actual existence of a hypervisor, and the particular frequency of a certain measurement together with variations of the utilisation of other guests [Mau+21].

Next, I describe considerable measurements and present three quintessential microbenchmarks and compare different platforms: the hypervisor-induced interrupt latency on an ARM platform, the moderation of the ICR register of the interrupt controller of x86 platforms, and the impact of Spectre mitigations on virtualised systems. These are three sources of workload-dependent hypervisor activity and hence, hypervisor noise.

5.2.1 Hypervisor Overhead

The following paragraphs give an overview of potential sources of interference and noise. It is important to remark that benchmarks do not measure the overhead of the hypervisor, but the overhead of the hypervisor when running on a *specific* hardware platform. Nevertheless, those measurements give insights of the impact due to the existence of a VMM.

Hypercalls One typical benchmark for hypervisors is the cost of hypercalls, as hypercalls are often used to implement paravirtualisation of devices. In case of Jailhouse, hypercalls do not need to be considered, as they are only used for cell management purposes, and never occur in the operational state or in hot paths.

Shared System Bus Jailhouse implements an AMP system. Different guests asynchronously access memory, and memory or I/O access may be serialised by hardware. Though starvation does not occur on supported architectures, heavy utilisation of memory or I/O busses may lead to significant slow downs of guests. While this problem is well-known for SMP applications, its impact must be evaluated when asynchronously executing multiple payloads that were designed for single-core platforms.

Interrupt Latency While on x86 architectures, VT-dallows for direct trap-free remapping of MSI-X interrupts to guests [Int18b], many ARM platforms miss equivalent extensions. While some ARM64 platforms support Software Delegated Exception Interface (SDEI) [ARM17], an extension that can be exploited to implement trap-free interrupt injection to guests, it is practically not available for 32-bit ARM platforms–interrupts must be reinjected by the hypervisor. I measure the overhead of the reinjection in Section 5.2.2.

Architecture-dependent Traps Because of architectural limitations, Jailhouse needs to emulate devices that are essential for a hardware platform and that cannot be virtualised in hardware, such as the interrupt distributor as part of the GIC in ARM architectures, or the ICR, as part of the interrupt controller on modern Intel® x86 platforms. Depending on the utilisation of those devices, the impact of the hypervisor must be analysed.

In Section 5.2.3, I quantify the latency overhead that is caused by hypervisor activity that is required to moderate accesses to the ICR on a modern Intel® platform.

Mitigations for Spectre-like Attacks Mitigations for Spectre-like attacks heavily rely on measures by system software and firmware, as well as updates for CPU microcode. For example, Meltdown enforces operating systems to isolate user and kernel address space to separate page tables, which is a cost-intensive change, as it requires TLB flushes on context switches. As mentioned before, in case of TLB misses, the existence of a hypervisor introduces additional paging layers, which, in turn, leads to higher latencies for page table walks.

Other platforms require assistance of the system's firmware to mitigate the attack surface. To mitigate, for example, CVE-2017-5715 (Spectre v2) [Koc+19], ARM64 platforms [ARM20b] implement a firmware-based workaround. The OS conducts a pseudo firmware call (*i.e.*, Secure Monitor Call (SMC)) that, as a side effect, results in a speculation barrier. However, a firmware call by a guest OS is trapped by the hypervisor which needs to moderate and forward the call to the firmware interface. Jailhouse implements

a fast dispatcher path for these kind of SMC calls. Nevertheless, those traps introduce additional latencies when speculation barriers are conducted.

As a base-line I present the impact of Spectre mitigations on conventional non-virtualised real-time systems, as well as their impact on virtualised, partitioned environments in Section 5.2.4.

In the following, I provide three quintessential microbenchmarks for four different platforms in total. I measure specific aspects of the overhead that is introduced by the hypervisor on modern Intel® x86 platforms (Intel® Xeon® E5-2683 v4 and a Intel® Xeon® Gold 5118) platform, a 32-bit ARMv7 platform (Nvidia Jetson TK1) and a 64-bit ARMv8 platform (Nvidia Jetson TX1).

5.2.2 ARM: The Cost of Interrupt Reinjection

Jailhouse supports two versions of ARM's Generic Interrupt Controller, GICv2 and GICv3 [ARM13b; ARM16]. Both implementations share the same architectural limitation: Interrupts do not directly arrive at the guest. They arrive at the hypervisor, and are then reinjected as virtual Interrupt Requests (IRQs) to the guest. This leads to an overhead in the hypervisor, as it must redirect the interrupt to the appropriate guest, followed by a switch of the privilege level.

My automated measurement setup consists of an Nvidia Jetson TK1 (quad-core Cortex-A15 @2.32 GHz) as target platform, and an AVR microcontroller for performing the actual measurement.

The AVR periodically toggles a general purpose input/output (GPIO) pin on the target board which causes an interrupt. The only task of the measurement binary is to answer as soon as possible to the interrupt by toggling a second GPIO that is connected to the AVR. Therefore, I implemented a minimalist application that uses Jailhouse's own inmate library.[2] To minimise code size for the response to make it as fast as possible, the instructions for toggling the GPIO are directly written in assembler in the interrupt vector table (*cf.* Listing 5.1).

```
1  vectors:
2    b       __reset_entry
3    b       vector_undef
4    b       vector_svc
5    b       vector_pabt
6    b       vector_dabt
7    b       vector_unused
8    //b      vector_irq
9    //b      vector_fiq
10   mov     r3, #0xd000
11   mov     r2, #0
12   movt    r3, #0x6000
13   str     r2, [r3, #0x520]
14   b       vector_irq
```

Listing 5.1: Fast IRQ response: Only four instructions are required to respond to the interrupt, that is, to toggle a GPIO.

[2] That is, a minimalist runtime environment for guests.

Any arriving IRQ will toggle the GPIO before the interrupt is actually dispatched and acknowledged. As no other interrupts than the GPIO IRQ will arrive, it will not introduce spurious answers. To measure the response latency, I compare the *bare-metal latency* (*i.e.*, the minimum latency without hypervisor and without an operating system) with the latency when the hypervisor is present and the application runs inside a non-root cell. The Capture Compare Unit (CCU) of the AVR ensures a precise measurement at a resolution of 62.5 ns. To validate automated measurements, I verified sample measurements with the latency manually measured by an oscilloscope.

The measurement without hypervisor (*i.e.*, VMM=off) represents the *bare minimum* latency achievable by the selected hardware platform. Latency difference with and without hypervisor presence measures the delay that is introduced when the hypervisor and other guests asynchronously access the system bus.

For the Jailhouse setup (VMM=on), I repeat the measurement under several environmental conditions (*e.g.*, additional load is placed on other guests to measure the influence on the shared system bus) and present the arithmetic mean as well as the standard deviation and the maximum latency. Every measurement runs for four hours, and was repeated with an interrupt frequency of 10 Hz and 50 Hz to determine the role of the frequency of the measurement. The *stress* parameter in Table 5.1 describes if other guests are put under CPU, I/O or memory load with the stress-ng benchmark.

Results can be found in Table 5.1. The first two lines show the minimum interrupt latency of the measurement without the existence of the hypervisor. The difference to other measurements denotes the overhead that is introduces by the hypervisor.

The latency that is introduced by the hypervisor does not significantly depend on the interrupt frequency, but on the utilisation of neighbouring guests. This effect is caused by the shared system bus: The hypervisor wants to access memory that is required for dispatching the interrupt, while other guests asynchronously access the same bus and potentially invalidate shared memory caches.

Additional VMM-induced interrupt latency amounts to $1.26\,\mu s$ - $0.45\,\mu s \approx 810$ ns on average, with narrow deviation. Still, outliers lead to additional latencies of $\approx 5\,\mu s$. Compared to the cycle times of typical industrial communication bus systems, the maximum delay is acceptable for many applications.

5.2.3 x86: The Cost of the Moderation of accesses to MSR

The APIC of x86 platforms provides a MMIO-based interface for accessing the controller; the interface is mapped to a single page in memory. Besides core-local interrupt config-

Table 5.1.: Interrupt reinjection latency on the Nvidia Jetson TK1 (in µs).

VMM	Freq	Stress	μ	σ	Max
off	10 Hz	no	0.45	0.02	0.50
off	50 Hz	no	0.45	0.02	0.50
on	10 Hz	no	1.26	0.07	2.81
on	50 Hz	no	1.25	0.04	2.94
on	10 Hz	yes	1.36	0.34	5.56
on	50 Hz	yes	1.35	0.34	5.38

uration, a special register of the APIC, the ICR, is used to send IPIs to neighbouring CPU interfaces. Sending IPIs to interfaces outside of the domain's scope violates isolation criteria: the domain must not be allowed to leave its scope. As the page size is the finest granularity for trap-free assignment of memory, MMIO-based write access to any register requires moderation by the hypervisor. The hypervisor forwards the access to the physical interface, and in case of ICR writes, it prevents cross-domain IPIs by masking out CPUs of other domains.

On modern x86 platforms, MSRs provide fine-granular access to various control registers of the platform, such as performance monitors, power management or debugging features. The `WRMSR` resp. `RDMSR` assembler instructions are used to write to, or to read from those registers. The address and value of the MSR are stored in CPU registers.

In virtualised environments, the VMM configures MSR bitmaps [UNR+05; AMD05] (each bit represents a MSR register) to selectively allow for unmoderated (*i.e.*, trap-free) access to those registers: If access to a register has no effect on neighbouring domains, then the access does not require any hypervisor intervention and it can be forwarded without moderation. Therefore, MSR bitmaps allow to selectively trap for read and write access. Among others, unmoderated access is granted for platform-specific information, Time Stamp Counter (TSC), or thermal information.

The successor of the APIC, the x2APIC [Int10], provides, in addition to a MMIO interface, a MSR-based interface to the registers of the controller. Nevertheless, as the x2APIC does not understand the semantics of isolated domains, write access to the ICR still requires moderation by the hypervisor. This affects domains with multiple CPUs that, for example, use IPI for inter-process signalisation. Any moderation leads to additional hypervisor activity, which can have impact on the system's response time in real-time scenarios.

I measure the additional latency that is introduced by Jailhouse and compare it against another real-time capable virtualisation solution, KVM that runs as VMM on top of a real-time patched Linux kernel. The setup is called RT-KVM [Rie16].

The measurement setup is as follows. To measure the duration of the WRMSR as precise as possible, I snapshot the value of the TSC,[3] a counter that holds the number of CPU cycles since system reset, right before and after the WRMSR to the ICR. Note that reading the TSC does not require any hypervisor intervention for both, Jailhouse and KVM–the TSC provides comparable measurement data. I repeat the measurement and collect $10 \cdot 2^{20}$ ($\approx 10.5 \cdot 10^6$) samples.[4] To further achieve minimal measurement overhead, the samples are appended to an array in memory during the measurement. After the measurement ended, the array is read in one bulk step. To evaluate the baseline of raw ICR write access, I additionally measure the duration of the WRMSR instruction on bare-metal without a hypervisor.

I measure the number of cycles that are required for executing the WRMSR instruction in the following three scenarios:

1. Baseline: bare-metal WRMSR duration without a VMM
2. Jailhouse: WRMSR duration as guest of Jailhouse
3. Qemu/RT-KVM: WRMSR duration as isolated guest of Qemu

```
1  void native_x2apic_icr_write(low, id)
2  {
3    [...]
4    raw_spin_lock_irqsave(&lck, flg);
5
6    delta = ___rdtscp();
7    __wrmsr(msr, low, id);
8    delta = ___rdtscp() - delta;
9    icr_measurement_enqueue(delta);
10
11   raw_spin_unlock_irqrestore(&lck, flg);
12   [...]
13 }
```

Listing 5.2: The core of the measurement of the ICR write duration. While interrupts are disabled, new measurement values are appended to an array.

In every scenario, I use Linux as payload OS to conduct the WRMSR. I use the same kernel binary in all three scenarios. I instrument and hook into Linux's native_x2apic_msr_write() wrapper and insert TSC reads before and after the measurement. Of course, I disable interrupts within the critical path of the measurement (*cf.* Listing 5.2) to avoid preemption within a single measurement. While the exact choice of the kernel version does not play an subordinate role for scenarios **(1)** and **(2)**, I patched (and tuned) the kernel with the real-time extension Preempt-RT (version 5.4.58-rt35), as I require a real-time capable host kernel in scenario **(3)**.

[3] RDTSCP [Int19b] is a serialised variant of RDTSC that avoids out-of-order execution of instructions of the measurement.
[4] I chose the size as it benefits from memory storage properties.

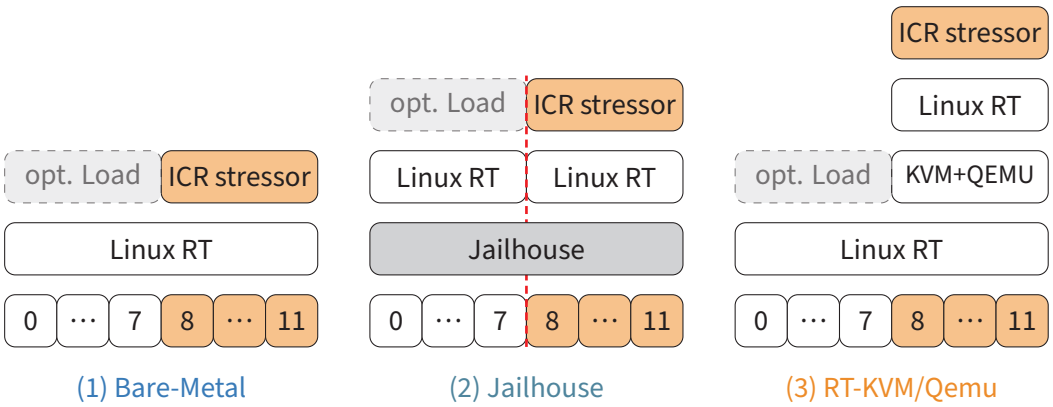

Figure 5.1.: Different measurement setups for the ICR MSR measurement. In every case, CPUs 0-7 serve uncritical payloads, CPUs 8-11 serve real-time payloads.

The measurement is conducted on a single-socket 12-core Intel® Xeon® Gold 5118 CPU running at a frequency of 2.30 GHz. CPUs 0-7 are assigned to the uncritical domain of the system CPUs 8-11 represent the critical real-time domain.

In case of **(1)** Bare-Metal, those CPUs are isolated from the rest of the system by using Linux kernel real-time configuration mechanisms, such as `isolcpus`, `nohz_full` and IRQ affinity rerouting. Any system noise, such as device interrupts, local timer interrupts and housekeeping activities are redirected to uncritical CPUs. Unless explicitly specified, processes will *not* be scheduled to those isolated CPUs.

In case of **(2)** Jailhouse, CPUs 8-11 are assigned to an isolated Jailhouse non-root cell by the hypervisor. As payload, the isolated domain is booted with the same Preempt-RT-patched Linux kernel as used in **(1)**.

The **(3)** Qemu/RT-KVM setup uses the same Linux-based isolation setup as in **(1)** with the difference that the ICR measurement is deactivated in the host kernel. In this case, virtual CPUs are explicitly pinned to the dedicated and isolated CPUs 8-11.

For each setup, I need a payload that causes a high amount of ICR writes in the real-time domain. Therefore, I use the stress-ng tool with the *fifo* stressor that starts workers which exercise a named pipe, which leads to frequent writes to the ICR. I repeat each setup under two additional environmental differences: with and without load on uncritical CPUs 0-7. Again, I use stress-ng on those to put additional stress on shared system busses and caches.

The results of the measurement can be seen in Fig. 5.2. The worst-case latency is of special interest, as it determines the maximum delay of the platform. Worst-case values are marked with horizontal lines. On bare-metal, a write to the ICR register takes at maximum, 296 cycles without load and 2216 cycles (factor 7.49 slowdown in comparison

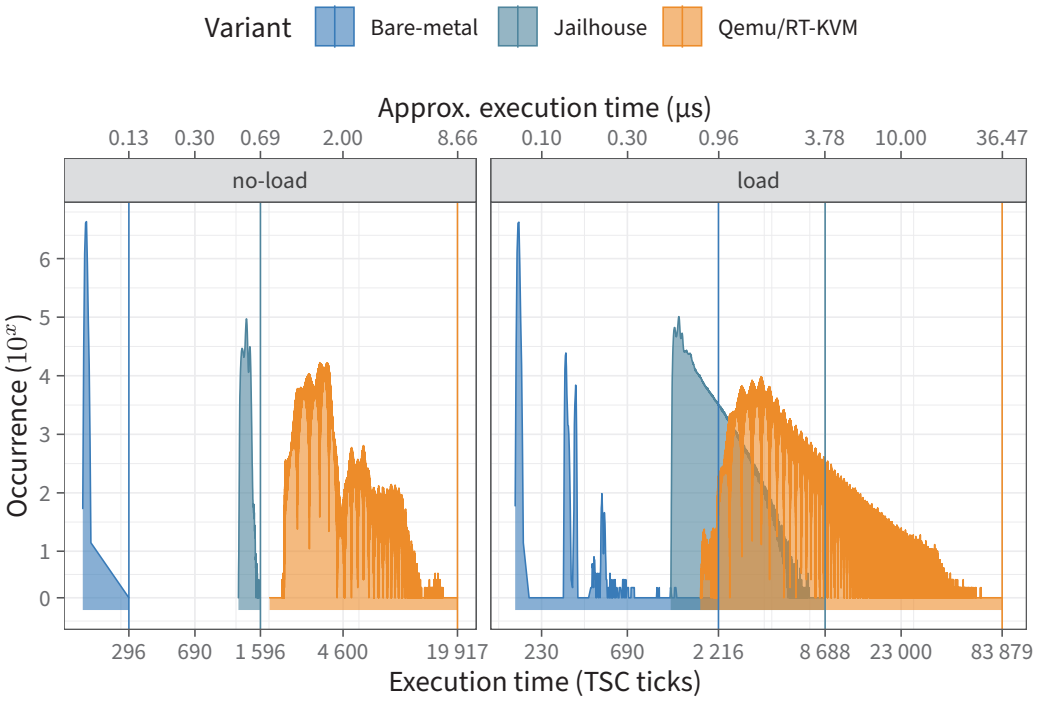

Figure 5.2.: Histogram of the execution time of `WRMSR`s to the ICR register of the x2APIC in TSC ticks and approx. transformed execution time in us. Vertical lines denote the maximum latency of a measurement. Both axes are scaled logarithmically.

to no load) under load. These values denote the bare ICR write access time that is achievable on the platform.

Under Jailhouse, the worst case amounts to 1596 without load, resp. 8688 (factor 5.44 slowdown) with load. The slowdown is better than on bare-metal, yet, the absolute maximum latency in the worst case (load situation) increased by 6472 ticks or 2.82 µs.

However, Qemu is 67 times slower than bare-metal without load, and 37 times slower under load. An additional worst-case delay of 35.51 µs can be unacceptable for many real-time applications, while an additional maximum delay of 2.82 µs is within a reasonable range of acceptable delay in many real-world real-time systems. The results can be used as actionable criteria, if a system architecture fulfils the requirements of a specific use case.

5.2.4 The Cost of Spectre Mitigations

Side-channel attacks on speculative execution of modern processors challenge system software to implement cost-intensive countermeasures, partly assisted by changes in

CPU microcode [Int18a]. While the impact of mitigations against speculative execution attacks on throughput-oriented systems is well-known [Pro+18], impact on virtualised or real-time system received less attention.

To quantify the impact on real-time systems, I will measure the interrupt response time for two different platforms: a Xeon®-based Intel® platform and a ARM-based Nvidia platform. At the core of the measurement, I use the real-time testbench cyclictest [GKW19] to compare latency differences of systems with and without mitigations that are required to protect against Spectre-like attacks.

Cyclictest is a multi-threaded high-resolution Linux user-space (US) benchmark that periodically configures a timer interrupt and measures the latency between the expected arrival of the interrupt, and the time where the interrupt notification actually arrived at the measuring thread. Threads of cyclictest typically run with a real-time scheduling policy (in my case SCHED_FIFO) and have the highest real-time priority. Each measuring thread is pinned to a specific CPU. Cyclictest supports REALTIME and MONOTONIC clocks, and uses Linux's high-resolution sleep system call clock_nanosleep(). The kernel receives the sleep request, arms the timer and changes the process state from running to sleeping. The kernel will then either idle and wait for the next interruption, or, if existent, reschedule processes of lower priority on the CPU. As the timer interrupt will first arrive in the kernel space before it is propagated to userland, cyclictest can be used to detect, trace and debug outlying latencies that occur between the system-level arrival of the interrupt, its dispatching, and the propagation to the userspace until cyclictest snapshots the timestamp of the arrival.

The goal of cyclictest is to not observe strong outliers that exceed a certain threshold over very long periods of time.[5]

As mentioned in Section 5.2.2, interrupts on ARM platforms require additional hypervisor activity due to interrupt reinjection by the hypervisor, which causes additional latencies. Those latencies can be measured with cyclictest. Under varying environmental conditions (*e.g.*, variations of additional lower-priority load), the long duration shall ensure a high coverage of rare events, such as, for example, SMIs, NMIs, machine check exceptions (MCEs), thermal exceptions, and others.

I use cyclictest to quantify the maximum latency that can be observed under the existence of a VMM. First, I measure the base-line of the system to determine latencies that can be expected without a hypervisor on bare-metal. Next, I repeat the measurement while the VMM is present. Analogous to the ICR measurement in Section 5.2.3, I repeat the measurement under two environmental variations: with, and without additional

[5]It is not unusual, that systems are benchmarked with cyclictest under varying conditions over weeks.

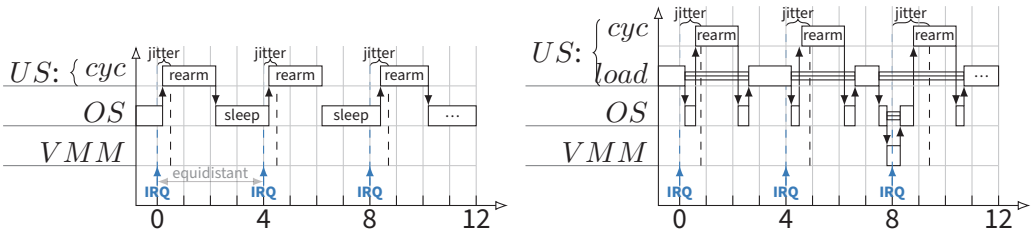

Figure 5.3.: Schedule of cyclictest. Left: The system is left without load, and no hypervisor intervention is required. Right, the system is under load, and the load causes VM exits.

peripheral load. In this measurement, I optionally put load with lower priority than cyclictest on the measuring CPUs to additionally stress the OS kernel (*e.g.*, to stress decisions of the scheduler), and to additionally stress the hypervisor (*i.e.*, by exits caused by the additional load).

To understand, why it is important to put load on those exact CPUs, I will first elaborate circumstances, when, and why (a) the hypervisor may introduce additional overhead, and (b) when Spectre mitigations are exactly required. Consider Fig. 5.3. In both figures, the x-axis denotes time in arbitrary units, and the y-axis denotes activity in different privilege levels, resp. US processes. On the left side, the system has no additional load–cyclictest is the only scheduled application in userspace of the measuring core. In case of my measurement on an Intel® x86 platform (Ref. Fig. 5.4), the hypervisor does not get active during the measurement; I achieve zero traps in that configuration. On the ARM system, in turn, every interrupt arrives at the VMM and requires reinjection to the guest. However, even in zero-trap scenarios additional latency may occur if the hypervisor is active, as virtualisation comes at a certain cost [Dre08]. Hence, and as additional load may (depending on the workload, and the platform) lead to additional hypervisor exits, I measure both: without the existence of a VMM (*i.e.*, bare-metal) and with existence of a VMM (*i.e.*, Jailhouse).

The IRQ directly arrives at the guest OS in the Intel® x86 setup. The OS acknowledges the interrupt and activates the high-priority cyclictest process. Cyclictest measures the jitter (as described before), rearms the timer and sleeps. As no other tasks are waiting on the measuring CPU in the no-load scenario, the system will idle.

When the interrupt arrives in the load scenario (right side of Fig. 5.3), the load-task is preempted by the OS, which acknowledges the interrupt and immediately reschedules cyclictest. Cyclictest, in turn, snapshots time and measures the jitter, rearms the timer and sets itself again to sleep state. The OS will then reschedule the preempted low-priority load task until the next interrupt arrives. Now, the load process may occasionally lead to activities of the hypervisor (*e.g.*, MSR writes to the ICR register of the x2APIC,

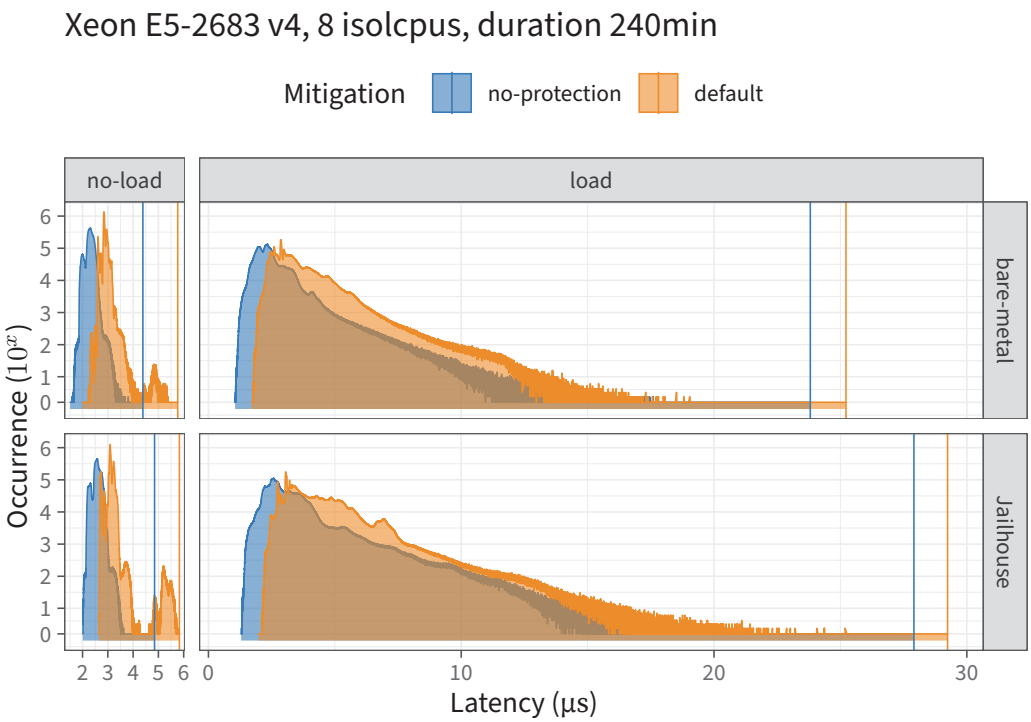

Figure 5.4.: Intel® Xeon® E5-2683 v4: Latency histogram of the influence of mitigations against speculation attacks on determinism and response latencies of a Preempt-RT extended Linux based system. Left side: without additional system load. Right side: measuring CPUs are put under additional non-real-time load.

Ref. Section 5.2.3). If the interrupt arrives while the hypervisor preempted the OS, at least two context switches are involved (VMM→OS, and OS→US), which leads to yet higher latencies.

This situation is aggravated by mitigations for attacks on speculative execution, as they are primarily conducted at these type of context level switches. Meltdown, for example, exploits speculation on otherwise prohibited reads on supervised kernel pages [Int19b] by leveraging side-channel attacks on caches [Lip+18] on Intel® x86 platforms as well as on some ARM variants [ARM20b]. To mitigate meltdown, Linux introduced PTI, a mechanism that separates user- and kernel-space page tables. Only thin supervised trampoline code to switch page tables on US→OSS context level switch is present when executing in user-land. On any context level switch (black arrows in Fig. 5.3) between US and OS, the kernel will invalidate the TLB and switch the tables, which introduces latencies due to massive changes of the microarchitectural state of the platform–independent of the existence of a VMM.

On ARM64 platforms, the kernel needs to invoke the system firmware to protect itself against speculation attacks [ARM20b]. This does, besides short firmware activity in any

case, involve additional hypervisor activity in virtualised setups, as the hypervisor needs to moderate the firmware call.

Other mitigations for Spectre-like attacks require speculation barriers if an untrusted user has control over input to the kernel, that may be used for later speculation.[6] This requires enforcement non-speculative indirect array referencing to mitigate Spectre v1. Another workaround to mitigate speculation of indirect jump addresses is the return trampoline (retpoline), which mitigates Spectre v2. To prevent speculation on the target of an indirect jump, the retpoline uses an never-used infinite loop to prevent speculation. Compilers typically implement this kind of mitigation.

In conclusion, the whole software stack is involved in the still growing number of mitigations.

Spectre Mitigations on x86 Consider the histogram in Fig. 5.4, and Table 5.2. I measure on eight cores of dual-socket Intel® Xeon® E5-2683 v4 CPU. Those cores are located at the same Non-Uniform Memory Access (NUMA) node to ensure Uniform Memory Access (UMA). For the isolation I used the same tuneables as in the previous ICR measurement in Fig. 5.2. In all variants of the measurement, we use the real-time patched Linux Kernel 4.19.69-rt43.

For the no-protection scenario, I disable all mitigations. Therefore, I use second variant of the kernel that disables compile-time mitigations (*e.g.*, retpoline). Additionally, I use the stock microcode version 0xb00001e in the no-protection variant, and 0xb00002e [Int18a] in the default setup, as the latter one contains additional instructions for microcode-assisted speculation control, such as Indirect Branch Prediction Barrier (`IBPB`) and Indirect Branch Restricted Speculation (`IBRS`).

[6] For example, users have control over system call numbers, which may be later be used for indirection.

Table 5.2.: Cyclictest on a Intel® Xeon® E5-2683 v4 platform: Minimum, average and maximum latencies (in µs).

VMM	Protection	no-load			load		
		min	avg	max	min	avg	max
bare-metal	no-protection	1.509	2.300	4.386	1.056	2.430	23.808
bare-metal	default	2.001	2.871	5.765	1.698	3.349	25.219
Jailhouse	no-protection	2.007	2.574	4.848	1.296	3.081	27.907
Jailhouse	default	2.639	3.101	5.831	1.980	4.136	29.238

Besides updates in microcode, the default configuration activates following OS-assisted mitigations:

- L1TF: PTE inversion, VMX conditional cache flushes
- Meltdown: PTI
- Spectre v1: `__user` pointer sanitisation
- Spectre v2: Full generic retpoline, IBPB, IBRS_FW

Note that cyclictest does not cause any hypervisor traps on the selected platform in case of no-load. I use the stress-ng tool in the load scenario as producer of additional load. Stress-ng is configured to sequentially change stressors that vary in workload types. Among others, workload types are computational loads that stress the CPU, memory-bound loads that stress the system bus, and load types that stress caches, the TLB, and cause a heavy amount of context switches between user-space and kernel-space. The measurement is run for 240 minutes, the cycle time for the cyclictest timer on a CPU is 1 ms. Per CPU, the histogram contains collect $14.4*10^6$ measured points with nanoseconds resolution.

Though minimum latencies are of minor interest in real-time systems, let me first mention that the minimum latencies achieve better results if the system is under load–independent of the existence of a VMM and the configuration of the protection. In my observation, the minimum latency is up to 711 ns faster if the system is under additional load. While this effect may seem contradictory, it can be explained as the result of multiple effects that occur if the system is under load. Additional load prevents the system from falling back to sleep states. Furthermore, caches and CPU pipelines are constantly under utilisation. Hot code paths that are shared between the load and the measuring task (*e.g.*, parts of the kernel code, such as IRQ handlers) are kept warm in caches: the load avoids cache misses in those code paths, which is beneficial for the measuring task. Additionally, the microstate of the CPU may be in a *graceful* state if the system is put under constant load.

On bare-metal, spectre mitigations introduce an additional latency of 1.4 µs (+32%) without additional load, and 1.4 µs (+6%) with additional load. The absolute time that is added by the existence of mitigations on bare-metal is constant, independent of additional system load.

Next, I quantify the latency that is added by the existence of a VMM. Therefore, I compare the bare-metal/no-protection configuration and the Jailhouse/no-protection configuration. Note that it is possible to achieve a zero-trap configuration in case of the presence

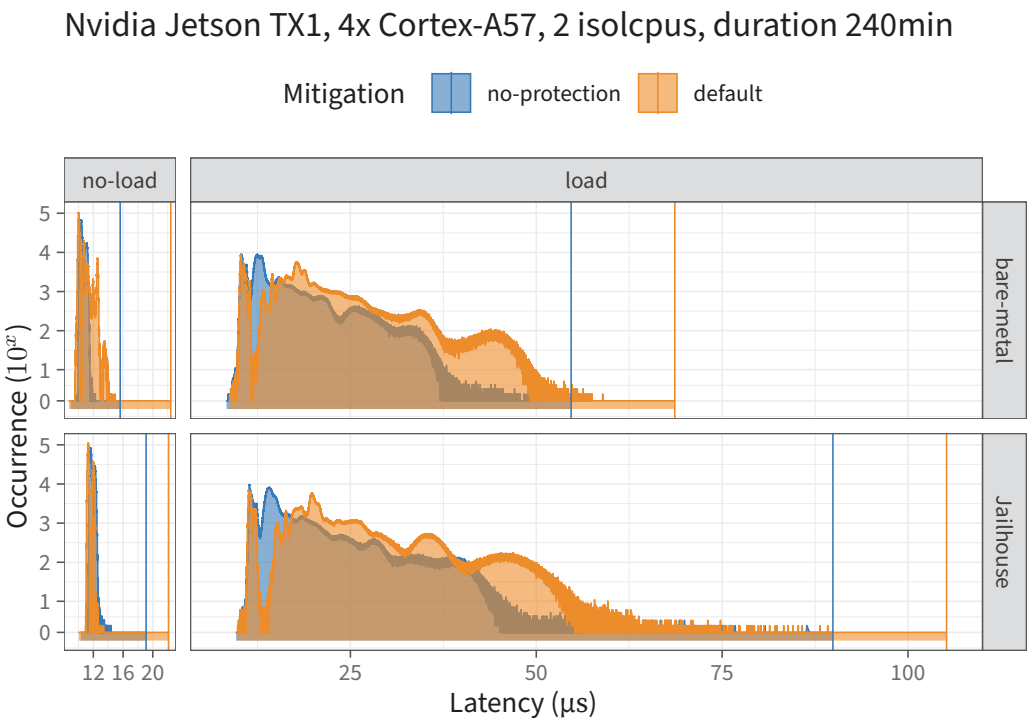

Figure 5.5.: Nvidia Jetson TX1: Latency histogram of the influence of mitigations against speculation attacks on determinism and response latencies of a Preempt-RT extended Linux based system. Left side: without additional system load. Right side: measuring CPUs are put under additional non-real-time load.

of the VMM. In maximum, the existence of a hypervisor introduces an additional latency of 462 ns (+11%) without additional load, and 4.1 µs (+17%) with additional load. The higher latency in the load-scenario can be explained with additional hypervisor activity that may be required if the hypervisor is active while the interrupt arrives (*cf.* Fig. 5.3). If mitigations are activated, the increase of the latency amounts to 66 ns (+1.1%) without load, and 4.0 µs (+16%) under load.

This means that the hypervisor adds less overhead–in proportional and absolute number–than spectre mitigations without additional load. Under load, the hypervisor adds slightly more overhead than the Spectre mitigations. Still, the maximum overhead that is introduced through the existence of a VMM amounts to 4 µs, which is acceptable for many real-time scenarios.

Spectre Mitigations on ARM64 Consider the histogram in Fig. 5.5, and Table 5.3. The measurement has the same setup as for the x86 platform with respect to the variations of the measurement. However, mitigations on ARM64 require to exchange the system firmware, as firmware support is required to protect against Spectre v2 [ARM20b]. Be-

Table 5.3.: Cyclictest on an ARM64 Nvidia Jetson TX1 platform: Minimum, average and maximum latencies (in µs).

VMM	Protection	no-load			load		
		min	avg	max	min	avg	max
bare-metal	no-protection	8.971	10.523	15.613	8.334	15.117	54.688
bare-metal	default	8.744	10.353	22.420	8.717	19.317	68.632
Jailhouse	no-protection	10.310	11.807	19.101	9.629	16.915	89.900
Jailhouse	default	10.054	11.487	22.122	9.738	22.283	105.185

sides firmware support, this mitigation requires support by the hypervisor, as it needs to forward the mitigation request of the guest, and, of course, OS support that discovers and enables the conduction of mitigations. The default configuration of the measurement setup was configured to activate mitigations against Spectre v2. As speculation barriers, firmware calls cause branch predictor invalidation. My target platform, the Jetson TX1 comes with a quad-core Cortex A57 CPU.

In alignment with the x86 measurements, I observe the same effect of improvements of the minimum latency if the system is under load. The same argumentation for those effects apply on the ARM64 platform.

Again, I first investigate the overhead of mitigations without a hypervisor. In maximum, mitigations cost $6.8\,\mu s$ (+45.4%) without load, and $14\,\mu s$ (+25%) under load. This significant increase of latency can be explained with additional context switches when mitigations are conducted. With additional load on the system, memory caches of the hypervisor can be evicted by the load. In these cases, the hypervisor will experience a high amount of cache misses during the interrupt reinjection.

Without protection, the hypervisor adds an additional latency of $3.4\,\mu s$ (+22%) without load, and $35.2\,\mu s$ (+64.3%) under load. With default protection, the hypervisor reacted faster than bare-metal $-298\,\mu s$ (-1.3%) without load, but $36.5\,\mu s$ (+53.2%) under load. The improvement of the latency under default protection as a measurement artefact that can be compensated with longer durations of the measurement.

However, as interrupt reinjection is required on the target platform, the existence of the hypervisor introduces significant latencies. This situation is aggravated, if mitigations are activated, as mitigations lead to further context switches that can additionally overlap with the arrival of the timer interrupt. The suitability of the platform for real-time applications requires careful consideration.

Nevertheless, there is ongoing work to exploit SDEI to avoid costful interrupt reinjection on ARM64 platforms. Additionally, cache-colouring [Bug+96; Sco+20] can be used to improve effects on caches that are shared between cores and between the OS and the VMM.

5.3 Discussion

5.3.1 The Jailhouse Approach

The minimalist approach of Jailhouse results in a manageable amount of Source Lines of Code (SLOC). This is a crucial factor for both, formal verification from an academic point of view and system certification from an industrial point of view. I am aware of the problem that a substantial chain of software besides the Linux kernel (*e.g.*, UEFI firmware code, bootloaders etc.) is required for the boot process, and needs to be considered in such certifications to some extent. There are various possible approaches to address these issues that go beyond the scope of this thesis.

Jailhouse, in total, consists of almost 30k SLOC for four different architectures. This includes the hypervisor core, example code, kernel driver, and userland tools and utilities. Substantial parts of the code are architecture-independent. The common critical hypervisor core code that is shared across all architectures amounts to less than 3.4k SLOC. Architecture dependent code amounts to ≈7.4k SLOC for x86 and implements both, Intel® and Advanced Micro Devices (AMD), and ≈5.4k SLOC for ARM (both, ARMv7 and ARMv8). Exemplarily, the whole hypervisor core for ARMv7 sums up to ≈7.4k SLOC.

Many research systems are developed from scratch and spend tremendous effort on re-implementing existing device drivers. But still, missing device support is a major obstacle for their practicability. More than half of Quest-V's source lines of code (≈70k SLOC of 140k SLOC) implement device drivers. With almost 27k SLOC, XtratuM is more lightweight than Quest-V and only implements basic drivers for debug output. Still, the publicly available versions of Quest-V and XtratuM currently only support the x86 architecture.

Jailhouse does intentionally not follow classical virtualisation approaches, but its design does not generally eliminate the use of those techniques. This opens the possibility to exploit Jailhouse as an experimental systems platform that allows for keeping focus on the actual problem instead of re-implementing fundamentals from scratch. Jailhouse is an ideal platform for investigating hardware and software behaviour under AMP

workloads. Furthermore, it provides a convenient and comfortable environment for executing DSP-like workloads on *raw* hardware.

Modern multi-core systems already provide enough physical CPUs to make scheduling in hypervisors unnecessary for many real-world embedded use cases. In fact, numerous essential requirements on real-time embedded hypervisors [CRM10], such as real-time scheduling policies, efficient context switches, or deterministic hypervisor calls, do not even need to be addressed in a partitioned setup. Those requirements actually reflect well-known issues of operating systems and should not be duplicated in hypervisor space for embedded systems with real-time requirements. As Jailhouse does not virtualise CPUs, overcommit hardware or schedule partitions, there are no expensive partition context switches or scheduling issues [Ves07] as they occur in other real-time hypervisors [KW07; MRC05; Pin+14; Xi+11]. Hypercalls are only used for management purposes and not for moderating access to shared hardware.

5.3.2 Hardware Limitations

In general, on all supported platforms, the concept of ideal hardware partitioning is not yet fully achievable. Depending on the interrupt system and the architecture, interrupts might arrive at the hypervisor. On such platforms, the interrupt reinjection to guests is a frequent job of the hypervisor that introduces unintended additional interrupt latencies. This issue is already solved for 64-bit x86 architectures that support *Interrupt Remapping* and will be solved in future ARM architectures that implement the GICv4 [ARM16] specification, which is beneficial to the final goal, to end up in no VM exits.

Virtualisation always comes at the cost of drawbacks and implications on performance and latency, especially when limitations of current state of the art hardware does not provide sufficient assistance for hardware partitioning. Hypervisor activity and the amount of traps can be significantly minimised by future improved hardware design. Placing different hardware devices on separate pages is a hardware design issue that could easily be solved.

Nevertheless, there are unavoidable traps that are caused by hardware design. On current ARM architectures, the interrupt distributor must be virtualised. Varanasi and Heiser [VH11] assume that this is not expected to cause performance issues. During the implementation of my demonstration platform I contrariwise observed that Linux kernels with the Preempt-RT real-time patch make heavy use of the interrupt distributor which causes high activity of the hypervisor. Such issues should be addressed by proper hardware design in order to be able to execute unmodified guests.

The physical design of many dual-homed hardware devices (*e.g.*, clock and reset controllers) forces us to fall back to paravirtualisation. While this is acceptable for many real-time scenarios, as those types of devices are typically not called in the operational phase, they still should be designed in a way, that settings of a device can be modified without any impacts on other devices.

The requirements of Popek and Goldberg were postulated in 1974, but are—almost half a century later—still applicable to modern systems. However, some adaptions and extensions are required to handle contemporary real-world use cases that need to satisfy real-time and mixed criticality requirements with slight adaptions. In Section 4.1 I have presented general, high-level criteria, and have also derived consequences for hypervisor assisted static hardware partitioning.

Many of the existing hardware virtualisation extensions reduce hypervisor interaction to optimise the average-case behaviour (throughput) of systems. As real-time systems are optimised for the worst-case, these extensions do not always necessarily meet real-time requirements in terms of low latency as avoidable hypervisor interception is required in many cases.

For ideal hardware partitioning, any remaining interception causes are mainly solved by software by policy-based decisions that can fully be offloaded to hardware. Hardware/-Software co-design can close the missing gap: software requirements on the system need to be carefully evaluated with systems designers.

Barrelfish [Sch+08] is an operating systems designed for heterogeneous multi- and many core systems. They focus on operating system scalability aspects, as, for example, the number of CPUs on a system grows more than individual clock rates. They argue that multi-core systems can be seen as a network of independent cores, and that no sharing at lowest level is required [Bau+09].

Other researchers analyse operating system overhead and try to offload, for example, scheduling decisions to hardware [DL17]. Microkernel approaches follow similar goals: a significant fraction of decision should be offloaded to hardware.

5.3.3 Speculative Execution and Static Hardware Partitioning

The disclosure of Spectre and related attacks target the complexity of modern systems, and require, depending on the workload, expensive mitigation measures. Many of these attacks exploit sharing of resources: hardware units can be shared across multiple tasks. This includes, for example, physical CPUs, if different workloads are scheduled on the

same execution unit, and caches, if different workloads on different execution units share the same caches.

With the existence of multi-core SMP CPUs, software-based sharing techniques, such as process or domain scheduling, can be avoided for many use cases. From a real-time perspective, this lowers system overhead and on the other hand, it avoids sharing of hardware resources and inherently reduces the attack surface for attacks based on speculative execution.

For mixed-criticality environments, the partitioning of systems is an architecturally more attractive alternative that can handle both, the inadvertent establishing of covert side channels and the safe coexistence of workloads of mixed criticality at the same time. This benefits, for instance, existing certified industrial codes that can only be modified at the expense of re-certification, which is both substantial in terms of monetary investment and required time to market: Instead of ensuring protection by adding explicit countermeasures to the code, it is run inside a isolated partition. Virtualisation technologies of modern CPUs provide mechanisms for strict and full isolation of computing domains [UNR+05; Int18b; AMD05; VH11], and the overhead (usually caused by the decreasing lack of hardware capabilities and imperfections) imposed by the cost of partitioning only marginally differs from the cost of mitigations in many situations (Ref. Section 5.2.4). Since partitioning provides additional possibilities to system architects, I perceive the approach to be a preferable solution as compared to only rectifying erratic CPU behaviour.

It is self-evident that a system operating at maximal capacity will not be able to satisfy its original constraints once such countermeasures will be in place, which gives additional justification to the use of over-provisioned hardware. I have already remarked that many real-world systems do not operate at the brink of their capacity, and will retain the ability to appropriately respond to events even in the presence of Spectre-type mitigations.

5.4 Smoke Test

To demonstrate the suitability of the approach especially for practical use, I implemented a (mixed-criticality) multi-copter control system that is shown in Fig. 5.6 The requirements on such platforms are comparable to many common industrial appliances: The flight stack, a safety and real-time critical part of the system with high reliability requirements, is responsible for balancing and navigating the aircraft. Sensor values must be sampled at high data rates, processed, and eventually be used to control rotors.

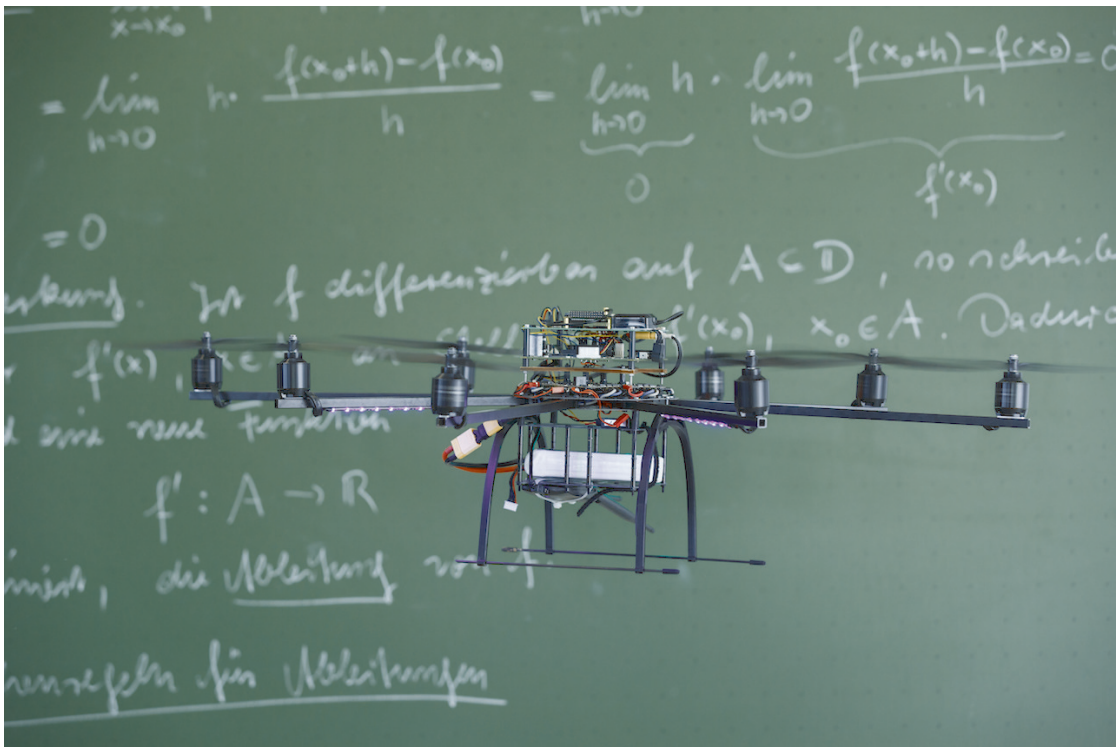

Figure 5.6.: A multi-copter platform that runs on the Jailhouse hypervisor. The real-time critical flight control executes within a Jailhouse cell, while uncritical parts of the system (*e.g.*, camera tracking) execute in an isolated domain.
Photo © OTH Regensburg / Florian Hamerich

For a safe and reliable mission, the control loop must respond deterministically. System crashes may result in real crashes with severe consequences.

The flight stack runs in a Jailhouse cell, while uncritical tasks, for example WiFi communication with a ground station or camera tracking, can easily be implemented in an uncritical part thanks to the available Linux software ecosystem. Critical hardware components, for example SPI, I^2C or GPIO devices, are assigned to the critical cell. The hardware platform is a Nvidia Jetson TK1 with a quad-core Cortex-A15 ARMv7 CPU, connected to a sensor board that provides accelerometers, GPS, compasses and gyroscopes. Two cores are assigned to the uncritical part, and two cores to the critical one.

The critical domain executes a second stripped-down Linux operating system with the Preempt-RT real-time kernel extension. Ardupilot provides flight control, and does not require modifications besides board support. This underlines that existing applications can be deployed in a Jailhouse setup with little effort, and that it is suitable for implementing real-time safety-critical systems mostly based on existing components. Nonetheless, I needed to solve various issues that do not arise on a purely conceptual

level or with systems tailored for very specific hardware, but endanger assumptions made in my and similar approaches.

5.5 Summary

The implementation of mixed-criticality systems requires safe isolation. Primitives that can be used to achieve safe isolation, in turn, depend on mechanisms provided by hardware. On the one hand, uncritical domains of a system demand for high flexibility with respect to their field of application, on the other hand, safety-critical applications demand for strict isolation from other domains of the system.

The one extreme is the individual conception of tailored hardware – processor architectures that precisely fulfil the needs of a specific system. This approach is accompanied by high hardware development costs and complicates future reconfigurations of the payload software [Die19]. The other extreme is to purely rely on software-based isolation primitives – OSs use standard isolation mechanisms and shall *guarantee* a *sufficient* level of isolation (Ref. Section 1.3.2, SIL2LinuxMP). Yet, this approach is accompanied by lower hardware costs, but also by lower isolation guarantees.

Virtualisation extensions of COTS hardware allow to introduce strong isolation barriers that are otherwise not used by standard system software. I presented the concepts of Jailhouse, a real-world Linux-based static partitioning hypervisor. Static and ideal hardware partitioning are promising approaches to fill the gap between both extremes: it enables dynamic reconfiguration of complex systems, and exploits strong hardware-based isolation mechanisms that are provided by economic COTS hardware components. For embedded real-time virtualisation, its ultimate goal is to minimise the interaction with guests. All issues that are introduced by conventional (para-)virtualisation approaches are deferred back to the operating systems of the guests, where they already existed before. Furthermore, the driverless approach tries to fill the gap between academic research systems and industrial practicability.

In comparison to paravirtualisation techniques, direct hardware assignment to guests allows for running unmodified legacy payload applications with no active hypervisor overhead. The minimalist hypervisor core simplifies certification efforts. By executing standard operating systems as guests, efforts that are required for porting existing legacy payload applications can be minimised. By implementing a complex demonstration platform, I successfully showed the practicability of hardware partitioning.

While standard virtualisation extensions provided by current hardware seem to suffice for a straight forward implementation of my and many other approaches, real hardware presents a number of limitations that can completely undermine the advantages and guarantees of partitioning and virtualisation-based approaches. As ideal hardware partitioning is not possible on supported platforms, I showed in Section 5.2 quintessential benchmarks that are required to quantify the performance of the hypervisor and to assess the suitability of the approach for specific use cases.

The measurement in Section 5.2.4 evaluates the impact of Spectre mitigations on real-time systems. I showed that for many real-world use cases that require Linux to run side by side with real time operating systems, static hardware partitioning is a viable alternative to classical OS-based isolation approaches, especially when legacy workloads must be protected against hardware weaknesses like Spectre-class speculation attacks without modifying certified components.

I defined the concepts of *ideal VMMs*, *ideal partitions* and *ideal partitioned systems* with the goal of establishing zero-trap hypervisors on real-world systems that only need to account for setting up partitions, but do not interact with the content of any partitions in the operational phase. Experiments with an implementation of the concept on multiple hardware platforms showed limitations inherent in current hardware. I discussed necessary improvements in future virtualisation techniques to facilitate a realisation of my approach on real-world systems.

In future, hardware manufacturers are asked to change design aspects with respect to the demand that hardware can be partitioned. Ongoing developments in this field already promise partial improvements with respect to system partitioning. Any software-based workaround leads to more avoidable hypervisor code and more hypervisor logic. This demand requires software engineers and hardware manufacturers to strengthen their focus on Hardware-Software Co-design.

End of Part II

6 Summary, Conclusions and Further Ideas

6.1 Summary of the Thesis and Conclusion

In this thesis, I presented and evaluated an architecture for mixed-criticality systems based on OSS components. After I elaborated common industrial requirements on such systems, I continued from the software engineering perspective in Part I, and from a system engineering perspective in Part II. The approach from those two sides is necessary to take all aspects (Real-Time and OSS, Software Certification and Strict Isolation) into account to derive an architecture that holistically respects all requirements.

In particular, I considered consolidated mixed-criticality systems, where Linux serves as feature-rich OS for less-critical aspects of the system, while critical payloads execute in strictly isolated domains. By exploiting virtualisation extensions of modern CPUs, I presented the concept of ideal hardware partitioning that enables strict isolation across execution domains. The concept was evaluated on COTS hardware. Ideal hardware partitioning allows for executing hard real-time payloads in dedicated domains, while they remain strictly isolated from less-critical aspects of the system, where Linux and its feature-rich ecosystem execute. The thin hypervisor layer of the approach eases certification efforts.

Yet, OSS components still demand for certification when being used in safety-critical environments, which, in turn, requires argumentations on their development process.

6.1.1 Software Engineering

The industrial deployment of OSS is often hindered by certification requirements on development processes. To judge wether OSS projects comply with relevant standards, it is necessary to quantify characteristics of the development process. To conduct analyses regarding traceability and auditability of development decisions, including code writing, reviewing, deployment, and maintenance activities, it is prerequisite to be able to reconstruct the development process. Hence, I asked the following research question.

Research Question 1: *Can complex OSS development processes be reconstructed in ex-post analyses?*

The open and community-driven development process of OSS does not provide full traceability of its process by design. However, my contributions in Chapter 2 showed that it is possible to reconstruct and analyse development processes in ex-post analyses. By restructuring otherwise disorganised publicly available development artefacts, it is possible to reconstruct the development process, that is, to arrange development artefacts in a normalised, queryable representation. My approach scales with the largest OSS projects of the world.

This representation is the basis for answering in-depth questions about the development process. Hence, I posed my second research question:

Research Question 2: *What are reasonable metrics to quantify the adherence to or violations of OSS development processes?*

To give an answer to this question, I exemplarily analysed the Linux kernel in Chapter 3. I gave an in-depth overview and formalisation of the Linux Kernel Development Process (LKDP), and quantified key development characteristics, such as the amount of ignored patches, or the amount of patches that were integrated in conformance with official development processes. Those metrics can be used as proxy metrics for safety assessments. Finally, I systematically investigated violations of the development process, and categorised different types of violations of the process. My framework gives the possibility to conduct further in-depth analyses that serve as an evidence-based input for safety certification efforts.

I showed that it is possible to quantify development process-related aspects of large-scale OSS development processes. The lack of traceability of OSS projects is no longer an obstacle to safety certifications. From the software engineering perspective, OSS software provide serious alternatives for being deployed in safety-critical environments.

6.1.2 System Engineering

While the methodology and analyses in Part I help to address the certification requirement (*cf.* Req. 2), they do not address remaining requirements on mixed-criticality systems: real-time capabilities of the system (*cf.* Req. 1) and strict isolation of domains (*cf.* Req. 3).

To address these requirements, I presented the concept of *ideal hardware partitioning* in Part II. By exploiting virtualisation extensions of modern COTS CPUs, ideal hardware

partitioning segregates SMP systems into isolated execution domains, while maintaining real-time capabilities of the platform. Ideal hardware partitioning eradicates software-induced overhead that is related to the partitioning of the system. The fist goal was to investigate if ideal hardware partitioning is a feasible approach for common COTS systems:

Research Question 3: *What are hardware requirements to implement ideal hardware partitioning?*

In Chapter 4, I defined ideal hardware partitioning, and derived hardware requirements that are mandatory to successfully implement the approach. In Section 4.2, I presented the concept and the architecture of the Jailhouse hypervisor, which aims to implement ideal hardware partitioning on four different architectures: 64-bit x86 (Intel® and AMD), 32-bit ARMv7 and 64-bit ARMv8 platforms. While modern architectures provide comprehensive virtualisation extensions, and development trends strive towards the full implementability of the approach, I concluded that ideal hardware partitioning is not yet fully achievable on those architectures: In certain situations, it is unavoidable that software-based intervention causes additional latency, which potentially endangers real-time capabilities of the platform. Hence, I systematically identified and quantified remaining overheads:

Research Question 4: *What are the limitations on current COTS hardware to implement ideal hardware partitioning? What are unavoidable overheads?*

In Chapter 5, I presented methodologies to systematically microbenchmark remaining overheads. Results of the analysis can be used to assess the suitability of a given platform for static hardware partitioning. I discussed and showed the suitability for real-world applications of the approach. While ideal hardware partitioning is not yet fully achievable on those platforms, I showed that limitations of current architectures are not a major drawback for the suitability of the approach on COTS hardware components in general.

6.2 Further Ideas

Software Engineering With respect to software engineering aspects of my thesis, I was able to show that OSS development processes can be reconstructed. My evaluations underline the practicability of the approach. Yet, they do not qualify OSS software for use in safety-critical environments. In the future, precise questions on processual ongoings

in projects that support safety certification endeavours need to be defined by safety experts, and can then be answered based on my methodology.

System Engineering Hardware manufacturers are demanded to change design aspects with respect to the requirements of ideal hardware partitioning. This demand requires software engineers and hardware manufacturers to strengthen their focus on Hardware-Software Co-Design: Open instruction set architectures (ISAs), for example RISC-V, offer the possibility to adjust the platform to individual requirements. Furthermore, open ISAs offer the opportunity to address the remaining issues at their root cause: the hardware. Further investigation allows for implementing ideal hardware partitioning on real-world hardware.

Appendix

A

This chapter shares updated material with the OpenSym '16 paper "Observing Custom Software Modifications: A Quantitative Approach of Tracking the Evolution of Patch Stacks" [RLM16].

A.1 Quantification of Mainlining Efforts

Special-purpose software, like industrial control, medical analysis, or other domain-specific applications, is often composed of contributions from general-purpose projects that provide basic building blocks. Custom modifications implemented on top of them fulfil certain additional requirements, while the development of *mainline*, the primary branch of the base project, proceeds independently.

Especially for software with high dependability requirements, it is crucial to keep up to date with mainline: latest fixes must be applied and new general features have to be introduced, as diverging software branches are hard to maintain and lead to inflexible systems [GS12]. Parallel development often evolves in the form of *patch stacks*: feature-granular modifications of mainline releases. Because of the dynamics exhibited by modern software projects, maintaining patch stacks can become a significant issue in terms of effort and costs.

My methodology presented in Chapter 2 can be used to quantitatively analyse the evolution of patch stacks by mining git [GIT16] repositories and produces data that can serve as input for statistical analysis. I use the methodology to compare different releases of stacks and groups similar patches (patches that lead to similar modifications) into equivalence classes. This allows for comparing those classes against the base project to measure integrability and influence of the patch stack on the base project. Patches that remain on the external stack across releases are classified as *invariant* and are hypothesised to reflect the maintenance cost of the whole stack. A fine grained classification of different patch types that depends on the actual modifications could function as a measure for the *invasiveness* of the stack.

In the following, I will present an approach for observing the evolution of patch stacks based on the methodology of PaStA, as presented in Chapter 2. I will present a case study on Preempt-RT [PRT20], a real-time extension of the Linux kernel that has been enjoying widespread use in industrial appliances for more than a decade, yet has not fully been integrated into standard Linux. I measure its influence on mainline Linux and visualise the development dynamics of the patch stack.

A.1.1 Approach

In general, a patch stack (also known as patch set) is defined as a set of patches (commits) that are developed and maintained independently of the base project. Well-known examples include the Preempt-RT Linux real-time extension, the Linux Long Term Support Initiative (LTSI) kernel, and vendor-specific Android stacks needed to port the system to a particular hardware. In many cases, patch stacks are applied on top of individual releases of an upstream version, but they do not necessarily have to be developed in a linear way [Bir+09]. The commits of the patched version of a base project are identified as the set of commit hashes that do not occur in the mainline project. My analysis is based on the following assumptions:

- Mainline *upstream* development takes place in one single branch.
- Every release of the patch stack is represented by a separate branch.

The work flow of this analysis is assisted by PaStA and consists of the following steps:

1. Set up a repository containing all releases of the patch stacks.
2. Identify and group similar patches across different versions of the patch stacks.
3. Compare representatives of those groups against mainline.
4. Use statistical methods to draw conclusions on the development and evolution of the patch stacks.

A *commit hash* provides a unique identifier for every commit: In the following, U is the set of all commit hashes of the base project, while P_i is the set of the commit hashes of a release i of the patch stacks. $P \equiv \bigcup_i P_i$ denotes all commit hashes on the patch stacks. Note that $P \cap U = \emptyset$. Let $H \equiv P \cup U$ be the set of all commit hashes of interest. I will use the PaStA methodology presented in Chapter 2 to cluster similar patches.

Grouping Similar Patches

Patch stacks change as they are being aligned with the changes in base project and additionally integrate or loose functionalities. New patches are pushed on top of the stack, existing patches may be amended to follow up with API changes, or patches are dropped. Because of the rapid dynamics and growth of Open Source projects [DR08], a significant amount of patches must manually be ported from one release of the base project to the next. Since the base project changes over time, it is necessary to continuously adapt the details of individual patches. Those adaptations can be classified in textual and higher-order conflicts [Bru+11]. Textual conflicts can be solved by manually porting the patch to the next version. In a series of patches, patches may depend on each other, so that textual conflicts in one patch lead to follow-up conflicts in further patches. Higher-order conflicts occur when a patch obtains a new (erroneous) semantic meaning after changes in the base project diverged, despite a lack of textual conflicts. Both types are known to induce high maintenance cost [MS13].

Even if the semantics of patches remain invariant over time (*e.g.*, a patch introduces identical functional modifications in subsequent revisions of the patch), their textual content can change considerably over time. To track patches with unchanged semantics over time, I use the `PaStA` methodology that places similar patches into equivalence classes R_j, so that $P = \bigcup_j R_j$. If sim were able to track the exact semantics of patches, it would hold that $\text{sim}(a, b) = \text{yes} \Leftrightarrow a \sim b$. But as sim can only compare textual changes, it follows that $\text{sim}(a, b) = \text{yes} \Rightarrow a \sim b$. This results from the fact that two similar patches between two successive versions usually have less textual changes than the first and last occurrence of the same patch. Approximate $P \approx \bigcup_j \hat{R}_j$.

Comparing Groups Against Mainline

After grouping all patches on the stacks in equivalence classes $\hat{R}_j$, a complete representative system $\mathcal{R} \subseteq P$ is chosen and compared against the commits in the base project. As representative of an equivalence class, I choose the patch with the latest version, as it very likely has the closest similarity to mainline, if it was integrated. $Q = \{(r, u) | r \in \mathcal{R}, u \in U, \text{sim}(r, u) \geq \text{ta}\}$ denotes the set of all patches that are found in the base project.

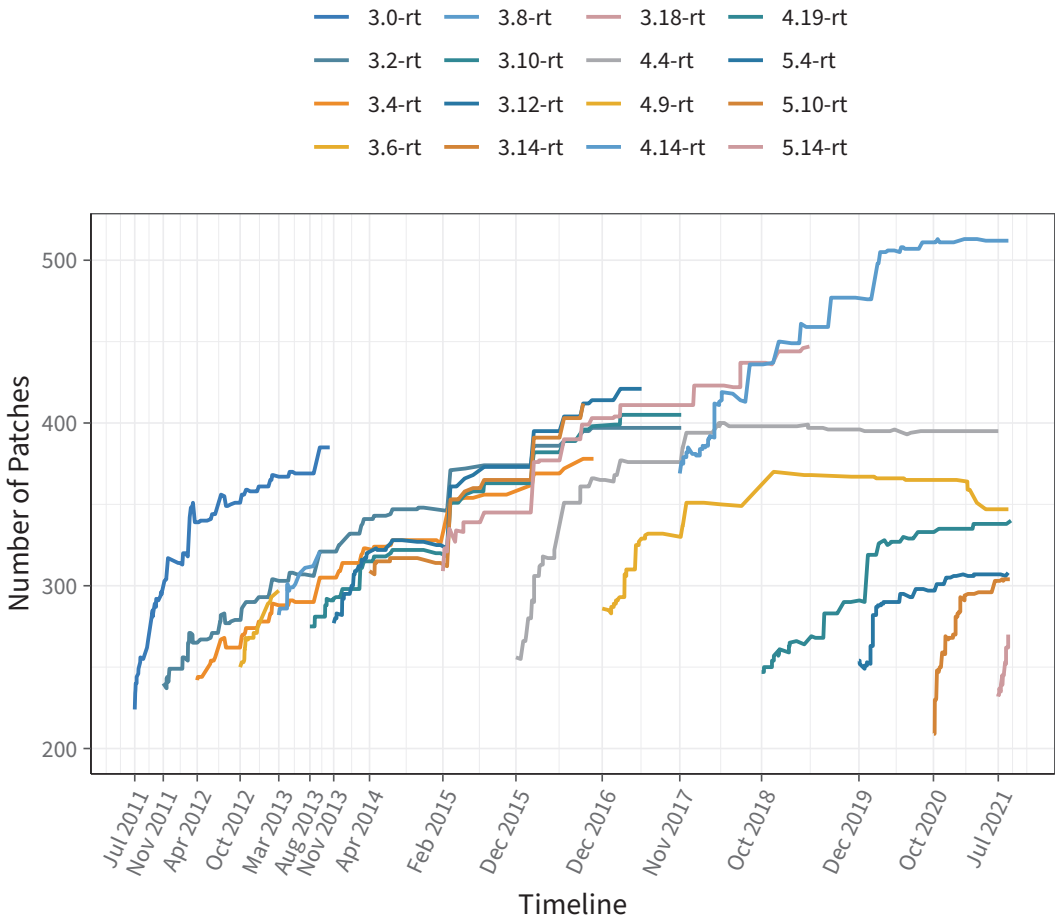

Figure A.1.: Preempt-RT patch stack: Evolution of the stack size since Linux kernel version 3.0.

A.1.2 Discussion

After grouping all patches into equivalence classes and linking them to optional commits of the base project, they can be distinguished between two temporal conditions: (1) patches that first appeared on the patch stack and later appeared in the base project (ports or *forwardports*) and (2) patches that first appeared in the base project and were ported back to older versions of the stack (backports). Patches that are not linked to a commit of the base project are called *invariant*, as they only appear on the stack. Across two releases of the patch stack, following flow of patches can be observed:

1. inflow – new patches on the patch stack and backports.
2. outflow – patches that went upstream or patches that were dropped.
3. invariant – patches that remain on the stack.

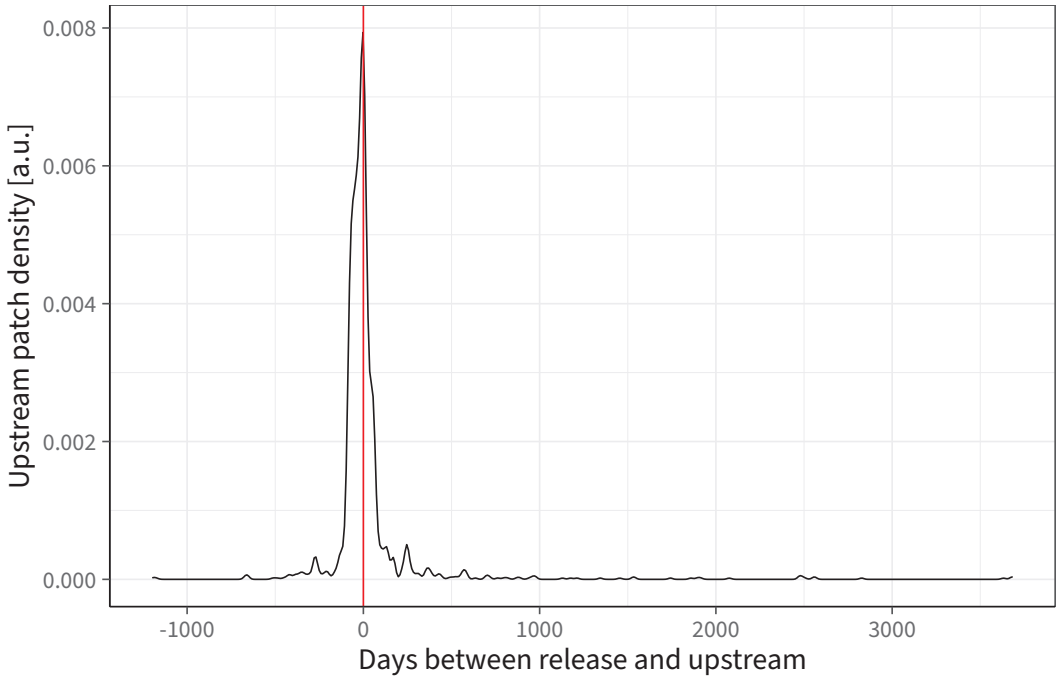

Figure A.2.: Preempt-RT patch stack: Distribution of integration duration (in days) for patches that are eventually integrated in mainline. Positive values indicate forwardports, negative values indicate backports.

In the following, I consider the evolution of the Preempt-RT patch stack as a case study: First, I inspect the temporal evolution of patch stack size, which is visualised in Fig. A.1. Among all $1{,}267$ releases of the patch stack published between July 2011 and September 2021 (that in total consist of almost $420{,}000$ patches), `PaStA` detected $2{,}995$ different groups of patches. $1{,}048$ of those groups were classified as backports, 706 groups were classified as forwardports.

Knowledge of the stack history allows for determining the composition of older patch stacks in terms of the direction of flow of constituents. Retroactively, I can determine which patches of the stack went upstream at a later point in time, and compute the amount of backported patches and invariant patches. Fig. A.3 shows the composition of the latest releases of major versions of the Preempt-RT [PRT20] patch stack. Green bars describe the amount of patches on the stack that eventually are integrated into the upstream code base, red bars describe the amount of backports, and the blue bars give the number of invariant patches.

Another covariate of interest is the time duration that a patch needs to go upstream (*i.e.*, the time between the first appearance on the patch stack and the integration with the base project). Fig. A.2 shows the result of this analysis for the Preempt-RT project. Positive values on the x-axis describe forwardports, negative values describe backports.

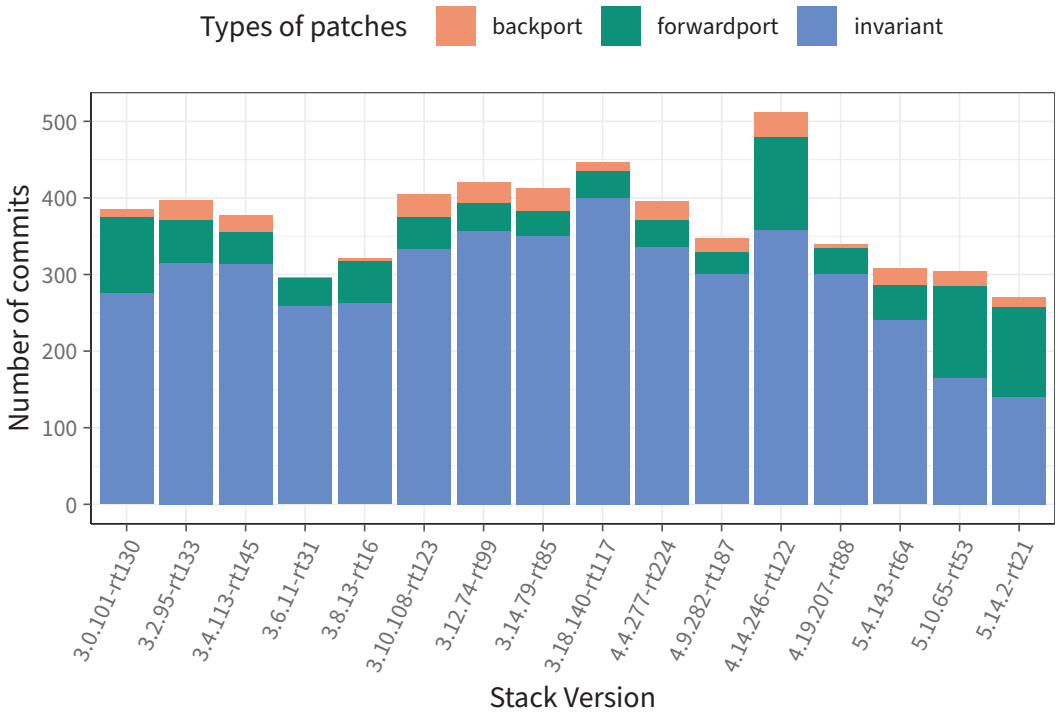

Figure A.3.: Preempt-RT patch stack: Comparing the composition of the last major releases.

There is a prominent hot spot around zero days. I interpret this spot to indicate close cooperation with the base project: backporting of many patches only takes few days while the author list of forward and backport patches overlaps.

A.1.3 Conclusion

I used the `PaStA` methodology to derive a method to systematically examine the temporal evolution of patch stacks, track non-functional properties like integrability and maintainability, and estimate the eventual economic and engineering effort required to successfully develop and maintain patch stacks. My results provide a basis for quantitative research on patch stacks, including statistical analyses and other methods that lead to actionable advice on the construction and long-term maintenance of custom extensions to OSS. The results of the analysis can be used to estimate future maintenance efforts of massive out-of-tree developments. An evaluation and visualisation of the Preempt-RT patch stack was presented as case study.

Lists

Acronyms

#AC alignment check. 127

ABS anti-lock braking system. 2

ACU airbag control unit. 2, 3, 9

AGL Automotive Grade Linux. 10

AMD Advanced Micro Devices. 145, 157

AMP asymmetric multiprocessing. 111, 131, 145

APIC Advanced Programmable Interrupt Controller. 109, 126, 133, 134, 137, 139

ARM Advanced RISC Machines. xiii, 68, 85, 106, 107, 109, 114, 115, 117, 119, 121, 122, 127–132, 138–140, 143–146, 149, 157, 188

ASIL Automotive Safety Integrity Level. 2, 8

AVR AVR (Alf and Vegard's RISC processor) microprocessor architecture. 132, 133

AVX Advanced Vector Extensions. 127

CAN Controller Area Network. 4

CAS collission avoidance system. 3

CAT Cache Allocation Technology. 110

CCD Code Clone Detection. 52

CCU Capture Compare Unit. 133

CIP Civil Infrastructure Platform. 10

COTS commercial off-the-shelf. 5, 12, 13, 18–21, 99, 102, 103, 150, 155–157

CP Control Coprocessor. 106

CP15 Control Coprocessor 15. 115, 129

CPU central processing unit. v, 5, 6, 13, 18, 19, 50, 91, 99–102, 105, 107–115, 117–123, 129, 131, 133–136, 138–144, 146–149, 155, 156

CR Control Register. 106, 109, 127

CVE Common Vulnerabilities and Exposures. 77, 78, 83–86, 89–91, 127, 131

DMA Direct Memory Access. 108, 114, 116

DRAM Dynamic Random-Access Memory (RAM). 13

DSP digital signal processing. 100, 146

ECC Error-Correcting Code. 127

ECM engine control module. 2, 3

ECU electronic control unit. 2–5

EL1 Exception Level 1. 122

EL3 Exception Level 3. 122

ELISA Enabling Linux in Safety Applications. 10

ESC electronic stability control. 2

GIC Generic Interrupt Controller. 115, 128, 129, 131, 132, 146

GPIO general purpose input/output. 132, 133, 149, 188

GPOS general purpose operating system. 15, 101

HMI Human Machine Interface. 5

HPC high-performance computing. 5

I/O input / output. 100, 111, 114, 115, 126, 131, 133

IBPB Indirect Branch Prediction Barrier. 141

IBRS Indirect Branch Restricted Speculation. 141

ICR Interrupt Control Register. 109, 126, 130, 131, 134–139, 141, 187, 188

IOMMU I/O memory management unit. 114

IPI inter-processor interrupt. 109, 126, 129, 134

IRQ Interrupt Request. 132, 133, 136, 139, 142, 188

ISA instruction set architecture. 158

KVM Kernel-based Virtual Machine. 123, 127, 134–136

L1TF Level 1 Terminal Fault. 77

LF Linux Foundation. 10

LFB Line Fill Buffer. 121

LKDP Linux Kernel Development Process. 55–58, 64, 73, 156

LKML Linux Kernel Mailing List. 40, 42, 46, 47, 53, 68, 187

LLC Last Level Cache. 110

LOC lines of code. 3

LTS Long Term Support. 60, 62, 85–87, 91

LTSI Long Term Support Initiative. 160

MC mixed-criticality. 5, 11

MC multi-core. 5, 6, 12, 13, 113

MCE machine check exception. 138

MDS Microarchitectural Data Sampling. 119, 120

ML mailing list. 16, 27, 28, 30, 35, 36, 46, 49, 51, 65, 66, 68, 73, 77, 78, 80, 87, 88

MMIO Memory Mapped I/O. 101, 106, 109, 114, 115, 125–128, 133, 134

MMU memory management unit. 4, 101, 102, 105, 107, 119, 120

MPU memory protection unit. 4

MSI-X Message Signalled Interrupts. 116, 131

MSR model-specific register. xiii, 101, 106, 109, 126, 133, 134, 136, 139, 187

MTRR Memory Type Range Register. 126

MUA mail user agent. 42

NDA non-disclosure agreement. 57

NFR non-functional requirement. 3, 10

NMI non-maskable interrupt. 127, 138

NUMA Non-Uniform Memory Access. 141

NVD National Vulnerability Database. 85

OS operating system. 9, 11, 12, 17, 100, 102, 105, 112, 119, 122, 123, 127, 131, 135, 139, 140, 142, 144, 145, 150, 151, 155

OSS open source software. v, vii, xi, 4, 8–10, 12, 14–17, 20, 25–27, 53, 55, 56, 78, 87, 88, 93, 111, 140, 155–157, 164

`PaStA` Patch Stack Analysis. 16, 20, 26, 28, 31, 33, 40, 42, 43, 46–53, 55, 57, 60, 62, 64–67, 73, 77, 80, 89, 160, 161, 163, 164, 188

PAT Page Attribute Table. 126

PCI Peripheral Component Interconnect. 113, 114, 116, 127

PIO Programmed Input/Output. 101, 126

PLC Programmable Logic Controller. 5

PMIO port-mapped I/O. 126

PSCI Power State Coordination Interface. 115, 129

PTE Page Table Entry. 120, 142

PTI Page Table Isolation. 120, 140, 142

RAM Random-Access Memory. 166

RDT Resource Director Technology. 110

RIDL Rogue In-Flight Data Load. 120

RT real-time. 9, 11, 18, 102

RTAI Real Time Application Interface. 9

RTEMS Real-Time Executive for Multiprocessor Systems. 117

RTOS real-time operating system. 9, 117

SCS safety-critical system. 7–9, 11, 16, 17

SDEI Software Delegated Exception Interface. 131, 145

SGI software generated interrupt. 129

SGX Software Guard Extensions. 120

SIL safety integrity level. 11–13

SLAT Second Level Address Translation. 113

SLOC Source Lines of Code. 145

SMC Secure Monitor Call. 131, 132

SMI System Management Interrupt. 129, 138

SMP symmetric multiprocessing. 111, 131, 148, 157

SMT simultaneous multithreading. 109, 110, 120, 121, 123

SOC System-on-a-chip. 100

SPI Serial Peripheral Interface. 106, 108, 128, 149

SSE Streaming SIMD Extensions. 127

SVM Secure Virtual Machine. 105, 115, 117

TLB Translation Lookaside Buffer. 130, 131, 140, 142

TSC Time Stamp Counter. 134, 135, 137

UART Universal Asynchronous Receiver Transmitter. 106, 114, 126

UMA Uniform Memory Access. 141

US user-space. 138–140

VCS version control system. 26, 30, 80

VE virtualisation extensions. 117

VM virtual machine. 111, 139

VMM virtual machine monitor. xii, 18, 19, 100, 102–106, 111, 112, 114, 115, 121, 125, 127, 130, 133–135, 138–145, 151

WCET Worst Case Execution Time. 104

x86 x86 microprocessor architecture. xiii, 101, 106, 109, 114, 115, 117, 122, 126, 127, 129, 131–134, 139–141, 143–146, 157

References

Own Articles

[Mau+21] Wolfgang Mauerer, Ralf Ramsauer, Edson R. Lucas F., and Stefanie Scherzinger. "Silentium! Run-Analyse-Eradicate the Noise out of the DB/OS Stack". In: *19. Fachtagung für Datenbanksysteme für Business, Technologie und Web* (Dresden, Germany). 2021 (cit. on p. 130).

[Ram+17] Ralf Ramsauer, Jan Kiszka, Daniel Lohmann, and Wolfgang Mauerer. "Look Mum, no VM Exits! (Almost)". In: *Proceedings of the 13th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT '17)* (Dubrovnik, Croatia). June 2017 (cit. on pp. 99, 111, 115).

[Ram+20] Ralf Ramsauer, Lukas Bulwahn, Daniel Lohmann, and Wolfgang Mauerer. "The Sound of Silence: Mining Security Vulnerabilities from Secret Integration Channels in Open-Source Projects". In: *Proceedings of the 12th Cloud Computing Security Workshop* (Virtual Event). Nov. 2020 (cit. on p. 55).

[RLM16] Ralf Ramsauer, Daniel Lohmann, and Wolfgang Mauerer. "Observing Custom Software Modifications: A Quantitative Approach of Tracking the Evolution of Patch Stacks". In: *Proceedings of the 12th International Symposium on Open Collaboration* (Berlin, Germany). Aug. 2016 (cit. on pp. 9, 25, 55, 159).

[RLM19] Ralf Ramsauer, Daniel Lohmann, and Wolfgang Mauerer. "The List is the Process: Reliable Pre-integration Tracking of Commits on Mailing Lists". In: *Proceedings of the 41st International Conference on Software Engineering* (Montreal, QC, Canada). IEEE. May 2019, pp. 807–818 (cit. on p. 25).

Standards and Norms

[AMD05] Advanced Micro Devices. *Secure Virtual Machine Architecture Reference Manual*. 2005 (cit. on pp. 18, 105, 107, 114, 117, 134, 148).

[ARINC653] *ARINC 653: Avionics Application Software Standard Interface*. Aeronautical Radio Incorporated (cit. on p. 3).

[ARM13a] ARM Ltd. *ARM Architecture Reference Manual*. 2013 (cit. on pp. 114, 117).

[ARM13b] ARM Ltd. *ARM Generic Interrupt Controller, Architecture version 2.0*. 2013 (cit. on pp. 109, 132).

[ARM16] ARM Ltd. *ARM Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0 and version 4.0*. 2016 (cit. on pp. 105, 107, 109, 128, 132, 146).

[ARM17] ARM Ltd. *Software Delegated Exception Interface (SDEI) - Platform Design Document*. 2017 (cit. on p. 131).

[ARM20a] ARM Ltd. *Arm Architecture Reference Manual, Armv8, for Armv8-A architecture profile*. 2020 (cit. on p. 117).

[ARM20b] ARM Ltd. *Whitepaper – Cache Speculation Side-channels*. 2020 (cit. on pp. 122, 131, 140, 143).

[CAN91] Robert Bosch GmbH. *CAN specification Version 2.0*. 1991 (cit. on p. 5).

[DO-178B] RTCA SC-167, EUROCAE WG-12. *DO-178B, Software Considerations in Airborne Systems and Equipment Certification* (cit. on p. 3).

[IEC60601] *IEC 60601: Medical Electrical Devices*. International Electrotechnical Commission (cit. on p. 3).

[IEC60880] *IEC 60880: Nuclear Power Plants – Instrumentation and Control Systems Important to Safety – Software Aspects for Computer-Based Systems*. International Electrotechnical Commission (cit. on p. 3).

[IEC61508] *IEC 61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems*. International Electrotechnical Commission (cit. on pp. 3, 10, 12, 13).

[IEC62304] *IEC 62304: Medical device software – Software life cycle processes*. International Electrotechnical Commission (cit. on p. 3).

[Int10] Intel Corporation. *Intel® 64 Architecture x2APIC Specification*. Mar. 2010 (cit. on pp. 105, 107, 134).

[Int15] Intel Corporation. *Improving Real-Time Performance by Utilizing Cache Allocation Technology Enhancing Performance via Allocation of the Processor's Cache*. Apr. 2015 (cit. on p. 110).

[Int18a] Intel Corporation. *Intel Microcode Revision Guidance*. Aug. 2018 (cit. on pp. 138, 141).

[Int18b] Intel Corporation. *Intel® Virtualization Technology for Directed I/O*. Rev. 3.0. June 2018 (cit. on pp. 18, 105, 107, 131, 148).

[Int19a] Intel Corporation. *Intel Resource Director Technology (Intel RDT)*. 2019. URL: `https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html` (cit. on p. 110).

[Int19b] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C & 3D): System Programming Guide*. May 2019 (cit. on pp. 117, 135, 140).

[ISO26262] *ISO 26262: Road vehicles – Functional safety*. International Organization for Standardization (cit. on pp. 2, 3, 8, 11).

[KRNL20] The Linux Kernel development community. *Linux Kernel Documentation on MDS - Microarchitectural Data Sampling*. see Documentation/admin-guide/hw-vuln/mds.rst (cit. on p. 121).

Related Work

[Ada18] Bram Adams. personal communication. June 21, 2018 (cit. on p. 49).

[AGL18] The Linux Foundation. "The Automotive Grade Linux – Software Defined Connected Car Architecture". In: (2018) (cit. on p. 10).

[Alj+09] Husain Aljazzar, Manuel Fischer, Lars Grunske, et al. "Safety analysis of an airbag system using probabilistic FMEA and probabilistic counterexamples". In: *2009 Sixth International Conference on the Quantitative Evaluation of Systems*. IEEE. 2009, pp. 299–308 (cit. on p. 8).

[Ami+09] Enrique Amigó, Julio Gonzalo, Javier Artiles, and Felisa Verdejo. "A comparison of extrinsic clustering evaluation metrics based on formal constraints". In: *Information retrieval* 12.4 (2009) (cit. on p. 43).

[And02] Ross Anderson. "Security in Open versus Closed Systems—The dance of Boltzmann, Coase and Moore". In: *Open Source Software: Economics, Law and Policy*. June 2002 (cit. on p. 86).

[Aro+10] Ashish Arora, Ramayya Krishnan, Rahul Telang, and Yubao Yang. "An Empirical Analysis of Software Vendors' Patch Release Behavior: Impact of Vulnerability Disclosure". In: *Information Systems Research* 21 (Mar. 2010) (cit. on p. 85).

[AT06] Christian Arwin and Seyed MM Tahaghoghi. "Plagiarism detection across programming languages". In: *Proceedings of the 29th Australasian Computer Science Conference-Volume 48*. 2006 (cit. on p. 52).

[Bac+09] Alberto Bacchelli, Marco D'Ambros, Michele Lanza, and Romain Robbes. "Benchmarking lightweight techniques to link e-mails and source code". In: *WCRE'09. 16th Working Conference on Reverse Engineering*. 2009 (cit. on p. 52).

[Bar16] Maxim Baryshnikov. “Jailhouse hypervisor”. Czech Technical University in Prague, 2016 (cit. on p. 117).

[Bat20] Christopher Batten. *Lecture notes ECE 5745 Complex Digital ASIC Design*. 2020 (cit. on p. 6).

[Bau+09] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, et al. “The Multikernel: A new OS Architecture for Scalable Multicore Systems”. In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating systems principles*. ACM. 2009, pp. 29–44 (cit. on p. 147).

[Bax+98] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant’Anna, and Lorraine Bier. “Clone detection using abstract syntax trees”. In: *Proceedings of the International Conference on Software Maintenance*. 1998 (cit. on p. 52).

[BB18] BusyBox authors. *BusyBox Project*. Aug. 2018. URL: `https://busybox.net/` (cit. on p. 31).

[BBL76] Barry W Boehm, John R Brown, and Mlity Lipow. “Quantitative evaluation of software quality”. In: *Proceedings of the 2nd international conference on Software engineering*. IEEE Computer Society Press. 1976, pp. 592–605 (cit. on p. 3).

[BD13] Alan Burns and Robert Davis. “Mixed criticality systems-a review”. In: *Department of Computer Science, University of York, Tech. Rep* (2013), pp. 1–69 (cit. on pp. 4, 17, 25, 99).

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, et al. “Xen and the Art of Virtualization”. In: *Proc. of the 19th ACM Symposium on Operating Systems Principles*. 2003 (cit. on pp. 17, 105, 112).

[BGD07] Christian Bird, Alex Gourley, and Prem Devanbu. “Detecting patch submission and acceptance in oss projects”. In: *Proceedings of the 4th International Workshop on Mining Software Repositories*. MSR’07. 2007 (cit. on pp. 28, 29).

[Bir+06] Christian Bird, Alex Gourley, Prem Devanbu, Michael Gertz, and Anand Swaminathan. “Mining Email Social Networks”. In: *Proceedings of the 3rd International Workshop on Mining Software Repositories*. MSR’06. 2006 (cit. on pp. 27, 42, 51).

[Bir+08] Christian Bird, David Pattison, Raissa D’Souza, Vladimir Filkov, and Premkumar Devanbu. “Latent Social Structure in Open Source Projects”. In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2008 (cit. on p. 25).

[Bir+09] C. Bird, P. C. Rigby, E. T. Barr, et al. “The Promises and Perils of Mining Git”. In: *Mining Software Repositories, 2009. MSR ’09. 6th IEEE International Working Conference on*. May 2009, pp. 1–10 (cit. on pp. 30, 32, 160).

[Bis+14] Matt Bishop, Heather M Conboy, Huong Phan, et al. “Insider threat identification by process analysis”. In: *IEEE Security and Privacy Workshops*. 2014 (cit. on p. 86).

[BLD11] Alberto Bacchelli, Michele Lanza, and Marco D’Ambros. “Miler: A toolset for exploring email data”. In: *Proceedings of the 33rd International Conference on Software Engineering*. 2011 (cit. on p. 52).

[BLR10] Alberto Bacchelli, Michele Lanza, and Romain Robbes. “Linking e-mails and source code artifacts”. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 2010 (cit. on p. 52).

[BN06] Eric W Biederman and Linux Networx. “Multiple instances of the global linux namespaces”. In: *Proceedings of the Linux Symposium*. Vol. 1. 2006, pp. 101–112 (cit. on p. 13).

[Bro06] Manfred Broy. “Challenges in Automotive Software Engineering”. In: *Proceedings of the 28th International Conference on Software Engineering (ICSE)* (Shanghai, China). New York, NY, USA: ACM Press, 2006, pp. 33–42 (cit. on pp. 4–7).

[Bru+11] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. “Proactive Detection of Collaboration Conflicts”. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ESEC/FSE ’11. Szeged, Hungary: ACM, 2011, pp. 168–178 (cit. on p. 161).

[BTH12] Nicolas Bettenburg, Stephen W Thomas, and Ahmed E Hassan. “Using fuzzy code search to link code fragments in discussions to source code”. In: *16th European Conference on Software Maintenance and Reengineering (CSMR)*. 2012 (cit. on p. 52).

[Bug+96] Edouard Bugnion, Jennifer M Anderson, Todd C Mowry, Mendel Rosenblum, and Monica S Lam. “Compiler-directed page coloring for multiprocessors”. In: *ACM SIGPLAN Notices* 31.9 (1996), pp. 244–255 (cit. on p. 145).

[Bul17a] Lukas Bulwahn. “Is Linux Kernel Development Good Enough to Make Your Life Depend on It?” In: *Embedded Linux Conference Europe*. 2017 (cit. on pp. 15, 25).

[Bul17b] Lukas Bulwahn. “Is Linux Kernel Development Good Enough to Make Your Life Dependon it? Progress on Procedures & Methods to Qualify the Linux Kernel Development Process”. In: *Embedded Linux Conference Europe (ELCE17)*. Oct. 2017 (cit. on p. 100).

[Can+18] Claudio Canella, Jo Van Bulck, Michael Schwarz, et al. “A systematic evaluation of transient execution attacks and defenses”. In: *arXiv preprint arXiv:1811.05441* (2018) (cit. on pp. 19, 113, 115, 117).

[CCR04] Hasan Cavusoglu, Huseyin Cavusoglu, and Srinivasan Raghunathan. “Emerging Issues in Responsible Vulnerability Disclosure”. In: *Workshop on Information Technology and Systems*. WITS. 2004 (cit. on p. 78).

[Chá+17] Alexander Chávez, Isabella Ferreira, Eduardo Fernandes, Diego Cedrim, and Alessandro Garcia. “How Does Refactoring Affect Internal Quality Attributes? A Multi-Project Study”. In: *Proceedings of the 31st Brazilian Symposium on Software Engineering*. SBES’17. 2017 (cit. on p. 86).

[Chu+12] Lawrence Chung, Brian A Nixon, Eric Yu, and John Mylopoulos. *Non-functional requirements in software engineering*. Vol. 5. Springer Science & Business Media, 2012 (cit. on p. 3).

[CIP17] The Linux Foundation. “The Civil Infrastructure Platform – Whitepaper”. In: (2017) (cit. on p. 10).

[CJ12] Georgina Cosma and Mike Joy. “An approach to source-code plagiarism detection and investigation using latent semantic analysis”. In: *IEEE transactions on computers* 61.3 (2012) (cit. on p. 52).

[Cor11] Jonathan Corbet. “How the Development Process Works”. In: *Linux docs*. The Linux Foundation. 2011 (cit. on p. 25).

[Cor19] Jonathan Corbet. “What to do about CVE numbers”. In: *Linux Weekly News (LWN)* (2019) (cit. on pp. 78, 90, 92).

[CRM10] Alfons Crespo, Ismael Ripoll, and Miguel Masmano. “Partitioned Embedded Architecture based on Hypervisor: The XtratuM approach”. In: *Proceedings of the 8th European Dependable Computing Conference (EDCC)*. IEEE. 2010 (cit. on pp. 17, 146).

[Cul+10] Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, et al. “Predictability Considerations in the Design of Multi-Core Embedded Systems”. In: *Proceedings of Embedded Real Time Software and Systems* 36 (2010), p. 42 (cit. on p. 118).

[CVE17] *CVE-2018-3615, CVE-2018-3620, and CVE-2018-3646*. Dec. 2017 (cit. on p. 77).

[CVE19] *CVE-2019-1125*. Nov. 2019 (cit. on pp. 83, 91).

[CW02] Steve Christey and Chris Wysopal. “Responsible vulnerability disclosure process”. In: *IETF draft* (2002) (cit. on p. 62).

[Die+12] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. “A Robust Approach for Variability Extraction from the Linux Build System”. In: *Proceedings of the 16th International Software Product Line Conference - Volume 1*. SPLC ’12. Salvador, Brazil: Association for Computing Machinery, 2012, pp. 21–30 (cit. on p. 9).

[Die19] Christian Dietrich. “Interaction-Aware Analysis and Optimization of Real-Time Application and Operating System”. PhD thesis. Leibniz Universität Hannover, 2019 (cit. on p. 150).

[DL17] Christian Dietrich and Daniel Lohmann. “OSEK-V: Application-Specific RTOS Instantiation in Hardware”. In: *Proceedings of the 2017 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES ’17)* (Barcelona, Spain). New York, NY, USA: ACM Press, June 2017 (cit. on pp. 100, 147).

[DR08] Amit Deshpande and Dirk Riehle. “Open Source Development, Communities and Quality: IFIP 20th World Computer Congress, Working Group 2.3 on Open Source Software, September 7-10, 2008, Milano, Italy”. In: Boston, MA: Springer US, 2008. Chap. The Total Growth of Open Source, pp. 197–209 (cit. on p. 161).

[DRD99] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. “A language independent approach for detecting duplicated code”. In: *Software Maintenance, 1999.(ICSM’99) Proceedings. IEEE International Conference on*. IEEE. 1999, pp. 109–118 (cit. on p. 35).

[Dre08] Ulrich Drepper. “The Cost of Virtualization”. In: *Queue* 6.1 (Jan. 2008) (cit. on pp. 105, 115, 130, 139).

[Edg19] Jake Edge. “CVE-less vulnerabilities”. In: *Linux Weekly News (LWN)* (2019) (cit. on p. 89).

[EKO95] Dawson R. Engler, M. Frans Kaashoek, and James O’Toole. “Exokernel: An Operating System Architecture for Application-Level Resource Management”. In: *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*. 1995, pp. 251–266 (cit. on pp. 112, 114).

[Ere03] Justin R Erenkrantz. “Release management within open source projects”. In: *Proceedings of the 3rd. Workshop on Open Source Software Engineering*. 2003 (cit. on pp. 27, 30).

[FM83] Edward B Fowlkes and Colin L Mallows. “A method for comparing two hierarchical clusterings”. In: *Journal of the American statistical association* 78.383 (1983) (cit. on p. 43).

[Fou18a] Linux Foundation. *Automotive Grade Linux*. Aug. 2018. URL: https://www.automotivelinux.org/ (cit. on p. 25).

[Fou18b] Linux Foundation. *Civil Infrastructure Platform*. Aug. 2018. URL: https://www.cip-project.org/ (cit. on p. 25).

[Fou19] Linux Foundation. *Civil Infrastructure Platform (CIP)*. 2019. URL: https://www.cip-project.org (cit. on pp. 5, 99).

[Fre+10] Stefan Frei, Dominik Schatzmann, Bernhard Plattner, and Brian Trammell. “Modeling the security ecosystem-the dynamics of (in) security”. In: *Economics of Information Security and Privacy*. 2010 (cit. on p. 62).

[GAH16] Daniel M German, Bram Adams, and Ahmed E Hassan. “Continuously mining distributed version control systems: an empirical study of how Linux uses Git”. In: *Empirical Software Engineering* 21.1 (2016) (cit. on p. 30).

[GBK11] David Gullasch, Endre Bangerter, and Stephan Krenn. “Cache games–Bringing access-based cache attacks on AES to practice”. In: *2011 IEEE Symposium on Security and Privacy*. IEEE. 2011, pp. 490–505 (cit. on pp. 110, 118).

[GIT16] Git Community. *Git Version Control System*. 2016. URL: https://git-scm.com/ (cit. on p. 159).

[GKW19] Thomas Gleixner, John Kacur, and Clark Williams. *The Cyclictest Real-Time Benchmark*. 2019. URL: https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start (cit. on p. 138).

[Gol73] Robert P. Goldberg. *Architectural Principles for Virtual Computer Systems*. Tech. rep. Harvard University Cambridge, 1973 (cit. on p. 111).

[GS12] Mário Luís Guimarães and António Rito Silva. “Improving Early Detection of Software Merge Conflicts”. In: *Proceedings of the 34th International Conference on Software Engineering*. ICSE ’12. Zurich, Switzerland: IEEE Press, 2012, pp. 342–352 (cit. on p. 159).

[Gui14] Nicholas Mc Guire. “SIL2LinuxMP: GNU/Linux Multicore platform for safety related systems”. In: *Linux Plumbers Conference*. 2014 (cit. on pp. 12, 13).

[Hei08] Gernot Heiser. “The role of virtualization in embedded systems”. In: *Proceedings of the 1st workshop on Isolation and Integration in Embedded Systems (IIES)*. 2008 (cit. on pp. 17, 102).

[Hem+13] Hadi Hemmati, Sarah Nadi, Olga Baysal, et al. “The MSR Cookbook: Mining a decade of research”. In: *10th Working Conference on Mining Software Repositories*. MSR. 2013 (cit. on p. 86).

[Heo+15] Tejun Heo, J Weiner, V Davydov, et al. “Control group v2”. In: (2015) (cit. on p. 13).

[Her07] James D Herbsleb. “Global software engineering: The future of socio-technical coordination”. In: *Future of Software Engineering. FOSE’07*. 2007 (cit. on p. 27).

[HH08] André Hergenhan and Gernot Heiser. “Operating systems technology for converged ECUs”. In: *6th Conference on Embedded Security in Cars Conf (ESCAR)*. 2008 (cit. on pp. 3, 5).

[HNH03] Guido Hertel, Sven Niedner, and Stefanie Herrmann. “Motivation of software developers in Open Source projects: an Internet-based survey of contributors to the Linux kernel”. In: *Research policy* 32.7 (2003) (cit. on p. 27).

[Hua+16] Zhen Huang, Mariana DAngelo, Dhaval Miyani, and David Lie. “Talos: Neutralizing Vulnerabilities with Security Workarounds for Rapid Response”. In: *IEEE Symposium on Security and Privacy*. SP. 2016 (cit. on p. 85).

[Int18c] Intel Corporation. *Resources and Response to Side Channel L1 Terminal Fault*. `https://www.intel.com/content/www/us/en/architecture-and-technology/l1tf.html`. 2018 (cit. on p. 77).

[Jää+12] Antti Jääskeläinen, Mika Katara, Shmuel Katz, and Heikki Virtanen. “Verification of Safety-Critical Systems: A Case Study Report on Using Modern Model Checking Tools”. In: *6th International Workshop on Systems Software Verification*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2012 (cit. on p. 10).

[JAG13] Yujuan Jiang, Bram Adams, and Daniel M German. “Will my patch make it? and how fast?: Case study on the linux kernel”. In: *Proceedings of the 10th Working Conference on Mining Software Repositories*. MSR’13. 2013 (cit. on pp. 45, 51, 80).

[Jia+07] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. “Deckard: Scalable and accurate tree-based detection of code clones”. In: *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society. 2007, pp. 96–105 (cit. on pp. 35, 52).

[Jia+14] Yujuan Jiang, Bram Adams, Foutse Khomh, and Daniel M German. "Tracing back the history of commits in low-tech reviewing environments: a case study of the linux kernel". In: *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM. 2014 (cit. on pp. 32, 47, 49, 52).

[Job+17] Mitchell Joblin, Sven Apel, Claus Hunsen, and Wolfgang Mauerer. "Classifying developers into core and peripheral: An empirical study on count and network metrics". In: *Proceedings of the 39th International Conference on Software Engineering*. ICSE'17. 2017 (cit. on pp. 27, 86).

[KG19] Beckhoff Automation GmbH & Co. KG. 2019. URL: `https://www.beckhoff.com/` (cit. on pp. 5, 99).

[Kis09] Jan Kiszka. "A Linux/Xenomai Platform for High-Performance Magnetic Resonance Scanners". In: *Xenomai User Meeting 2009 (XUM2009)*. 2009 (cit. on pp. 5, 9, 99).

[Kis11] Jan Kiszka. "Towards Linux as a Real-Time Hypervisor". In: *Proceedings of the 11th Real-Time Linux Workshop*. 2011 (cit. on p. 17).

[Kle+14] Gerwin Klein, June Andronick, Kevin Elphinstone, et al. "Comprehensive formal verification of an OS microkernel". In: *ACM Transactions on Computer Systems (TOCS)* 32.1 (2014), pp. 1–70 (cit. on p. 10).

[Kle09] Gerwin Klein. "Operating System Verification — An Overview". In: *Sādhanā* 34.1 (Feb. 2009) (cit. on p. 117).

[Kle19] Andi Kleen. *Linux Kernel MDS mitigation patches*. Available at `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=95310e348a321b45fb746c176961d4da72344282`. 2019 (cit. on p. 121).

[Kni02] John C Knight. "Safety Critical Systems: Challenges and Directions". In: *Proceedings of the 24th International Conference on Software Engineering*. 2002 (cit. on pp. 1, 8).

[Koc+19] Paul Kocher, Daniel Genkin, Daniel Gruss, et al. "Spectre Attacks: Exploiting Speculative Execution". In: *40th IEEE Symposium on Security and Privacy (S&P'19)*. S&P. 2019 (cit. on pp. 19, 90, 91, 109, 110, 113, 115, 117–119, 131).

[Kop11] Hermann Kopetz. *Real-time systems: design principles for distributed embedded applications*. Springer Science & Business Media, 2011 (cit. on p. 9).

[KP13] Florian Kammüller and Christian W Probst. "Invalidating policies using structural information". In: *IEEE Security and Privacy Workshops*. 2013 (cit. on p. 86).

[Kro07] Greg Kroah-Hartman. “Linux kernel development”. In: *Proceedings of the Linux Symposium*. 2007 (cit. on p. 60).

[Kro16] Greg Kroah-Hartman. “Why kernel development still uses email”. In: *Linux Weekly News (LWN)* (2016) (cit. on p. 58).

[Kro19] Greg Kroah-Hartman. “MDS, Fallout, Zombieland & Linux”. In: *Embedded Linux Conference Europe (ELCE)*. 2019 (cit. on pp. 83, 86, 89).

[Kro20] Greg Kroah-Hartman. personal communication. Jan. 2020 (cit. on pp. 62, 79, 87, 90, 91).

[KW07] Robert Kaiser and Stephan Wagner. “Evolution of the PikeOS microkernel”. In: *First International Workshop on Microkernels for Embedded Systems*. 2007, p. 50 (cit. on pp. 9, 17, 112, 146).

[KZ09] Andrew Kornecki and Janusz Zalewski. “Certification of software for real-time safety-critical systems: state of the art”. In: *Innovations in Systems and Software Engineering* 5.2 (2009), pp. 149–161 (cit. on p. 7).

[LDW11] Ye Li, Matthew Danish, and Richard West. “Quest-V: A virtualized multikernel for high-confidence systems”. In: (2011) (cit. on p. 112).

[Lev66] Vladimir I Levenshtein. “Binary codes capable of correcting deletions, insertions, and reversals”. In: *Soviet physics doklady*. Vol. 10. 8. 1966 (cit. on p. 37).

[LF18] The Linux Foundation. “Technical Charter (the “Charter”) for ELISA Project a Series of LF Projects, LLC”. In: (2018) (cit. on pp. 10, 25).

[Li+19] Hao Li, Xuefei Xu, Jinkui Ren, and Yaozu Dong. “ACRN: a big little hypervisor for IoT development”. In: *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 2019, pp. 31–44 (cit. on p. 17).

[Lin20] Linux Kernel Community. *Linux – How the development process works*. 2020 (cit. on p. 58).

[Lip+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, et al. “Meltdown: Reading Kernel Memory from User Space”. In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018 (cit. on pp. 19, 90, 109, 115, 117–119, 140).

[LKP18] Linux Kernel Community. *A guide to the Kernel Development Process*. 2018 (cit. on p. 13).

[LP17] Frank Li and Vern Paxson. “A large-scale empirical study of security patches”. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 2017 (cit. on p. 85).

[LWM14] Ye Li, Richard West, and Eric Missimer. “A Virtualized Separation Kernel for Mixed Criticality Systems”. In: *Proceedings of the 10th USENIX International Conf. on Virtual Execution Environments (VEE)*. Salt Lake City, Utah, USA: ACM, 2014, pp. 201–212 (cit. on p. 114).

[Man+00] Paolo Mantegazza, E Bianchi, Lorenzo Dozio, et al. “RTAI: Real-time application interface”. In: (2000) (cit. on p. 9).

[Mau10] Wolfgang Mauerer. *Professional Linux kernel architecture*. John Wiley & Sons, 2010 (cit. on p. 60).

[MES13] David MacKenzie, Paul Eggert, and Richard Stallman. *Comparing and Merging Files*. 2013 (cit. on pp. 29, 35, 37).

[Min+19] Marina Minkin, Daniel Moghimi, Moritz Lipp, et al. “Fallout: Reading Kernel Writes From User Space”. In: (2019) (cit. on pp. 19, 109, 115, 117).

[MJ13] Wolfgang Mauerer and Michael C Jaeger. “Open source engineering processes”. In: *it–Information Technology* 55.5 (2013) (cit. on p. 25).

[Moy13] Bryon Moyer. *Real World Multicore Embedded Systems*. Newnes, 2013 (cit. on p. 6).

[MRC05] Miguel Masmano, Ismael Ripoll, and Alfons Crespo. “An overview of the XtratuM nanokernel”. In: *Proceedings of the 1st Workshop on Operating System Platforms for Embedded Real-Time Applications (OSPERT)*. 2005 (cit. on pp. 17, 146).

[MS13] Hisao Munakata and Tsugikazu Shibata. *The Economic Value of the Long-Term Support Initiative (LTSI)*. Linux Foundation. 2013 (cit. on p. 161).

[Mül+14] Rainer Müller, Daniel Danner, Wolfgang Schröder-Preikschat, and Daniel Lohmann. “MultiSloth: An Efficient Multi-Core RTOS using Hardware-Based Scheduling”. In: *Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS ’14)* (Madrid, Spain). Washington, DC, USA: IEEE Computer Society Press, 2014, pp. 289–198 (cit. on p. 100).

[Mur+19] Emerson Murphy-Hill, Edward K. Smith, Caitlin Sadowski, et al. “Do Developers Discover New Tools on the Toilet?” In: *Proceedings of the 41st International Conference on Software Engineering*. ICSE. 2019 (cit. on p. 89).

[MW09] Andrew Meneely and Laurie Williams. “Secure open source collaboration: an empirical study of Linus’ law”. In: *Proceedings of the 16th ACM conference on Computer and communications security*. 2009 (cit. on p. 58).

[OSADL14] OSADL eG. *SIL2LinuxMP – Letter of Intent V11*. 2014 (cit. on p. 12).

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. “Cache attacks and countermeasures: the case of AES”. In: *Cryptographers’ track at the RSA conference*. Springer. 2006, pp. 1–20 (cit. on p. 118).

[PG74] Gerald J Popek and Robert P Goldberg. “Formal requirements for virtualizable third generation architectures”. In: *Communications of the ACM* 17.7 (1974), pp. 412–421 (cit. on pp. 103, 115).

[Pin+14] Sandro Pinto, Daniel Oliveira, Jorge Pereira, et al. “Towards a lightweight embedded virtualization architecture exploiting arm trustzone”. In: *2014 IEEE Emerging Technology and Factory Automation (ETFA)*. IEEE. 2014 (cit. on pp. 17, 146).

[Pin+17] Sandro Pinto, Jorge Pereira, Tiago Gomes, Adriano Tavares, and Jorge Cabral. “LTZVisor: TrustZone is the key”. In: *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2017 (cit. on p. 17).

[PMB18] Andreas Platschek, Nicholas Mc Guire, and Lukas Bulwahn. “Certifying Linux: Lessons Learned in Three Years of SIL2LinuxMP”. In: *Proceedings of the Embedded World Conference*. 2018 (cit. on pp. 10, 12, 13, 112).

[Pro+18] Andrew Prout, William Arcand, David Bestor, et al. “Measuring the Impact of Spectre and Meltdown”. In: *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE. 2018, pp. 1–5 (cit. on p. 138).

[PRT20] *Linux Kernel PreemptRT real-time extension*. 2020. URL: `https://rt.wiki.kernel.org` (cit. on pp. 9, 123, 160, 163).

[Qui+09] Morgan Quigley, Ken Conley, Brian Gerkey, et al. “ROS: an open-source Robot Operating System”. In: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan. 2009, p. 5 (cit. on pp. 5, 99).

[Ray99] Eric Raymond. “The cathedral and the bazaar”. In: *Knowledge, Technology & Policy* 12.3 (1999) (cit. on p. 77).

[RH07] Andrew Rosenberg and Julia Hirschberg. “V-measure: A conditional entropy-based external cluster evaluation measure”. In: *Proceedings of the 2007 joint conference on empirical methods in natural language processing and computational natural language learning (EMNLP-CoNLL)*. 2007 (cit. on p. 43).

[Rie16] Rik van Riel. “Real-time KVM from the ground up”. In: *LinuxCon NA*. 2016 (cit. on pp. 17, 123, 134).

[RMF19] Federico Reghenzani, Giuseppe Massari, and William Fornaciari. “The real-time linux kernel: A survey on Preempt_RT”. In: *ACM Computing Surveys* 52 (Feb. 2019), pp. 1–36 (cit. on p. 9).

[Rom85] G-C Roman. “A taxonomy of current issues in requirements engineering”. In: *Computer* 4 (1985), pp. 14–23 (cit. on p. 3).

[Rut06] Joanna Rutkowska. “Introducing blue pill”. In: *The official blog of the invisiblethings.org* (2006) (cit. on p. 111).

[Sæb+09] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. “Detecting code clones in binary executables”. In: *Proceedings of the eighteenth international symposium on Software testing and analysis*. 2009 (cit. on p. 52).

[Sch+08] Adrian Schüpbach, Simon Peter, Andrew Baumann, et al. “Embracing diversity in the Barrelfish manycore operating system”. In: *Proceedings of the Workshop on Managed Many-Core Systems*. Vol. 27. 2008 (cit. on p. 147).

[Sch+19a] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, et al. “RIDL: Rogue In-flight Data Load”. In: *S&P*. May 2019 (cit. on pp. 19, 109, 110, 115, 117, 121).

[Sch+19b] Michael Schwarz, Moritz Lipp, Daniel Moghimi, et al. “ZombieLoad: Cross-Privilege-Boundary Data Sampling”. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. CCS. 2019 (cit. on pp. 83, 91).

[Sco+20] Claudio Scordino, Ida Maria Savino, Luca Cuomo, et al. “Real-Time Virtualization For Industrial Automation”. In: *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. Vol. 1. IEEE. 2020, pp. 353–360 (cit. on p. 145).

[SH09] Randy Smith and Susan Horwitz. “Detecting and measuring similarity in code clones”. In: *Proceedings of the International Workshop on Software Clones (IWSC)*. 2009 (cit. on p. 35).

[SK10a] Udo Steinberg and Bernhard Kauer. “NOVA: A Microhypervisor-based Secure Virtualization Architecture”. In: *Proceedings of the 5th European Conference on Computer Systems*. EuroSys ’10. 2010 (cit. on p. 17).

[SK10b] Udo Steinberg and Bernhard Kauer. “NOVA: a microhypervisor-based secure virtualization architecture”. In: *Proceedings of the 5th European conference on Computer systems*. ACM. 2010, pp. 209–222 (cit. on p. 112).

[SMR08] Hinrich Schütze, Christopher D Manning, and Prabhakar Raghavan. *Introduction to information retrieval*. Vol. 39. 2008 (cit. on pp. 35, 43, 44).

[SN05] Jim Smith and Ravi Nair. *Virtual machines: versatile platforms for systems and processes*. Elsevier, 2005 (cit. on p. 103).

[SPL95] Olin Sibert, Phillip A Porras, and Robert Lindell. “The Intel 80x86 processor architecture: pitfalls for secure systems”. In: *Proceedings of the 1995 Symposium on Security and Privacy*. IEEE. 1995, pp. 211–222 (cit. on p. 118).

[SSL12] Muhammad Shahzad, Muhammad Zubair Shafiq, and Alex X Liu. “A large scale exploratory analysis of software vulnerability life cycles”. In: *34th International Conference on Software Engineering*. ICSE. 2012 (cit. on p. 85).

[The20] The Kernel Community. *Submitting patches: the essential guide to getting your code into the kernel*. 2020 (cit. on pp. 62, 78).

[UNR+05] Rich Uhlig, Gil Neiger, Dion Rodgers, et al. “Intel virtualization technology”. In: *Computer* 38.5 (2005) (cit. on pp. 18, 105, 109, 114, 117, 134, 148).

[Val+07] Giuseppe Valetto, Mary Helander, Kate Ehrlich, et al. “Using Software Repositories to Investigate Socio-technical Congruence in Development Projects”. In: *Proceedings of the 4th International Workshop on Mining Software Repositories*. MSR’07. 2007 (cit. on p. 27).

[Van+18] Jo Van Bulck, Marina Minkin, Ofir Weisse, et al. “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution”. In: *Proceedings of the 27th USENIX Security Symposium*. See also technical report Foreshadow-NG [Wei+18]. USENIX Association, Aug. 2018 (cit. on pp. 19, 77, 109, 115, 117, 120, 185).

[Ves07] Steve Vestal. “Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance”. In: *28th IEEE International Real-Time Systems Symposium (RTSS)*. 2007 (cit. on p. 146).

[Vya13] Valeriy Vyatkin. “Software Engineering in Industrial Automation: State-of-the-art Review”. In: *IEEE Transactions on Industrial Informatics* 9.3 (2013) (cit. on p. 1).

[Wei+18] Ofir Weisse, Jo Van Bulck, Marina Minkin, et al. “Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution”. In: *Technical report* (2018). See also USENIX Security paper Foreshadow [Van+18] (cit. on pp. 19, 77, 109, 110, 115, 117, 120, 185).

[Wen+15] Thomas F. Wendt, Wolfgang Bernhart, Jiten Behl, Dagan Mishoulam, and Ethan Goldsmith. “Consolidation in Vehicle Electronic Architectures”. In: *Think Act* (July 2015) (cit. on p. 3).

[Wir71] Niklaus Wirth. “Program development by stepwise refinement”. In: *Communications of the ACM* 14.4 (1971) (cit. on pp. 25, 27).

[Wol94] Wayne H. Wolf. “Hardware-Software Co-Design of Embedded Systems”. In: *Proceedings of the IEEE* 82.7 (1994), pp. 967–989 (cit. on p. 4).

[Xi+11] Sisu Xi, Justin Wilson, Chenyang Lu, and Christopher Gill. “RT-Xen: Towards real-time hypervisor scheduling in Xen”. In: *2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*. Oct. 2011 (cit. on pp. 17, 146).

[YF14] Yuval Yarom and Katrina Falkner. “FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack”. In: *23rd USENIX Security Symposium*. 2014, pp. 719–732 (cit. on pp. 110, 118, 119).

[Yod99] Victor Yodaiken. “The rtlinux manifesto”. In: *Proceedings of the 5th Linux Expo*. 1999 (cit. on p. 9).

[Yun+14] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. “PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platforms”. In: *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2014, pp. 155–166 (cit. on p. 13).

[Zyn19] Marc Zyngier. personal communication. Dec. 2019 (cit. on p. 85).

List of Figures

1.1. Evolution of processors since 1975 . . . 6
1.2. SIL2LinuxMP architecture proposal . . . 12
1.3. Structure of this thesis . . . 21

2.1. Typical patch integration workflow. . . . 28
2.2. Illustration of the generation of patch clusters . . . 36
2.3. Boxplot of irrelevant parameters . . . 44
2.4. Illustration of the influence of other parameters . . . 45
2.5. eCDF if the integration duration of patches on the LKML . . . 46
2.6. Evaluation of the Plus-Minus-based approach . . . 48

3.1. Linux development timeline . . . 61
3.2. Clusters of similar patches . . . 63
3.3. Composition of type of patches on Linux kernel mailing lists . . . 68
3.4. Regular and ignored patches between v3.0 and v5.10 . . . 69
3.5. Ratio of ignored patches per week between v3.0 and v5.10 . . . 69
3.6. Regular and ignored patches between v3.0 and v5.10 on the top four high patch traffic lists. . . . 70
3.7. Ratio of ignored patches between v3.0 and v5.10 on the top four high patch traffic lists. . . . 71
3.8. All mailing lists: Fraction of (in-)correctly and unintegrated patches. . . . 74
3.9. Top four high-volume mailing lists: Fraction of (in-)correctly and unintegrated patches. . . . 75
3.10. Disclosing secret integration channels . . . 79
3.11. Public observable and non-public integration channels . . . 81

4.1. Activation sequence of the Jailhouse hypervisor . . . 113
4.2. Ideal Hardware Partitioning . . . 116
4.3. Hardware Partitioning on Real Hardware . . . 116

5.1. Different measurement setups for the ICR MSR measurement . . . 136
5.2. Histogram of the execution time of the ICR MSR measurement . . . 137
5.3. Schedule of cyclictest . . . 139
5.4. Intel® Xeon® E5-2683 v4: Latency histogram . . . 140
5.5. Nvidia Jetson TX1: Latency histogram . . . 143
5.6. A multi-copter platform that runs on the Jailhouse hypervisor . . . 149

A.1. Preempt-RT patch stack: Evolution of the stack size. . . . 162

A.2. Preempt-RT patch stack: Distribution of integration duration 163
A.3. Preempt-RT patch stack: Comparing the composition of the last major releases. 164

List of Tables

2.1. Set of parameters used for evaluation. 43

3.1. Ratio of ignored patches per year. 66
3.2. Composition of all unique patches on all mailing lists. 67
3.3. A list of vulnerabilities that were detected by PaStA. 84

5.1. Interrupt reinjection latency on the Nvidia Jetson TX1 134
5.2. Cyclictest on a Intel® Xeon® E5-2683 v4 platform 141
5.3. Cyclictest on an ARM64 Nvidia Jetson TX1 platform 144

List of Listings

2.1. Example of an mapping that was found by PaStA. 33

3.1. The APPLETALK and NETWORKING DRIVERS section in MAINTAINERS. . 59

5.1. Fast IRQ response to toggle a GPIO 132
5.2. The core of the measurement of the ICR write duration. 135

List of Algorithms

1. Measure the similarity of two patches. 39

Ralf Stefan Ramsauer

Curriculum Vitæ

Ralf Ramsauer

Dienstadresse: Ostbayerische Technische Hochschule Regensburg, Labor für Digitalisierung
Galgenbergstraße 32, 93053 Regensburg

E-Mail: ralf.ramsauer@oth-regensburg.de

Akademischer Werdegang

1996–2000	**Grundschule**, *Grund- und Teilhauptschule Weiherhammer*
2000–2009	**Abitur**, *Kepler-Gymnasium Weiden*
Oktober 2009 –Februar 2013	**Studium: Technische Informatik, B. Sc.**, *Hochschule Regensburg*, Bester Abschluss des Wintersemester 2013
März 2013 –September 2015	**Studium: Applied Research in Engineering Sciences, M. Sc.**, *Ostbayerische Technische Hochschule Regensburg*
November 2015 –	**Wissenschaftlicher Mitarbeiter**, *Labor für Digitalisierung, Ostbayerische Technische Hochschule Regensburg*

Forschungsaufenthalte

Oktober 2014 –Dezember 2014	**Forschungsaufenthalt**, *School of Electrical Engineering, Tel Aviv University*

Lehrerfahrung

08/2017–02/2018	**Lehrbeauftragter**, *Ostbayerische Technische Hochschule*, Regensburg Theoretische Informatik

Forschungsinteressen

(Echtzeit-)Betriebssysteme, Statische Hardwarepartitionierung, Echtzeitvirtualisierung, Quantitatives Softwareengineering