Improving the Security of Multiprocessor-Based Embedded System Designs

Peng Seng Benjamin Tan

A thesis submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy in Computer Systems Engineering,
The University of Auckland, 2018.

Abstract

Designers are ambitious; we want to make embedded systems that are more capable, more connected, and ultimately, more complex. To tackle myriad design goals, the development of heterogeneous Multiprocessor System on Chips (MPSoCs) has emerged as a fashionable design paradigm. Embedded systems can be implemented more easily when various components, such as processors, memories, hardware accelerators, and other Intellectual Property (IP) blocks are integrated into a Network on Chip (NoC). This approach allows designers to better leverage parallelism and reduce costs through consolidation of many different functionalities into a single chip.

However, as embedded systems become more capable and Internet connected, so too are malicious entities—as we come to deploy embedded systems in more application domains, attackers have greater incentive to discover and exploit vulnerabilities for illicit gain. Where once security was handled as a secondary concern, or even worse, as an afterthought, designers acknowledge the value in treating security upfront. This thesis investigates strategies to introduce security into the design of multiprocessor-based systems from the very outset. The research we present is focussed on improving security, with the aim of reducing the impact of successful compromises.

After identifying the opportunities for hardware-based support for security mechanisms from a thorough examination of the literature, we begin by developing a conceptual model for describing the relationships between components in a MPSoC and the potential threats in a design. This leads us towards research into potential low-level mechanisms for improving security, and we present digital hardware for implementing decentralised and dynamic access controls in an MPSoC. One challenge in design for security is that design flows are often ad-hoc, so we propose a security-aware design process that systematically generates a security-enhanced MPSoC. Decentralised and dynamic access controls form the foundation for security improvement. Raising the abstraction level once more, we present

research on context-aware protections, where we re-frame memory accesses as service consumption, and enhance access controls with information about when an access should occur as part of a service. Our contributions are a system-level security-aware approach for MPSoC design, hardware support for decentralised and dynamic access controls, and systematic automated design methodologies.

To my family, for supporting my whims
and encouraging me to do them as well as I can

Acknowledgements

and words of gratitude

Doctoral-level study is a challenge. The years involved are filled with moments of confusion, doubt, and anxiety…and then, suddenly (and hopefully), moments of profound insight. My experience has been enjoyable and satisfying, and thankfully, moments of intellectual loneliness have been rare. This is in large part due to the people whose paths have intersected with mine for various reasons. I consider myself very fortunate, and these acknowledgements can only begin to reflect my utmost gratitude for the many different people in my life. Alas, this list is not exhaustive, but as I've come to discover, research work can never completely cover everything either.

To begin, I must acknowledge my supervisors. My main supervisor, Dr. Morteza Biglari-Abhari, has been a pivotal figure in my education thus far. Without the unexpected fortune of a summer research scholarship under his supervision (during my undergraduate study), I expect my life at this point would be very different. It was this first experience that really built our relationship, without which my path to research would not have been set. Since the beginning of my doctoral life, his everlasting encouragement has often allayed many of my fears, and our frequent detailed discussions about the research provoked many important strides. Sufficed to say, I have learned a lot under his guidance. I also have to express my gratitude for the teaching opportunities that he provided to me, for the counterbalance of teaching was instrumental in keeping my sanity in check.

Professor Zoran Salcic, my co-supervisor, has been a dependable source of wisdom throughout my studies. As my studies progressed (and his tenure as HOD wound down), the frequency of our interactions increased. He impressed upon me the importance of clarity (particularly when I became lost in a mire of my own ideas and terminology). I am grateful for his poignant and fair criticisms throughout the course of my research. I fondly recall an early meeting which preceded our first accepted publication; we went through page-by-page, and he highlighted all

the flaws and inconsistencies. In many respects, that exercise provided a launching pad for the main bulk of this work.

Outside of my supervisory team I have been privileged to interact with many inspiring academics in our department. Random day-to-day encounters are more valuable than one might expect! Dr. Kevin Wang was an encouraging voice; always sympathetic, and quite fun to chat with about all things University-related. At various times I had insightful and supportive discussions with Dr. HeeJong Park, Associate Professor Catherine Watson, Dr. Mark Andrews, Associate Professor Partha Roop, Dr. Avinash Malik, Dr. Oliver Sinnen, Dr. Bernard Guillemin, Professor Kevin Sowerby, Dr. Matthew Kuo… amongst others… not necessarily discussions about my research, but often about academia as a whole. I'd also like to acknowledge *all* of my past school teachers. Too many to name, but every single one has imparted some lesson, be it in knowledge, skills, or life itself. Thank you.

I have to express my gratitude also to the Embedded Systems lab technicians, first Sunita Bhide, then Hoda Najafi. Always eager to lend a hand, their expertise and support in getting whatever random item I might need was invaluable. I also want to thank the administrative support I received throughout my studies—Aruna Lal and John Lau. Efficient, understanding, and pleasant. Real superstars.

I'm incredibly lucky to have a fantastic group of friends, fellow doctoral candidates with whom to celebrate and commiserate. I've spent countless hours talking in depth with Andrew Chen, Ameer Ivoghlian, Hammond Pearce, Krystine Sherwin, Michael Orr, Nathan Allen, Nicholas Harvey-Lees-Green, Ryan Kurte, and Zac Roberts. We've all voiced our frustrations about silly undergrads, we've shared lunch sessions, and we've built things together. These fine folks have constantly broadened my horizons (largely so I can keep up with conversations), so I thank you all for that! There are also plenty of other fellow postgraduates who have made the overall experience so much richer, and I fear I'll run out of space if I name any more—so thank you to my colleagues too! We'll all get there.

My Mum (Patricia), Dad (Raymond), and my brothers, Benedict and Benett. Always there, always patient—what can I say? It's been a long journey. Mum and Dad have always supported me in whatsoever I desire, the only proviso being that I commit fully to the endeavour. Thank you for letting me do what I do, for all the many years. So, here we are. I can only hope that I've made everyone proud.

And finally, my thanks to Suraksha. You've been unconditionally supportive, of *all* my various enterprises. Always by my side, no matter the stresses and hurdles—so I hope you're proud too. Here's to whatever comes next. Thank you for everything.

THE UNIVERSITY OF AUCKLAND
Te Whare Wananga o Tamaki Makaurau
NEW ZEALAND

School of Graduate Studies
AskAuckland Central
Alfred Nathan House
The University of Auckland
Tel: +64 9 373 7599 ext 81321
Email: postgradinfo@auckland.ac.nz

Co-Authorship Form

This form is to accompany the submission of any PhD that contains published or unpublished co-authored work. **Please include one copy of this form for each co-authored work**. Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements. Co-authored works may be included in a thesis if the candidate has written all or the majority of the text and had their contribution confirmed by all co-authors as not less than 65%.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work. Material in Chapter 3 and 4 is based on work published in the following paper: Benjamin Tan, Morteza Biglari-Abhari, and Zoran Salcic, "A system-level security approach for heterogeneous MPSoCs," 2016 Conference on Design and Architectures for Signal and Image Processing (DASIP), Rennes, 2016, pp. 74-81. doi: 10.1109/DASIP.2016.7853800	
Nature of contribution by PhD candidate	Problem formulation, experimental work, manuscript writing
Extent of contribution by PhD candidate (%)	80

CO-AUTHORS

Name	Nature of Contribution
Morteza Biglari-Abhari	Paper Review/Editing
Zoran Salcic	Paper Review/Editing

Certification by Co-Authors

The undersigned hereby certify that:

- the above statement correctly reflects the nature and extent of the PhD candidate's contribution to this work, and the nature of the contribution of each of the co-authors; and
- that the candidate wrote all or the majority of the text.

Name	Signature	Date
ZORAN SALCIC	Z Salcic	9/05/18
Morteza Biglari-Abhari	Morteza Abhari	10/5/18

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

School of Graduate Studies
AskAuckland Central
Alfred Nathan House
The University of Auckland
Tel: +64 9 373 7599 ext 81321
Email: postgradinfo@auckland.ac.nz

Co-Authorship Form

This form is to accompany the submission of any PhD that contains published or unpublished co-authored work. **Please include one copy of this form for each co-authored work.** Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements. Co-authored works may be included in a thesis if the candidate has written all or the majority of the text and had their contribution confirmed by all co-authors as not less than 65%.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work. Material in Chapter 2, 3, and 4 is based on work published in the following paper: Benjamin Tan, Morteza Biglari-Abhari, and Zoran Salcic, "Towards decentralized system-level security for MPSoC-based embedded applications", In Journal of Systems Architecture, Volume 80, 2017, Pages 41-55, ISSN 1383-7621 doi: 10.1016/j.sysarc.2017.09.001.

Nature of contribution by PhD candidate	Problem formulation, experimental work, manuscript writing
Extent of contribution by PhD candidate (%)	80

CO-AUTHORS

Name	Nature of Contribution
Morteza Biglari-Abhari	Paper Review/Editing
Zoran Salcic	Paper Review/Editing

Certification by Co-Authors

The undersigned hereby certify that:

- the above statement correctly reflects the nature and extent of the PhD candidate's contribution to this work, and the nature of the contribution of each of the co-authors; and
- that the candidate wrote all or the majority of the text.

Name	Signature	Date
ZORAN SALCIC	Z-Salcic	9/05/18
Morteza Biglari-Abhari	Morteza Abhari	10/5/18

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

School of Graduate Studies
AskAuckland Central
Alfred Nathan House
The University of Auckland
Tel: +64 9 373 7599 ext 81321
Email: postgradinfo@auckland.ac.nz

Co-Authorship Form

This form is to accompany the submission of any PhD that contains published or unpublished co-authored work. **Please include one copy of this form for each co-authored work**. Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements. Co-authored works may be included in a thesis if the candidate has written all or the majority of the text and had their contribution confirmed by all co-authors as not less than 65%.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work. Material in Chapter 5 is based on work published in the following paper: Benjamin Tan, Morteza Biglari-Abhari, and Zoran Salcic. 2017. "An Automated Security-Aware Approach for Design of Embedded Systems on MPSoC". ACM Trans. Embed. Comput. Syst. 16, 5s, Article 143 (September 2017), 20 pages. doi: 10.1145/3126553

Nature of contribution by PhD candidate	Problem formulation, experimental work, manuscript writing
Extent of contribution by PhD candidate (%)	80

CO-AUTHORS

Name	Nature of Contribution
Morteza Biglari-Abhari	Paper Review/Editing
Zoran Salcic	Paper Review/Editing

Certification by Co-Authors

The undersigned hereby certify that:

- the above statement correctly reflects the nature and extent of the PhD candidate's contribution to this work, and the nature of the contribution of each of the co-authors; and
- that the candidate wrote all or the majority of the text.

Name	Signature	Date
ZORAN SALCIC	Z-Salcic	9/05/18
Morteza Biglari-Abhari	Morteza Abhari	10/5/18

THE UNIVERSITY OF AUCKLAND
Te Whare Wananga o Tamaki Makaurau
NEW ZEALAND

School of Graduate Studies
AskAuckland Central
Alfred Nathan House
The University of Auckland
Tel: +64 9 373 7599 ext 81321
Email: postgradinfo@auckland.ac.nz

Co-Authorship Form

This form is to accompany the submission of any PhD that contains published or unpublished co-authored work. **Please include one copy of this form for each co-authored work**. Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements. Co-authored works may be included in a thesis if the candidate has written all or the majority of the text and had their contribution confirmed by all co-authors as not less than 65%.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work. Material in Chapter 6 is based on work published in the following paper: Benjamin Tan, Morteza Biglari-Abhari, and Zoran Salcic, "Towards ContextAware Service Protection for NoC-based Heterogeneous MPSoCs"	
Nature of contribution by PhD candidate	Problem formulation, experimental work, manuscript writing
Extent of contribution by PhD candidate (%)	80

CO-AUTHORS

Name	Nature of Contribution
Morteza Biglari-Abhari	Paper Review/Editing
Zoran Salcic	Paper Review/Editing

Certification by Co-Authors

The undersigned hereby certify that:

- the above statement correctly reflects the nature and extent of the PhD candidate's contribution to this work, and the nature of the contribution of each of the co-authors; and
- that the candidate wrote all or the majority of the text.

Name	Signature	Date
ZORAN SALCIC	Z-Salcic	9/05/18
Morteza Biglari-Abhari	Morteza Abhari	10/5/18

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Computer systems are ubiquitous, and designs are becoming more complex as designers become more ambitious. This thesis begins with an introduction to the landscape of embedded system design, and highlights some motivations for research into techniques for improving security. After presenting some examples of security exploits and their significant impacts, we examine different facets of security. This leads us towards a discussion of important challenges and opportunities for embedded systems security with a specific focus on multiprocessor system on chip designs. From there we lay out the progression of research work presented in this thesis, and describe the contributions resulting from our efforts in this domain.

1.1 Security: An Ongoing Battle

Computers are everywhere. People use computers for work or pleasure, and the entire world is becoming more interconnected through the Internet. Alongside explicitly visible desktop and laptop computers, the number of computers *hidden* and *embedded* within specific application domains has risen exponentially. Embedded systems can be found in numerous sensing and actuating contexts [38], such as within automobiles, industrial automation, medical devices, smart homes, and robotics. Various reports estimate the number of embedded systems in the so-called *Internet of Things* (IoT) to swell to at least the *tens of billions* [81, 89] by 2020. Designs are becoming more complicated and feature-rich [103], and designers often need to create solutions that satisfy several competing requirements, such as resource cost, performance, safety, and timing [93]. The economic value of the IoT has been estimated to reach between $3.9 trillion to $11.1 trillion annually by 2025

[82]. The scale and rapid pace of technological developments in the embedded space is incredible.

Embedded systems are often a difficult class of computer systems to categorise, as their functionality can be wide and varied [128]; however, each design typically has an element of being application specific, performing a specialised role in a specific environment, for example:

- **Cyber-physical systems**, such as in automotives, robotics, customizable and reconfigurable manufacturing
- **Financial systems**, such as point-of-sale (POS) terminals, Automatic Teller Machines (ATMs)
- **Medical systems**, such as pacemakers, insulin pumps, wireless health monitoring
- **Infrastructure**, such as smart power grids, building condition sensors, intelligent transportation systems
- **Consumer products**, such as IoT-enabled appliances, wearables, and mobile devices

These types of applications often have sensitive or valuable components, such as private data (for example, financial data, or medical data), or safety-critical parts. In recent years, technology improvements have increased the performance of embedded systems, and ambitions for more efficient and smaller devices has created a shift from uniprocessor systems towards consolidation and integration of many functionalities into heterogeneous *multiprocessor systems on chip* (MPSoC) [103]. And yet, while there is great potential for new innovations and greater human conveniences in the design of new embedded systems, such designs may be deployed into a potentially dangerous and hostile environment. Malicious entities aim to reap financial benefit through illicit means, sneak and steal information to further political agendas, and disrupt or destroy valuable targets for some insidious gain. Designers should be vigilant; they need tools and strategies to prevent, detect, and mitigate security attacks.

Currently, devices ranging from cars to insulin pumps have all been demonstrated to contain critical vulnerabilities [134]. To get a sense of the current landscape, consider a few examples of recently disclosed attacks[1].

[1]Not all attacks or vulnerabilities are disclosed by victims in a timely fashion, and some may not even be disclosed at all. Even so, these *known* security breaches stir up plenty of anxiety about the present state of affairs by themselves

In 2010, the Stuxnet worm [65] was discovered and publicised, when it was found to have infected at least 14 industrial sites in Iran, including at least one uranium enrichment plant. In addition to being considered a prime example of state-sponsored cyberwarfare, the worm was especially noted for specifically targeting and subverting an *industrial* control system, upending the assumption that industrial systems are fundamentally robust by default, or obscure enough to be "immune" to security exploits. As discussed in [70], the developers of the worm had significant understanding of the target operating environment, and worked to undermine the system's performance over a period of time, instead of causing a single catastrophic failure.

In 2011, vulnerabilities in cars were reported [17], demonstrating that determined attackers with modest resources were able to exploit a variety of different physical channels for causing harm on modern vehicles. Subsequent research presented in 2014 [2] highlighted risks in "smart cars", where less-critical parts of the system (like the CD player) could be exploited to access and disrupt control functionality. The impact and reach of a successfully compromised part of the design was increased by the consolidation and interconnection of separate Electronic Control Units (ECUs) to improve performance and reduce costs. Modern cars provide a classic example of a mixed-critical embedded system, with a mixture of sensitive tasks for actuation, alongside less-critical tasks for things like infotainment.

In 2013, Target, a large consumer retail company in the United States of America, was attacked by malware known as BlackPOS [90]. Approximately 70-110 million people had credit card information, mailing and email addresses, phone numbers, and names stolen from Target's system, resulting in significant financial fraud and identity theft [41]. Point-of-sale systems are a type of embedded system, designed to facilitate financial transactions in stores. The BlackPOS malware operated by quietly installing itself in the device's operating system (OS) after which it monitored and scanned process memory to identify credit card information. The sensitive information was logged, and exfiltrated on a daily basis. This anomalous (and ultimately, malicious) behaviour was not immediately detected, resulting in losses for many ordinary citizens.

In 2016, a huge Denial-of-Service (DoS) attack was carried out on parts of the Internet's infrastructure, affecting the availability of many popular websites [117]. While DoS attacks are not new, the novelty in this particular instance was the malicious manipulation of millions of vulnerable IoT devices for generating the excessive and disruptive web traffic in the attack. The IoT devices were commandeered to do something they were not designed to do. The sheer volume of devices,

coupled with the immature state of IoT security meant that the attack had quite a significant impact.

Despite the disparate nature of these different attacks, several commonalities emerge. Security was either ignored, or not successfully incorporated into the design of each system. Attacks caused the embedded systems to behave in a way that was unintended by designers, and merely responding to security exploits is insufficient; simply reacting to threats can still lead to considerable losses. Because there can be huge motivation for malicious exploitation, attackers often have significant resources at their disposal. Hence, work needs to be done to improve the security of future devices—we need to develop strategies to add more hurdles for potential attackers. Security should be considered from the very start of the design process instead of being treated as an afterthought.

Consider a future where embedded systems perform many different functions. The design is composed of many different components, sourced from many different places, with many different parts working in parallel. How do we reduce the impact of a compromised part on the system as a whole? Is it possible to improve security such that critical parts of the system continue to operate, even in an attack? The best place to consider security is at design time, where security is treated as a first-class design requirement, as opposed to an after-thought, and with this challenge in front of us, this thesis proposes contributions for improving security. But first, before we proceed, we must first consider the fundamental question: what *is* security? What does it mean for something to be *secure*?

1.2 Understanding Security

1.2.1 What is Security?

Intuitively, people want to protect their valuable "things" from being harmed or stolen by a malicious entity. Some things might be physical objects, like a favourite toy or electronic device, or even perhaps other people. Other things might be more abstract, such as private information, or *secrets*, such as intellectual property. We can refer to these valuable or sensitive things as *assets* [75]. Security is therefore a type of non-functional requirement which specifies that a given asset should be *protected*. In the literature, security is typically considered as the problem of managing the relationship between attackers (malicious entities that aim to disrupt, damage, or steal assets), and defenders (who are trying to make a system more secure by

introducing various mechanisms) [114, 122]. In an ideal case, there should be no relationship at all—a defender should be able to block all attackers.

However, security is itself a broad concept, which can have different meanings for different people in different contexts [3]. As such, security is often decomposed into several smaller properties [3, 10, 11, 75], which includes the properties of:

Secrecy—*sensitive information is kept private/confidential, such that only a specific entity (or set of entities) can access said information*

Integrity—*specific assets are protected from unwarranted modification*

Availability—*resources in a system which need to be accessible are accessible, or that the system functions as expected when expected*

By devising a design that can satisfy a specific mix of different security properties, we aim to make it *trustworthy*, and develop confidence that our system behaves exactly as we *intend*. The different security requirements indicate properties of the states that our system as a whole is allowed to be in (e.g. the house is always comfortable). If our system is always in one of the allowed states, our system is *secure* [10, 68]. On the other hand, if our system can be made to enter a disallowed state (e.g. the occupancy data is extracted by a malicious party), the system is *insecure*. Naturally, as designers, we should aspire towards the design of a *perfectly* secure system. However, in the next section, we will explore why development of *completely* secure systems proves to be an ongoing challenge.

1.2.2 In Pursuit of Perfect Security

Consider the scenario where we are tasked with the design of an internet-accessible embedded system that is used in automation of a "smart" home (Figure 1.1). The embedded system interacts with various sensors and controls various actuators. As part of its functionality, this system should detect for the presence of occupants, and ensure that the environmental conditions are maintained based on the preferences set by the owners.

We want this design to be *secure*, and so at an informal level, we might want to define a few "high-level" security requirements:

1. We don't want an attacker to make the house uncomfortable (by misconfiguration of the actuators)

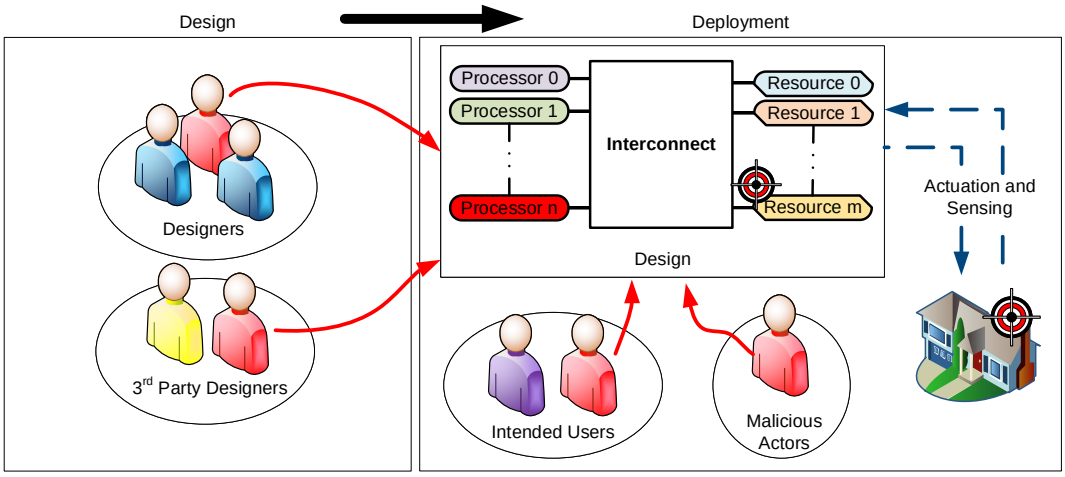


Figure 1.1: An embedded system for controlling a "smart" home—different threats from different potential adversaries provides a system-wide challenge

2. We don't want an attacker to know when we are in (or out) of the house (by stealing occupancy sensor data)

The first requirement relates to the *functional* operation of the embedded system; it deals with what the system *does*. The second requirement relates to *data* or *information flow* related aspects of the embedded system. However, this is not enough detail for trying to design a secure system. To accommodate these requirements, we need to consider where potential threats might come from, or try and determine the potential *attack surface*.

For one, the users might be worried about unknown external malicious actors; they may try to access the embedded system through the internet. Or perhaps within the group of legitimate users, some are more *trustworthy*, or have more *authority* than others, and one or more of these users might need to be prevented from misconfiguring the design.

How might we find ways to prevent malicious actions? If we want to ensure that the environment in the house is always comfortable, we may need to consider elements *within* the design; the software (SW) and hardware (HW) that constitutes the interface between the embedded device and the actual actuators in the house may need to be protected in some way. However, let's consider further potential risks. What if a designer that is involved in the production of the embedded system (either in-house or 3rd party) has malicious intentions? They might introduce bugs, or even deliberately incorrect functionality, which can trigger after the embedded system is deployed. Or perhaps consider issues *outside* the embedded system itself;

what happens if the physical sensors in the house are tampered with such that they give misleading (false) readings to the embedded system, thus triggering unwanted actuation? If an attacker wanted to really cause damage, they might even simply break in to the house and cut power to the device, or perhaps steal the embedded system itself!

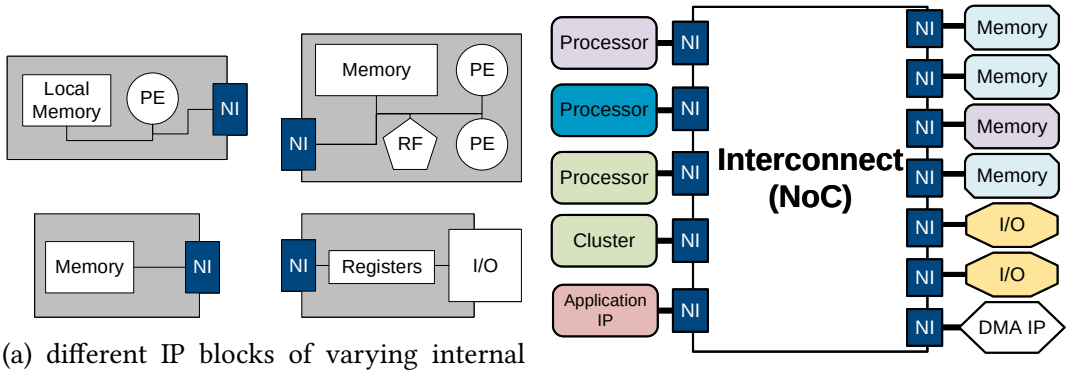
As a means of addressing different security requirements, we need to design and apply an appropriate *security policy*, which specifies the details about how the system should operate, and the environment that it is in. A security policy aims to capture, in detail, the relationships between all the entities inside and outside a system. From our informal discussion thus far, it is clear that security, as a whole, is non-trivial. Security is a cross-system issue, where technical, logistical, and human aspects should be considered to better safeguard assets.

The key point here is that "high-level" security requirements may appear to be misleadingly "simple", and they often entail complex issues across many abstraction levels for real-world systems. One should appreciate that implementing and guaranteeing *complete* or *total* protection in the face of *all* potential adversaries and risks is a gargantuan undertaking. Obviously, the scope of research presented in a single doctoral thesis cannot address or solve *all* aspects of security, so in the next section we examine more closely some of the technical security challenges for embedded systems, with full acknowledgement that our contributions should be considered one part of a far larger picture.

1.3 Challenges for Embedded Systems Security

1.3.1 Multiprocessor Systems on Chip

With the growing complexities of embedded system applications, heterogeneous MPSoCs have become a popular platform architecture paradigm. In this thesis, we explore issues related to MPSoC security. MPSoCs are an evolution of the System-on-Chip concept, where a fully fledged computing system is produced by combining a processor (or microcontroller), memories, peripherals, and accelerators into a single chip [108]. One of the key benefits is the adoption of pre-designed and pre-verified hardware and software components, also known as Intellectual Property (IP) blocks, which improves designer productivity, particularly when designers need to implement several different functionalities, with a multitude of various timing, safety, or security requirements [59, 93]. As the next logical progression, MPSoCs integrate more processors to leverage parallelism and further consolidate



(a) different IP blocks of varying internal complexity

(b) integration using a common interconnect

Figure 1.2: A Generalised MPSoC architecture

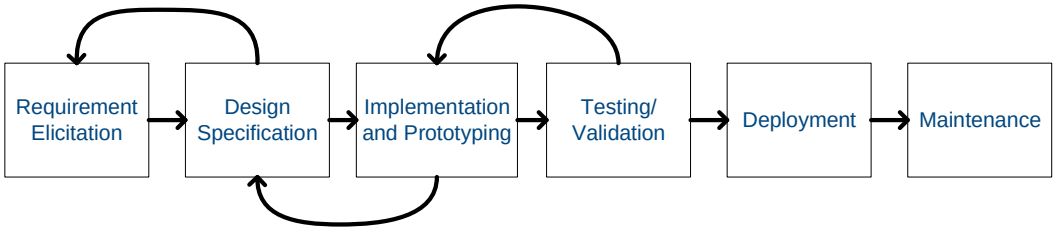


Figure 1.3: A general embedded systems design flow, adapted from [116]

different parts of an application. Various heterogeneous blocks (with varying levels of complexity, as shown in Figure 1.2(a)) are interconnected using some common communication infrastructure, such as shared buses, or a Network on Chip (NoC) [22, 31, 36, 106], as shown in Figure 1.2(b). Designers increasingly use IP blocks sourced from different vendors for their MPSoC designs [77, 103].

1.3.2 Integrating Security into the Design Flow

A general embedded system design flow is illustrated in Figure 1.3. First, designers must elicit and refine both functional and non-functional requirements for a given application. These requirements are then transformed into more detailed design specifications, which are used as the basis for prototyping/implementation of the design. This design process is often iterative, with further refinements to requirements and specifications as more information or gaps in knowledge are revealed, and trade-offs are made. The implementation is then tested, validated, and refined, before ultimately being deployed. In some applications, the final design may also undergo periodic maintenance, such as via firmware updates.

Throughout the design process, high level descriptions and requirements are progressively refined into lower level implementation details, either manually, or

through the use of some sort of automated framework [96]. Various levels of abstraction (and automation techniques) are useful for designers, particularly to improve productivity, and to reduce errors from manual implementations. This also helps with design exploration, where different potential configurations can be explored to see how different design metrics are affected by different design decisions.

In order to improve the security of a design, security requirements should be introduced and managed throughout the entire design process [116]. The challenge is in selecting appropriate levels of abstraction throughout the design process, and how to design-for-security. Firstly, different abstraction levels can better model various threats and adversaries for a design, and so to achieve certain security requirements, we need to decide how we represent threats, and how to represent our mechanisms for mitigating those threats. The choice of abstraction affects how well a design "factors in" security issues, and describes the coverage of the chosen security approach. This is an important consideration if we aim for security to be built into the design. Essentially, we need to formulate a *security model* which is compatible with the design flow. Another challenge that exists is that it is difficult to *measure* security—there are different perspectives with which to reason about security, and thus challenges with respect to defining how security could fit in to an exploration/trade-off of different design requirements [94, 119].

Furthermore, when describing complex behaviours, abstraction can help make aspects of the design process more manageable by hiding certain complexities. Abstractions are also useful when utilising 3rd party components where implementation details are obfuscated or unavailable, as designers can focus on an application's intended behaviour and functionality separately from work on underlying architecture or implementation. As part of the design flow, some assumptions and simplifications are made, and these need to be consolidated with the intended security approach. For example, we might assume that an IP block that is used for accelerating signal processing is used *only* for signal processing, but in reality, it may have the ability to access arbitrary regions of memory. This type of extra-functional capability should be considered as part of our chosen design abstraction so it can be addressed at the same level of abstraction, particularly if we want to achieve the required protection against spurious accesses.

1.3.3 Heterogeneity in Embedded Systems Design

Embedded systems are typically characterised as being tightly constrained (in terms of resource cost, form factor restrictions, computation capabilities, power

consumption etc.) [102, 128]. Applications often have real-time and reactivity requirements as they perform continuous sensing and actuating to control the environment in some way. Some systems generally operate autonomously (such as in certain control applications), while others (like IoT devices) feature human interfaces, either directly on the device, or remotely through an Internet-enabled interface. The design environment has rising time-to-market pressures, which encourages design re-use [48, 59, 101, 103], whereby general architectures are customised and tuned for specific applications, and deployed on execution platforms like Field Programmable Gate Arrays (FPGAs) [48]. As such, MPSoCs are becoming more heterogeneous, with many parallel and concurrent behaviours implemented in both hardware and software. Platform building blocks can be sourced from a variety of vendors, and different parts of an application can have differing levels of criticality and vulnerability.

These trends highlight a challenge for embedded system security; how do we integrate different functionalities and requirements, while also building security into the design? The challenge lies in creating security approaches that are usable in the context of heterogeneity, as well as in the context of many different concurrent and loosely-coupled functionalities.

In uniprocessor platforms, it is typical to employ a central privileged authority for managing security, usually an operating system (OS) (such as Linux [99]), or hypervisor (such as Xen [137]). This central authority then manages the security features provided by the execution platform. There is an explicit privilege hierarchy, where highly privileged entities (such as the OS) can control access, to all the resources in the system; hence, privilege escalation attacks can cause a significant negative impact. However, in heterogeneous systems, it may be challenging or inconvenient to deploy a single OS across the entire platform. Furthermore, some tasks may execute "bare-metal" for performance or timing reasons, and there may be multiple IP blocks (that implement some fixed functionality) with direct memory access (DMA) capabilities. These system components need to be managed properly to ensure that security restrictions that are defined by the designer are respected.

As will be explained later in this thesis, there are many different security mechanisms that could be used to better safeguard assets in a system. Development of new security approaches, especially for a multiprocessor context, remains an ongoing topic for exploration. However, it is challenging to select and integrate a given mechanism to address a given security requirement. For example, designers might identify memory protection [76] as a useful *general* approach for improving a

design's security. Part of the challenge is knowing where to deploy the memory protection mechanism, and how the specific parts of the mechanism can be mapped to entities within the specific application (or vice versa). In other words, which permissions should be created, and where should they be enforced—security mechanisms should be linked to specific security requirements. We need to develop a suitable protection infrastructure for the given application, even when components do not have a common software foundation, are not *security-aware* (i.e. without built-in security mechanisms), or even if their internal designs are not fully accessible by system designers. Moreover, designers should be able to quantify the implications of adding a mechanism into the design (for example, in terms of cost overhead), or understand the implications of omitting security additions in lieu of other design requirements. As such, the process for exploring and incorporating security features into an embedded system design should be systematic.

1.4 Aims and Contributions

As a response to these various challenges in the arena of embedded system security, this thesis proposes and explores novel approaches for improving security of MPSoCs. Broadly, our primary aims are to:

- Reduce the impact of an initially successful incursion in a heterogeneous MPSoC
- Facilitate easier accommodation of security concerns into the design flow
- Provide frameworks for design space exploration which are security-aware
- Investigate the feasibility of customising MPSoC architectures to provide a hardware-supported security *foundation*

To achieve these aims, our research story begins at an abstract level, where we first develop a conceptual model for describing the relationships between components in a MPSoC and the potential threats in a design. This leads us towards development of low-level mechanisms for improving security; these are then used to influence the concrete design of digital hardware for supporting decentralised and dynamic access controls. To deal with the bigger problem of designing for security, we then create a security-aware design flow to systematically generate a security-enhanced platform. This thread culminates in our research on context-aware protections at

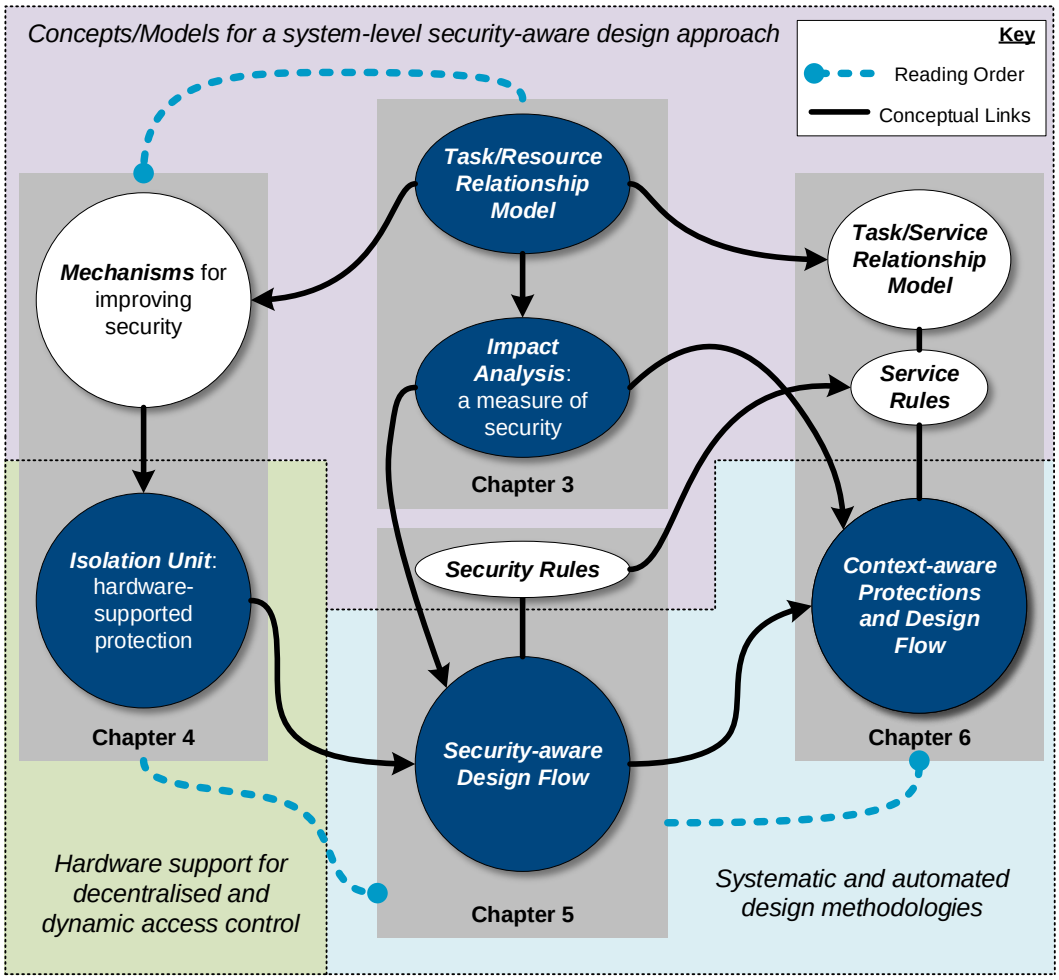


Figure 1.4: A high-level view of research progression in this thesis

a higher service-level abstraction. The progression of this research is illustrated in Figure 1.4. Overall, this thesis will present the following contributions:

- **A system-level security-aware approach for MPSoC design**: A *security model* provides the framework for formulating security requirements/specifications, capturing potential security risks, and management of potential security issues. At the system-level we want to reason about the design at a high level of abstraction. By considering several general approaches, alongside the specific context of heterogeneous MPSoC design, we propose a security approach and novel abstractions to incorporate security into the design flow. With this approach, we can define security rules to specify requirements for protecting parts of a design with different levels of criticality. The proposed approach serves as the foundation for novel system-level techniques and tools for improving security that are presented in this thesis.

- **Hardware support for decentralised and dynamic access control**: A security infrastructure as part of the MPSoC architecture is necessary for improving security. In this thesis we present prototype implementations of the *Isolation Unit* (IU), which are customisable hardware blocks that can be inserted into a MPSoC design to improve system security. These IUs are dedicated memory protection blocks that do not require an OS for run-time customisation. Two variations of the IU, implemented in VHDL, are described and characterised by their resource cost when synthesised for an FPGA-based execution platform. We also investigate the feasibility and utility of enhancing and integrating the functionality of the Isolation Unit into a Network-on-Chip (NoC) network interface. Access controls are also further enhanced with the notion of *context*, where the time of an access or sequence of operations is also checked for correctness.

- **Systematic and automated design methodologies**: To move away from *ad-hoc* design methodologies, we present some novel approaches to improve security as part of the design flow. We propose a systematic methodology for generating and exploring customised MPSoC architecture configurations, and analysing the specific realisation of the application to check for compliance with certain security rules. This presents groundwork for facilitating a more robust trade-off between security and other design metrics. We also present a novel top-down design flow for customizing a *protection infrastructure* to improve MPSoC security, even where components are heterogeneous or not security-aware. As part of this, we also describe a service-level abstraction to deal with complex IPs, to re-frame security as protection of services, and protection from service providers.

1.5 Publications and Manuscripts

This thesis features manuscripts that have been published:

- Benjamin Tan, Morteza Biglari-Abhari, and Zoran Salcic, "A system-level security approach for heterogeneous MPSoCs," 2016 Conference on Design and Architectures for Signal and Image Processing (DASIP), Rennes, 2016, pp. 74-81. doi: 10.1109/DASIP.2016.7853800

  – Material from this paper can be found in Chapter 3 & 4

- Benjamin Tan, Morteza Biglari-Abhari, and Zoran Salcic, "Towards decentralized system-level security for MPSoC-based embedded applications", In Journal of Systems Architecture, Volume 80, 2017, Pages 41-55, ISSN 1383-7621 doi: 10.1016/j.sysarc.2017.09.001.
  - Material from this paper can be found in Chapter 2, 3 & 4
- Benjamin Tan, Morteza Biglari-Abhari, and Zoran Salcic. 2017. "An Automated Security-Aware Approach for Design of Embedded Systems on MPSoC". ACM Trans. Embed. Comput. Syst. 16, 5s, Article 143 (September 2017), 20 pages. doi: 10.1145/3126553
  - Material from this paper can be found in Chapter 5

as well as manuscripts that are under preparation/under review:

- Benjamin Tan, Morteza Biglari-Abhari, and Zoran Salcic, "Towards Context-Aware Service Protection for NoC-based Heterogeneous MPSoCs"
  - Material from this paper can be found in Chapter 6

1.6 Thesis structure

The remainder of this thesis is as follows. In Chapter 2 we provide a critical discussion of the state-of-the-art for embedded systems security, with a particular focus on mechanisms and frameworks for MPSoC design. In Chapter 3 we explore a variety of security models, examining the utility of each in the context of MPSoC design. We present a threat model that underpins and motivates our research contributions, and propose a design abstraction for security analysis. In Chapter 4 we apply our MPSoC security model, and discuss various mechanisms needed to provide a more secure foundation. This leads us to the development of the Isolation Unit. In Chapter 5 we propose a systematic and automated approach for designing embedded systems on MPSoCs, where Isolation Units are customised and inserted into the platform to address designer-specified security rules. In Chapter 6 we present a novel service-oriented approach to MPSoC design, and move Isolation Unit functionality into network interfaces, while also providing transparent permission management as part of a service consumption paradigm. Finally, in Chapter 7 we make some concluding remarks, and contemplate potential future directions for this research.

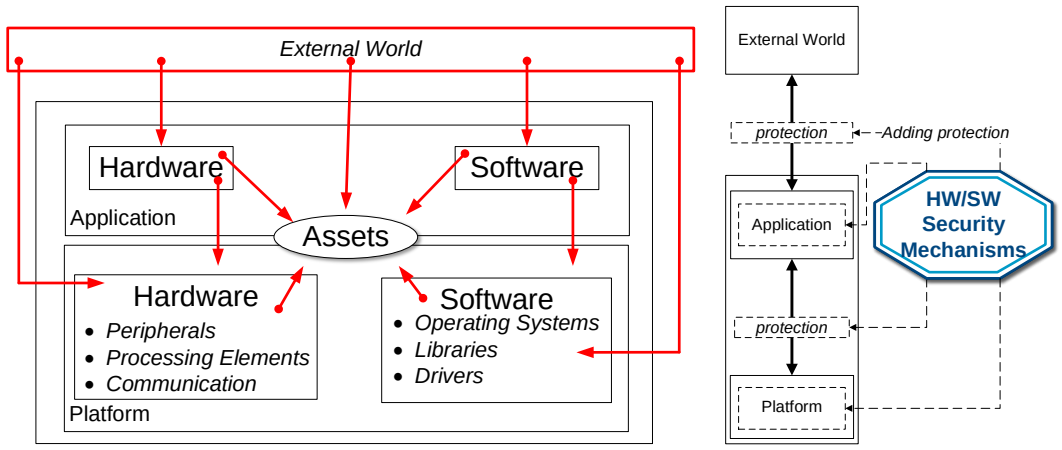
CHAPTER 2

Background and Existing Approaches for Improving Security

Researchers have tried to consider and manage security issues for a long time. In this chapter we discuss some background concepts on security issues, such as examples of mechanisms used in attacks, and then critically examine various techniques for improving SoC security, with particular interest at the implications when we consider the multiprocessor context. In looking at different types of attacks, and different types of defences, we present a view of the state-of-the-art for computer security, examining both more established techniques for computers generally, followed by some emerging approaches for MPSoCs. We identify common themes and opportunities within existing literature for improving computer system security, and this provides the context for the research contributions presented in this thesis. Our story begins by looking at attacks in a general sense (§2.1), before moving into a discussion of defences (§2.2 and §2.3).

2.1 Background on Security Attacks

Security is about protection of *assets* which are specific to an application. Consider Figure 2.1, which represents an MPSoC as logically partitioned in two: hardware/-software components related to implementation of the application's *functionality*, and the hardware/software components that form the underlying *platform*. The application level contains the "logic" that designers implement, such as control algorithms, user interface behaviours, and data processing operations. The platform consists of components that we can consider to be the infrastructure, such as processing elements and memories that application software executes on, as well as other hardware, such as I/O interfaces, accelerators, peripherals, and on-chip



(a) Internal and external threats/attack vectors to a MPSoC (b) Different possible protection boundaries

Figure 2.1: Threats and Defences for a MPSoC

buses/networks. In addition, we consider system software, like operating systems, shared libraries, and drivers as part of the platform, as they support the application's execution. Security exploits occur as attackers attempt to take advantage of inadvertent (or deliberate) flaws, or vulnerabilities, within the application or platform. Various attack strategies might target the application (exploiting issues in application logic, or implementation), target the hardware/software in the platform, or employ a combination of attack *vectors* in the mission to compromise an asset.

In 1994, Landwehr et al. formulated and presented a useful taxonomy for categorising computer security program flaws [69]. Their taxonomy provides three dimensions of classification regarding *how* a flaw is introduced, *when* the flaw is introduced, and *where* in a system it appears, referred to as *genesis*, *time of introduction*, and *location* respectively. Several key points are raised in their discussion, reflecting issues in the design practices of the day, and their view of the "terrain of computer program security flaws":

- flaws can be introduced *intentionally*, where malicious parties directly add malicious code to systems, or where flaws are introduced by functionalities that are added non-maliciously by legitimate designers that can later be exploited (for example, debugging interfaces for making the design process easier)

- flaws can also be introduced *inadvertently*, especially as a result of designs that are "composed of many modules, and involving many programmers", as

issues such as under-resourcing might result in insufficient oversight (leading to bugs which are missed in reviews), or inadequate documentation (leading to implementation mistakes)

- the system life-cycle can be abstracted to three phases of *development*, *maintenance*, and *operation*—flaws introduced during development may appear during requirement/specification design, in source code, or object code
- security issues are likely to arise due to "competition between security requirements and other functional requirements", or where solutions are "not deemed to be cost effective"—which implies design trade-off
- risk from flaws introduced by hardware, while not prevalent at the time, still warranted a discussion, with the view that there will be an increasing need to consider the impacts of hardware-based exploits

Since [69] was published, the nature of fundamental flaws used in security attacks have persisted; the same grievances regarding validation errors, privilege escalation, and exploitation of poorly documented (or undocumented) "features" are present in recently disclosed vulnerabilities. Despite decades of work in computer security, our current state of affairs is still hamstrung by the prevalence of applications that are fraught with vulnerabilities resulting from poor implementation. In short, poor **software**. In an analysis of vulnerability trends from 2008-2016 [64], Kuhn et al. discovered that simple coding errors continue to be widespread. Such errors, like omitting a bounds check (resulting in buffer overflows) are trivial, yet continue to have drastic impacts on overall security. It should be unsurprising then, that the literature often features technical mechanisms and strategies to bolster security at the platform level, accepting that applications are likely to be vulnerable, or already compromised.

Research in improving security is typically contextualised as a response to security attacks. To better understand the trends and opportunities in security research, we will now look at some exemplary attack types, particularly as we shift focus from single to multiprocessor systems.

An example attack vector: Buffer Overflows

A common aim for attackers is to seize control of a computer by controlling a processor's execution. One example of a common attack vector is the targeting of software flaws that deal with memory buffers. Useful programs usually receive data

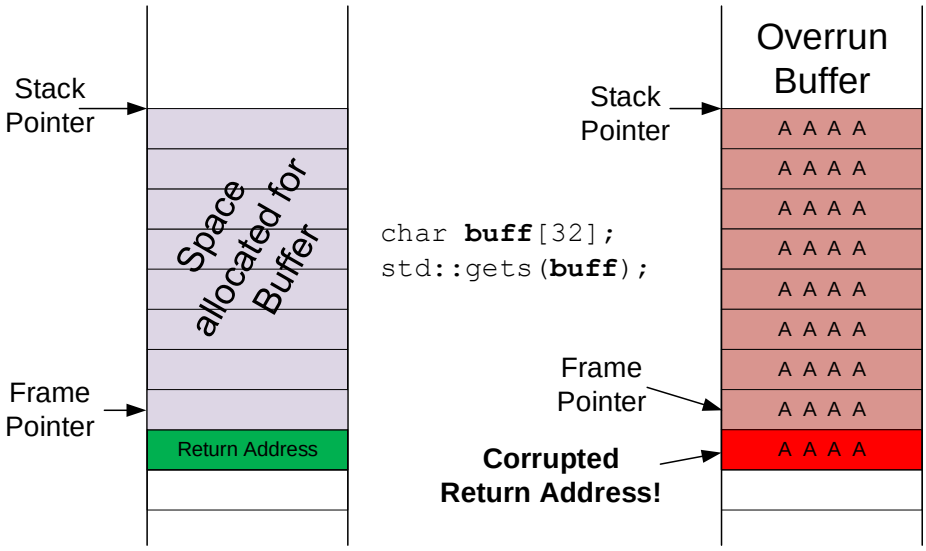


Figure 2.2: An example of the quintessential overflow

from the external world, and this data needs to be stored in memory. When using lower level programming languages, such as C [54], care must be taken to manage memory appropriately; after all, a processor will *blindly* execute whatever instructions it is given. Consider Figure 2.2, which illustrates a *stack*, the memory structure used as temporary storage in a program's execution. The stack holds data for the currently executing function, as well as other critical data such as the return address. Space on the stack is allocated for a buffer, `buff`, into which user input is stored. However, the C standard library function `gets()` does not perform any input validation or checks, simply copying the input data into memory—input data that exceeds the allocated size simply *overflows*, over-writing whatever was in the memory. This means that attackers can cause program crashes (by causing the function return to redirect to some random address), or hijack the program (and then perform some nefarious actions by injecting their own code). In fact, the memory compromise might only be the starting point of a multi-vector attack, where other flaws may be exploited to circumvent security mechanisms. The specific case illustrated here is trivial, but many variations of this sort of attack mechanism continue to exist "in the wild" [122].

Where do we start if we want to improve security? It might be useful to think in terms of the taxonomy discussed earlier: how is the flaw introduced, when is it introduced, and where is it located? Memory overflow flaws are inadvertent, stemming from an omission of boundary condition checking. It is a flaw that is introduced during software development, and might be located in the application, operating system, or other supporting utility programs. Understanding the *how*,

when, where of a flaw provides a good starting point for researchers and designers to improve security by flaw elimination or mitigation.

We might want to tackle potential overflows from the perspective of *when* the flaw is introduced, for example, during development of the application. One option is to mandate that designers use higher level languages with automatic memory management. Alternatively, we might mandate more frequent and thorough code reviews, or adoption of a standardised development guidelines [84]. But what if we are still uncertain about the assurance level of the software? We can turn to technical solutions, perhaps introducing new mechanisms for run-time monitoring [92] to instead *detect* if a violation occurs, and then perform some corrective action, instead of preventing the incursion outright. Each approach for improving security, be it in changing development principles, or deploying technical mechanisms, carries some element of cost, and therefore trade-off.

In single processor contexts, existing approaches often focus on execution of a single thread; in the case of overflows, we are interested to know how the control flow can be manipulated. Often, the focus is on software issues, with execution assumed to be on commodity architectures. Other issues that have been examined include the concept of privilege, where a single processor, which has control of the entire system (such as in a System on Chip context), might need to coordinate different concurrent behaviours safely, usually by some means of hardware-supported privilege level enforcement. As we will discuss in Section 2.2 of this chapter, classical exploits in the context of single processor systems, like buffer overflows, are well-studied in the literature, with different security mechanisms aiming to improve different parts of the system life-cycle.

An example emerging issue: Hardware Trojans

Research in the context of multiprocessor security is less mature, particularly as design environments evolve and embedded systems become more complex. Single processor embedded systems can obviously harbour the same risks and vulnerabilities of well-studied flaws such as memory overflows in vulnerable software, but the impact of a successful exploit can potentially have a greater scope once many other processors, memories, and peripherals all come into play. Insights from research in more traditional single processor contexts are useful, but also need to be considered alongside additional issues that are introduced in the design and use of multiprocessor systems. MPSoC design brings many of its own new emerging challenges, such as *hardware* Trojans, where a malicious IP block is included into a MPSoC

design, as shown in Figure 2.3. As design processes become more distributed and decentralised with design re-use strategies, malicious IP blocks that masquerade as legitimate and useful can be introduced into the supply chain.

Ordinarily, IP blocks are used to accelerate parts of an application, often operating at the request of a processor (Figure 2.3(a)). During system operation, a hardware Trojan can cause trouble in various ways after being activated (after some time, or after a specific trigger); it might start misbehaving, either denying service, or providing spurious data (Figure 2.3(b)), or surreptitiously try to retrieve or corrupt data from elsewhere in the system (Figure 2.3(c)).

Using the same flaw taxonomy as before, hardware Trojans can be categorised as an intentional, malicious flaw, introduced during development in the hardware part of the system. In fact, the hardware aspect provides an interesting challenge when trying to improve security, not least because of the complexity arising from having many parallel entities operating during run time. Once again, there are many potential avenues for designers and researchers; we might enforce security policies that preclude the use of untrusted IPs, or build-in some sort of Trojan detection by analysing register-transfer level (RTL) models [95], with the aim to exclude Trojans entirely. Alternatively, we might instead tolerate the *potential* existence of Trojans in a design, but use techniques to minimise their impact if they manifest, such as high level synthesis approaches to isolate potential malicious IPs [101], or scheduling to try and prevent collaborating Trojans from triggering [77].

In the multiprocessor domain, the challenge is that we face a scenario where any one of *many* concurrent behaviours is compromised or malicious. Architectural features like on-chip component interconnections provide new areas of risk,

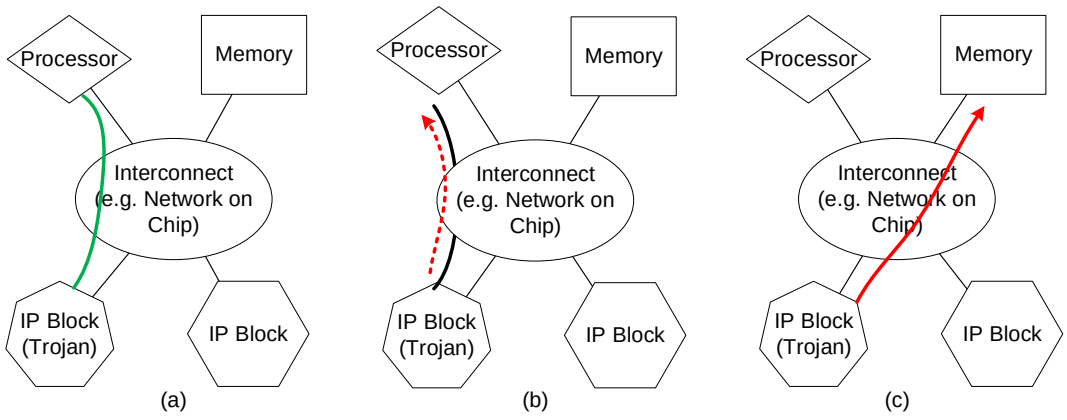


Figure 2.3: Behaviour of a HW Trojan, (a) functioning as expected, (b) generating spurious data, (c) performing unexpected actions/accesses

especially when they enable resource sharing amongst potentially untrustworthy components. Furthermore, whereas the "main processor" in an SoC may have been the key target for attacks, new approaches instead target other hardware components in a system, such as the GPU [67], which implies the need for more comprehensive, system-wide security. Issues that are well-studied in the single processor domain are also exacerbated, particularly as designers consolidate multiple independent systems into a single system. There is no panacea for security; solutions need to target a specific security goal, be it in removing a flaw, preventing the exploitation of a flaw (while the flaw remains), or mitigating the impact of a flaw's exploitation. Such emerging security issues requires novel design approaches and security mechanisms, and it is in this developing area that this thesis contributes.

Other types of attacks: Side-channels

However, before we explore some trends in security defences, side-channel attacks should be mentioned for the sake of completeness. Side-channels are indirect channels that can (often unintentionally) communicate information, and are used maliciously for extracting sensitive data from systems. Types of side-channels include timing variations during execution [61], and power consumption changes [62]. Side-channels are highly dependent on implementation specifics, and are closely linked to the platform architecture.

For example, consider encryption. The secrecy provided by encryption relies on the prerequisite that encryption keys are private. However, implementations of cryptographic algorithms are often optimised to increase performance, which might result in timing variations when executing on a given architecture. If timing is affected by the message that is encrypted, and the key used for encryption, a malicious actor may be able to perform statistical analysis to infer the key used, thus breaking security of the system, as demonstrated by Kocher in [61].

Side-channel attacks are an issue in single and multiprocessor contexts, especially in the context of secrecy/confidentiality, but we will not deal with these directly in this thesis.

2.2 Defences: Design Approaches and Software

In general, existing approaches for securing systems aim to introduce mechanisms that tackle small parts of the security problem, addressing one or two of the key security properties (§1.2.1) to create some sort of protection boundary. Approaches

can be classified by which part of an attack is addressed, be it in attack prevention, detection, mitigation, or recovery. Broadly speaking, security-hardening techniques include proposed software/programming language changes, hardware architecture modifications, custom-hardware, application of formal methods, or combinations of hardware/software additions. Essentially, the security design problem is selecting and implementing mechanisms to realise a chosen security policy[1] throughout various levels in the design. Mechanisms are chosen to perform one or more of the following:

- Detect an attack attempt
- Prevent infiltration into the system
- Reduce the impact of a successful intrusion
- Prevent exfiltration of assets
- Allow recovery from an attack

At the application level, security issues are often focussed around protection of data, either when it is *in motion* (data in transmission from one device to another over a network) or *at rest* (data stored on a device, or some repository, like a database hosted in the cloud). In the 1980's, discussions of security issues were as much about managing people within an organisation as it was about technical implementation of security mechanisms, with access controls designed to reflect the roles, responsibilities, and motivations of different people in an organisation [109]. Information flow was a key consideration, and early security models enforced concepts such as a hierarchy of secrecy levels, which were adapted from military security policies (as evidenced in the development of the Bell and LaPadula model [68]). Later developments then accommodated commercial requirements, such as the Chinese Wall model for preventing information leakage to conflicted parties [14]. A security policy is essentially a representation of actors, objects, and processes in a

[1] If we want a system to be "secure" in an *ideal* sense, designers must derive a set of ideal security requirements that completely capture the needs/wants of all stakeholders. The ideal security policy must then be formulated to satisfy *all* security requirements, and then a set of security mechanisms must be *flawlessly* implemented to realise the ideal security policy. If any of these things are incomplete or inconsistent, *successful* attacks are possible, and we don't have "perfect" security. Trying to decide if a designer has completely captured requirements, addressed them in a policy, and implemented the policy, is, in my opinion, a huge (and potentially impossible) undertaking of requirements/systems engineering on top of purely technical issues with respect to security mechanism design and implementation. We will focus only on mechanisms here, with the fundamental assumption that we aren't aiming for *perfect*, but that there is some sense of "good enough".

system, and how they should (or should not) interact; designers implement security mechanisms specifically to enforce the security policy.

As computer systems have become distributed and decentralised, the desire for information security has required that security properties are addressed by technical innovation, such as the development and use of novel and secure communication protocols to protect data in motion. Such protocols usually involve *cryptography*, such as use of the well-known RSA cryptosystem [104] for secure data transmission in distributed systems. Other aspects of application level security also involve managing human behaviours or business processes related to security, like mandating the use of strong passwords, and ensuring frequent firmware updates which address and patch vulnerabilities that are discovered after a device is deployed [6]. Unfortunately, trying to encourage good behaviour for human elements of a system doesn't always work [12].

Existing work on improving security can be coarsely categorised as techniques for improving the design process, and techniques that deploy mechanisms into the system. Security mechanisms can be inserted to place restrictions on the interaction *between* different parts of a system, or *within* each part of the system (such as around a particular asset). Use of different mechanisms is known as the *multiple independent layers of security* approach [132]. This is pertinent as attacks can originate from external sources—exploiting exposed interfaces—or internal sources, where components in the design have been compromised during design or the manufacture process [76, 101]. *Isolation* is a key concept in security, especially if we want to reduce the likelihood of unwanted interference. In this section, we will discuss some design approaches that factor-in security issues (§2.2.1), as well as explore some software-focussed techniques for improving security (§2.2.2).

2.2.1 Improving Implementation

Application logic is typically implemented in software—after all, a computer is useless if it is not executing any useful instructions. There are many approaches in the literature to enhance security at the application level, where the aim is to avoid issues that manifest from designers dealing directly with low level aspects of the design.

In the design process, there are several industrial standards that guide the development of secure software. In [59], the practice of embedded software development is discussed, highlighting concepts such as the principle of least privilege (software components should only be given access to resources that are necessary for their

functionality, nothing more), as well as various attempts at standardisation, such as Motor Industry Software Reliability Association (MISRA) coding guidelines [84]. Additionally, there exists an international standard for evaluating security of IT systems known as Common Criteria (ISO/IEC 15408), which defines different assurance levels and the use of protection profiles for evaluation [54]. In the Common Criteria framework there is a mix of peer-review practices for code verification in industry, as well as requirements for static and dynamic code checking at higher assurance levels. Threat modelling as part of the design process has also been presented, often drawing from insights in human psychology [114].

Another design process improvement is the increased use of formal methods with concepts like model driven design gaining some traction. Some examples of the use of formal methods include [79] where an attempt is made to bring security design requirements out of documentation and into the design flow, with proposals for things like security annotations that can be checked throughout the design process. Formal methods can also be used for specifying requirements and checking their consistency/correctness [87], particularly with complex distributed systems. Model checking is used in [29] together with dataflow analysis to identify potential sources for malicious input, labelling "tainted" sources and their eventual "sinks", thus providing useful feedback for fixing flaws in a system. Formal methods offer the promise of guaranteeing certain security properties such as isolation guarantees, and this has been used in the development of secure operating systems such as seL4 [60], which can be used to increase the assurance of the platform. However, formal methods are not without limitations. In particular, the cost (in terms of time, expertise required, and ultimately, financial investment) has been identified as prohibitive in many cases, as discussed in a recent National Institute of Standards and Technology (US) round-table [55].

2.2.2 Supporting Security with Software

As a complimentary approach to improving design processes before a system is deployed, run-time mechanisms seek to improve security while a system is operating. Instead of assuming that all flaws are eliminated during the design process, research into run-time mechanisms usually assumes that security exploits are possible, and hence focuses on detection of attacks. Once an attack is detected the system can then try to prevent damage, recover from the attack, or log the attack for later analysis.

One strategy for improving security is the enhancement of programming languages (and their run-time behaviour). Different programming languages have been designed to provide different features, with the aim to reduce the design effort required from programmers. Features include automatic memory management, execution platform portability, and higher levels of design abstraction (such as Object-oriented programming, concurrency models etc.); the idea is to reduce programmer-introduced flaws by "taking care" of finicky details. Languages like Java [34] and Go [32] purport to make programming easier, with low level details managed by interpreters, virtual machines, or compilers, such as built-in bounds checking, or discouraging direct manipulation with pointers. Even so, a lot of embedded system software remains written in lower level languages like C, where problems with buffer overflows, code corruption, control flow hijacking, and information leaks present a large potential for exploitation [122]. In fact, even in higher-level languages like Java, underlying run-time environments (e.g. Java virtual machine implementations) are typically written in lower level languages, which presents considerable risks owing to design complexity [33].

With this in mind, there have been some efforts in trying to improve the security of low level languages like C, such as the extension CCured [86], one of several modifications to the language. In this work, the primary aim was to enhance the security of legacy software by recompiling source to produce "safer" programs. The main addition is around the pointer—essentially, pointers are enhanced with more information, such as permissible bounds (this is also known as the concept of *fat pointers*), or surrounded with run-time checks, in order to reduce the risk of memory exploitation. This is also complemented with enhanced static checks for type safety. Run-time checks of enhanced pointers added 3-87% overhead in execution time, and also introduced compatibility issues with external C libraries. Furthermore, because of the software-based nature of this mechanism, multi-threaded contexts require that locks are acquired to ensure consistent states for the enhanced pointers.

Instead of directly changing programming languages, other run-time monitoring techniques operate alongside application software, acting like an overseer. Intrusion detection systems (IDS) [85] aim to identify that an attack has been attempted successfully. For example, we can attempt to detect malicious software (malware) directly by recognising known *signatures*. This checking of software samples against a database of known threats forms the basis of antivirus software (AV) approaches [125], with coverage that varies depending on the vendor [1, 74]. However, because of the reliance on a database of known threats, so-called zero-day

vulnerabilities are often undetected [125]. Furthermore, AV software typically requires a high level of privilege in a computer system, particularly as it requires the ability to inspect data for many parts of the system. Because use of AV software is part of a "common sense" approach to desktop security, AV software provides a lucrative target, whereby an attacker can abuse a highly trusted part of a system; in using software to defend software, the same risks for inadvertent software flaws exist for exploitation [21]. Firewalls [78] are another widely deployed mechanism (and is often bundled with AV). By means of packet inspection and filtering, firewalls create a boundary between networks, often providing a barrier between local networks and the wider Internet; instead of detecting malicious software, firewalls focus on communication, or network *traffic*.

Alternatively, we can instead monitor legitimate applications to identify where attacks occur. For example, the approach in [92] proposes decomposing programs into "basic blocks" in a control-flow graph and using these blocks to generate information about what software does, and how long it should take to undertake each operation. Code injections are detected by recognising failures in meeting expected checkpoints that are instrumented into the binaries.

An alternative approach uses performance counters provided in hardware to analyse software behaviour, with machine learning models trained to detect variations in "program phases" [24]. This work is based on the underlying assumption that most legitimate software behaviour is cyclic with regards to the resources that are used, and the time it takes, hence recognisable in performance data. Malicious programs also have signatures, for example, a program designed to steal location data from a GPS will exhibit certain behaviours like turning on the GPS unit even if its code has been mutated to avert traditional anti-virus detection mechanisms. Using machine-learning, a monitor was trained to identify malware based on performance counter information provided by processors, with some success.

Another run-time monitoring strategy is the use of a technique called Dynamic Information Flow Tracking (DIFT), also known as *taint tracking* in the literature. The main idea in this technique is that most security exploits arise from misused data. For example, in the case of code injection, untrusted data from external sources is used as program instructions or branch targets. With regards to information leaks, sensitive information is used or handled in an improper way, such as when secure data is prepared for transmission outside the system. By *tagging*, *tainting*, or *marking* information and tracking it throughout the system, we can add safeguards to program execution. Similar to enhanced pointers, we add

metadata to the system so we can make judgements as to the legitimacy of the operations in our program. A typical use case involves detection of a stack overflow attack—when a stack is overflowed from an external source (like while reading a file through a vulnerable function) and the return address is corrupted, attempts to jump to the overridden address will trigger an exception as the address will have been tagged as "dirty" or some other identifier.

DIFT has appeared in various forms over the years. Initially, DIFT techniques were implemented in software, typically introducing fairly high overhead (slowing down application performance 24–37x in [88]) with subsequent developments typically targeting performance improvement. Strategies for implementation include techniques like use of dynamic binary translation [100] to provide transparent security without modifying pre-existing binaries, but again with fairly high worst-case overhead (12.0–46.5x without optimisations).

2.3 Defences: Hardware Support

Looking at security from a hardware perspective is an emerging trend, especially as software-only techniques for security continue to be thwarted or circumvented [121], and hardware-based threats increase [66, 72]. As with software-based techniques, different hardware approaches serve different purposes; some techniques act transparently to applications, usually in some sort of monitoring capacity that is not easily disrupted by flaws in software. Other hardware additions implement mechanisms that are used *alongside* software to realise security policies, with the belief that hardware-based implementations are less susceptible to attack, or that important primitive operations, like key generation or cryptographic verification, are better insulated when they essentially exist at a different "level" to at-risk software. Hardware implementations of security mechanisms can also provide valuable performance improvements by reducing the overhead associated with executing protections as software (such as from context switching by OSes). Furthermore, hardware, being the platform that sits "under" software, seems to be a natural place to create a foundation that is more resilient to predominantly software-focussed exploits. With the rise of custom MPSoC design there is also greater opportunity for customising a platform with hardware-based mechanisms, which was not previously possible with commodity mass-market platforms.

In this section we first begin by looking at the processor, being the focal point of a computer (§2.3.1). We then move outside the processor, examining support

hardware and the move towards multiprocessor system security (§2.3.2).

2.3.1 Processor-centric Protections

Commodity Processor Evolution

Two of the main reasons for implementing security mechanisms in hardware are to improve performance, and to *hide* things from software. In making certain information inaccessible to user software (such as status flags or identification registers), or handling certain checks without software intervention (such as permission checking), security mechanisms can be hardened against software-based attacks.

One of the most influential ideas in processor-supported security is the notion of *protection rings*, the idea that different software should execute with different levels of privilege. The Multics operating system featured one of the earliest realisations of these hierarchical protection domains [112], where each higher privilege level provides a superset of the access rights of the lower levels—software in ring 0 can access all parts of the system, while ring 1 upwards can access less and less. The key insight was that, over the course of an application's execution, different external functions and libraries require different access rights, and so by supporting easy reduction of privilege, and carefully regulating increasing privilege as well defined "gates", a program could change protection domains to more closely match the application's needs. Furthermore, different levels provide *isolation*—while misbehaving or faulty programs in ring 0 could result in a system crash, the impact of user program failures in higher rings would be less catastrophic to the system as a whole.

Initially implemented in software, a request for changes in access controls were trapped and then executed by the OS. Hardware support for protection rings was subsequently introduced, for performance improvement (or increasing "economy", as discussed in [112]) and easier use by software designers. Hardware was added to track the current protection ring (new register flags), as well as to perform access control checks as part of virtual memory management, where current privilege level can be checked against segment descriptors of smaller areas of memory. New machine instructions for changing privilege level (and essentially, the "mode" of the processor) were introduced, which was then used by software developers to implement their protection domains.

This notion of different privilege levels proliferated to other processor architectures (such as x86, since the 80286 [49]) allowing the separation of software into at

least two levels, nominally the privileged *supervisor* (for the OS kernel) and non-privileged *user* (for all application programs).

Virtualisation was the next evolution, where an even higher privileged, so-called ring -1 was added into processors to allow an underlying virtual machine monitor (VMM) or *hypervisor* to manage multiple *guests* co-existing on a single host machine. The hypervisor is endowed with the ability to set-up and coordinate several *virtual machines* (VM), where software running on each VM can "see" an entire (abstract) computer system, but not necessarily other VMs executing on the same hardware, thus isolating different applications. This approach introduces some overhead however, as the hypervisor is required to perform some translation or trapping every time the guest OS wants to do anything like access peripherals [59]. Hardware support like Intel VT-x [50] improved virtualisation performance by adding new instructions for guests to enter and exit virtual machine modes while maintaining the protection of the host. Virtualisation is used extensively in large cloud-based services, and there is a recent trend of using virtualisation in the context of embedded systems [43], although problems like guest-to-host escape (where privilege levels are violated) are possible [16, 81]. Subsequent attempts to improve security have tried to add functionality into hypervisors, such as the Memory Introspection Engine (MIE) presented in [81], which analyses OS kernel code for malicious modification. The hypervisor is able to run independently to vulnerable code because of hardware-based mechanisms.

In a related approach, but orthogonal to hierarchical rings, ARM's TrustZone [75] adds a secure mode to the processor, thus creating, in essence, two logical processors on one physical processor. TrustZone enforces the notion of separating processes—in this case, along the lines of secure and insecure (or normal) processes, where the two operating modes, normal and secure, have address space separation enforced in hardware. An important feature of the TrustZone architecture is the extension of the secure/normal distinction *beyond* the processor into the rest of the SoC; on-chip peripherals are set as secure or normal. Software in the secure world have full access to the address space, while normal world processors cannot access secure world memory, and this extends into peripherals, where secure peripherals can only be accessed by a processor in secure mode. This allows two operating systems, one secure, the other not, to share the processor. Sensitive data and functionality, such as biometric data, or cryptographic operations, can be placed in the secure world. Normal world software is completely oblivious to the secure world memory (and cannot *physically* access it), and can only request secure world functionality through well defined hardware implemented interfaces (such as the

secure monitor call instruction). Privileged entities (such as an OS), can add further protections by configuring components such as a Memory Protection Unit (MPU) [4] which can perform access control checks on regions of memory, within the context of the given world. Such techniques have only recently started appearing in embedded systems contexts, such as in ARM-V8 based Cortex M series, (in the Cortex-M23 which appeared in 2016 [80]).

In contrast to TrustZone's secure/normal delineation, Intel's Software Guard Extensions (SGX) [83] provide hardware-based isolation for multiple smaller protected areas. Hardware support is added for *enclaves* that are protected even from higher privileged programs; in *enclave mode* additional address checks are performed to ensure that only enclave code executes on enclave data irrespective of privilege. Additional facilities are added to allow selected data to be encrypted upon exit of the processor (such as during a cache eviction), as well as cryptographic measurement to check for integrity of the enclave.

Security-focussed Processor Architectures

In parallel to developments introduced in commodity architectures, where security features have been added, there also exists some architectures where support for security was a first-class consideration. One of the major works often cited in the realm of secure processor architectures is the XOM (pronounced "zom") or eXecute-Only Memory machine, presented in [126]. This work was largely motivated by anti-piracy efforts. Execution of pirated software is prevented by providing secure execution environments, described in this work as *compartments*, and heavy reliance on encryption and cryptography for isolating program code and data of concurrently executing applications. Each *compartment* has a unique session key, except for the *null compartment* (where insecure software can execute). The session key is used to decrypt all machine instructions, as well as determining if data reads are allowed. Intercommunication between *compartments* is performed using machine instructions that allow moving to and from the null compartment. As all data is tagged, each application can only interact with the data it creates itself, or explicitly takes from the non-secure part of memory. Untrusted external memory is accommodated by encryption, as well as hashing, to verify the integrity of anything sent off-chip (violation of code integrity should result in non-matching hashes). Context switching for interrupt handling is also allowed, with special instructions to encapsulate secure machine state for moving, as well as hashing for verifying integrity. As may be expected, this architecture has not

been widely adopted, largely due to significant overheads in terms of performance (cryptographic functions), and resources needed (extra bits for tagging, dedicated cryptographic functions).

The more recent AEGIS processor design [121] attempts to secure execution by providing an execution platform with different operational modes, cryptography, and ultimately, better performance than an architecture like XOM. AEGIS features four execution modes, STD (Standard), TE (Tamper Evident), PTR (Private Tamper-Resistant), and SSP (Suspended Secure Processing), each with a different level of security. STD is essentially a normal execution mode; TE involves checking hashes to verify information from off-chip memory; PTR adds encryption in stores to off-chip memory; and SSP allows an application in PTR or TE modes to perform insecure operations (such as accessing functions from a shared library) with reduced overhead. The idea behind these modes is that a more flexible approach to security can be taken; in architectures like XOM, the entire application is considered secured, and so must operate with the overheads inherent to the security approach. In AEGIS the central tenet is that only *parts* of an application need to be secure, that only a subset of the application requires encryption or hashing.

Memory in this architecture is partitioned according to the level of security (whether it needs encryption, or hashing, or nothing at all), and the proposed programming model has procedures and data structures identified as *unprotected*, *verified*, or *private* in program code, being placed in the appropriate region by a compiler. Another interesting thing that underpins AEGIS is the use of Physical Uncloneable Functions (PUF) for generation of processor *secrets*. PUFs provide an alternative to storage of digital keys in non-volatile memory; using physical properties such as timing delays (which differ from device to device due to variations resulting from manufacturing processes), secret keys can be generated for each processor. The implementation used in [121] uses a race among multiple-bit inputs, and BCH code (a type of error correction code) to make generated keys reliably consistent. Multitasking applications are managed by the use of a secure kernel.

CHERI [135, 136] takes a different approach to secure design; instead of cryptography, CHERI provides hardware support for *capabilities* [25] by means of an extended 64-bit MIPS ISA. The concept of capabilities is intuitive to grasp; memory accesses are only possible via capabilities which define the actions that a certain process may perform on a certain part of memory (for example, managing permissions for reading, writing, and execution). Memory is protected by mandating that all accesses have to be made through these unforgeable safe pointers. Capabilities also facilitate application sandboxing. Woodruff et al. suggest that the use of

capabilities provides greater protection as the granularity is much finer than that of traditional page-based mechanisms (conventional virtual memory). Low level languages like C can have pointers implemented instead as capabilities, adding bounds checking as well as permissible action enforcement. To demonstrate their work, CHERI has been implemented as an FPGA prototype (by extending BERI—the Bluespec Extensible RISC Implementation from the University of Cambridge [19]), adding a new coprocessor, a modification of LLVM for capabilities, and a modification of FreeBSD. Additionally, the authors highlight that their work supports incremental adoptability—legacy programs can run without modification, minor modifications can add some additional security, and new programs can be created from scratch.

Other recent work in protecting memory involves the creation of secure sections of memory, and the use of cryptography for attestation (proving code integrity). Iso-X [28] introduces modifications to both processor architecture and instruction set to provide support for secure *compartments*. A virtual-memory-like system is implemented, which performs permission checks on memory accesses. The OS creates compartments; once control has been transferred to a secure compartment, the processor remains in that compartment until relinquishes control.

Processor modifications to address the performance shortcomings of software-based implementations can also be found in the literature. In fact, returning to DIFT (§2.2.2), initial work involved making extensions to the processor datapath, extending the width of registers, ALUs, and other components [23]. The FlexiTaint approach [129] added an additional pipeline stage to the processor after the *commit* stage to perform the required taint propagation operations, adding a new taint register file and cache structure. By moving taint management to hardware, the added overhead was within 1-8.4%, as opposed to the higher overheads of software-only approaches. Another recent work [26] introduced a cache-like structure to support multiple metadata security policies.

2.3.2 Moving Beyond the Processor to the Platform

Aside from enhancements and modifications to the internal processor architecture, other works have looked at improving security by changing facilities *outside* the processor. This is particularly pertinent as the embedded system design paradigm shifts towards heterogeneous multiprocessor systems, which encourages the execution of multiple parallel applications on different types of cores. Security is im-

proved by adding cores/accelerators dedicated to security functions, or by managing how on-chip resources are used or partitioned.

In industry, there is a widely adopted standard known as Trusted Platform Module (TPM), standardised as ISO/IEC 11889 and maintained by the Trusted Computing Group [127]. The TPM is a security-hardened microprocessor that is specifically dedicated for improving security. As a *trusted* module, the TPM can perform cryptographic and authentication functions, and is used to check the integrity of various parts of an execution platform, such as the BIOS, to establish a trusted computing base (TCB). This is illustrated by Intel's Trusted Execution Technology (TXT) [39], where a "chain of trust" can be established at boot-time. The code of each software module involved in the boot process is checked against a hash of a previously known "good" version; if the module passes, it is trusted, and can then be used for loading the next module—which is also checked—thus setting up each link in the chain of trust. This technology is useful for establishing a known, trusted initial state.

As we move towards multiprocessor contexts, additional hardware blocks dedicated to security can be added to implement strategies like run-time monitoring. For example, SHIELD [91] adds a separate processor dedicated to security, which is then connected with the multiple application processors via FIFOs. At run-time, instrumented applications communicate with the security processor, which checks that instructions are executing in the expected order and with expected timing. In many ways, work in this domain is similar to work in reliability, with SHIELD also able to detect the effects of bitwise faults. Such techniques often have software-based analogues (c.f. §2.2.2), but offering improved performance, reduced overhead, or isolation from direct malicious manipulation. For example, DIFT techniques have been implemented as hardware accelerators, which offload some of the computation and memory access overhead for dealing with metadata [18, 57]. In [98], tag management is considered at the Network on Chip (NoC) level, where network interfaces provide support for proper retrieval and storage of tags.

Another security issue that emerged as research began considering the wider computer system (as in SoCs and MPSoCs) is communication, both on- and off-chip. As described earlier in §2.3.1, encryption has often featured as a means of protecting memory, particularly when considering off-chip (external) memory as untrustworthy. The survey presented in [44] discusses memory encryption, and its use in securing systems, where there is a large emphasis on the concept of *confidentiality*. One key observation of this survey was that encryption typically incurred

large performance overhead; where overheads impact on response time, adoption of a security strategy is less likely.

Hence, work such as SecBus [15] aims to address performance issues, while supporting increased security. Instead of changing the processor, SecBus adds an additional module between the SoC and off-chip memory for cryptography, with security policies defined and managed in software. These security policies are used in a virtual memory type mechanism where each security policy has page-level granularity. Each policy specifies the cryptographic keys needed, read/write permissions, integrity mode, and whether or not the policy is valid. A cache structure is implemented for these data structures to improve performance. The hardware additions are then combined with a software flow to allow for trusted secure booting.

Another approach, presented in [45], similarly attempted to address performance cost due to memory encryption. DynaPoMP looks at memory protection for scratchpad memory (SPM), small memories that often have software controlled allocation, unlike caches. The main idea of this scheme is the use of SPM as the main work memory for sensitive data; only once sensitive data is finalised it is "evicted" from the SPM to insecure RAM, where it is encrypted en-route. To improve performance, the policy-driven memory allocation methodology attempts to minimise data transfer costs from SPM to RAM, factoring in encryption/decryption costs for sensitive data, as well as the balance between allocating space for sensitive and non-sensitive data. The two main policies are *SensitivityFirst* (keeping sensitive data in SPM as long as possible) and *AccessFirst* (keeping frequently accessed data in SPM as long as possible). The SPM is divided into two regions S1 and S2, where each of these policies govern allocation (SensitivityFirst and AccessFirst respectively). New data is attempted to be allocated into S1, if this succeeds and if there is victim data, this victim data is moved to S2, and any victims in S2 are then evicted to the RAM. Benchmarks are then used to trade-off the sizes of the two regions to try and reduce latency.

Aside from encryption, multiprocessor system security has also been addressed by means of platform customisation and policy generation, where the key aim is to isolate the many different parts of the platform for a given application. In an explicit multiprocessor context, the policy-based approach, PoliMakE [7], proposes a means of securing software execution in a multi-processor environment, by generating a secure-schedule, memory mappings, and a set of requirements that are enforced at run-time. The policy engine first analyses an application that has had its sensitive

buffers/variables identified by code annotation, determining the required buffers, processing requirements, and communication channels required to ensure security of the sensitive data. A task graph is generated, and this informs the mapping of data and generation of policies. Each task in the system is essentially provided its own sandbox. The security-scheduler then loads tasks at run time based on the generated policies. Essentially, each task is isolated, and is provided only the resources it requires at any given time. The follow up, TrustGem [8], extends PoliMakE and uses the notion of Embedded RAIDs-on-Chip to provide logical scratchpad memories. The enhancement here is to reduce the memory requirements of generated policies, thus increasing overall performance.

Another prominent class of improvements in multiprocessor security has been in the area of NoCs, particularly for ensuring that access to on-chip resources are well regulated.

For example, [35] introduces a firewall at network interfaces to enforce segment-level rules. The security model implemented is based on "deny rules", which means that unless explicitly protected, any processor has access to that memory location. The observations in [36] highlight the benefits for providing such protections, especially for reducing communication traffic during exploitation.

Another firewall is presented in [73], where entire OS instances are isolated. Dynamic permissions are managed by an integrity kernel, which runs on an isolated lightweight processor. This processor is connected to all firewalls in the system, and any entities that can influence policies (modify rules). The firewall only stores a limited set of rules; if there is no rule for a transaction the integrity core searches for and loads the required rule, or generates an error response. The integrity kernel could potentially be a bottleneck, depending on the size of local firewalls and the number of memory transactions.

Fiorin et al. [31] propose distributed Data Protection Units (DPUs), managed by a central Network Security Manager (NSM). In order to reconfigure DPUs at runtime, tasks that are identified as "supervisor" running on processors that are "secure" send requests to the NSM. This relies on processors having different operating modes, as well as the need for a privileged entity in the system. In order to provide for greater scalability (in terms of number of permissions), [97] proposes the use of a memory hierarchy, where a Permission Lookaside Buffer is accessible within the NoC-MPU at each processor, and permissions are stored in a table in memory. Identification of tasks is based on information provided by the interconnection protocol (in this case, OCP or VCI), which relies on trusted software for correct identification.

With respect to handling violations once they are detected, [20] proposes the reconfiguration of hardware firewalls to facilitate a heightened security mode where memory accesses are severely restricted. A separate security bus is used, allowing the firewalls to be managed by a central trusted processor. However, this means that run-time reconfiguration of protected regions depending on the state of tasks is not possible as there is no interaction between tasks and the security manager.

2.4 Trends and Opportunities

There has clearly been a variety of different proposed approaches for increasing the security of computer systems, and it is generally acknowledged that security is of increasing importance. From our exploration of some background in security, we can observe the following trends:

There is a shift in focus towards hardware

Good software design is difficult; good software design *with security* seems at times like an almost insurmountable challenge. As we have seen from the early days of software-mechanisms in Multics, through to the addition of additional hardware components like the TPM, research and development of hardware-based approaches is becoming more widespread, especially to provide a certain level of insulation from vulnerable software. Having that said, hardware additions often do not work alone; they simply implement mechanisms that are only fully useful if properly utilised by application software. Additionally, we are now seeing more work that looks at the platform as a whole, where designers are also considering the potential risks of the many other components in a system aside from the processor. While we can improve security by adding dedicated security-related components in the design, we must also be wary about attacks originating from otherwise inconspicuous, "normal" hardware, especially when multiple processors are involved.

The OS is all-important in many security techniques

In several of the works we have discussed, there is a prerequisite that there is some sort of privileged entity in the system, some trustworthy authority that can configure and manage security functionality. Usually, this is in the form of an OS, which is, as part of resource management for applications, responsible for setting up isolation boundaries, and ensuring that hardware mechanisms in the underlying platform are properly used. As such, a tight coupling between privileged software and

hardware mechanisms is highly critical. This proposes an interesting challenge—can we decouple this relationship? What happens if there are bare-metal components in the design? What happens in highly heterogeneous systems, where several complex processing cores execute in parallel, but there is no overarching OS across the different cores? There is a research opportunity to explore strategies that might reduce the reliance on an OS, while supporting security across the different layers of a system.

Heterogeneity is tough

Many embedded systems are designed for mixed-critical applications [93, 105], where we may have varying requirements for timing, performance, or security. To address these different constraints, we can see that multiple cores, some general, some highly specialised, can all be incorporated into a single chip. And yet, existing research has until recently focused only on security improvements of a single core. In many threat models, there is an assumption that all components on-chip are trustworthy, and the security boundary is needed for when data moves off-chip. However, as MPSoCs become more complex, and with issues like hardware Trojans, there is a research opportunity for considering security in heterogeneous systems, where security should be considered within the platform from the outset of the design process. Where a lot of existing work focuses on the implementation details of specific security mechanisms, work in multiprocessor systems and network on chip architectures is less mature. Raising the abstraction level to a system-level helps designers manage heterogeneity; instead of a low-level focus on individual components, designers can reason about application tasks and their interactions. In fact, a system-level security approach would be useful, particularly as different subsystems might have different requirements, and so a mix of security customisations might be necessary. This is especially important when trade-offs inevitably occur—in deciding how best to improve security in a design, we need to understand the costs involved, be it in resource overhead, reduction in throughput, or effects in other design factors. With many different low level mechanisms possible, MPSoC design needs to focus more on how and where to improve security, be it manually, or through an automated process. By having a system-level view of security, designers might be better equipped to make decisions about how to add security into a design, and for which parts of the system.

2.5 Summary

In this chapter we presented some background on security, looking at the evolving nature of attacks, the different angles at which security improvements have been attempted, and the rise of hardware-based approaches to security, particularly in the shift from processor-centric enhancements, to security in the wider platform architecture. While this review has not been exhaustive, we highlight some interesting areas in the literature, noting the need for research in more heterogeneous contexts, as well as opportunities in finding security approaches that will work without reliance on operating systems. In the next chapter, we turn our focus to MPSoCs, examining some of the security issues, and potential models for providing a foundation to improve security.

CHAPTER 3

Foundations for a Security Approach for MPSoCs

In this chapter we lay foundations for improving security in an MPSoC, and work towards a conceptual model for thinking about heterogeneous components and their roles in an MPSoC-based platform. Using a motivating example application we begin to identify general security properties which are useful for future embedded systems, and this allows us to identify security mechanisms that can be implemented in the execution platform. We also examine different access control models, and consider their applicability to our motivating example. Finally, we propose a high level abstraction for quantifying the potential impact of a security exploit, which is useful as a starting point for comparing different architectures in later chapters of this thesis.

3.1 Motivating Scenario

3.1.1 Concurrent, Consolidated, and Complex

Having shed some light on some background in security, we now turn our attention to the specific challenges of embedded systems and MPSoCs. As a means of establishing the context of our research, and as a means to explore the nature of complex embedded systems applications, let us discuss a motivating scenario, inspired by the domain of Internet of Things (IoT) enabled smart homes [5, 71]. We draw from this example to discuss the security requirements and potential threats to complex embedded systems, and work towards mechanisms that would be useful for a security approach for MPSoCs.

Task Group (processor)	Tasks		Functional Description	"Criticality" Remarks
Environmental Control (P_{Y0})	**E0**	Light	Controls the lights	Physical control with human impact
	E1	AC	Controls the heating for each room	Physical control with human impact
	E2	Sensor	Receives and processes sensor data	Physical sensing, real-time requirements
	E3	Environment	Uses sensor data to manage environment	Physical control, real-time requirements
Home Security (P_{Y1})	**S0**	Security camera	Controls security cameras, processes footage	Sensitive information, real-time requirements
	S1	Door access	Controls door locks	Physical control, sensitive
	S2	Biometric access	Manage fingerprint entry system	Real-time requirements, sensitive data
Fire Detection (IP core)	**F0**	Fire manager	Detects fires, raises alarm, and extinguishes	Real-time requirements, must always be active
Entertainment (P_{X0})	**N0**	Speaker	Controls multi-room speaker system	Physical control, less-critical
	N1	Media decoder	Manages media files, and processes audio	Potentially vulnerable, non-sensitive data
	N2	Local interface	Manages local control panel for UHAH	User interface, potentially vulnerable
Remote management (P_{X1})	**R0**	Remote interface	Web interface for remote management	User interface, potentially vulnerable, exposed
	R1	Update	Manages remote update of controller software	Potentially accesses critical data
	R2	Authentication	Authenticates remote users (checks passwords)	Accesses sensitive data

Table 3.1: Task groups (subsystems) of the Integrated Home Automation Hub

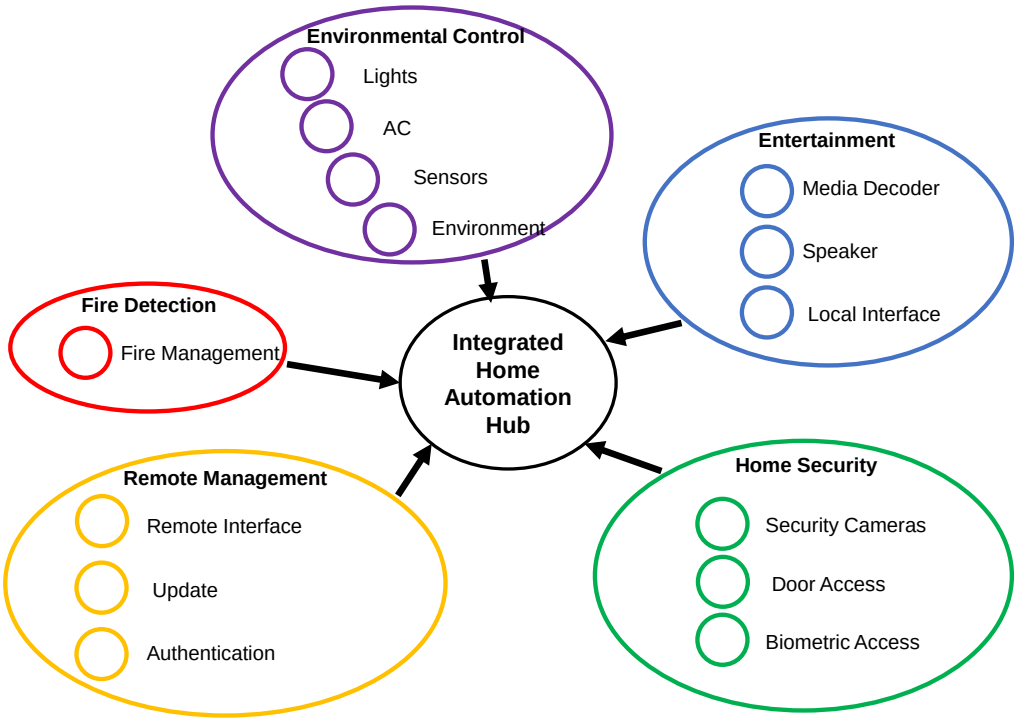


Figure 3.1: The Integrated Home Automation Hub

Consider the Integrated Home Automation Hub application (henceforth referred to simply as, the *Hub*), shown conceptually in Figure 3.1. This system integrates different functions in a home: environmental control, home security, fire detection, and entertainment. Each of the different functions in the *Hub* are partitioned into several tasks within task groups (or subsystems), shown in Table 3.1. Each task contains its own program code, and can execute concurrently with other tasks. These tasks interact with one another, both within their own group, and with tasks from other groups. For example, an owner might wish to program her *Hub* to turn on lights and music automatically when she unlocks her front door with her fingerprint—this is an interaction between *Home Security*, *Environmental Control*, and *Entertainment* task groups.

Whereas traditional design approaches might implement the application across several individual devices, we instead envision the scenario were a designer choose to *consolidate* the various subsystems and execute the *Hub* on a single heterogeneous MPSoC, such as one shown in Figure 3.2. In this general heterogeneous architecture, we have several different types of processing elements with several different types of resources. Initially, this platform has no security additions and the on-chip interconnect carries memory transactions initiated by processors and

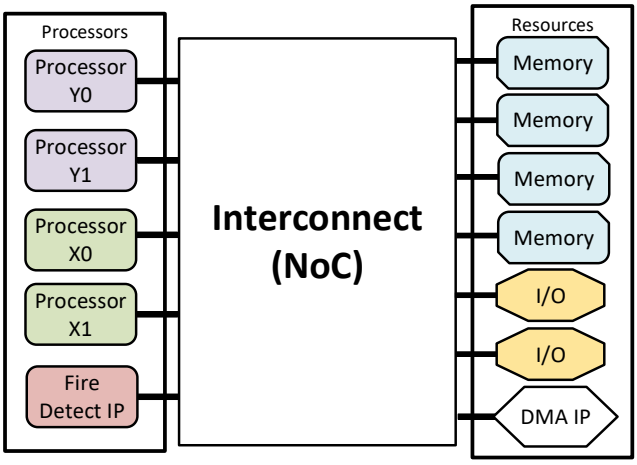


Figure 3.2: A typical MPSoC for the Integrated Home Automation Hub

DMA-capable IP blocks, as well as the responses from resources that are accessed. In general, the interconnect allows all-to-all communication.

Let us assume that we have two different types of processors (PX, PY), together with several different types of shared resources (memories, I/O, and hardware IP blocks). The processor variants provide a different trade-off between complexity and capability: PX is used for the *Entertainment* and *Remote management* task groups, which might make use of an OS (which provides multimedia functionality and external connectivity); PY is a simpler processor that is used primarily with bare-metal programs that can be statically scheduled. We also implement the *Fire Detection* functionality as a dedicated hardware block for reliability. This block can directly access memory. Other IP blocks with DMA capabilities could include hardware accelerators, such as a vector unit for signal processing, or a graphics controller for managing a HMI panel. The configuration registers for these IPs, as well as I/Os, and memories are mapped into a globally shared address space. Over the course of the *Hub's* lifetime, the various task groups all operate in parallel, sharing the different on-chip resources by making memory reads and writes.

3.1.2 Characteristics of MPSoCs

From this description we can make some informal generalisations about characteristics of embedded systems implemented as MPSoCs:

- there are numerous *concurrent* behaviours implemented as numerous *tasks*, with different relationships between different task groups

- the platform features many different components from different sources (heterogeneous)
- components can be shared by different task groups at different times
- the platform is customised to the application
- different parts of an application have different criticality requirements
- there is a common communication backbone that integrates all the various parts of the platform (and allows memory transactions between components)

These characteristics need to be accommodated as we consider how best to improve security.

3.1.3 Threat/Attack Model

As we have seen in previous chapters, security attacks can take many forms, targeting many parts of the system. Unfortunately, it is infeasible for designers to concretely identify and address all potential threats to a system. Hence, to constrain the design problem, we have to decide on a *threat model* [114] which gives us a (non-exhaustive) view of the issues we may need to address in our security policy. While this makes the design process more manageable, our choice of threat model carries the obvious caveat that security issues may still be present and simply undiscovered due to limitations on the expressiveness of a given model. As such, we assume that we are dealing with highly motivated attackers who have a thorough understanding of the system design.

In this motivating scenario, let us focus on two key aspects of the general MPSoC architecture: the existence of shared resources and the all-to-all communication capability provided by the interconnection infrastructure. Taking a less optimistic stance, our threat model assumes that a task in the system has been compromised in its entirety, either through remote exploitation of flaws, or due to its untrustworthiness (i.e. a Trojan task was introduced by a malicious designer during design).

From the vantage point of a compromised task, and in the absence of security mechanisms, we give the attacker *full* capabilities to arbitrarily access parts of the address space. In addition to this, we also consider the threat that one of the DMA-capable IP blocks has been compromised instead of task; in much the same way, the IP block can issue arbitrary memory transactions. We assume that our interconnect

behaves correctly as expected, and that any security components we chose to add are trustworthy. As all the resources on our platform are mapped into a global address space and can be accessed through the shared interconnect, an attacker could perform the following attacks:

A1 If the compromised task is running bare-metal it can generate a memory access transaction to any part of the platform. This can result in disruption of other tasks or access of sensitive data

A2 If the compromised task can configure an IP with direct memory access capabilities, it can configure the peripheral to access a part of memory on its behalf.

In addition to this, we consider additional threats where an OS is used to implement memory protection (such as through configuration of MPUs [4], or enforcing access control in software), by acknowledging that privilege escalation is possible, thus introducing the following possible threats:

A3 Any MPU is disabled, thus enabling unprotected access to the rest of the platform.

A4 The MPU is reconfigured to disrupt the operation of other tasks located on that processor, i.e. by restricting memory accesses such that the other tasks cannot perform their intended functionality.

In these discussions, we are not concerned with how an attacker initially compromises the system; instead, we focus on limiting the impact of a successful initial attack, although mechanisms like buffer overflows described in §2.1 provide a good starting point.

As an example, one possible attack vector could be code injection via a buffer overflow in the Remote Interface. Without any security additions, the task R0 can then be used to access parts of the platform that it would not ordinarily access. The attacker can perform some malicious activities, such as disabling the Fire Detection. Alternatively, a potential A2-type attack could involve the Local Interface being compromised such that it re-programs the graphics controller to stealthily eavesdrop on sensitive data from the memory of other tasks, such as the case presented in [67].

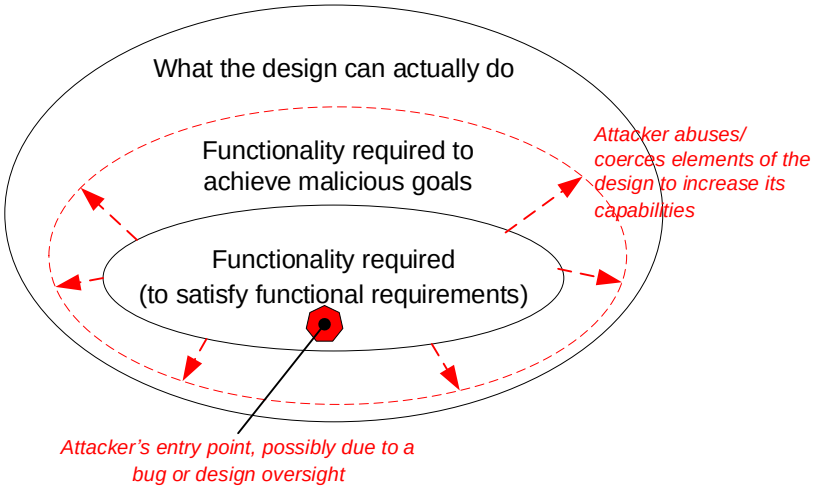


Figure 3.3: Exploiting the capabilities of the platform

The key problem that we face is that the underlying platform typically provides more capabilities than is explicitly needed by an application; this is especially pertinent in general-purpose platforms where we over-provision resources to provide maximum flexibility. Consider Figure 3.3, which represents what a component can do. The aim for attackers is to *acquire* enough capabilities to perform malicious actions; for example, an attacker can compromise one of the tasks in a system. From that entry point, the task can be used to abuse the actions that it *can* do, but wouldn't do under normal circumstances.

3.1.4 Security aims and requirements

With the threat model in mind, we now consider the security aims and requirements of our system.

Intuitively, the different task groups in the *Hub* example (Figure 3.1) have different levels of *importance* and *risk*. For instance, *Fire Detection* has strong real-time requirements, whereas *Entertainment* provides convenience functionality only. Deciding what we want to protect is largely a question of values; we may wish to ensure *safety*, which means we need to protect the *Fire Detection* system. Alternatively, we might highly value *privacy*, in which case, data logs about comings and goings in the *Home Security* subsystem may need to be protected. These systems can be considered as "security critical", in that we require some means of protection.

There are also several other ways in which we might wish to classify criticality—a task might have stricter real-time requirements, or perform physical control, or

deal with sensitive/private data, and these could be taken into account when forming security requirements. It is also worth noting that security is a precondition for safety and timing criticality, which might encourage the designer to guarantee the security of those parts of the system.

From a different security perspective, a task might even be thought of as "more-critical" if it is likely to be exposed to malicious entities. Such exposure might be due to its use in interfacing with wider networks, or if it was developed by a 3rd party. Combining the different criteria results in a complex classification, particularly when we also consider the capabilities added by DMA-capable IP blocks that are used by various tasks. We might opt to draw boundaries between critical/non-critical tasks so that a hijacked task will be *isolated* from other parts of the system, thus limiting the impact of an attack. However, deriving a binary "critical/less-critical" classification for the purposes of deciding whether things should be made "secure/non-secure" is challenging, and may not provide sufficient expressiveness to capture the design requirements. Still, the aim to isolate the various systems is clearly necessary, which leads us to the idea that we should isolate at the level of task group, where all groups are protected from the undue influence of one another.

While it would be desirable to completely isolate task groups from one another, particularly when different functions have different tolerance and susceptibility to attack, we need a means to ensure safe, fine-grained resource sharing where required, thus, we need the ability to modify access permissions. For example, consider the situation where a remote user wishes to examine an Environmental Data log. The Remote Interface should not necessarily have *permanent* access to the environmental data; the more prudent design would involve R0 only accessing the resource when it is required, i.e. when a user has properly authenticated and requested that specific function. In the case that R0 is compromised, as per the threat model, temporary access in this way would complicate getting access to the desired data.

Stepping back to the bigger picture, in our embedded system design we need a security model, and a means of enforcing such a model, that can be used across all entities in the heterogeneous MPSoC platform. The designer can then trade-off the potential costs of introducing isolation and protection enforcement, granularity of the protection, and their evaluations of the potential risk and impact of any given attacks. In the next section, we explore various access control policies as the basis of a potential security model.

3.2 The Need for a Security Model

In order to improve the security of a system, we need to have a clear understanding of what a system should be doing when it behaves correctly. We also need to understand what our security requirements are (as part of the security policy), so we can consider the "important aspects of security and their relationship to system behaviour" [133]. A security model allows us to represent (informally or otherwise) the various entities in a system, and how security mechanisms affect their interactions. Furthermore, they allow us to make originally vague security objectives more concrete (for example, "this device should not be hackable" can be revised into a requirement about protecting a specific area of memory from spurious modification). Security models also allow designers to check the implementation throughout the design process against the intended, ideal model.

3.2.1 Access Control Policies

To develop an appropriate security model, we first consider access control policies in the context of how they might be applied in an embedded system such as the *Hub*. Fundamentally, entities in our system, either software tasks or DMA-capable IPs, need to have their (memory-access) capabilities constrained. As discussed in [111] there are several different types of access control policy that can be implemented, each with their own merits and drawbacks.

Mandatory Access Control (MAC) [111] policies make use of a central privileged administrator, managing access permissions for all tasks. Requests for new access need to go to the administrator, typically the OS kernel, as in SE Linux [99], by means of performing system calls (which typically invoke high overheads from context switching [9]). It is challenging to directly apply a MAC policy in a heterogeneous embedded system, particularly as tasks do not easily map to the "security levels" as described in [111] due to the lack of a clear hierarchy in terms of data sensitivity.

Discretionary Access Control (DAC) [111] policies provide a more flexible way to manage access control; entities in the system grant and revoke access to the objects of each other. This could be applied to our embedded system scenario, as tasks intuitively "own" data, and transfer data to each other at run-time. For example, the Sensor task collects and processes data from the environmental sensors; the data log is then displayed by a different task.

Role-based Access Control (RBAC) [30] policies associate permissions with roles, as opposed to tasks; tasks might perform different roles at run-time, and thus require different permissions. This too is useful for our embedded system scenario, where IP blocks and shared code lends itself well to RBAC, as they can be seen to performing different roles depending on the task it is working for; this is appropriate given the shared nature of on-chip components. Tasks themselves may also have changing roles over time; for example, the Local Interface might display only a subset of system data depending on the user that is authenticated at any given moment.

Attribute-based Access Control (ABAC) [46] (also known as Next-Generation Access Control) policies present a more sophisticated approach to dealing with access controls, especially at run time. Access decisions are made based on examining a combination of rules, the requesting entity's attributes, the target object's attributes, and environment conditions (such as time of day, location, or other *contextual* information). While implementation and enforcement of these policies is complex, the nuances of ABAC would also be useful for an MPSoC, particularly where legitimate actuation and sensing, or movement of data, can be linked to contextual information.

3.2.2 Implications of Sharing

With these options in mind, we look now at our application to decide which policies to adopt. In our motivating scenario, we identify three main types of sharing: shared data, shared code, and shared IPs.

In the case of shared data, take the following example. A local user may access the control panel with the intention to review images from the security camera (S0). The local interface (N2) could have permanent access to the footage, and so the local interface task itself can simply guard access to the sensitive data. However, should task N2 become compromised, the data is unprotected. Instead, we can set up the relationship between the S0 and N2 as a case of controlled/regulated data sharing.

Shared libraries are often employed to reduce engineering effort. Consider the Media Decoder and Local Interface; these tasks can share a code library that assists in displaying contents from a media file. In many ways, shared libraries have similarities with IP blocks. They provide a "service" for tasks; in other words, they take an input and produce some output on behalf of another entity. As demonstrated

in return-oriented programming style attacks, shared code can be exploited by an attacker to add additional capability to a compromised task.

Shared IP blocks in many ways combine the notion of shared data and shared libraries; tasks share data (or memory regions) with IPs to make use of extra functionality. For example, an accelerator for signal processing could be shared between two tasks. However, we need to prevent data leakage; the IP should not be able to access simultaneously the data regions belonging to multiple tasks, nor should one task be able to configure the IP to access the data of a different task. This is akin to changing the role of the IP block at run time, provided we use something like a hardware mutex to support safe resource sharing.

From this discussion, it can be seen that a *mixture* of access control policies would be most useful, and access controls need to be applied universally. All tasks and IP blocks in the system need to have a set of *permissions*, some of which are temporary. It is quite natural to express access controls as relationships between tasks and the resources shared between them, so we therefore need two key features:

1. The ability to set up permissions for managing read/write access to regions of the address space which can be shared between two (or more) tasks/IP blocks

2. The ability to grant/revoke permissions at run-time

Where resources are exclusive to a task, or always shared between tasks, we need to define fixed permissions, as in a MAC policy. For dynamic permissions, we can distribute authority for access control among tasks to reduce the potential risks involved with relying on a single central authority.

This brings us to our proposed security foundation for MPSoC-based embedded systems, one based on *isolation* between tasks, while supporting legitimate task interaction through shared resources. Instead of classifying parts of the design as secure/non-secure, we instead propose multi-domain isolation between tasks or task groups, with enforcement across the entire platform. To achieve this, we need to abstract the heterogeneous components in the design, upon which we can apply a universal access control policy.

3.3 Abstracting MPSoC components

3.3.1 The Task/Resource Relationship abstraction

Historically, access control models were used to help manage the flow of information, and developed as a means to deal with *people* in an organisation [111] (as evidenced by the terminology of *users* and *roles*). To successfully apply those ideas to our MPSoC design, we need to decide how best to *abstract* the various components (tasks, IPs, resources, I/Os etc.) such that we can analyse and manage their interactions at a higher level of abstraction to achieve our security aims.

One way to conceptualise the MPSoC is as a collection of *active* elements that *initiate* memory transactions, and *passive* elements that respond to memory requests. The active elements are components such as software tasks executing on processing cores, or IP blocks that can act as bus-masters. On the other hand, the passive elements are components like shared memories, or the IP blocks that act as bus-slaves. Hence, we propose the following abstractions:

> **Task** — an abstraction that represents the implementation of an algorithm for processing or control. For example, tasks could be a software process running bare-metal on a processor, a DMA-capable IP block, a software thread running on an Operating System, or an algorithm implemented directly as a digital hardware component. Tasks can operate concurrently.
>
> **Resource** — an abstraction that represents assets which are utilised by tasks. Resources can include memories, peripherals, and input/output devices or their registers (I/Os). We assume that all resources are memory-mapped.
>
> **Relationships** – describe how tasks and resources are related. For example, we can describe when a task needs to read or write to a resource, or if a task has the capability to access a resource. As resources are memory-mapped, relationships can be represented by memory accesses.

Using these abstractions, we can represent an MPSoC design at various levels of detail. As an illustrative example, consider a four-core, four-resource application, shown in Figure 3.4(a). Here we *graphically* represent four processing cores (P0–P3), four memory-mapped resources (R0–R3), and their functional relationships as a bipartite graph, tasks on the left, and resources on the right. Edges indicate that a task needs to *read* and *write* to the resource, unless explicitly labelled. This

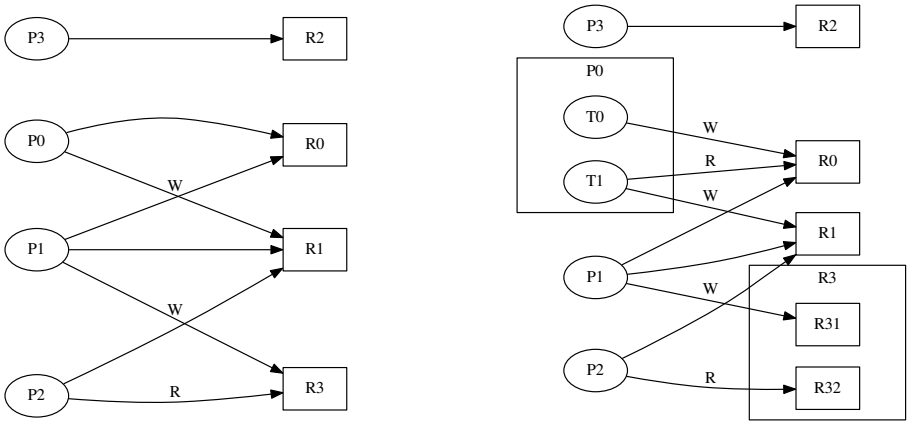
Table 3.2: Task/Resource Relationship Matrix for the four-core example

	R0	R1	R2	R3
P0	RW	W		
P1	RW	RW		W
P2		RW		R
P3			RW	

Task/Resource relationship graph can also be represented as a matrix, as shown in Table 3.2.

If we had more details about the design, for example, that P0 is running two software threads, T0 and T1, and that R3 actually contains two memory-mapped regions, R31 and R32, we can change the Task/Resource relationship graph to provide finer grained information, as shown in Figure 3.4(b).

This graphical representation allows a designer to more easily visualise dependencies between resources and tasks in the application, irrespective of whether they are implemented in hardware or software. By using different levels of abstraction we can also begin to make some informal inferences about potential security risks; for example, P0 is shown to have a read/write relationship with R0, but on refinement, we can see that T0 should only *write*, while T1 only needs to be able to *read* from R0. If P0 is not subject to any access controls, we run the risk that T0 or T1 can be coerced to do more than it should. Similarly, in the case of P1 and P2, they may both be able to affect R31 and R32 if the processors have physical access



(a) Four-core app, processor-level (b) Four-core app, with some refinement

Figure 3.4: Illustrative example of different levels of abstraction in a Task/Resource Relationship Graph

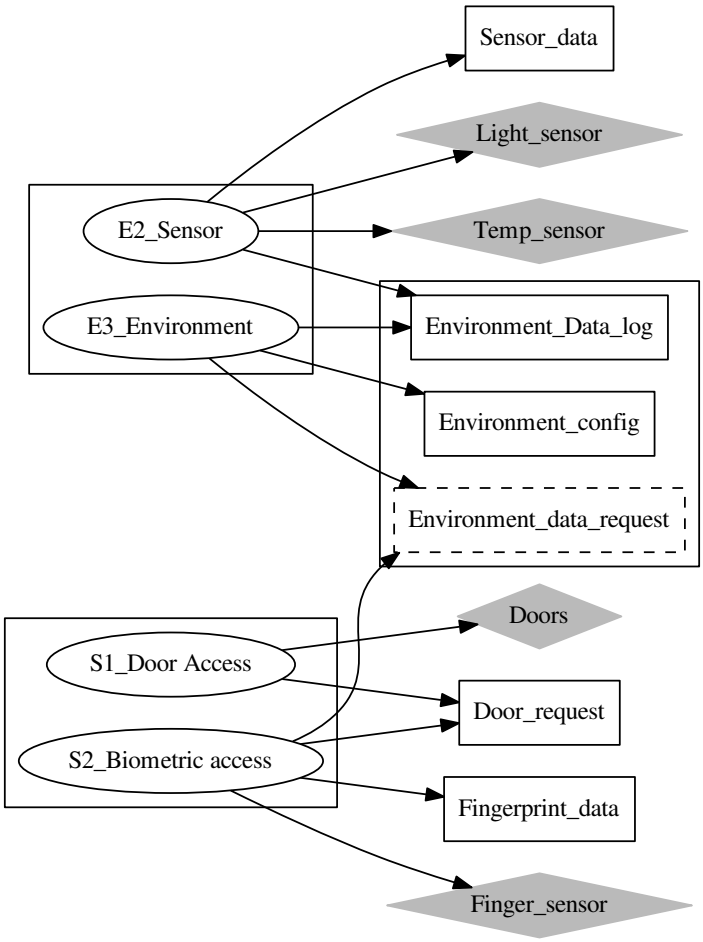
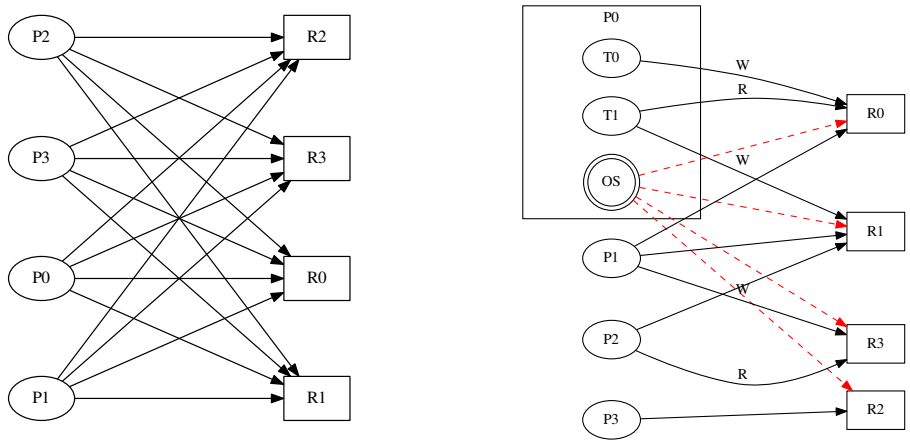


Figure 3.5: A simplified T/R graph

to that memory block. As we will see in later chapters, this abstraction can be used to automate the addition/removal of such capabilities in an MPSoC, particularly as we incrementally introduce security mechanisms. This also provides us with the opportunity to explore the design space, where we might opt for a finer or coarser granularity for setting up protections.

For a less trivial example, let us now revisit the *Hub* scenario presented earlier, and re-cast the inter-task interactions as a Task/Resource relationship graph. A subset of the Hub is illustrated in Figure 3.5. Using different node shapes, we can capture even more information about an MPSoC; in this case, physical resources are represented in diamonds, and memory resources in rectangles. For discussion, we group the Environment-related data into a super-node that represents a single physical memory block.



(a) All to all (physical) access (b) Extra capabilities due to privilege

Figure 3.6: Illustrating the implications of certain architectures

In this example, the *Environment_data_request* resource exists as a sort of "criticality boundary"—it is a resource that is used by two different task groups. Within task groups the resources needed are different from task to task; we would not expect that the Sensor task needs to check to see if there is an environmental control request. While access to *any* part of the environment-related memory block is physically possible, S2 has no real need to access the environment data log, and a more secure design would prohibit purposeless access. In fact, ensuring that tasks can access only the nodes that are present in the Task/Resource relationship graph would better satisfy the Principle of Least Privilege [110].

Because the *task* abstraction allows us to encapsulate any active component in an MPSoC (processing cores, IP blocks, or other such transaction initiators) and the *resource* abstraction represents all memory-mapped components (sensors, actuators, memory, or programmable accelerators) we can apply the security model ideas of §3.2 universally at the task/resource level. Later chapters in this thesis present our proposed techniques for implementing and enforcing access controls at the system-level as a means to improve MPSoC security.

3.3.2 Gauging the Impact of an Attack

Let us consider the threat model once more (§3.1.3), where a compromised task can issue arbitrary memory accesses. This becomes a potential problem if the task has physical access to resources in a system. One way in which we can quantify

the *impact* of a successful incursion is to look at what parts of the system each compromised task can access. The *reach* of a given task is primarily a result of the underlying platform architecture.

Returning to the four-core example, imagine that P0–P3 are connected together using a shared bus, where each core is able to issue a memory request to any part of the system. We can represent this as the Task/Resource graph in Figure 3.6(a)—note how all resources can be directly accessed by all processors. There are many more edges compared to what the application actually needs (as in Figure 3.4(a). MPSoCs designs often feature this situation where our processors can do more than they actually need to; after all, as designers we naturally assume that our designs behave only as we intend, so architectural features like a shared common on-chip interconnection is standard practice as a convenience, and for maximising flexibility.

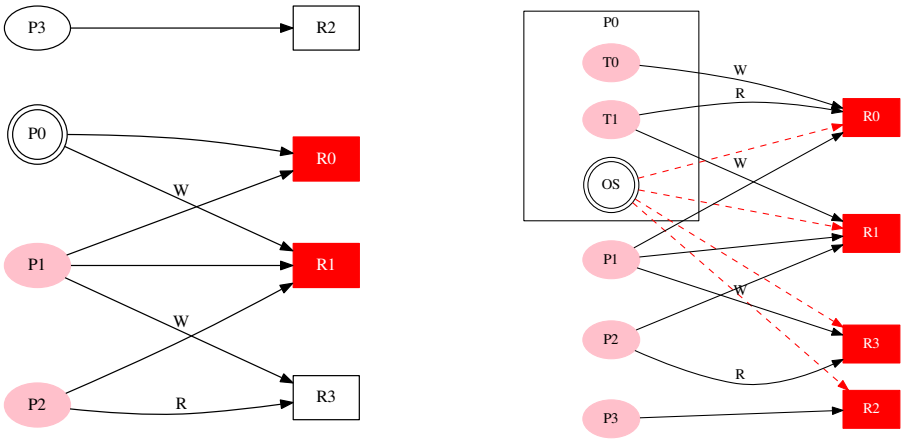
Another issue that our threat model features is that privileged entities, such as OSes, should be considered as similarly at-risk (akin to the other application tasks). For illustrative purposes, let us imagine that the tasks T0 and T1 are managed by an OS, which executes in a privileged mode. While T0 and T1 might be able to have their capabilities restricted by the OS (for example, by means of MPU configuration §2.3.1), the OS has free reign over the system. Figure 3.6(b) illustrates the OS as another task in the Task/Resource graph, with edges connecting it to all the resources in the system.

Using the notion of "task reach", we can more formally gauge the impact of an attack in a process that we call *Impact Analysis*.

For each task in the system, we create a *Task Attack Impact Profile* (TIP), by identifying the set of tasks and resources that represent the scope of what a compromised task can affect.

Initially, before we add any security mechanisms, we assume that a task is able to *write* to any of the resources it has physical access to—all relationships are initially *unrestricted*. Starting with a task, we follow all relationships of that task to construct the *Immediate Impact List*, which contains the set of resources that the task is able to write to, either by design, or as a result of platform-added capabilities.

We then examine how the resources in the Immediate Impact List are used by other tasks to determine which of the other tasks could be affected by a task's compromise. If a task has a read relationship with any of the resources in the Immediate Impact List, we add it to the *Secondary Impact List*, based on the notion that a compromised task can write erroneous data into any resource within its Immediate Impact List, potentially disrupting/manipulating tasks with read-dependencies.



(a) Impact of P0's compromise (b) Impact due to OS compromise

Figure 3.7: Illustrating the potential impact of a compromised task

It is also possible to find further impact levels by going backwards and forwards through the relationships, particularly if the impact of a task in the secondary list can cause errors to be propagated to other resources, thus affecting other tasks with dependencies on those resources.

For example, let us consider the impact of P0's compromise in the four-core example (Figure 3.7(a)). In an ideal case, the designer has implemented only the connections required by the application (perhaps through a custom point-to-point on-chip network). R0 and R1 are in P0's Immediate Impact List (shown as the dark shaded nodes). Because P1 and P2 have read dependencies on resources in the Immediate Impact List, they are potentially affected by the compromise of P0; hence, they are placed in the Secondary Impact List (shown as the light shaded nodes).

Each TIP essentially characterises the reach an attacker has if they were to compromise the associated task, and Impact Analysis can be performed to help designers identify potential paths from which certain assets could be attacked given an architecture configuration. Intuitively, the more shared resources a task uses, the greater the potential impact if compromised. The Impact Analysis algorithm is presented in Algorithm 1, where the input is a Task/Resource Relationship Matrix.

3.3.3 Using the results of Impact Analysis

Having the TIPs for each task can be useful for a designer, especially as a baseline for an unsecured system. It allows the designer to get a feeling for how much inter-

Algorithm 1: Algorithm for Impact Analysis

```
Input: Relationship Matrix rm[][]
Output: Task Impact Profile List tipList[]
for each task (row) in the matrix do
    for each resource(column) in the matrix do
        if rm[task][resource] == one of (R, W, RW, RWd, a, b) then
            tipList[task].addToImmediateList(resource);
        end
    end
    for each r (resource) in the tipList[task].immediateList do
        for each t (task) in the system do
            if rm[t][r] contains ("R") then
                tipList[task].addToSecondaryList(t);
            end
        end
    end
end
```

dependency there is between tasks in the application. By combining other security information with impact analysis (such as probabilistic risk assessment [115] of the compromise of each task, based on factors such as its origin, or its role), the designer could be guided towards parts of the system that require more attention. Resource sharing could be minimised, or specific security mechanisms could be implemented for tasks that have high risk and high impact (such as control flow checking §2.2.2), while other, more trusted tasks, can have such mechanisms omitted to avoid the overhead of added security.

Measuring security is difficult due to its multidimensional nature, and the challenges involved in ascertaining *perfect* knowledge of all parts of a system [94]. Hence, impact analysis provides another useful metric for evaluating whether or not a solution improves security. While a platform might allow all-to-all communication, we can add access controls which essentially remove unintended Task/Resource relationships; in comparing the changes to TIPs, we can coarsely measure an improvement in security. This idea, alongside checking and addressing constraints on the contents of impact lists, is deeply explored in Chapter 5. *Isolation*, as our security foundation, naturally aims to minimise (or at least constrain) the impact of any given attack.

3.4 Summary

Embedded systems can be quite complex, with many components operating in parallel, and a variety of different requirements and constrains. In this chapter, we examined a motivating scenario to provide the context for MPSoCs that have concurrent execution, consolidate multiple functionalities, and ultimately, complex designs, with heterogeneity and design re-use as a feature. We presented a threat model that generously assumes that a compromised task can be fully hijacked by an attacker to issue arbitrary memory accesses, and highlight the dangers that are present with privileged entities in a design. By exploring the multi-domain nature for security requirements in the *Hub* example, we examined some existing access control policies, to come up with a potential foundation for improving MPSoC security. As a means of visualising and handling access controls at different abstraction levels, we presented the Task/Resource abstraction, and showed how this could be used in *Impact Analysis* to measure the potential impact of the various tasks in a given design. In the next chapter, we investigate how we might implement some hardware-supported permissions in a heterogeneous MPSoC, as a response to the need for permanent and temporary permissions, as identified in this chapter.

CHAPTER 4

HW Support for Distributing Access Controls

Having now established an idea for how we can approach security in MPSoCs, in this chapter we present hardware for supporting isolation in an MPSoC. First, we take a more detailed look at the type of mechanisms that we need to implement, in light of the requirement for fixed and dynamic permissions that were identified in Chapter 3. Furthermore, we propose a way to distribute access controls, in a bid to avoid a central trusted authority, thus reducing dependency on an OS, better supporting heterogeneous architectures, and facilitating multiple security domains. We present the Isolation Unit (IU), as a hardware component that implements this security strategy at run time by checking memory transactions, and allowing both local and remote reconfiguration of permissions.

4.1 Overview

Given the trend towards the integration of multiple functionalities for execution on heterogeneous MPSoCs, we need to support designs with many co-existing parallel behaviours. In uniprocessor platforms it is typical to employ a centralised authority for managing security, usually an operating system (OS) [99], or hypervisor. This central authority then manages the security features provided by the execution platform, such as memory protection units (MPUs), or input/output memory management units (IOMMUs) [9], thus placing restrictions on the *other* software and hardware elements in the design. There is an explicit privilege hierarchy, where highly privileged entities (such as the OS) can control access to *all* the resources in the system. As such, successful privilege escalation attacks can cause a significant negative impact.

For improved resilience in the face of security breaches, the central authority-based approach should be reconsidered. In heterogeneous systems it may be challenging or inconvenient to deploy a single OS across the entire platform. Furthermore, some tasks may execute bare-metal for performance or timing reasons, and there may be multiple IP blocks that have DMA capabilities—these system components need to be managed properly to ensure that security restrictions that are defined by the designer are respected. To provide a *system-level* approach, we propose distributing access control across tasks in an MPSoC, where each task manages its own regions of the address space. This is supported by mechanisms for sharing code/libraries, memory regions, and IP blocks with direct memory access capabilities.

In the following sections, we will discuss the security implications and trade-offs of implementing these mechanisms. We discuss issues that need to be considered when adding architectural support for our security approach. As a means of investigating the feasibility of hardware-supported enforcement, we implement the Isolation Unit (IU), which provides a simple scheme for dynamic access reconfiguration at run-time without the need of the OS. We also present an alternate IU variant that employs an alternate bus-based interconnection between IUs as design option. To characterise the performance implications of adopting our security mechanisms into a MPSoC infrastructure, we use a cycle-accurate simulation model of representative multicore scenarios.

4.2 Proposed Mechanisms

4.2.1 Permissions

Continuing from the discussions of the previous chapter (§3.2.2), we now consider permissions for an MPSoC context in more detail.

Our alternative to having a single centralised authority in the system is to have *multiple* decentralised authorities. In other words, we explore the notion of distributing responsibility for managing dynamic permissions in the system. This strategy does imply greater complexity within tasks, but allows better isolation of systems should there be a successful compromise. There is no entity that has complete access to all regions as a default, and designers would need to *explicitly* allow unfettered access. With parallels to capability-based security [25] we enforce the requirement that *all* memory accesses are checked against permissions. Using a

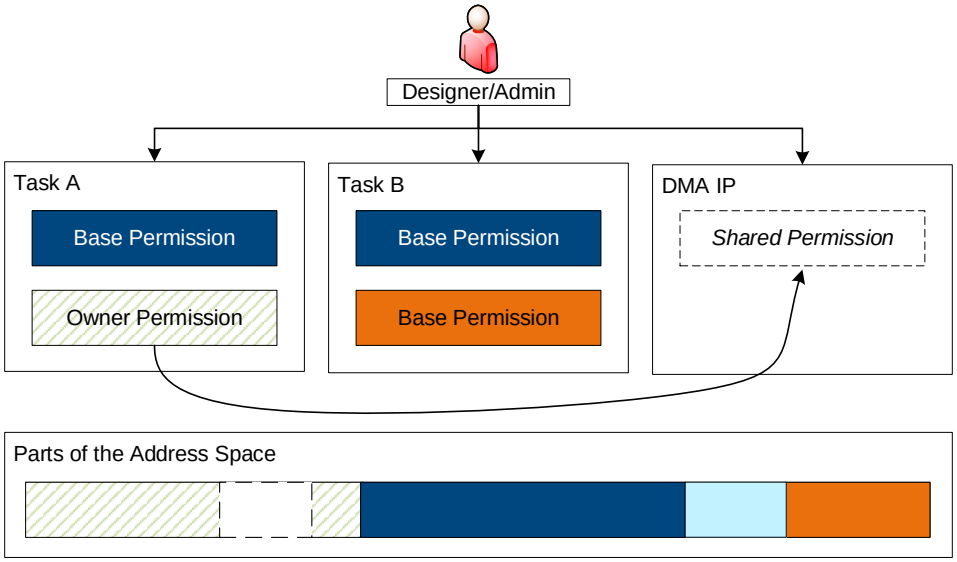


Figure 4.1: Delegating authority with different types of permissions

partitioned global address space we allocate all resources and memories (as everything is memory-mapped, we essentially allocate memory regions), to the tasks in the design. We specify three types of permissions:

> **Base** permissions are statically allocated by the designer, and cannot be transferred to anybody else. They indicate regions that are accessible to a task/IP block.
>
> **Owner** permissions are statically allocated (they can be considered an extension to base permissions), and indicate regions that a task has authority to share. It is not necessary that a task "owns" any regions. Owner permissions should be reserved for regions that are intended to be shared.
>
> **Shared** permissions are dynamic and used to grant temporary access to regions to other tasks; the owner can revoke access at any time.

As shown in Figure 4.1, the designer is the primary trusted authority and is responsible for determining, during the design phase, the regions each task/IP block can access. This carries the caveat that the efficacy of this proposed approach relies on the trustworthiness of the designer's decisions with regards to granularity, and proper delegation of authority. Base and Owner permissions are always active; once the system starts executing , tasks that own regions can share subsets of those regions with other tasks. One implication of this approach is that the entire

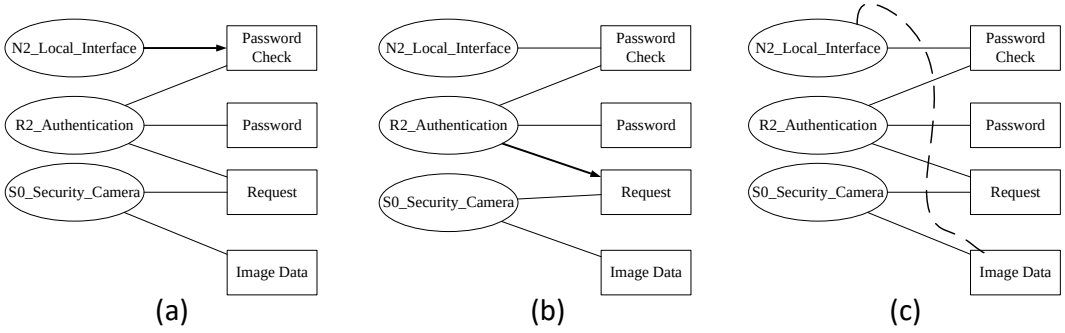


Figure 4.2: Illustrating a temporary relationship

address space (including memory-mapped peripherals) should be allocated in advance. Tasks are completely isolated if there is no interaction between them, i.e. if they do not share resources.

By analysing the task-resource relationships in our system (or designing with these in mind), we can identify the permissions, and nature of those permissions, that align with what tasks require during execution.

For example, consider the scenario illustrated in Figure 4.2, where a user (operating a local interface) wants to get access to the Image Data of a security camera. The process for accessing the data is as follows:

(a) The Local Interface task authenticates by placing a password in to the *Password Check* memory region

(b) The Authentication task checks the password; if it passes, the Authentication task places a request in the *Request* resource to the Security Camera task

(c) The Security Camera task gives access to the Local Interface so it can access *Image Data*

This Task/Resource relationship illustrates where we would use Base permissions to allow fixed, always available inter-task communication (for the *Password Check* and *Request* resource), Owner permissions for tasks that can share resources (between Security Camera and Image Data), and Shared Permissions for temporary access (between Local Interface and Image Data). An alternative scenario might involve access request and permission handover directly between N2 and S0, cutting out the Authentication task—the same fundamental permission types are still used. It is ultimately up to designers as to how "clean" or modular their functional decomposition of the application is.

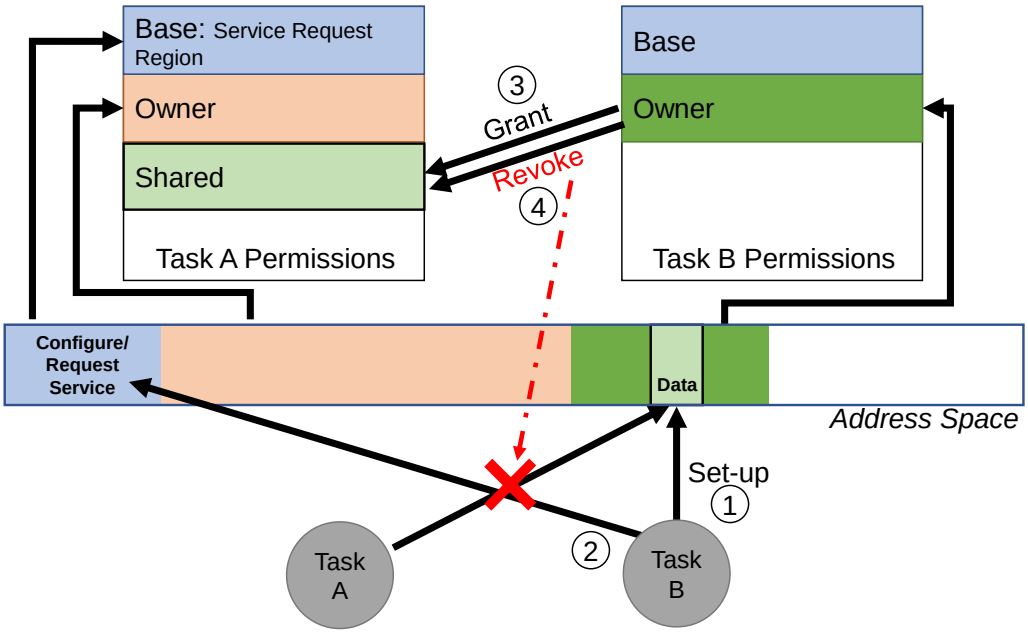


Figure 4.3: An example of temporary region access setup when Task B requests a service from Task A

4.2.2 Implementing sharing

To implement the data, library, and IP sharing identified in the previous chapter (§3.2.2), we use a mixture of the three types of permissions.

Firstly, let us consider sharing IP blocks and peripherals. Usually, they contain a set of memory-mapped registers for configuration. In this security strategy, we group these registers into a region which is collectively referred to as a Service Request Region (SRR). Use of shared code is similar, in the sense that the calling task can "request" a certain function to be executed—hence, we can come up with a generalised mechanism for these shared *processing resources*.

By examining the operation of each task, we can identify where a task requires the use of a shared *processing resource*. Essentially, these tasks are *consumers*, and we can provide a Base permission for each consumer which allows access to the SSR of each required *processing resource*. Each *processing resource* will itself not need any Base permissions, unless it has its own (possibly private) memory-mapped working memory.

Figure 4.3 indicates an example with two tasks, Task B requests a service from another task, Task A. Task B first acquires control of that resource (such as by acquiring a mutex lock). Then:

1. Task B sets up a region for the data the *processing resource* will process, as well as a region for storing the results of that processing
2. Task B writes the request to the SRR
3. Task B also sets up a Shared permission for the resource, which grants temporary access to the shared region
4. Upon completion, Task B revokes the Shared permission

A variation of this scheme can be used for simple data sharing. A task shares data by granting another task access to a region using a Shared permission if the shared memory region is temporary, or if the location of the region changes during execution. To facilitate run-time requests for data sharing, tasks can have base permissions that are allocated at design time for fixed, always-accessible communication buffers in shared memory.

4.2.3 Security Implications and Limitations

The approach we present here allows us to describe and enforce specific relationships between tasks and resources. Through careful design, we can isolate systems that do not interact. Furthermore, this strategy avoids the implications of attacks that aim for privilege escalation; as authority for access controls is distributed throughout the system we can *limit* the impact of an attack on the *whole* system.

However, like all approaches, this strategy has several limitations. Firstly, we introduce additional complexity into tasks where they are expected to dynamically share access to regions in memory. Secondly, this approach relies heavily on the quality of the design; the designer must properly partition the design, and ascertain the appropriate level at which to implement and enforce the necessary permissions.

It is also possible that the designer is using several off-the-shelf "black-box" components, either software or hardware, that are not able to be tightly integrated with the security approach. While these black-boxes might not be able to leverage the dynamic permission scheme directly, the designer can use this approach

to restrict their capabilities, thus incorporating these potentially less-trusted entities while still better protecting trusted tasks. In Chapter 6 we explore further techniques to deal with security un-aware components.

Furthermore, if we can detect violations, our dynamic permissions allow us to respond with strategies like quarantine of compromised tasks, at least in terms of revoking shared permissions, thus allowing for continued (although possibly degraded) system operation. As one might expect, the efficacy of this security model depends heavily on the underlying implementation of permission enforcement. We discuss this and other hardware issues in the context of MPSoCs in the following subsection.

4.2.4 Architectural Support: Issues and Trade-offs

There are several considerations that need to be taken into account when implementing our security model. A software-only approach is inadequate, particularly when the underlying hardware architecture allows violation of the task-resource relationships that we are trying to enforce (i.e. it *physically* allows access). Hence, we propose the use of hardware to guarantee that all parts of the overall system are subject to the same security enforcement. There are three key elements that need to be implemented, namely: permission enforcement and storage, task identification, and permission management/reconfiguration. There is a trade-off to be made between security and the cost of adding additional hardware into our designs.

Every memory transaction in the system needs to be checked against a list of permissions. We need to consider two aspects: where permissions are stored, and how we check every transaction. Firstly, we need to consider the number of permissions in the system, and how many of these permissions are "active" at any point in time. This is largely a function of the complexity of tasks in the system, and how many distinct regions they need access to.

To reduce the amount of storage required at local permission *checkpoints*, we could adopt a central permission repository. However, this introduces overheads where permissions need to be "fetched" at moments like context switching. Furthermore, the checking of every transaction results in additional resource usage and performance overheads. These could be reduced however; the work in [42], suggests that the performance cost can be reduced by checking transactions on cache misses, if caches are available, thus reducing checking frequency. There is however the possibility for information leakage where tasks that are co-resident on a processor access data of other tasks that have remained resident in the cache.

Further consideration might also be required if tasks are able to migrate across processors, but we do not consider this case here.

There is a close coupling between task identification and permission enforcement; we need to correctly and irrefutably identify the currently running task, particularly to ensure that tasks do not circumvent isolation boundaries by disguising themselves as other tasks. Illegitimate accesses need to be blocked, and identified for handling. Traditionally, tasks are identified by the OS, making use of task ID registers that are available in several processor architectures. However, when software is generally untrustworthy, we cannot always rely on software-based identification. Depending on the granularity of our permissions, it may be sufficient simply designing and enforcing permissions at the level of processors in the MPSoC. This coarse-grained identification is particularly useful in the case that only a single task is running on the processor, or where all the tasks on a processor have identical security requirements.

However, in our heterogeneous MPSoC it may be more appropriate to identify tasks *within* processors, with permissions enforced at a higher level of granularity. Once again, there is a trade-off between security and increasing the complexity of hardware. One option is to adopt a hardware-based task loader/scheduler; the hardware performs task switching by directly controlling processor state and internal registers. While immune to software-based exploits, it incurs the cost of invasive hardware modifications.

Program Counter Based Access Control (PCBAC) has recently been proposed by several authors as a means for identifying tasks and enforcing permissions [27, 63, 120]. Task identities are linked to the addresses of the program instructions. As the program counter is not typically *directly* manipulated (aside from jump and branch instructions), we can have a fairly high level of trust in the task identification. Should a task be compromised and attempt to use instructions from program memory associated with another task, the task ID would also change, resulting in a different set of permissions, thus making a successful attack more complicated. Shared code would need to be marked as a task; our approach of setting up shared permissions to temporarily extended access at run-time nicely complements this approach.

Memory protection is often managed by software making use of special processor instructions, such the approach presented in [136]. If we want to integrate different types of processing elements while allowing for dynamic permission changes across the system on chip, a more generalised approach is required, especially if

the processor or IP does not have built-in support for different privilege modes. If a satisfactory mechanism for identifying tasks is employed, permission management could be performed simply through configuration of certain "permission management" registers, mapped into the address space of each task. While processors and IPs may be heterogeneous, they will all be able to read and write using some sort of memory interface. Hardware would then need to ensure that the correct permissions are modified. All tasks would essentially be in "user space", and can grant and revoke permissions at run-time as required. This also has the added benefit of avoiding the costly overhead that is associated with performing system calls to request an OS handle permission changes.

4.3 The Isolation Unit

We explore our proposed security strategy by designing the Isolation Unit (IU), a hardware block that checks permissions and allows permissions to be reconfigured at run-time. The idea is that IUs are customised and added at the interface between processor or IP and the interconnect, based on the needs of the given application.

Permissions are classified as either local or remote from the perspective of each task. If a task is granting/revoking access to a region to another task on the same processor, the task will modify a locally stored permission; if the task or IP is elsewhere in the MPSoC, a remote permission is modified. Using the task-resource relationships in the application, we can customise the capabilities of each IU. The number and mixture of statically allocated, local, and remote permissions affect the complexity of each IU.

We present two IU design approaches/architectures, which we call the Original and Alternate design. Each IU is then modified to produce one of four different variants:

1. The *base variant*, an IU that enforces base permissions only—for use with processors that have no dynamic relationships between tasks. This is essentially a Memory Protection Unit that has been configured at design time, and synthesised

2. The *base+local variant*, an IU that enforces base permissions and can support local reconfiguration—for use with processors that have several local tasks that have shared permissions

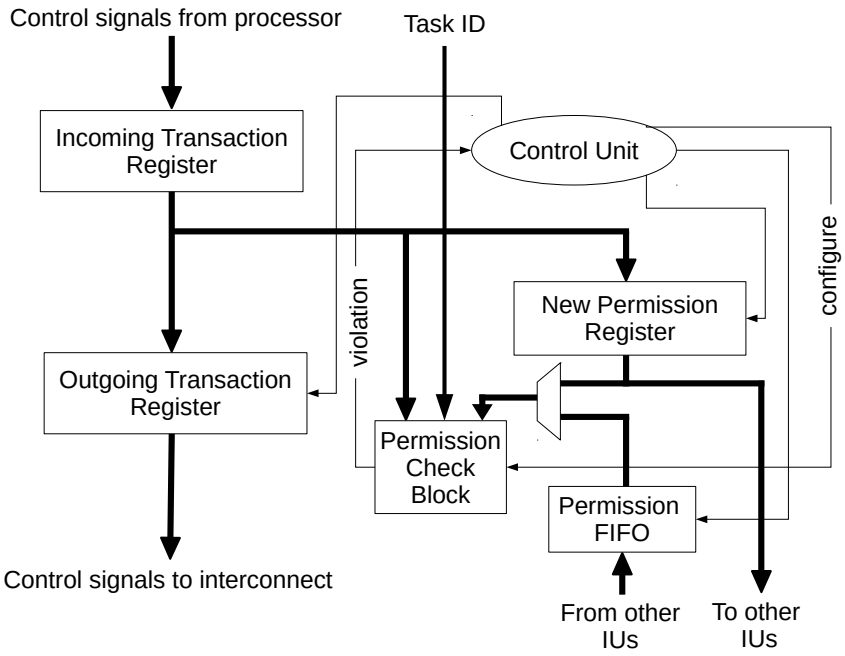


Figure 4.4: Simplified view of the original IU internal design

3. The *complete variant*, an IU that enforces base permissions and can support local and remote reconfiguration—for use with processors which have several tasks that have shared permissions, including with "remote" tasks and IPs

4. The *remote-only variant*, an IU that enforces only shared permissions—for use with IPs that receive permissions only. This variant could be used to restrict the domain of "black-box" IPs

4.3.1 Original IU

Figure 4.4 shows a simplified view of the original IU that supports both local and remote reconfiguration. All original IU-variants contain a register that holds the incoming transaction from the processor/IP block, a register that holds the transaction that is released to the interconnect, as well as a Permission Check Block. Registers for all permissions that apply to the tasks on the local processor are stored in the Permission Check Block. Local shared permissions are all initially disabled. When the IU detects an incoming transaction (typically, if a read or write signal is asserted), the target address is checked against all permissions in parallel (simultaneously). If the address falls within any of the regions specified within the permis-

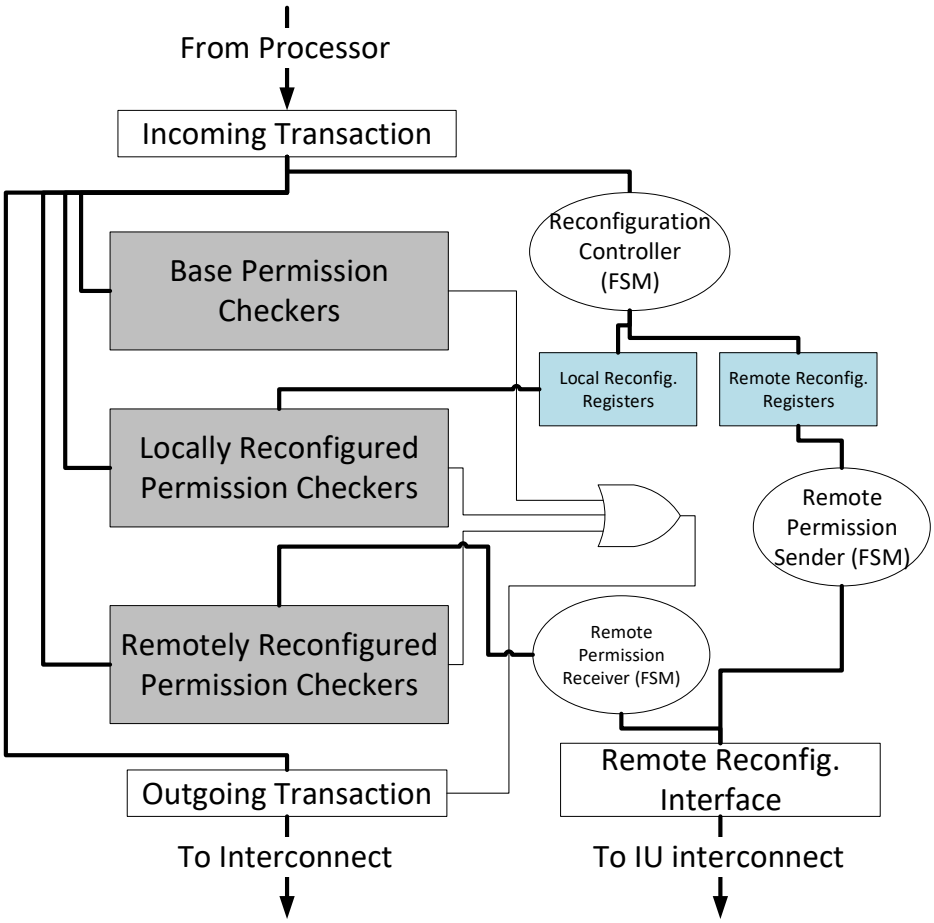


Figure 4.5: Simplified view of the alternate IU internal design

sions that correspond to the current task ID from the processor, alongside appropriate read or write access, the transaction is forwarded to the outgoing transaction register in the following clock cycle. The Control Unit manages the modification of permissions. FIFO buffers are used to store new permissions that are coming in from remote IUs. If remote reconfiguration is not needed, the Permission FIFO is removed. If the IU does not need to be reconfigured at all, we remove the Control Unit completely.

The IU is memory-mapped into the address space of the local processor for reconfiguration; when a task writes to the IU's addresses, the New Permission Register stores the base address, end address, and read and/or write access attributes.

Table 4.1: Permission Format for the Original IU

Field	Region Start	Region End	Owner PID	Owner TID	Task ID	R	W	V	O
No. of bits	32	32	2	2	2	1	1	1	1

Table 4.2: Permission Format for the Alternate IU (base permissions in square brackets)

Field	Region Start	Region End	Task ID	R	W	V / [O]
No. of bits	32	32	2 [4]	1	1	1

4.3.2 Alternate IU

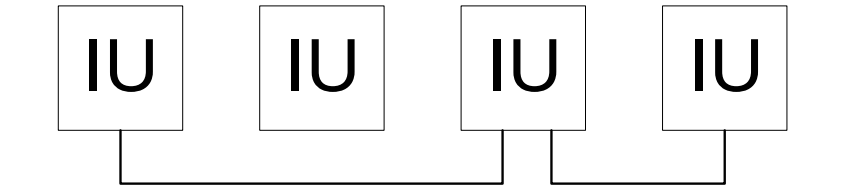
Figure 4.5 shows an alternative design, where the Permission Check Block is split into distinct blocks for three different types of permissions, base permissions (which may contain the Owner attribute), locally reconfigured permissions, and remotely reconfigured permissions, as required. Instead of register-based storage of permissions, permissions are stored in on-chip memory. As with the original design, the incoming transaction is registered, and the address, current task ID, and read/write control signals are passed to the permission checkers, which will check all permissions simultaneously. Control is divided into several parallel Control Units, one for managing the local reconfiguration process (Reconfigure Controller), one for managing permissions that arrive from other IUs (Remote Permission Receiver), and one for sending permissions to other IUs (Remote Permission Sender).

4.3.3 Permissions

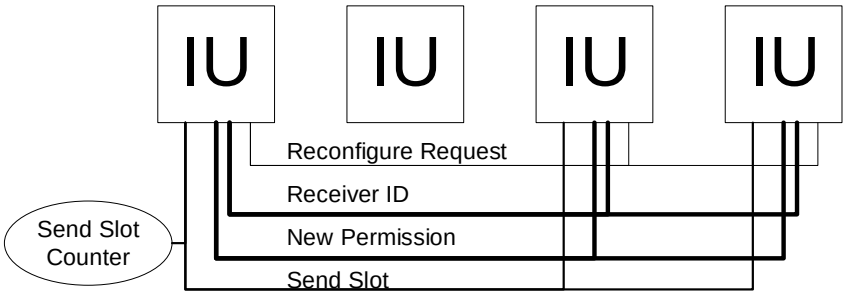
The permission formats can be found in Table 4.1 for the original IU, and Table 4.2 for the alternate IU.

In the original IU, each permission consists of 74 bits: 32 bits used for the upper and lower region boundaries, 4 bits for the owner of the region (2 bits for processor (PID), 2 bits for the task ID (TID), 2 bits for the task ID of the task that has received the permission, and 1 bit each for whether the permission is for read (R) and/or write (W) access, whether the task owns (O) the region, and whether the permission is valid (V).

The format used in the alternate IU requires fewer bits, 69 for shared permissions, and 71 for base permissions. In base permissions, four separate bits are used to identify four tasks—bit 0 represents Task 0 through to bit 3 representing Task 3 in the local processor. As all base permissions are valid throughout execution, a bit



(a) Point to Point



(b) Bus-based with Slot Counter control

Figure 4.6: IU interconnection variants

is used to indicate Owner status in lieu of Valid status. The Owner Processor and Owner Task ID fields are omitted in the alternate IU in lieu of additional logic in the Remote Permission Receiver for ensuring the correct permission is modified at the recipient.

4.3.4 Connecting IUs

If a task remotely configures permissions for other tasks, or IP blocks, the corresponding IUs need to be connected. We separate the IU interconnection from the general system interconnect, to prevent interference with task-resource traffic. IUs are assigned a unique ID number. The original IU is designed with point-to-point connections between IUs, so if a task shares regions with two different remote tasks, the local IU would have two point-to-point connections (Figure 4.6(a)). These point-to-point connections feature a permission bus and a request signal. On the receiving end, a FIFO buffer stores the incoming permission. The original IU control unit checks to see if a reconfiguration operation is currently underway, and if not, loads the new permission into the Permission Check block. The IU checks a different IU-pair buffer each cycle to ensure that all new permissions are eventually added.

The alternate IU implementation instead uses a time-division multiplexed bus for remote reconfigurations, which we call the inter-IU bus, as shown in Figure

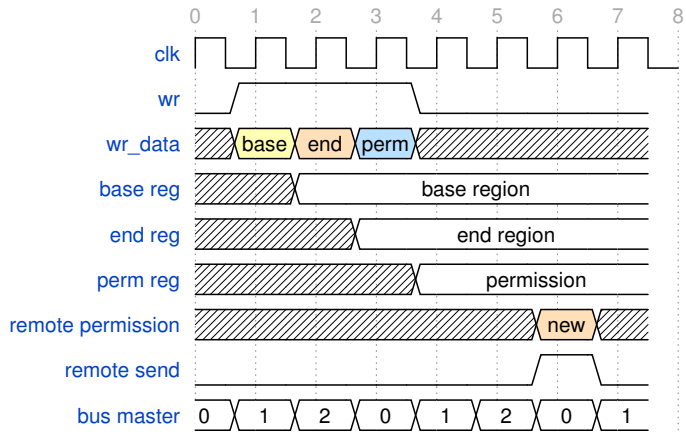


Figure 4.7: Timing Diagram for Remote Permission Reconfiguration

4.6(b). The IU reconfiguration bus contains four elements, the permission bus, a request line, the send slot count bus, and a receiver ID bus. A Global Send Slot counter is used to control which IU can send in any given clock cycle, ensuring that all tasks that need to reconfigure remote permissions can eventually do so.

When a remote permission needs to be modified, the IU waits until its slot appears on the slot count bus; it now has control of the bus, and so places the remote permission on the permission bus, the ID of the IU to receive the new permission, and asserts the request line. On the receiving end, the IU waits until it identifies its ID on the receiver ID bus. When this occurs, the remote permission is sent to the appropriate remote permission block. This sequence is represented in Figure 4.7 as a timing diagram.

4.3.5 IU Operation

The IU has two operating modes; the Normal mode, which performs permission checking as tasks issue memory transactions, and Reconfiguration mode, which is used for managing permissions.

Normal mode operation In each clock cycle memory accesses from the local processor are captured in the Incoming Transaction Register; the permission checking occurs, and the transaction is released to the system interconnect if no violation is detected. This operation adds a fixed single cycle delay to all transactions. In the event a violation is detected (when the access is unauthorised, or the current task is invalid) the offensive transaction is blocked. A violation interrupt signal is asserted, which can be used to interrupt the processor or to notify another part of the system.

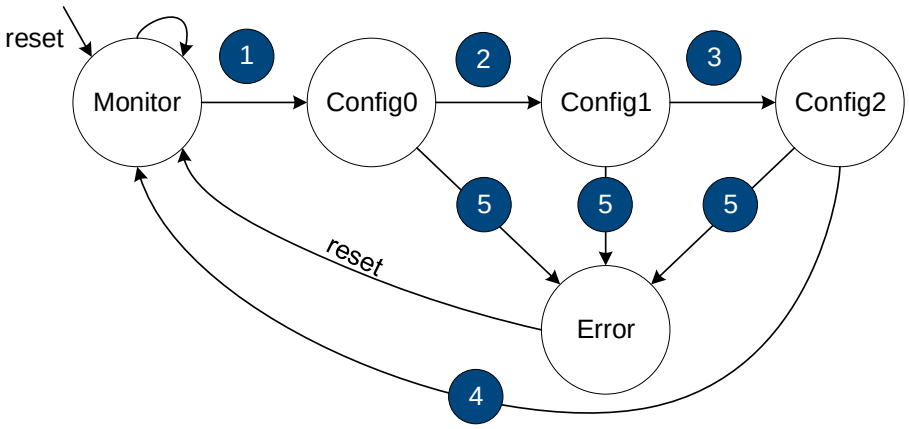


Figure 4.8: IU reconfiguration state machine

Reconfiguration mode operation Reconfiguration mode is triggered by writing to a memory address reserved for the IU. The Control Unit/Reconfigure controller then transitions into the first of three configuration states, as shown in Figure 4.8. Tasks reconfigure permissions by performing a series of successive memory writes to the IU's reserved addresses. The Control Unit/Reconfigure Controller transitions to successive Configuration states, where it moves the permission parameters written by the task into the appropriate reconfiguration registers, loading the permission into the permission check block in the final state. If any of the reconfiguration steps fail, the IU is moved into the Error state.

Reconfiguration procedure (for the Original IU):

Step 1: The task writes the base address of the shared region to the New Permission Register; the control unit redirects the write data into the Permission Check block to check that the task is sharing a region that it owns

Step 2: The task then writes the end address of the shared region to the second reconfiguration register

Step 3: To complete reconfiguration the task writes a word with the target task ID and desired permissions to the final reconfiguration address.

Step 4: The permission is then updated (if local), or placed on the correct IU-pair connection (if remote)

Step 5: In the event that the base or end address of the shared region is not owned by the task, or the reconfiguration is interrupted, the IU goes into an error state, preventing all memory accesses until a reset signal is asserted.

Table 4.3: IU Synthesis Results

Variant	Supported no. of tasks	Permission Configuration (L—local, R—remote)	Total no. of Permissions	Analysis & Synthesis (LE use)		Theoretical Max Freq. from Slow 85℃ Model (MHz)	
				Original	Alternate	Original	Alternate
Base	1	4 Base Permissions	4	237	473	196	833
Base	2	8 Base Permissions	8	262	747	181	582
Base	3	12 Base Permissions	12	275	1020	159	524
Base	4	16 Base Permissions	16	295	1293	171	680
Base+local	2	6 Base, 2 Shared (L)	8	562	894	118	117
Base+local	3	6 Base, 6 Shared (L)	12	700	1185	117	110
Complete	2	2 Base, 2 Shared (L), 4 Shared (R)	8	1361	1187	106	119
Remote-only	1	4 Shared (R)	4	1280	548	134	114

Reconfiguration for the alternate architecture follows the same procedure, with the following internal differences:

- In config0 and config1 states, the Reconfigure Controller redirects the write data into only the Base Permission Checkers
- In Step 4, if the permission update is for a remote target, the new permission is stored in a remote reconfiguration register and an internal send signal is asserted.

The Remote Permission Sender controller then monitors the Remote Reconfiguration interface for the IU's sending slot. In both the original and alternate IU, reconfiguration takes three cycles, with the local permission updated in the fourth cycle. Remote permissions take longer to become updated. In the Original IU design, a cycle is used to transfer the permission from one IU to another. The destination IU then takes a cycle to place this new permission into its new permission register before it becomes active. Further delays are incurred if the IU is already undergoing reconfiguration, or when a permission from a different IU-pair is being added causing a delay until the correct buffer is accessed. Similarly, the Alternate IU design has several cycle delays where the IU has to wait for its slot on the inter-IU bus, but does not have delays if the receiving IU is being reconfigured locally.

4.3.6 Synthesis

We synthesised several IU variants and permission configurations for both the Original and Alternate architecture, targeting the Altera Cyclone IV FPGA (EP4CE115F29C7). Altera Quartus II v15.1 was used. The total logic element use and theoretical maximum frequency (from the TimeQuest Timing Analysis tool) are presented in Table 4.3. In general, the resource usage increases as more tasks

and more permissions are supported; addition of the control logic for local reconfiguration adds several hundred LEs for the same total number of permissions. In the Original design, remote management adds a significant LE-usage cost, mainly due to the added FIFO (36% of the resource usage comes from the addition of this remote connection logic). The alternate design is instead significantly less costly as it forgoes FIFOs, as can be seen in the results for the remote-only IU.

4.3.7 Design Trade-offs

The Original IU design has several limitations:

- The point-to-point interconnection scheme used to connect IUs where tasks have remote shared permissions requires FIFO buffers, and has issues with scalability. As more IUs are interconnected, the complexity of the interconnect, and thus, the entire system, increases significantly
- Local and remote reconfigurations are not able to happen simultaneously, as the Control Unit can only deal with one reconfiguration process at a time, as there is only a single permission check block
- Base permissions, which are used to represent static, always accessible shared regions, are required for every task that shares that region; if four tasks share a region, each of the four tasks needs to have a corresponding base permission

The Alternate IU design addresses these limitations as follows:

- The bus-based implementation is more scalable, especially in the case where there is a lot of shared permissions between a group of tasks/processors. One drawback of using a bus is that IU pairs that do not need to communicate are still connected; to avoid errant tasks attempting to grant or modify permissions to IUs that should not receive them, we add checks in the Remote Permission Receiver logic so that IUs will ignore any reconfiguration requests from senders it is not configured to receive from.
- By separating out the control logic for reconfiguring remote permissions from the local reconfiguration logic, and by splitting up the permission check blocks, the Alternate IU can handle both local and remote reconfiguration at the same time

- Base permissions are also able to be shared by multiple tasks, so redundant base permissions are not required

However, as can be seen from the synthesis results, the Alternate IU design carries the cost of increased resource usage in general, compared to the Original IU, and as such, designers will need to perform a design trade-off when choosing which IU design, variant, and permission configuration to adopt.

The Original IU, with its register-based implementation, can be further optimised and customised depending on the base permissions that are defined; for example, the comparison logic can be simplified, allowing storage of less than the full number of bits per permission, depending on the permission. However, use of embedded memory blocks results in the base-variant Alternate IUs having a higher theoretical maximum frequency, which could be useful for faster designs.

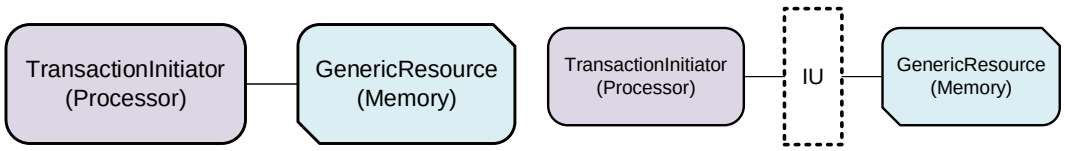
Point-to-point connections between IUs are useful if there are multiple unique task-resource pairs across the system. On the other hand, if multiple tasks share the same IP block, or frequently temporarily share regions with each other, the inter-IU bus approach may be more appropriate, in which case the remote-only variant of the Alternate IU design could be quite efficient. Additionally, if there are several distinct sub-systems, a mixture of Original and Alternate IU designs may be useful.

Ultimately, IUs should be customised for a specific application. By incorporating only the capabilities and permissions required for a specific set of task and task-resource relationships we can introduce better security into a heterogeneous MPSoC with relatively little additional cost.

In general terms, we can further optimise IUs and adjust the granularity of protection regions, checking only the upper bits of memory transactions to reduce the complexity of permission checking and the size of permission lists. Furthermore, permission checks could be pipelined to reduce the cost of comparing the incoming transaction and the stored permissions.

4.4 Experimental Characterisation and Discussion

To explore the performance impact of our security mechanisms in the context of an MPSoC design, we developed cycle-accurate simulation models in SystemC for the IU, the different IU interconnection schemes, and a NoC architecture with associated network interfaces. To model processors/tasks, we designed a Transaction-



(a) Scenario without IU (b) Scenario with IU

Figure 4.9: Single core experiment set-up

Initiator module which issues memory read/write transactions after loading a list of transactions from a file. Each transaction is tagged with the ID of the task that is sending the transaction. TransactionInitiator modules perform blocking reads and non-blocking writes, and can also issue requests for multiple data responses. We model resources (memories, accelerators, and I/O) generally as GenericResource modules, which consume read and write transactions. GenericResource modules issue responses to read requests, and can also emulate DMA-capable IP blocks by generating memory reads or writes upon receipt of a DMA request from a TransactionInitiator. We instantiate these various modules in several different configurations to develop functional simulation models of an MPSoC.

For the interconnect, we model a multi-stage interconnection (MIN) network as presented in [105]. The MIN network is comprised of simple two-port cross-bar switches that are set in either straight or cross mode. A global controller is used to control the mode of the switches in the network. The global controller is configured to apply a time-division multiple access (TDMA) scheme in the network by generating a repeated sequence of control values to allow all cores in the MPSoC to communicate to all other cores exactly once per TDMA round. We use this TDMA-MIN interconnect architecture as the infrastructure for models of selected representative subsets of the Hub motivating scenario, given that TDMA-MIN is time predictable, and appropriate for mixed-critical systems (for example, the Fire Manager task is critical).

First, we characterise the performance overhead of the fundamental operations of our security approach: permission checking, and local IU reconfiguration. Initially, we consider a single processor connected to a single resource, without an IU as shown in Figure 4.9(a).

In this profiling, we use a synthetic workload to represent an extreme case where a task performs only multiple sequential memory accesses. This is because our security approach focuses on memory protection. In more practical cases, the relative number of memory transactions compared to computational operations could be less. The TransactionInitiator is configured to issue 200 memory requests,

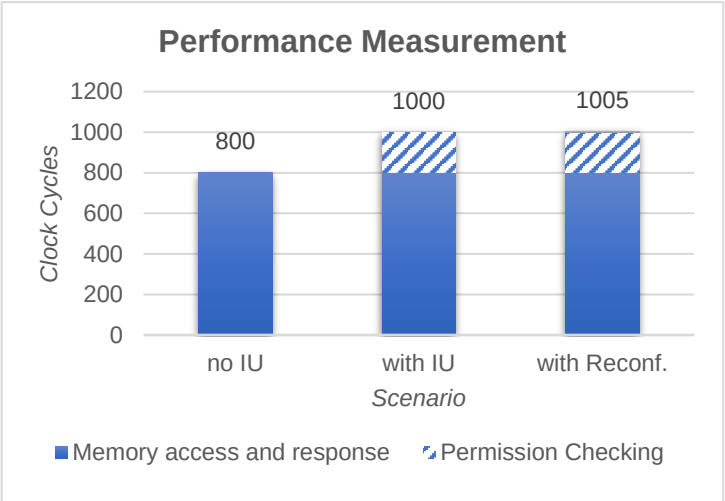


Figure 4.10: Performance penalty of fundamental operations

and the number of clock cycles taken to issue transactions and receive responses is measured. We then add an IU, as shown in Figure 4.9(b), and measure the time taken to complete the accesses with the added security. The IU is configured with a base permission granting access to the GenericResource module.

We also model the scenario where 100 memory accesses are first performed by one task, which then configures a shared permission for a shared region that a second task uses as the target of its 100 memory accesses. The IU is configured with an Owner permission for the first task, and no base permissions for the second task. The results are shown in Figure 4.10, and indicate that memory checking adds an overhead of about 20% in this case. This arises from the addition of a single cycle delay to every memory access for permission checking. In practice, the exact percentage overhead from memory checking will vary depending on the characteristics of the specific application. Factors that will affect overhead include the ratio of memory access instructions to other processor instructions, as well as characteristics like interconnection and memory latency.

The cycle cost of modifying a shared permission adds only a further 0.625%. To simulate an A1-type attack, where a task has been compromised, we populated the transaction file used by the TransactionInitiator with transactions that access arbitrary regions of the GenericResource—as expected, the IU successfully blocks these malicious access attempts.

We then explore a more complex setup, with two processors, one DMA-capable IP, and a shared memory, interconnected using a TDMA-MIN configuration for 4 cores. This is a simulated subset of the Hub (as discussed in §3.1, featuring the

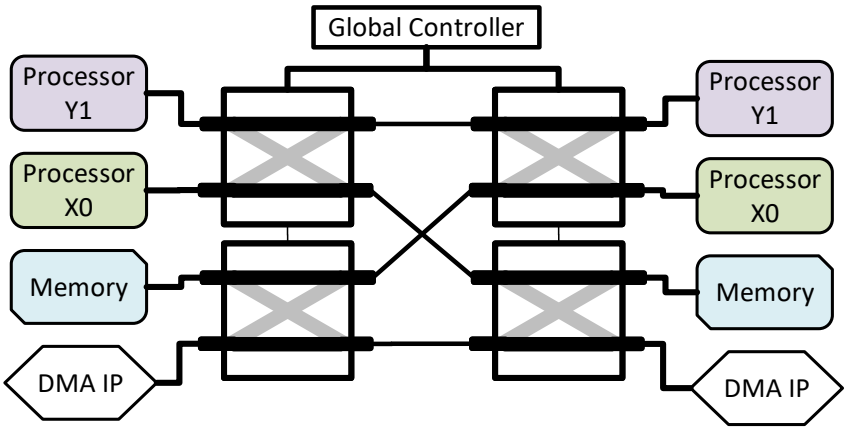


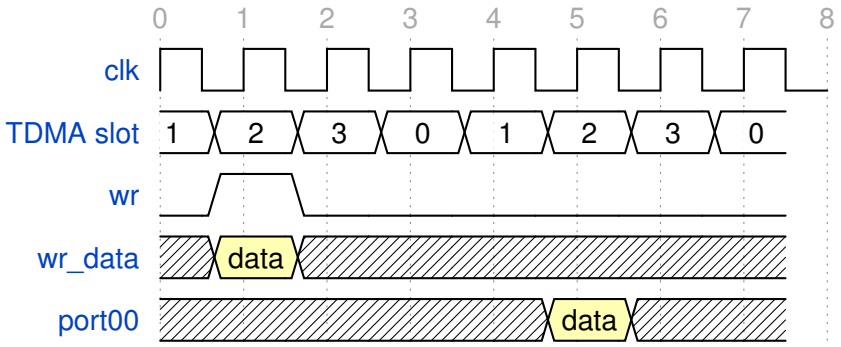
Figure 4.11: A four-core subset of the Hub (§3.1), without IUs

Environment and Entertainment processors, as well as an accelerator, as shown in Figure 4.11. The memory is modelled with four regions: M_0, M_1, M_2 and M_3. M_0, M_2, and M_3 are regions owned by X0, and M_1 is owned by Y1. In this simulation, we assume that X0 and Y1 execute only a single task. The cycle timing for transactions sent using the TDMA-MIN NoC with and without IUs can be seen in Figure 4.12.

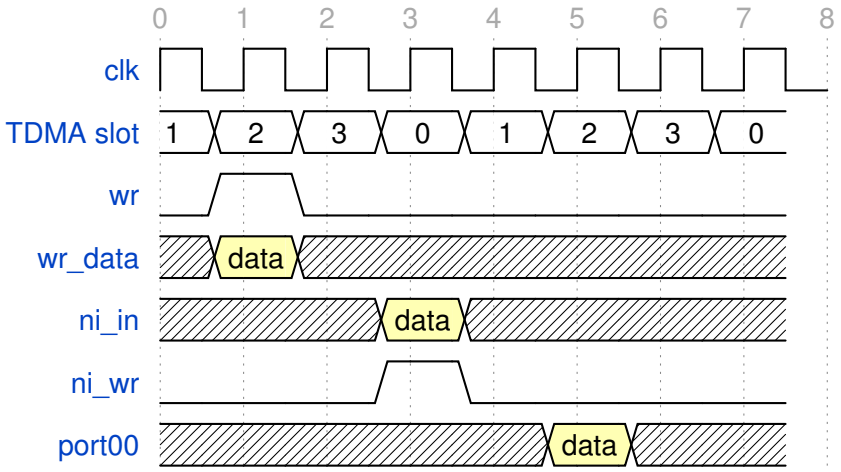
To simulate the execution of the entertainment task, we load the Transaction-Initiator that models X0 with a synthetic transaction list consisting of randomly dispatched reads and writes to the M_0 region of the shared memory, as well as a configuration request of the DMA IP to perform its own independent read and write of regions M_2 and M_3 in the shared memory. We measure the clock cycles required for X0 to complete 400 memory accesses, as well as the time required for the DMA IP to complete its read and write operations on behalf of X0. 2184 clock cycles are required.

We then add IUs to the processors to prevent Y1 and X0 from accessing each other's regions in the shared memory, as well as the configuration where we add an additional IU to the DMA IP.

In the case with two added IUs, 2188 clock cycles are required to complete the memory accesses, which represents an additional 0.18% overhead on the performance without any IUs at all. However, while this configuration can prevent A1-type attacks (i.e. we can set base permissions such that tasks running on X0 are unable to access M_1 directly), the MPSoC is still susceptible to A2-type attacks, where the DMA IP could be configured to access M_1. This issue is addressed by adding a remote-only IU to the DMA IP; if X0 wants the DMA IP to read from M_2 and write to M_3, X0 must first use its IU to configure two remote permissions, one



(a) Delay between sending and delivery of transactions



(b) Delays from IU "hidden" by NoC operation—notice the delay caused by the IU check between the core write transaction and its release to the NI

Figure 4.12: Timing Diagrams for IUs in the TDMA-MIN NoC

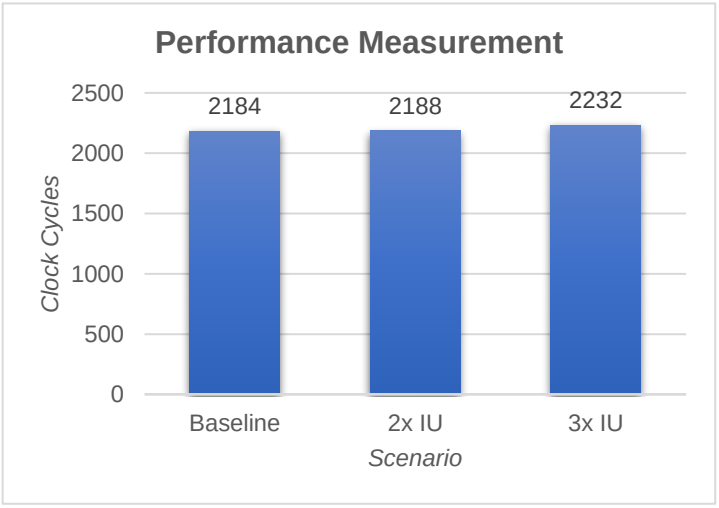


Figure 4.13: Performance penalty of multiple IUs

to allow the DMA IP to read from M_2, and one to allow the DMA IP to write to M_3. Because of the extra reconfiguration commands, and permission checking of the DMA IP transactions, 2232 clock cycles are required (2.2% additional overhead) for both the point-to-point and TDM-based IU interconnect schemes. These results are presented in Figure 4.13. With the added IU, a successful A2-type attack could not be triggered.

These results indicate that performance penalties depend on the application and implementation of the MPSoC architecture. In the simulation of the Hub subset, most of the memory checking cost is absorbed by the latency of the interconnection. Similarly, in a simulation of the full Hub MPSoC (Figure 3.2), with a random workload on each core, the addition of IUs for memory checking added only 1.2% additional clock cycle overhead.

Finally, for comparison purposes we model a centralised approach based on a privilege entity (e.g. an OS), by configuring our IU models to allow a compromised task to modify/configure all the IUs in the system (to model malicious modification of MPUs as a result of privilege escalation); as a result, task accesses are able to be disrupted, and the compromised task has full reign over all resources. This is in stark contrast to our proposed approach, where the capabilities of tasks can be limited by the immutable nature of the base permissions.

4.5 Related Works

To our best knowledge this work is the first to consider a decentralised approach to security in a heterogeneous multiprocessor system.

For single processor systems TrustLite [63] proposes the use of a Memory Protection Unit (MPU) to check all memory accesses; the MPU is extended to be "execution-aware" by directly associating executable code regions with memory access permissions, and identifying the currently executing "trustlet" by examining the instruction pointer provided by the CPU. We use this technique for task identification. This approach is subsequently extended in TyTAN [13] to allow dynamic loading of tasks, and a special driver to manage memory protection regions. Comparatively, these approaches provide higher protection granularity compared to the frequently used ARM TrustZone [75], which offers hardware supported isolation for two "worlds" and a reliance on software for managing access control in each world. There is also extension of virtual memory management ideas specifically for peripherals through OS-managed use of IOMMUs [9], although these incur

overheads for virtual-to-physical address translation, and require storage for page tables.

An alternative approach to isolation uses cryptography; in [126], tasks are isolated by encrypting tasks with different keys, and tagging memory such that a task can only work with data it has created itself, or taken from the insecure part of the system. As expected with cryptography-based solutions, there is a large performance overhead. Another isolation strategy involves generation of schedules for temporal isolation [7], where a central loader dispatches tasks depending on the generated security policy.

In a similar bid to remove reliance on "trusted" software, the work presented in [58] illustrates an architecture for cloud infrastructure, whereby virtual machines are loaded and managed by hardware components. In contrast to this work we investigate the requirements for embedded systems with heterogeneous processors, providing isolation as well as sharing where needed.

Capability-based security has long been discussed as a means to enforce the Principle of Least Privilege [110], [25], where access to all objects in the system have to be performed through capabilities, unforgeable pointers that carry access rights to specific regions in memory. The focus of the recent work in [136] is the implementation of a hybrid capability model based on additions to the instruction set and the addition of a capability co-processor to protect physical addresses. Our work takes a similar approach, where our permissions are akin to their capabilities; however, we generalise the ability to manipulate permissions using conventional writes, and examine a multiprocessor context.

A similarly "resource-aware" approach for multiprocessors is described in [27] where PCBAC is also used to enforce access controls. However, unlike our distributed approach, a centralised OS handles configuration of access protections. What is interesting to note here is the use of cryptography to add additional guarantees for confidentiality and integrity of code and data.

There are also several NoC-based approaches (which were discussed in Chapter 2). Such works propose "firewalls" at the network interface [20, 31, 35, 73, 97] which filter transactions. However, these approaches rely on an OS or other central authority to manage rules at run-time, such as in [73] where a dedicated processor is used. The approach proposed in [31] only allows processors with a supervisor mode the ability to request a permission change, which limits the type of processors we are able to employ. With respect to handling violations once they are detected, [20] proposes the reconfiguration of hardware firewalls to facilitate a heightened

security mode where memory accesses are severely restricted. This work requires a complete system reset for continued execution, whereas our approach allows isolated task groups to continue running.

4.6 Summary

There are several competing concerns when it comes to the design of embedded systems, and heterogeneous MPSoCs are used to satisfy complex design requirements. However, heterogeneous execution platforms provide a security challenge as different processors, IP blocks, and resources are all integrated. In this chapter we presented hardware support for enforcing permissions based on the specific task/resource relationships required by an embedded system design. This stemmed from our exploration of MPSoC security requirements in Chapter 3. As a result we are able to introduce isolation where needed, thus reducing the potential impact of a compromised task. To avoid the notion of a privilege hierarchy, we investigated distributing the management of access control among tasks themselves. This was explored through the design of Isolation Units, hardware blocks that can be used to check memory transactions before they enter the on-chip interconnect, and can be used to grant/revoke permissions to both local and remote tasks at run-time. As a means of addressing some shortcomings from the original FIFO-based implementation, we also discussed an alternate IU design which is amenable for interconnection using a bus-based architecture. We then characterised the performance penalty of our security approach by developing simulation models of Isolation Units, as well as an overall MPSoC infrastructure. The IU implements the fundamental isolation mechanisms that are used to improve security in the following chapters of this thesis. In the next chapter, we move to a higher level of the design abstraction, and investigate how we can augment an MPSoC design flow to incorporate automated addition of IU into the platform to achieve specific security goals.

CHAPTER 5

Towards Security-Aware Enhancement of an MPSoC Platform

The ability to consider and address potential security issues in system-level design is of paramount importance, especially when designs consist of hardware and software components integrated into heterogeneous multiprocessor systems on chip. In this chapter, we address the challenge of designing an MPSoC-based platform which satisfies designer-defined security requirements. To better support the design process, designers need to be able to quantify the potential impact of possible attacks, and subsequently implement strategies to mitigate these so that the extent of damage can be limited. There is a need for systematic and automated approaches for architectural exploration as part of the overall design process for a specific application. Building on the idea of Impact Analysis from Chapter 3, and using the hardware Isolation Unit presented in Chapter 4, we present an approach for specifying security rules, augmenting the platform, exploring the design space, and automation of the entire process to create several design options.

5.1 Overview

In MPSoCs we typically employ a regular shared interconnection architecture that facilitates communication between all platform nodes in the absence of any security restrictions, as in regular NoC approaches [31, 36, 106]. The underlying platform typically provides surplus capabilities that may provide attackers with the opportunity to violate the intended design. As such, *isolation* is a key element of a defence strategy, especially in mixed-critical systems where we might wish to guarantee

that certain critical parts of a system can continue correct operation despite an attack affecting elsewhere in the design.

On one hand, we can customise a given architecture with some form of fine-grain memory protection, and check *all* read and write accesses, potentially adding overheads in both resource cost and latency. On the other hand, if we can trust (or tolerate the compromise of) certain parts of our design, we may have opportunities to explore the design space, to trade-off between the security of a given architecture configuration, and the cost and overhead of the architectural additions required for satisfying a set of security requirements.

A large part of the design problem, however, is deciding how best to integrate different security solutions, particularly if our application has numerous security domains. Design space exploration (DSE), seeks to better support the design process, especially in the trade-off of design objectives. For example, in the software design domain, the work in [56] argues for DSE to incorporate security requirements to better guide designers. The work presented in [118] formalises an approach for the design of distributed ECUs in automotive applications. An ILP-based approach is used in [47] to optimise resource cost when introducing memory protection based on minimization of packet header size.

In a similar vein, we investigate, in this chapter, a more general design approach that takes a different angle to facilitate the exploration of the trade-off between security and resource cost. We propose an automated systematic framework for the generation and exploration of architecture configurations for an MPSoC execution platform, and tailor the configurations for a given application-specific design and its associated security requirements. Our intention is that this systematic approach can be subsequently augmented with other objectives and constraints to further explore the design space, while considering security as an integral part of the exploration.

In the following sections, we present this approach, drawing from an illustrative case study to discuss the processes in detail. We propose a way to define security rules to specify requirements for protecting parts of the design with different criticalities. To investigate the feasibility of our approach, we discuss a prototype implementation, and explore the implications of synthesising the generated configurations for an FPGA-based platform.

5.2 Preliminaries

In addition to the Task/Resource abstraction we presented in Chapter 3 (§3.3.1) we add the following terms in this chapter:

Platform Nodes — are hardware blocks (components) which are the nodes interconnected in our MPSoC NoC-based execution platform. Platform Nodes are classed as *processing* nodes (which implement tasks), or *non-processing nodes* (which implement resources).

Clusters — are platform nodes of related resources that have been grouped together by our tool (as explained in §5.5.2).

We consider the embedded system *application* as a collection of tasks, resources, and their relationships, where the designer has a priori knowledge of the design.

We also continue to use the threat model presented earlier in §3.1.3, and we consider the situation where an attacker can gain full control of a task. We are not concerned with the exploit method but are instead interested in the problem of restricting the ability for the compromised task to affect other parts of the system.

In any given design, we assume that different parts of the system have different levels of criticality. For example, safety-critical tasks should not be affected by the compromise of any other task. Furthermore, the designer may have varying levels of trust in the tasks and resources of the system. The designer's assessment of risk needs to be considered when deriving an architecture configuration on which to execute the embedded application.

Our aim is thus twofold: (1) for a given application, generate potential execution platform configurations derived from a given general MPSoC architecture, and (2) subsequently customise the configurations to ensure that the designer's security requirements are satisfied.

5.3 Security-aware Design

The primary goal of our proposed approach is to generate several potential execution platform configurations that are tailored to a given application and *compliant* with an associated set of security requirements.

Figure 5.1 presents a top-level view of the proposed approach, which features two main stages: **Architecture Configuration Generation**, and **Configuration**

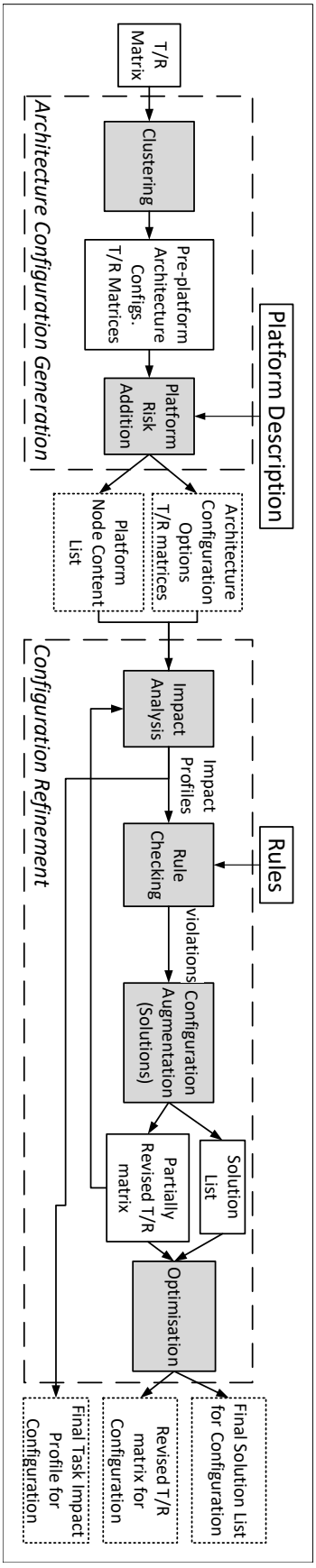


Figure 5.1: Overview of the Proposed Approach

Refinement. Each of these stages has several processes, which are indicated by the shaded boxes.

First, we take an application, represented as a Task/Resource Relationship matrix (T/R matrix), where each column represents a resource, each row represents a task, and the contents of each element in the matrix represents the relationship between task and resource (as discussed in §3.3.1). An example is shown in Table 5.1, and represented graphically in Figure 5.2. The T/R matrix is processed by **Architecture Configuration Generation**, where the primary aim is to develop several potential architecture configurations that are derived from a given general MPSoC architecture. Although the approach is applicable to a range of architectures, we consider a NoC-based architecture in this chapter.

The first process is *Clustering*, which analyzes the T/R matrix and generates several possible cluster configurations where the resources are grouped/clustered together into platform nodes. Initially we assume that each task is directly mapped to its own processing node (each processing node executes a single task). During this process, we also modify the T/R matrix to incorporate any potential risks that may arise as a result of forming a particular cluster from a given set of resources.

The T/R matrices for each generated configuration are then modified by the *Platform Risk Addition* process to further incorporate information about the capabilities that are introduced by our choice of interconnection architecture (NoC interconnect). This produces the architecture configurations that we subsequently analyze and refine from a security perspective.

In the security-driven **Configuration Refinement** stage, we take each configuration in turn, and perform *Impact Analysis*, which generates Impact Profiles for each task. These profiles are used to check designer-defined security constraints in the *Rule Checking* process. Detected violations are used to inform the *Configuration Augmentation* process, where a list of security additions is iteratively created to attempt to address the rule violations. If all rules are passed, the final configuration and solution list is optimised to reduce the total number of restrictions added (thus reducing the cost of security additions), and the final list of security additions, the final T/R matrix, and the final Task Impact Profiles are generated for each configuration. The designer can then weigh up the different configurations in terms of the resource cost or other design metrics. In the next section, we present a case study scenario that is used to illustrate the application of our approach as we discuss the details of each stage.

Table 5.1: Initial Task/Resource Relationship Matrix for the SHCS application case study (R = read, W = write, d = conditional)

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
T0	RW	W																	
T1		R	R	RW															
T2					RW	W													
T3						R	R	RW											
T4		Rd				Rd			RW	RW									W
T5			RWd				RWd					RW							W
T6													RW	RW					
T7												RW			RW	RW	RW		
T8																RW	RW	RW	
T9											R								R

5.4 Case Study

5.4.1 The Smart Home Control System

Consider an example embedded system application—the Smart Home Control System (SHCS). The SHCS combines environmental monitoring and control, multimedia capabilities, and provides an interface accessible over the internet. A local user can use a control panel to set lighting and climate preferences, display images, or play music on a sound system. Users can also monitor the status of their home remotely. After setting their preferences, the SHCS performs some actuation, controlling lights and HVAC systems. Additionally, the SHCS features a fire detection system, which monitors a fire sensor, and can trigger an alarm. To implement this application, the designer specifies the following tasks at the system-level:

T0: Light sensing; reads, processes, and stores light sensor data, also performs setup/configuration of sensors

T1: Light control; reads stored light sensor data and controls lights based on stored configuration

T2: Temperature sensing; reads, processes, and stores temperature sensor data

T3: Temperature control; reads stored temperature data, controls heating/cooling

T4: Remote interface; reads stored temperature and light data, displays them over the internet if a user is authenticated

T5: User management; manages the local control panel interface subsystem so an authenticated user can set their environment preferences

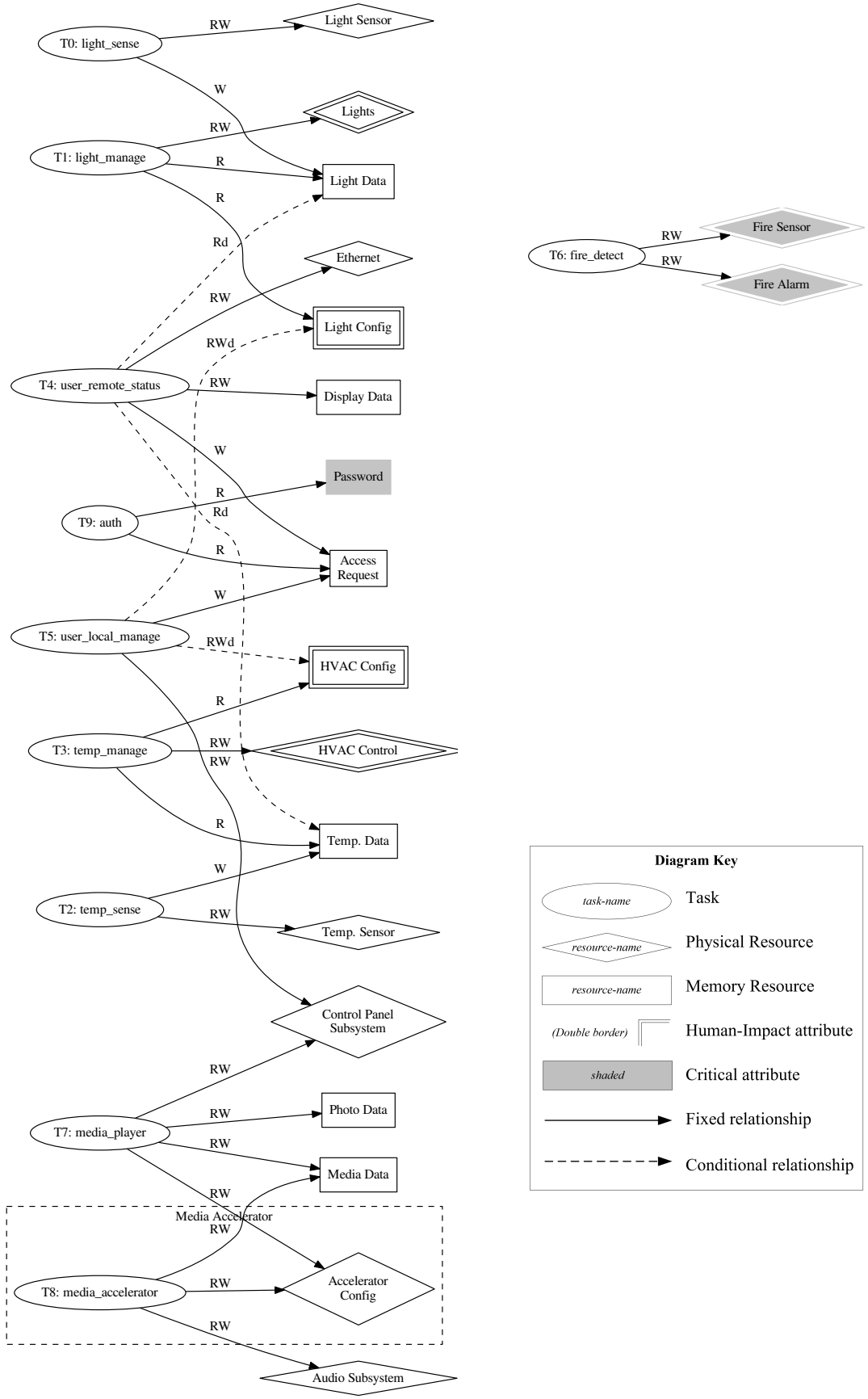


Figure 5.2: The Smart Home Control System as a T/R Graph

T6: Fire detection; implemented as a digital hardware component, monitors a fire sensor, and controls the fire alarm

T7: Media player; uses the control panel subsystem to display images, and manages the configuration of the media accelerator IP

T8: Media accelerator; a programmable hardware IP that can perform accelerated decoding/processing of multimedia, also controls the audio subsystem

T9: Authenticator; A task that checks the password when a user enters it locally, or when a user requests remote access

We associate the resources in our system with additional attributes defined and assigned by the designer; they further characterise the various components of the design, and are referred to when defining the security constraints for our security analysis. In this example, we define the following attributes:

- *Critical*, resources that have a high importance, or nodes that should be kept secret/private (c, represented as shaded in Figure 5.2)
- *Human-impact*, (also considered critical) to indicate resources that have a role in cyber-physical interactions (h, represented with double borders in Figure 5.2)
- *Physical*, to indicate that a resource represents a physical component (p, represented as diamond nodes in Figure 5.2)

The following is a list of all resources, and their attributes:

A: light sensor – (p) Detects ambient light

B: light data – Memory buffer for light data

C: light cfg – (h) Memory buffer for light preferences

D: light ctrl – (ph) Controller for lights

E: temp sensor – (p) Temperature Sensor

F: temp data – Memory buffer for temperature data

G: temp cfg – (h) Memory buffer for temp. preferences

H: temp ctrl – (ph) HVAC controller

I: display data – Memory for Control Panel UI

J: ethernet – (p) Ethernet block for networking

K: password – (c) Memory for password

L: control panel sub-system – (p) Block for Control Panel

M: fire sensor – (cph) Smoke detector

N: alarm – (cph) Emergency alarm

O: photos – Memory for photos

P: media data – Memory for multimedia playback

Q: accelerator cfg – (p) Control registers for Media Accelerator

R: audio system – (p) Audio playback sub-system

S: request – Memory buffer for requesting authentication

The system-level T/R matrix for the SHCS is presented in Table 5.1 with its corresponding T/R graph in Figure 5.2. The Media Accelerator is shown as both a task (T8) and resource (Q), as it can be used by other tasks, and can itself access other resources.

The designer can also specify some higher-level security conditions on relationships, such as conditional accesses, where a relationship is only "active" if a certain condition is met (these are shown in matrix elements with "d"). This is useful for representing behaviors such as authentication, where T4 and T5 are only allowed to access potentially sensitive data *after* a user is successfully authenticated.

5.4.2 Execution platform building-blocks

In this case study we utilise a NoC-based architecture, and aim to derive a suitable NoC configuration with security additions for a given application and its requirements.

For demonstrative purposes, we explore the use of the following building-blocks for our execution platform.

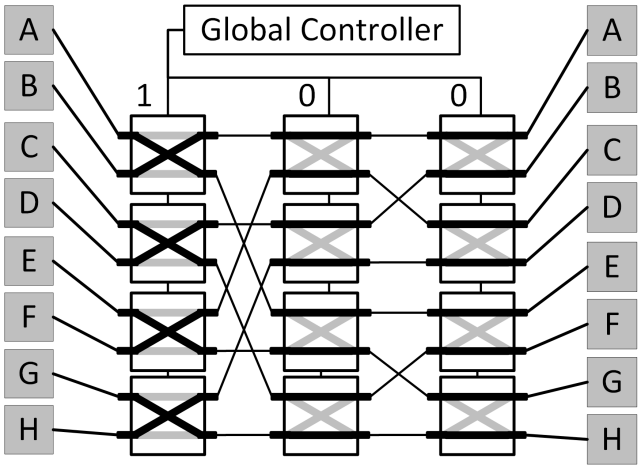


Figure 5.3: TDMA-MIN NoC with 8 platform nodes

The basic NoC architecture

The interconnect architecture has a crucial role in any MPSoC design, and as in Chapter 4 we adopt a multi-stage interconnection (MIN) network inspired by the work presented in [106], which features simple two-port cross-bar switches that can be set to either straight or cross mode. Our variation arranges the switches in a butterfly topology. The mode of the switches is dictated by a global controller, which sets all switches in a stage to the same mode. For example, in Figure 5.3, an 8-node network is shown with the control value of "100" indicating that the first stage of switches is in the cross mode, and the second and third stages are in the straight mode. The global controller can employ a time-division multiple access (TDMA) scheme by generating a repeated sequence of control values, so this network ensures that all platform nodes in the network are able to communicate with all other nodes, exactly once per TDMA round. We adopt this network for the SHCS application because:

- This network architecture is time predictable, which is useful for the parts of the system that have hard real-time requirements (T6, for example, has hard real-time constraints). In addition, the TDMA scheme establishes *temporal isolation*, which adds further protection where necessary.
- The network is resistant to Denial-of-Service type flooding by an errant core, as bandwidth between all platform node-pairs is guaranteed through the TDMA scheme.

- No routing information is required to be encoded in packets, as the routing is managed by the global controller
- Network Interfaces (NIs) are not complex — packet injection for receipt at the correct target port is based on a simple mechanism, where the destination port in a given cycle can be determined using the simple function:

$$destination \ = \ (portnumber >> 1) \ \oplus \ current\ control\ value \qquad (5.1)$$

The *current control value* is provided by the global controller. This relationship allows the NI to work out the source/destination of a packet without needing a tag or header. It also allows us to introduce very simple packet filtering, as explained in the following paragraphs.

Security Mechanism Additions

The NoC can then be further enhanced through the addition of security components. In example scenario we will only consider two options.

The first option is using a coarse-grain mechanism to block specific node-pairs from communicating. To do so, the receiving NIs can determine the source of a given packet given the current time slot through this relationship:

$$source \ = \ (receiving\ port \ \oplus \ current\ control\ value) << 1 \qquad (5.2)$$

A simple check can be added into the NI to prevent the delivery of packets from certain forbidden ports, or to allow only the delivery of packets from specific ports.

The second option employs some form of Memory Protection for fine-grain protection, whereby a hardware block is added to check memory accesses against stored designer-defined permissions. This regulates whether or not transaction can go through to the destination. For example, a permission may exist that allows one task to read but not write to a specific region of memory, and so the hardware block should reject any attempts by that task to write to the protected region.

This is typically achieved by using some form of Memory Protection Unit (MPU), and the use of MPUs for protecting memory is well established. Related approaches can be found in [36] and [31].

In this scenario, we use as our building blocks the basic MPU, which corresponds to the *base variant* of the Isolation Unit (IU) presented in the previous chapter. To support conditional accesses, we need to be able to configure permissions at run-time, so we should also employ the more complex IU variants. Instead of

using an OS or special instructions to configure/manage permissions, tasks, including bare-metal programs, can use a series of memory writes to its local Task IU to form a new permission. The IU checks designer-defined Owner permissions to ensure that the task has the right to give other tasks certain access permissions to a given resource. Owner permissions are unforgeable; they are defined by the designer during the design process, and enforced by IUs which are preconfigured with these permissions at synthesis time. Tasks are therefore prevented from granting access to resources they do not own. When the permission has been properly formed, it can be sent through the NoC and received by the appropriate resource IU, where the permission is loaded into its Permission Checking block, thus enabling another task's access to (a subset of) the resource. This is particularly useful for the implementation of password-protected accesses; in our example scenario, T4 can only access environmental data if T9 grants access—the use of IUs can facilitate this operation at a platform-level.

5.4.3 The Design Problem

For brevity, we apply the following constraints to our design problem:

- We initially assume that each task is mapped to its own single processing core which will be connected to one of the ports in the NoC
- Physical resources will also directly become platform nodes, also using a port each

The remaining memory resources can then be grouped/clustered to determine the remaining platform nodes in the network.

The basic NoC architecture supports 2^N nodes, where N refers to the number of stages. So, if our design requires 12 nodes, we need to use a 4-stage network (which has a maximum capacity of 16 nodes) and the capacity of the network won't be fully utilised. With regards to the security additions, the coarse-grain protection of port blocking is low cost in comparison to fully-fledged MPU-based protection.

The aim of our design exploration is therefore to determine which combination of fine-grain and coarse-grain protections should be added into the architecture to support the designer-specified security requirements. Our design space involves various ways to cluster memories; if we map memory resources to separate platform nodes, we might be able to use the less costly port block approach for protecting that resource, but may incur greater costs if we need a larger NoC size. We

might be tempted to cluster nodes to reduce the network size, but possibly incur larger costs by requiring fine-grain protections.

Other design factors, such as performance, will also be affected. For instance, the use of our MPU introduces a clock cycle delay for each check. While these additional factors could also be added in the design exploration, we will only explore the security/resource dimension in the following discussions.

5.5 Configuration Generation

The Architecture Configuration Generation stage is designed to generate a number of candidate architecture configurations for a given embedded systems application. Configuration generation involves two main processes, *Clustering*, followed by *Platform Risk Addition.*

5.5.1 Inputs

Application Description

The first input, used in *Clustering*, is the T/R matrix for our application. There are several ways in which this matrix can be derived. Besides a manually specified system-level description, as in our case study, there exist several automated approaches that can be used to automatically derive task/resource relationships.

For example, an algorithm is presented in [7] which can be used to analyze a program to identify shared memory buffers with security requirements; this can be extended so that we can automatically identify all memory buffers in a design, and potentially unearth hidden dependencies between tasks.

We also assume that in the design process, designers will generally explicitly define shared memory buffers for inter-task communication, which can be modelled in the task/resource relationship matrix.

In C-like languages we might use specific data structures to store resources like passwords; by modifying linker scripts a designer can explicitly place these structures in their own memory region. Deciding the granularity of representing resources in the task/resource matrix is also a design decision—one might choose to represent individual memory words as discrete resources, or simply say that a certain task's heap space is a resource. In our approach, the final physical address map can be determined after we have decided how to group resources into platform nodes.

Platform Description

The second input, used in *Platform Risk Addition*, is a description of how the general MPSoC interconnect architecture affects the relationships between tasks and resources. For example, an architecture that allows all-to-all communication enables all tasks access to all resources; this introduces risks that should be quantified and potentially addressed by the designer. Conversely, a custom point-to-point architecture might enable communication only between specific tasks and resources.

Different underlying architectures affect task/resource relationships in different ways, so we use an algorithm in Platform Risk Addition that modifies the given T/R matrix to indicate how a given architecture affects the capabilities of platform nodes.

5.5.2 Clustering

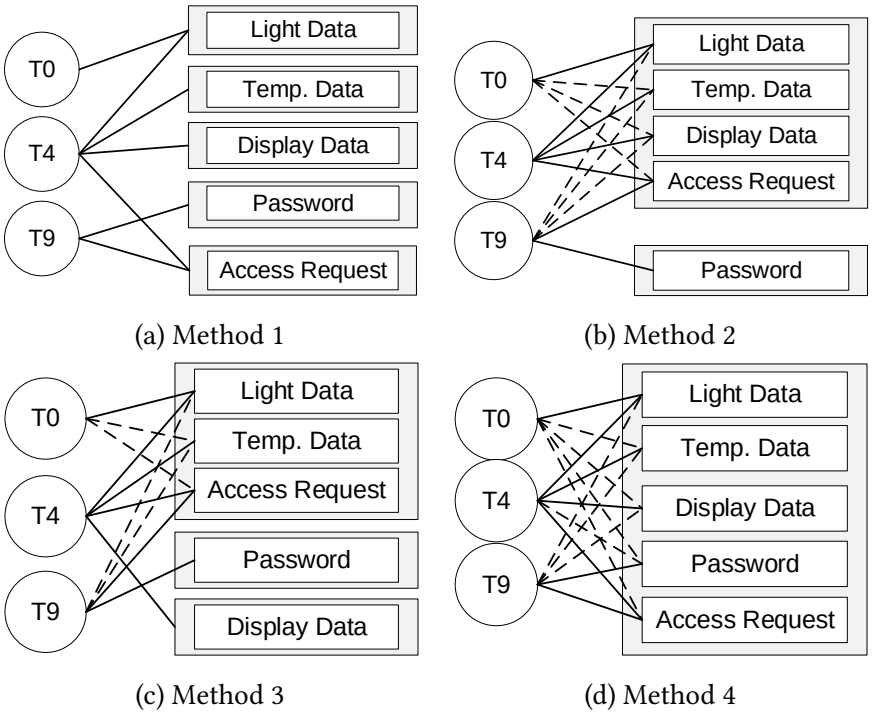
The Clustering process generates several options for how resources can be grouped into platform nodes in order to reduce number of NoC ports used. Physical resources are first assigned as their own platform nodes, which will be referred to as physical resource nodes. Designers define their own approaches for determining how resources can be clustered.

For the SHCS, we apply four different clustering methods to explore the design space. Each method produces a different number of platform nodes, which can affect the overall network size. Furthermore, these different configurations result in different potential security risks. Figure 5.4 shows a subset of the SHCS in graphical form, and the effect of the four clustering methods described in the next section.

Clustering Methods

The first method results in a configuration with the greatest number of platform nodes (and by implication, potentially the largest required NoC configuration). In this method, we simply make each resource a platform node, as illustrated in Figure 5.4(a).

The second method (Algorithm 2) is designed to reduce the number of protections we are likely to introduce. First, we sort the tasks by the number of resources they use (from most to least), and then go through each task and place the resources that are used by that task in a platform node (as shown in Figure 5.4(b)). The intuition here is that, for example, if we wanted to prevent T9 from accessing *Light Data*, *Temp. Data*, and *Display Data* which are co-located in a cluster with *Access*



(a) Method 1 (b) Method 2

(c) Method 3 (d) Method 4

Figure 5.4: Cluster Methods applied to SHCS (subset). Dotted edges represent added risks

Request, we would only need to enforce one access control such that T9 can only access the *Access Request* resource on that cluster.

The third method is a variation of the second method. We first identify the resources that are used exclusively by a task. Each of these exclusively used resources is then grouped into its own cluster (for example, *Password* is used only by T9). The remaining resources, which are all shared, are then clustered using the same algorithm as in the second method. This is shown in Figure 5.4(c).

The final method simply involves grouping all the nodes into a single cluster (Figure 5.4(d)), thus minimising the total number of platform nodes, and NoC ports required.

Adding in cluster risks

When we cluster resources, we potentially introduce risks in our design, based on the assumption that if a task uses a resource in a cluster, it will be given physical access to that cluster, and therefore, without any security additions, that task can access all the resources in that cluster. We represent that with the dashed edges in Figure 5.4. This allows us to model potential issues such as overflow, either deliberate or accidental; when resources are clustered, we assume that they will be

Algorithm 2: Cluster Method 2

```
Input: List of tasks: taskList[], Set of unclustered resources: rL{},
       relationship matrix: rm[][]
Output: List of platform nodes: clusterList[]
count no. of resources used by each task by adding number of "useful"
 relationships;
sort taskList by resource_count;
for task in sorted taskList do
    for resource used by task do
        if resource in rL then
            if not yet started_a_cluster then
                clusterList.add(new Cluster);
                set started_a_cluster;
            end
            /* -1 indexes the most recently added cluster     */
            clusterList[-1].add(resource);
            rL.remove(resource);
        end
    end
end
```

Table 5.2: Added relationships due to Cluster Method 2 (shaded resources are in the same cluster). "a" elements are due to clustering, "b" elements are due to the platform

	Light Data	*Temp. Data*	*Display Data*	*Access Request*	*Password*
T0	W	**a**	**a**	**a**	**b**
T4	Rd	Rd	RW	W	b
T9	**a**	**a**	**a**	R	R

assigned contiguous addresses in the memory map. An errant task that writes to one of the regions in the cluster can inadvertently overwrite neighboring regions.

To capture this, we produce an intermediate task/relationship matrix for each configuration, adding an "a" relationship between tasks and resources it does not actually need (but that are located within clusters that the task uses, as shown in Table 5.2).

5.5.3 Platform Risk Addition

After generating the cluster configurations and their respective T/R matrices, additional task capabilities (and therefore, risks) that are introduced by the choice of interconnect platform are added. In our case study, TDMA-MIN NoC allows all-

to-all access between platform nodes, so we express these additional capabilities into each T/R matrix using Algorithm 3, where the introduced relationships are represented as "b", as shown in Table 5.2.

We assume that, by default, NIs are preconfigured with port blocks that will prevent processing nodes from communicating with other processing nodes, and non-processing (resource) nodes from communicating with other non-processing nodes. The final T/R matrix for each configuration is finally written to a file for further processing in the second stage.

Algorithm 3: Platform Description for the case study NoC

```
Input: Relationship Matrix rm[][], List of platform nodes clusterList[]
Output: (modified) Relationship Matrix rm[][]
for each task (row) in the matrix do
    for each cluster in clusterList do
        for each resource in cluster do
            if rm[task][resource] == empty then
                rm[task][resource] == "b";
            end
        end
    end
end
```

5.6 Configuration Refinement

The second stage of our approach is Configuration Refinement, where we consider each of the previously generated configurations in turn, and determine what security additions need to be made in order to satisfy designer-specified security constraints. The Configuration Refinement Stage takes in the T/R matrix for each configuration, as well as information about the contents of each platform node. The three main processes in this stage are *Impact Analysis*, *Rule Checking*, and *Configuration Augmentation*.

5.6.1 Rules

A key input to the refinement step is a set of security constraints, which can be set by the designer to express security requirements for the final design.

Security requirements might be fairly general (e.g., reduce the impact of an attack originating from a certain task), or more specific (e.g., prevent *all* human-critical tasks from being impacted by a compromised by a certain task). These requirements are motivated by the functional requirements of the design. We assume that designers have performed some prior work to decide what risks they want to mitigate. As such, we are primarily concerned with the ability for a compromised task (potential threat) to affect other tasks/resources. For example, if we doubt the complete trustworthiness of a certain task in our system, we need a way to specify the risk posed by the task so that our tool can automatically address it.

In our T/R abstraction, security can be improved by restricting the connections between specific task/resource pairs, and at a fundamental level, security *rules* could be specified in this way. To better support automated checking, it may be convenient to check a "group" of rules that imply restrictions between certain tasks and resource types, or vice versa.

Hence, we propose a set of rules to help establish desired boundaries between different parts of the system with respect to their criticality. As a starting point, we provide five types of rules to express relationship restrictions between assets (tasks or resources), and potential threats (tasks).

Type 1 Impact restriction: We use this to specify that a task asset is not "impacted" by another task.

Type 2 Resource class access restriction: We use this to specify when a task should not have access to resource assets with specific attributes.

Type 3 Specific resource restriction: This is the same as Type 2, except for a specific resource asset.

Type 4 Untrusted task: This is used to indicate that an entire task is not trustworthy (i.e. the task may cause a possible threat), so it must have its capabilities limited.

Type 5 Resource exclusivity: This is the complement of Type 3, where we want to ensure that only one task can write to a resource.

These rules are not "exhaustive" in the sense of capturing all possible security requirements, and it is possible to add new rules as required (for example, adding restrictions on things like the number of resources accessible by a task). It is assumed that the original application description satisfies the rules as specified by

the designer; if the application itself does not satisfy a rule, the automated process for adding security additions to the architecture configuration may be insufficient for resolving the rule violation. Having that said, the detection of rule violations may be able to guide the designer towards refinement of application-level logic. For the SHCS case study, we define the following *optimistic* rule-set with the form {type, potential threat, asset}, except for Type 5, which does not specify a single specific threat:

Rule 1: 1, T8, T4: Remote interface should not be affected by compromise of the media accelerator

Rule 2: 1, all other tasks, T6: Fire detect should not be affected by compromise of any of the other tasks (this expands out to nine separate rules, e.g. {1,T0,T6}, {1,T1,T6} etc. ...)

Rule 3: 2, T5, ph: User management should not affect resources that are physical and have human-impact

Rule 4: 2, T4, h: Remote interface should not affect resources that have human-impact

Rule 5: 1, T4, T1: Light control should not be affected by the Remote Interface

Rule 6: 2, T9, h: Authenticator should not affect resources with human-impact

Rule 7: 3, T3, I: Temperature control should not affect the display data resource

Rule 8: 4, T7: Media Player is untrusted

Rule 9: 5, T9, K: The password resource is accessible only by the Authenticator

For comparison, we also consider a *pessimistic* rule-set where the designer has decided that they do not fully trust any of the tasks. In this rule-set, Type 4 rules for all tasks are specified to restrict all task capabilities to the minimum needed (i.e. allowing only the relationships specified in the original T/R matrix).

5.6.2 Impact Analysis

Impact Analysis (§3.3.2) forms the basis of our security-driven approach. For our SHCS case-study however, we restrict our analysis to the Immediate and Secondary levels only, and consider tasks to be sufficiently isolated from each other if they do

not appear in each others' two-level impact profiles. Each TIP essentially characterises the reach an attacker has if they were to compromise the associated task, and Impact Analysis can be performed to help designers identify potential paths from which certain assets could be attacked given an architecture configuration. Intuitively, the more shared resources a task uses, the greater the potential impact if compromised.

5.6.3 Rule Checking

With the TIPs as determined by Impact Analysis, we can then proceed with rule checking. One rule is checked at a time; if a rule passes, the next rule is checked, if a rule is violated, we proceed to *Configuration Augmentation* to try and resolve the violation. Four of the five rules are easily checked by examining the contents of the immediate and secondary impact lists in the TIP of the potential threat task (threat) specified in each rule.

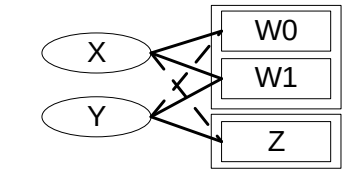
Type 1 rules are checked simply by examining the contents of the potential threat's secondary impact list for the task asset. If the task asset is in the list, the rule has been violated.

Type 2 rules are checked by examining the threat's immediate impact list, and checking the attributes for each resource in that list against the resource asset attributes. If any of the resources in the list have the attributes, the rule has been violated.

Type 3 rules are checked by examining the threat's immediate impact list for the specific resource asset. If it is in the list, the rule has been violated.

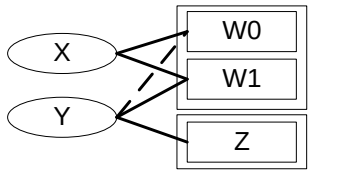
Type 4 rules are checked by comparing the size of the threat's immediate impact list against the number of resources it actually uses. If the size is greater than the number of resources it needs, the rule has been violated.

Type 5 rules could be checked by counting the number of times a resource appears across all immediate impact lists (it should appear at most, once, otherwise, the rule has been violated), but this can also be achieved by counting the number of tasks that have access to that resource in the relationship matrix. Because the proposed rule types simply involve looking up the rows/columns of the relationship matrix, the rule checking is easily automated. While scalability is affected by the number of tasks/resources, it is possible to raise the level of abstraction to consider even larger clusters of tasks/resources.



	W0	W1	Z
X	RW	R	b
Y	a	RW	RW

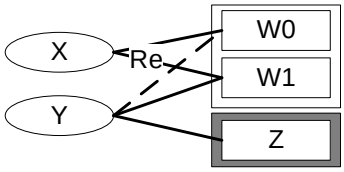
(a) The initial T/R relationship with added risk



	W0	W1	Z
X	RW	R	
Y	a	RW	RW

Solutions:
+ Port Block at W

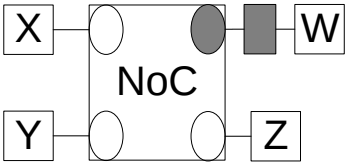
(b) Port block solution



	W0	W1	Z
X	RW	**Re**	
Y	a	RW	RW

Solutions:
+ Port Block at W
+ MPU write-deny at W

(c) MPU permission solution



(d) Final config. with port block (ellipse) and MPU (rectangle)

Figure 5.5: Iterative Solution Addition

5.6.4 Configuration Augmentation

Our approach for resolving rule violations involves incremental addition of solutions to the configuration. In our case study, these are NI port blocks, and MPUs (as described in §5.4.2).

We adopt a two-level approach. First, we attempt to add in a port block to prevent processing node to non-processing node (task to cluster/physical resource node) communication. If Impact Analysis and Rule Checking reveals that the rule is

still violated, we then add a MPU-based restriction. As we add solutions to restrict task capabilities, we delete elements in the T/R matrix, or mark where relationships are restricted or *enforced* (by adding "e" to the element, for example, marking a relationship as "Re" where a Read-only is enforced).

For Rule Type 1 violations, we first determine the resources that the task asset reads from (the task asset's *needs list*)—these are the resources that the threat should not be able to *impact* (maliciously modify) in order for the rule to be satisfied. At the node level, we check to see if the threat has any need to access a given non-processing node which contains resources in the *needs list*. If it does not, we can introduce a port block to prevent the threat from affecting any of the resources in that non-processing node. In other words, if the threat has a "b" relationship to any resource in the *needs list*, by implication, it does not use **any** of the resources in the non-processing node that the affected resource belongs to.

If we are unable to resolve the rule violation with port blocks alone, we then introduce MPU-based restrictions. If a resource in the *needs list* is not used by the threat, we add a *deny* restriction to stop the task from accessing that resource. If the threat has a read relationship with that resource, we add a *deny-write* restriction, so that it cannot maliciously modify the resource.

For example, consider Figure 5.5. X and Y are tasks on their own processing node, W0, W1, and Z are resource nodes (where W0 and W1 are clustered together in node W), and we use the same building-blocks as in our case study. Their initial T/R relationship is shown in Figure 5.5(a). We define a rule {1, X, Y} to ensure that X cannot impact Y, and this rule is initially violated. In the first step (Figure 5.5(b)), we identify that {W1, Z} are in Y's *needs list*. X has a "b" relationship to Z, so we first add a port-block to prevent interaction between X and Z. However, X can still impact (maliciously write to) W1, so in the second step (Figure 5.5(c)), we introduce an MPU to ensure that X cannot write to W1. The rule violation is resolved, so we end up with the architecture configuration as in Figure 5.5(d).

For Rule Type 2 and 3 violations, we undertake a similar process, but instead construct a *forbidden list* of resources that the threat should not potentially impact. We proceed to add port blocks or MPU-based restrictions, the same way we address Rule Type 1 violations, to protect the resources in the *forbidden list*. On the other hand, for Rule Type 4 violations, we take a slightly different approach, whereby we identify all the relationships to resources that are not needed by the threat. We add port blocks when access to entire non-processing nodes are not required, and MPU-based restrictions to restrict the threat task's capabilities as much as possible.

Finally, for Rule Type 5 violations, we identify the tasks that should not use the resource asset (which should be all tasks except for the task stated in the rule), and add port blocks so that the non-processing node is not accessible to tasks that have no need to access the non-processing node on which the exclusive resource is found, and MPU-based restrictions to all other tasks that require access to the non-processing node.

In the current prototype, rules are checked and addressed in the order in which they have been specified by the designer. Each introduced solution restricts the capability of tasks, and so, intuitively, it is possible that addressing a certain rule violation may inadvertently address a subsequent rule violation. The dependency between augmentations introduced and the order of rule checking may thus affect the overall solution in several ways, such as resource cost. While considering the implications of different rule ordering is out of the scope of this thesis, in future we could further explore this notion, as well as ways to incorporate the choices even when more solution options are given for addressing rule violations.

5.6.5 Optimization

In the final part of the Configuration Refinement stage, we perform a simple optimization pass to reduce the number of restrictions added into the system. For each non-processing node in the network, if the number of processing nodes (tasks) that use it is less than the number of tasks that are denied access to it through port blocking, we change the port blocks to that node into port grants—i.e. we check if a task can access that node instead of seeing which tasks cannot.

Similarly, we can also check the MPU-based restrictions added to each non-processing node; if the number of deny rules is greater than the number of useful relationships between tasks and resources in that node, we can replace all deny restrictions with grant permissions instead, where needed. Finally, we also identify processing/non-processing node relationships that are not restricted by any MPU-based restrictions; we add an MPU-bypass to the final solution list so that read/write packets from certain tasks do not have to go through permission checking.

5.7 Results and Discussion

Table 5.3 shows the total number of nodes, and the contents of each cluster, for the four configuration options generated by the Configuration Generation Stage for

Table 5.3: Output of Configuration Generator

Config	Nodes	Cluster Contents
1	29	All resources nodes are network nodes
2	23	c4: light data,temp data,display data, request c5: light cfg,temp cfg c7: photos,media data c9: password
3	26	c4: light data,temp data,request c5: light cfg,temp cfg c7: media data c9: password ce9: password ce4: display data ce7: photos
4	21	Cluster (c8) with all memory resources

the SHCS case study.

For our case study, the NoC size in all cases will be the configuration for 32 nodes (with some unused ports). For resource usage comparisons, we compile our designs using *Analysis and Synthesis* from Altera Quartus II 15.0 for a Cyclone IV target (set to AUTO). Synthesis of the base 32-node NoC (for packet sizes of 66-bits) indicates a base resource cost of **12675** logic elements. The packet payload consists of a 32-bit address, 32-bit data, 1 bit for read, and 1 bit for write. For other applications, it is possible that the network size required varies between each configuration.

Table 5.4 contains a summary of the number of security features in the solution list for each configuration (that satisfies the *Optimistic* rule-set), in terms of the total number of port checks, total number of MPU checks, and the number of MPUs and IUs (IUs are MPUs with additional logic to support run-time conditional access controls). This is also represented graphically in Figure 5.6, where dark ovals indicate where the tool suggests adding active port protections, and dark rectangles indicate the addition of MPUs/IUs. As expected, the more constrained *Pessimistic* rule-set, where the designer identifies all tasks as potential threats, requires more solutions, as presented in Table 5.5.

The different MPUs and IUs added into the system have different resource requirements; IUs are more complex due to the additional logic required for processing and loading new permissions. The size of MPUs and IUs also increase as the

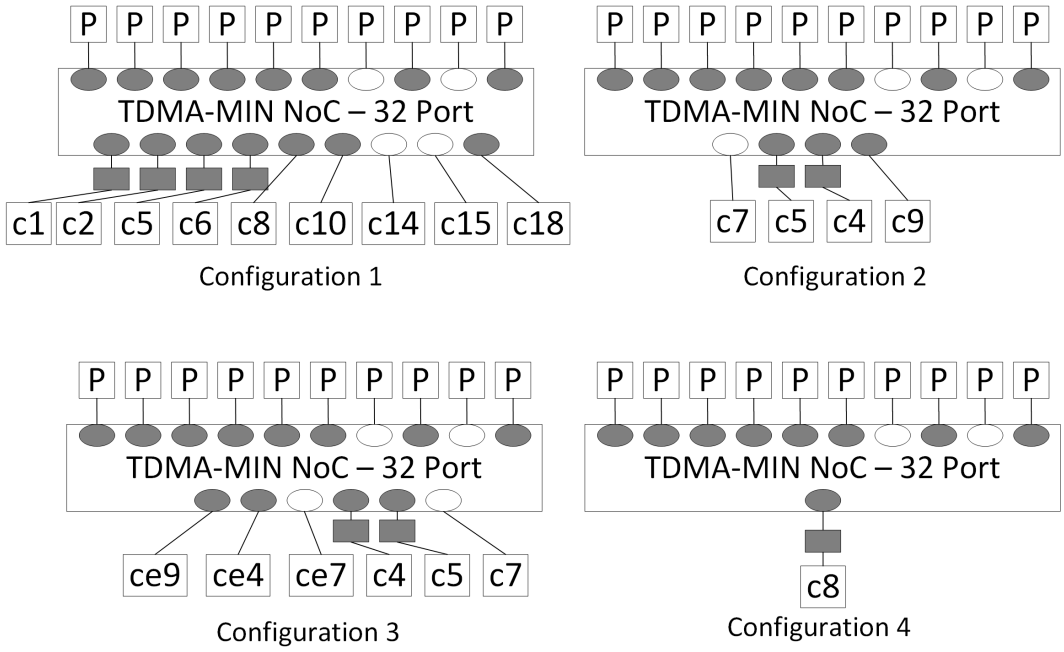


Figure 5.6: Graphical representation of Optimistic rule-set satisfying configurations (un-used ports and processing nodes omitted)

Table 5.4: Solutions Added to Satisfy Optimistic rule-set

Config	Total Port Checks	Total MPU Permissions	MPUs	IUs
1	19	4	0	4
2	14	5	0	2
3	15	4	0	2
4	9	18	0	1

Table 5.5: Solutions Added to Satisfy Pessimistic rule-set

Config	Total Port Checks	Total MPU Permissions	MPUs	IUs
1	28	10	2	4
2	21	14	2	2
3	23	14	1	2
4	13	18	0	1

Table 5.6: Resource overhead from MPU/IU additions (as % of base NoC resource usage)

Config	Optimistic Rule-set	Pessimistic Rule-set
1	+10.7%	+15.0%
2	+7.3%	+15.7%
3	+6.5%	+14.1%
4	+12.2%	+12.2%

number of permissions to be checked are increased. To get an idea of the resource cost for implementing our security additions, we synthesize the various MPU/IU configurations that appear in the solution list, where each MPU/IU checks the full 32-bit address from the incoming packet. We present the resource cost overhead introduced by the security additions as a percentage of the base network cost in Table 5.6. Additionally, in all configurations, the Authenticator task will have a Local Task IU, which requires 627 logic elements (this cost is omitted from the table).

Port checks, by comparison, remain small; in fact, the addition of port checks adds about 9% to the base cost of the receiving NI. Each receiving NI requires around 70 logic elements. As a result, we can see that Config. 3 provides the most resource usage efficient configuration for satisfying the Optimistic rule-set, largely because we can utilise a port check instead of an MPU permission to protect *Display Data* unlike Config 2. In Config. 1, separate IUs need to be added to support the dynamic permissions; adding a full IU is costlier than adding an additional permission to an existing IU. This is also the reason that Configs. 2 and 3 require more resources in order to satisfy the Pessimistic rule-set compared to Configs. 1 and 2.

Another way in which we might choose to compare solutions from a security perspective, is to examine the final TIPs generated by the Configuration Refinement stage. With these, the designer can check to see which assets remain vulnerable to certain task compromises even after satisfying the initially specified rules, or compare the relative security of different architecture configurations and their resulting resource requirements. For example, after satisfying the Optimistic rule-set, the Media Accelerator still has six tasks in its secondary impact list. With this in mind, a designer could factor in other information (such as how "likely" the Accelerator could be abused) to decide whether they should attempt to reduce the TIP by modifying the design (i.e. by re-writing the design, or adding additional rules). In this example, we could add a Type 5 rule to ensure that only the Media Player can configure the Accelerator, or a Type 4 rule to restrict what the Accelerator can ultimately access. The designer can use our tool to explore different trust scenar-

ios by specifying different security constraints as we have done here with the two presented rule-sets.

5.8 Summary

In this chapter we presented a systematic security-aware approach for the design of an MPSoC-based embedded system. Our two-stage approach first generates several potential architecture configuration options for a given application, and then analyzes each from a security perspective to detect designer-defined security rule violations. We then demonstrated how each configuration can be augmented with security mechanisms in order to resolve any rule violations, to give a designer options from which to select a satisfactory design solution. Our systematic approach is also further extensible; for example, more complex clustering approaches could be investigated, such as the formation of clusters of tasks together with resources, or even the clustering of tasks onto the same processing node, resulting in further possible configurations. Alternative general architectures and other security mechanisms could also be incorporated into our approach. In the next chapter, we further extend our automated approach to consider an even higher level of design abstraction as a strategy to safely incorporate IPs whose internal implementation details are not completely known to the designer.

CHAPTER 6

Introducing Context-Aware Protections to the MPSoC Platform

In the previous chapter we proposed a system-level approach for exploring potential application-specific protection options. With impact analysis and user-defined rules, designers can use IUs with different complexities. However, as embedded systems become more complex, we also need to support heterogeneous clusters of pre-designed HW/SW IPs. In this chapter, we introduce the notion of context to our proposed security protections. This allows designers to ensure that memory accesses occur only when expected, and in the right order. To support this, we adopt a novel service-level abstraction, framing our security as protection of services, and protection from service-providers. While the IUs in Chapter 4 and 5 assumed that application tasks were explicitly aware of security, and thus able to manage dynamic permissions directly (§4.3.5), we present here a strategy where dynamic permissions are managed by enhanced NIs directly at run-time, even when IPs are not designed with security as a design goal. We present an extension of the IU as the context-aware permission checking (PCheck) block.

6.1 Overview

IPs are often provided with full access to parts of the system, through a shared bus or regular NoC architecture. This has increased the potential severity of exploits such as the recent *Broadpwn* vulnerability [130], where a compromised wireless chipset could be commandeered to affect an OS kernel despite no ordinary need for the chip to access that memory region. IPs with direct memory access capabilities (DMA) could be manipulated to circumvent traditional software-based security

measures (as in [67]) particularly when they have access to more parts of the system than required. Our aim is to prevent or limit a potential attack's impact on the design as a whole.

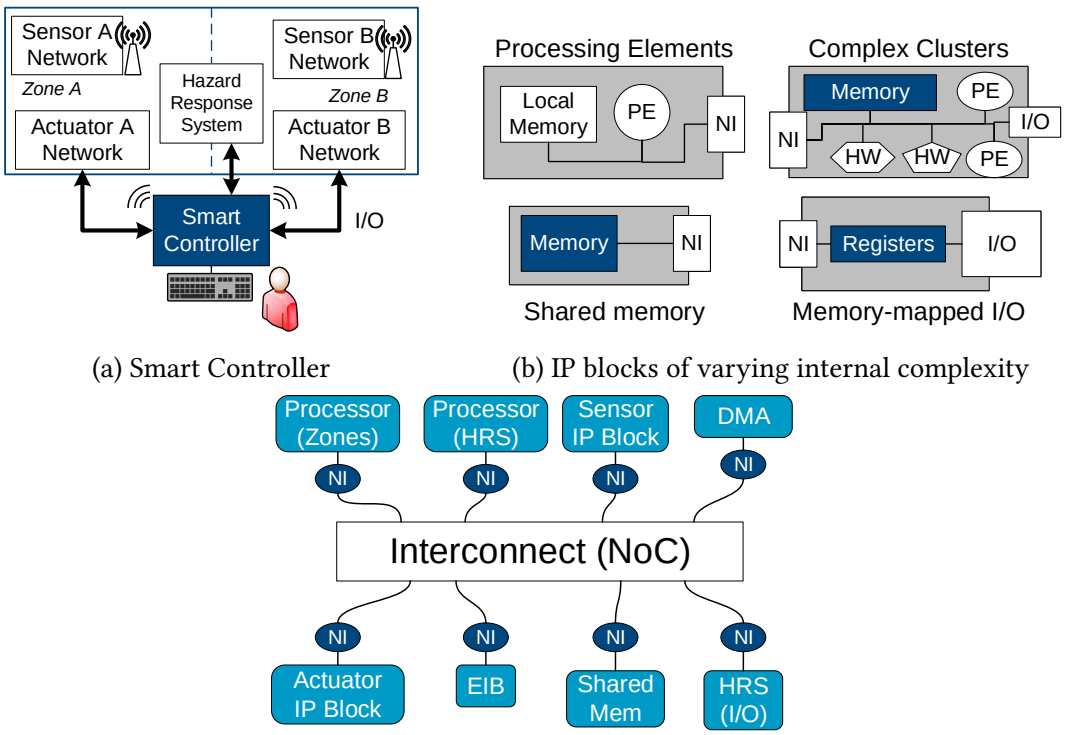
In this chapter, we propose an alternative, more decentralised approach for reducing an attack's impact across different parts of the system. This is especially important when we consider future application-specific MPSoC designs, where a general NoC architecture is customised with many concurrent behaviours and heterogeneous components from different sources. In the security approach we presented in Chapter 4, we assumed that applications were explicitly security-aware, and that each task would explicitly manage the shared regions.

However, faced with the possibility that tasks are not able to make use of IUs directly, we need to develop a suitable protection infrastructure. The infrastructure should support components that do not have a common software base, and enforce protection even when components are not *security-enabled* (i.e. without built-in security mechanisms). Components with internal designs that are not fully accessible by system designers should also be accommodated in some way. This can be achieved with a separation-of-concerns approach, where we consider security separately to application logic.

In other words, we need to be able to realise a security infrastructure that can operate independently and transparently to the application logic if required. Furthermore, access controls should be enhanced with a notion of *context*, where we take context to be the circumstances surrounding an access (e.g. time of access, sequence of accesses, or the reason for/role of an access). We use knowledge of expected interactions between components in the SoC as an input in the design of security enforcement.

Hence, in this chapter, we build upon the work presented in the previous chapters. Using a design process which re-frames IP blocks as service providers, and crafting protection around service consumption, we propose an architecture for dynamic access controls in heterogeneous MPSoCs. Instead of hierarchical privilege, we make use of the decentralised approach of Chapter 4, re-implementing the IU as an enhancement to NIs.

First, in Section 6.2 we describe another motivating scenario, and illustrate a variation of the threat model we presented in Chapter 3. This leads us to Section 6.3 where we present our proposed architecture and mechanisms. In Section 6.4 we describe our initial work on a design flow which incorporates our security infrastructure. To explore the feasibility of the approach, we perform experimental



(a) Smart Controller (b) IP blocks of varying internal complexity

(c) General NoC architecture

Figure 6.1: Motivating embedded system design scenario, where we want to integrate different IPs for a given application

evaluation, discussing in Section 6.5 the implications of our proposed architecture. In Section 6.6 we position this work in the context of other related works.

6.2 Motivation and Background

6.2.1 Smart Controller

To describe the added changes when integrating complex pre-designed IPs, let us examine another motivating scenario. Consider a smart controller used in an Internet of Industrial Things context (Figure 6.1(a)). The controller needs to bridge various networks, and provides an interface for a user to set preferences that are used in autonomous actuation and control of two industrial zones. The controller also needs to independently manage a Hazard Response System (HRS), which performs the critical function of hazard detection and management. The sensitive sensor data from both zones is stored for archival purposes, where the control tasks periodically make use of a DMA block to store data.

To implement the controller, designers have several IPs at their disposal to use as *nodes* in the MPSoC (Figure 6.1(b)). Designers can incorporate simple nodes, like processing elements, shared memory blocks, or DMA-capable accelerators, as well as more complex IPs, like a pre-designed Sensor Block (to interface Sensor A and Sensor B networks), or an External Interface Block (EIB), which can run a local graphical display and provide Internet connectivity. Complex IPs, like the Sensor Block, can themselves be clusters that integrate other components, such as a wireless modem, DSP hardware, and a full software stack. From the designer's perspective, IPs like the Sensor Block largely appear to be black boxes where only certain things are visible, such as some directly addressable memory-mapped registers for programming its operation, or accessing the data it produces.

In our design, some IPs act as *masters*—they are *IP tasks* that can *issue* memory requests. Other IPs act as *slaves* that can only respond to memory requests; in other words, they *provide* memory-mapped *resources*. Some IPs can act as tasks while simultaneously providing resources, such as a programmable DMA block (where it can issue memory transactions, after being configured through its configuration registers).

To control and coordinate the IPs, designers implement their application logic as concurrent software entities. We will refer to these as *application tasks*. IPs are integrated into a target NoC architecture (Figure 6.1(c)), together with processing elements which execute the application tasks. Directly addressable parts of each IP are mapped into a global address space, and NIs manage the delivery of memory transactions. As part of the design process, we assume that designers make explicit decisions as to which resources are used by each application task. In Figure 6.2 we can see the designer's initial abstract view of the tasks and resources in the smart controller design as a task/resource relationship graph.

6.2.2 Threat Model

We use the threat model presented in Chapter 3, and consider the threat that a task is compromised such that it is made to issue arbitrary memory transactions (A1 from §3.1.3). We assume the worst-case, where a task has full control of a given processing element (akin to privilege escalation). This means that a malicious task can directly corrupt stored data (by corrupting it through spurious writes) or attempt to access sensitive data (and potentially leak it). This also means that a malicious task can manipulate DMA-capable IPs in the MPSoC to perform malicious accesses on its behalf (A2 from §3.1.3).

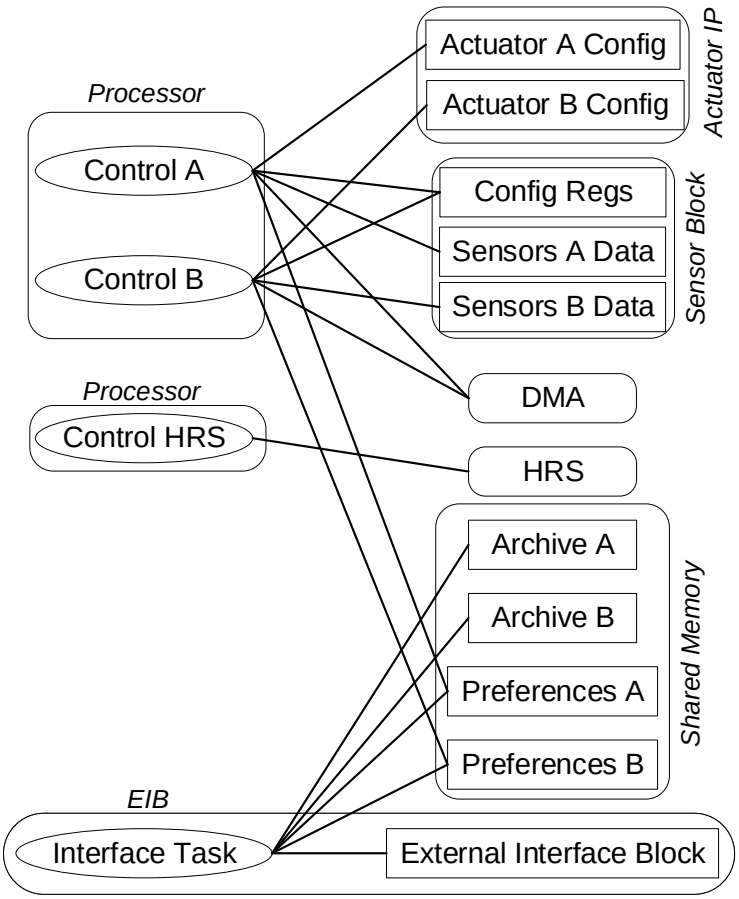


Figure 6.2: Task/Resource relationships for the smart controller example (ellipses represent tasks, rectangles represent memory-mapped regions within resources, rounded rectangles represent physical components in the MPSoC)

However, in addition to the aforementioned threats, the presence of complex IPs adds further potentially hidden threats. Recall, for example, the Sensor Block. The designer makes use of the data it produces (as accessed through *Sensors A Data* and *Sensors B Data* memory buffers), whereas the block actually contains a processor core with a master-side interface to the NoC. It has DMA capabilities (say, for streaming data to shared memory), and while the designer does not use this capability in this system, it remains available for an attacker to abuse.

Initially, we assume that our MPSoC features a general NoC interconnect that allows all-to-all communication. There are several threats that may arise due to resource sharing. Figure 6.3 illustrates some potential attack scenarios (AS) in the smart controller:

AS1: Control A could be compromised, and instead of using the DMA block to archive sensor data, it abuses the DMA to disrupt the HRS operation, manip-

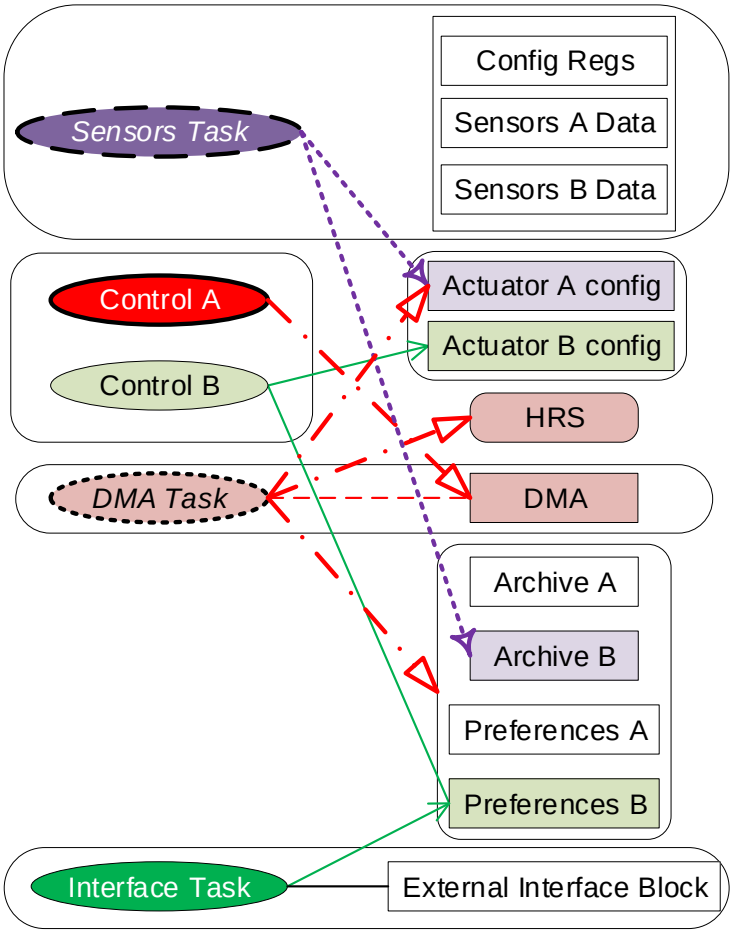


Figure 6.3: Potential attacks in the MPSoC (simplified view)

ulate the actuators, or illicitly access the shared memory. This is a "confused deputy" scenario [40]

AS2: Interface Task could be compromised to change Zone B preferences, thus causing Control B to perform actuation at an undesirable time

AS3: The Sensor Block's processing element (represented as Sensors A Task) could be compromised, and attempt to directly manipulate Actuator A inappropriately, or exfiltrate sensitive data from Zone B

AS4: A compromised task might also be able to cause issues by deliberately misconfiguring a block, such as the case where the DMA block is programmed to do more transfers than expected, potentially causing a denial-of-service for other tasks

Ultimately, there are two main issues that we want to reduce the severity of: the direct impact of a compromised task, and the impact resulting from a compro-

mised task's abuse of components in the MPSoC. We want to prevent these types of attacks from being carried out successfully.

6.2.3 Security Design Challenges

By designing our system with the Principle of Least Privilege [76], we can attempt to provide a security foundation. Potential windows for exploitation can be reduced by ensuring that access to resources is granted only as, and when, required. Critical components can be isolated from potentially vulnerable components when resource sharing is properly managed. Conventionally, we can implement some sort of access controls, prohibiting access between components that do not have any ordinary need for interaction.

As we discussed in Chapter 2, one security approach might involve partitioning of the system into so-called "secure" and "non-secure" parts, and enforcing isolation between them. Traditionally, as in the case of TrustZone [75], this additionally entails a privilege hierarchy, where the secure part can access (and possibly disrupt) the non-secure part. That introduces the problem of deciding how to partition the design appropriately. To alleviate attack AS1, we might say that the HRS and its control software, considering its importance, should be made exclusively *secure*, thus isolating it from the Zone control tasks, as well as the DMA block entirely. However, this opens up the possibility that non-secure parts are able to affect each other, as in the attack AS2 and AS3, or that a malicious design in the secure part can cause havoc across the entire system.

A subtler approach involves considering each application task and its associated resources as their own individual domains. This approach can be more resilient than a hierarchical one, because it does not encourage overarching privilege. By restricting communication in the NoC, we can aim to isolate uncoupled domains entirely. Yet, even if we restrict communication solely to the required links as indicated by the task/resource relationships, the system remains vulnerable to exploits that can cause impacts *across* different domains.

For example, the DMA block requires access to the memory regions which house archival data for A and B. It needs *physical* access (which is provided by the NoC), but its ability to modify *Archive A* or *Archive B* depends on *context*, that is, which task the DMA block is working for. Giving the DMA block access to both A and B opens up the risk of an exploit akin to attack AS1, where a compromised task might try to extend its reach across domains by abusing an IP.

We might also have other requirements. For example, we could desire that Control B's ability to manipulate Actuator B is conditional on contextual information such as *time*, whereby actuation changes should only occur during pre-set operational hours.

Different IPs may have varying levels of "security-awareness"—if an IP cannot be modified to deal with access control directly, an independent protection infrastructure is necessary. Hence, we need some strategy for *dynamic* conditional permissions for access to resources. The design challenge lies in how to specify the system's security requirements, and the implementation of a security infrastructure to support and manage the required dynamic controls, especially given a situation where:

- elements in the design are heterogeneous (e.g. in terms of instruction sets, capabilities, functionality)
- some processing elements have no OS (i.e. the software runs bare-metal) and are thus without any notion of privilege restriction
- an OS (when present) can be compromised (with successful privilege escalation if possible), thus precluding a security strategy that exclusively relies on *privileged* software for managing security
- hardware or software components in the design are "black-boxes", and the designer does not have the ability to make invasive modifications

Our underlying security approach involves identifying and managing the necessary relationships between components in the design. These relationships are used to generate and enforce access controls, as well as dynamic run-time permission enabling or disabling. As such, our proposed approach is not designed to be an exclusive security strategy; we do not explicitly deal with side-channels or prevent any initial means of incursion. Rather, we position this work as a possible foundation in a multiple independent layers of security (MILS) approach, where the architecture provides a fall-back that will guarantee that certain properties are not violated.

In the following section, we describe an approach for framing security specifications, mechanisms for context-aware protections, and a hardware architecture for implementing our security infrastructure.

6.3 Proposed Security Approach

Every memory access in an MPSoC is performed for a reason. Legitimate accesses are well structured; for example, a processor making use of an IP block will perform memory accesses to a set of memory-mapped control registers in a specific order with specific values to set up a specific action. When components in the design attempt to access memory regions that are outside where they usually operate, when accesses are in an unusual order, or when they are performed at unexpected times, we expect the presence of design error or malicious activity. As such, we should check that accesses are allowed by considering some notion of *context*, i.e. that memory accesses are appropriate to what a given application task is supposed to be doing. For this purpose, we propose to re-frame accesses on chip in terms of *service consumption*, and base our protection scheme around this abstraction.

6.3.1 Services in a MPSoC

The service-oriented paradigm provides a flexible abstraction for integrating reusable modules, especially given a well-defined interface [131]. This matches well with using heterogeneous IPs in an MPSoC, so we represent IPs that can perform several different functions as *service providers* that provide multiple *services*. Services are the operations performed by an IP on behalf of application tasks, ranging from simply responding to memory requests (e.g. shared memory), to more complex operations (such as a DMA block performing a sequence of memory transfers). We classify services as *active* when the service provider performs memory accesses, and *passive* when a provider only exposes *resources*.

As an example, let us return to the smart controller. We can describe access to the directly addressable data buffer in the Sensor Block IP as use of the *Read Sensors A data* service. It is *passive*, it does not require any resources or configuration, and it simply provides a readable memory region. Alternatively, configuring the Sensor Block to use its DMA capability to move data to elsewhere in the MPSoC entails consumption of an *active* service.

6.3.2 Protecting Services

With this abstraction we can express security specifications as protection *of services* (from misbehaving tasks), and protection *from service providers* (working on behalf of misbehaving tasks), instead of designing security at a lower level of abstraction. Access restrictions become centred around ensuring that an IP only has

access rights needed to provide a service for a specific task, and that the IP's memory accesses are consistent with the service interface.

We assume that it is the responsibility of designers to conceive the security policy for any given system design. To improve security, we thus propose a way for designers to specify, and then enforce, certain *service-level security specifications*. Essentially, the aim is to regulate the capabilities of application tasks and IP tasks in two ways: access controls for using services, and access controls for service providers.

On the service consumer side (i.e. application tasks), we need to make sure that access to services is only on an as-required basis. Conventionally, this is achieved by specifying static permissions. If a task wants to use an IP, it needs to have access to the corresponding memory-mapped registers of that IP. In this work, we propose the optional addition of *context* to these permissions, specifically, making them conditional on:

- Sequence: where access to a service depends on some pattern of service consumption, or
- External status: such as *time*, where access to a service depends on some global notion of time, or some other attribute, such as authentication or attestation status provided by components elsewhere in the system

In this way, we can reduce the risk of successful AS2-type attacks, where damage is caused by accesses that violate designers' expectations of when certain accesses should occur.

Furthermore, we also need to examine the steps leading up to service use. Take as an example, the DMA block which transfers data from one part of shared memory to another. It has memory-mapped registers for the read and write start addresses, a register for the size to transfer, and a control register for starting the transfer. Correct configuration of the DMA block follows a sequence akin to something like that shown in Figure 6.4. The various configuration "parameters" for the service could be checked for validity, particularly for ensuring that something like a DMA block is not misused. Activation of the service should be contingent on legitimate set up, as a means to prevent abuse (thus reducing the risk of an AS4-type attack).

On the service provider side, we need to make sure that the service provider only has access to the resources required to deliver the service. In some cases, such

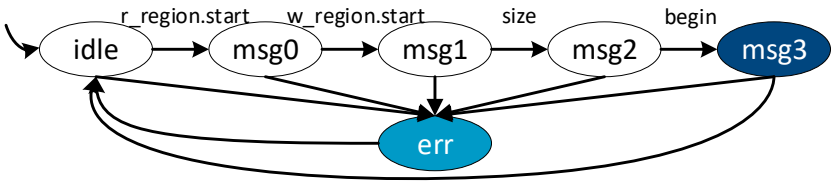


Figure 6.4: Example service configuration: DMA block

as with passive services, the service provider does not need access to anything else in the system. For example, in the smart controller, the Actuator block receives a configuration directly from a Control task, and sends its outputs off-chip.

However, for active services, we should place restrictions on what they can do in the system, particularly to prevent AS1-type attacks where an IP is manipulated to breach security domains. To do so, we can specify a permission *activation* and/or *deactivation* scheme. Security schemes manage the dynamic permissions given to a service provider.

For activation schemes, we propose the following:

- No Check (NC)—the service provider can access all memory
- Fixed-Always Active (F-AA)—permissions are always-enabled based on the T/R relationships associated with the service
- Fixed-On-demand (F-OD)—permissions to fixed regions are enabled only on service request
- Variable-On-demand (V-OD)—as above, but where the protected shared region's location is variable and specified as part of the service request

We also propose three options for deactivation schemes for On-demand permissions:

- Explicit—permissions are disabled when a service is completed (only usable if the requesting task can "acknowledge" service completion)
- Time-limited—permissions are disabled after some time has elapsed
- Message-count—permissions are disabled after a certain number of messages are sent

By specifying and enforcing a mix of security specifications, designers are able to improve the security of the system. For example, we can specify a conditional

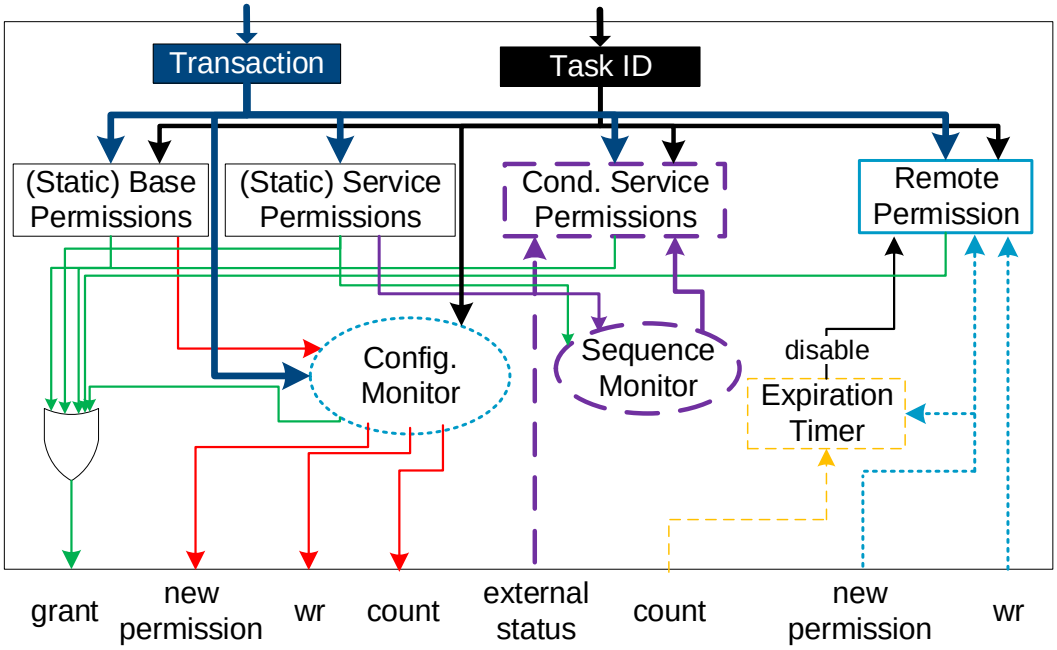


Figure 6.5: Permission Checking Block (PCheck)

permission based on *time* to protect access to the actuators in the smart controller. A compromised task can be stopped from creating malicious outputs at unexpected times. The DMA block can be protected from abuse (as in attack AS1) by specifying that its access permissions are V-OD, thus preventing unfettered access to the overall system. We have the added benefit of being able to detect some forms of attack, by identifying failed permission checking due to unexpected malicious behaviour.

6.3.3 Hardware Support

6.3.3.1 Permission Checking

We turn now to *enforcement*. Static access controls can be implemented by incorporating a firewall or MPU at each network interface (NI), so each memory transaction is checked against a list of permissions. Permissions specify read/write restrictions on given memory regions, and messages are prevented from release into the NoC unless access is granted by a permission.

As an augmentation to static permissions, we propose that dynamic controls and monitoring are also performed in hardware. Our intention is that security enforcement in hardware can be somewhat self-managing, thus insulating the fundamental access controls from rampant compromised software. Accordingly, we add

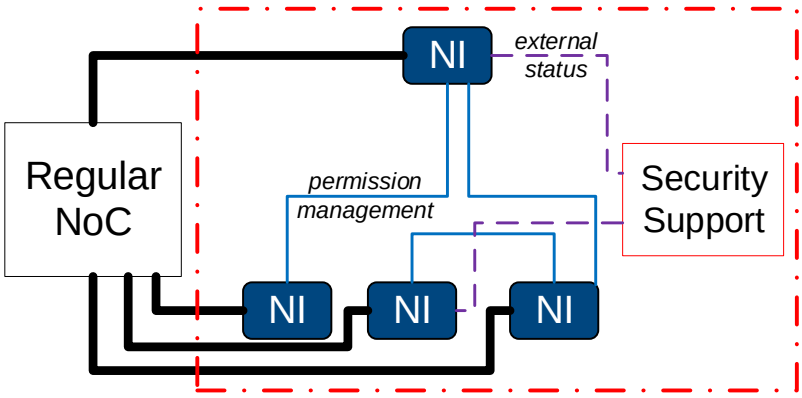


Figure 6.6: Security Infrastructure

a *Permission Checking block* (PCheck) to NIs (Figure 6.5). The PCheck adds IU-like functionality into the NI.

In its most basic form, the PCheck resembles a statically configured MPU; transactions from the local core are compared with static (*Base*) permissions, generating a grant signal if the read/write request is bound for an accepted region. To enforce dynamic parts of the security specification, we propose these additional PCheck customisations (as shown in Figure 6.5):

- an additional block of conditional service permissions, where conditions can be checked inside the PCheck, or fed in via an external interface. An optional sequence monitor which tracks the order of service use can also be added (long dashed components)
- a timer/counter for disabling permissions (medium dashes)
- a configuration monitor for ensuring that services are configured properly, upon which dynamic permissions for the service provider are enabled/disabled (small dashes)
- a remote permission block if dynamic permissions are enabled/disabled by other NIs

External conditions, provided by components that mark some part of system's context, like a counter which records system time, can be housed in a *Security Support* block, as shown in Figure 6.6. The security block is connected to the NIs separately to the regular NoC, and we assume that the components within are trusted.

6.3.3.2 Granting Permissions to Service Providers

As opposed to a centralised or privileged entity (such as a security manager [31]) for changing permissions during run-time, we instead add support for decentralised dynamic permission management through a *protection infrastructure* consisting of interconnected NIs (illustrated in Figure 6.6). These NIs can be interconnected in the same way we proposed in Chapter 4 (Figure 4.6). This is particularly important in the case of active services, as we need to ensure that the service provider receives the necessary permissions when a service is used. We also need to ensure that the service consumer configures the service provider correctly.

In the Fixed-Always Active case, the service provider has a statically configured MPU, which permanently provides the necessary permissions. For On-demand permissions, we need to implement checks on the consumer (to ensure that consumer configuration is correct), and place dynamic controls on the provider. Our idea is to support this directly in hardware, thus avoiding the reliance on privileged software.

On the consumer side, we can use a Configuration Monitor to ensure that services are not abused. For each active service used, we create a finite state machine to check each part of the IP block configuration. Take as an example, the DMA block, which has a configuration process as illustrated in Figure 6.4. Each step involves a write to a memory-mapped register, and we can check that the data for each write conforms to what the designer expects. Enforcing correct configuration thwarts abuse of the service providers.

For this approach to work, we adopt the notion that service providers operate on data that logically "belongs" to the service consumer. Hence, to prevent attacks where a compromised task tries to extend its reach, we give designers the responsibility of assigning regions of the address space to each application task. These regions are the parts of memory that an application task has authority to *share* with other entities in the system, and are represented in a *Base* permission in the PCheck. As the configuration process occurs, the Configuration Monitor checks that the region(s) being sent to the service provider lie within the allowed region(s). This stops malicious entities from successfully breaching security domains. Other aspects of the configuration (such as transfer size in the case of the DMA block), can also be checked by the monitor.

If the configuration is successful, the corresponding permission can be enabled at the service provider's NI. If the application task tries to configure the IP incorrectly, the monitor goes to an *error* state, and indicates a security violation. These operations are handled directly by the PCheck, so in this way, customised NIs can

manage dynamic permissions even if an IP is not "security-enabled". The complexity of the monitor can be relatively low, considering that it consists mostly of comparators and the number of checking states is directly proportional to the number of control registers to write (for example, a typical Scatter-Gather DMA block has 8 memory-mapped registers, a PLL block has 3, and only a subset of these registers are used for configuring the IP's behaviour [52]).

6.3.3.3 Task Identification and Location of Checks

Enhancing NIs with the PCheck introduces protection at node-level granularity. For increased granularity, a Task ID can be checked if multiple tasks are resident on the same node, although this does require the IP to be more security-aware. Strategies such as program counter-based access control [63] may be useful for task identification, and we assume that any adopted technique to identify tasks is sufficiently trustworthy.

Additionally, where traditional service-oriented approaches in the Internet feature permission checks at the service provider, our scheme involves checks at *both* consumer and provider to address our threat model. This has the added benefit of providing effective protection even when NoC protocols allow parameters such as packet headers to be forged, or where packet origins are not easily determined. Preventing the injection of packets also helps reduce the risk of spurious traffic in the NoC, which could lead to congestion (and Denial-of-Service as a result).

6.4 Integration and Customisation

In order for our proposed security approach to be effective, the protection infrastructure should be closely aligned to the embedded systems application. Security should be considered as part of the design process, so in this section we present a top-down system design flow that integrates customisation of the protection infrastructure with the customisation of the execution platform for a specific application. This design approach makes further use of our proposed service-level abstraction.

6.4.1 Approach Overview

An overview of our proposed design flow is presented in Figure 6.7, with corresponding details in Figure 6.8.

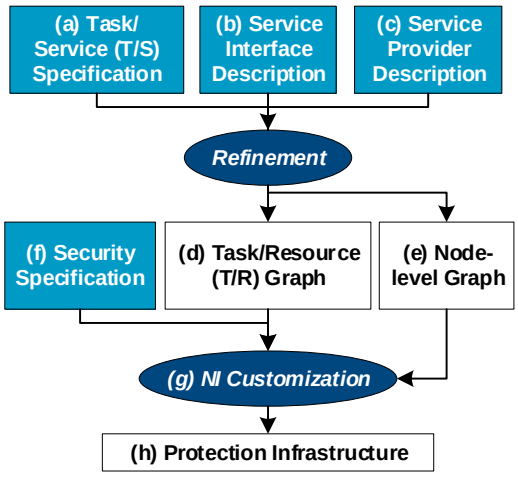


Figure 6.7: Our proposed design flow

First, designers specify the application in a high level Task/ Service (T/S) specification. The T/S specification represents all the application tasks in the system, and the services that each task consumes. These services are drawn from a library of services, where IP designers have specified a set of *Service Interface Descriptions* which indicate the resources and tasks that are involved in the service. Corresponding *Service Provider Descriptions* indicate the IP tasks and resources that each potential IP block provides. Initially, this specification is abstract, and is independent of any service providers.

This specification undergoes *refinement* to produce a lower level Task/Resource (T/R) graph, and a Node-level graph. The T/R graph represents the read/write relationships between *application tasks*, *IP tasks*, and all the memory-mapped *resources* in the system. The Node-level graph represents the required physical links between the IPs in the design. Refinement uses the service interface descriptions for each IP, as well as a logical description of the visible components of the IP.

To generate the protection infrastructure, designers also create a security specification at the service-level, indicating where a service use is conditional, as well as a security scheme for enabling/disabling permissions for the *IP task* that provides a given *active* service. The security specification is then used in conjunction with the T/R graph and Node-level graph for *NI customisation* to produce the customised protection infrastructure.

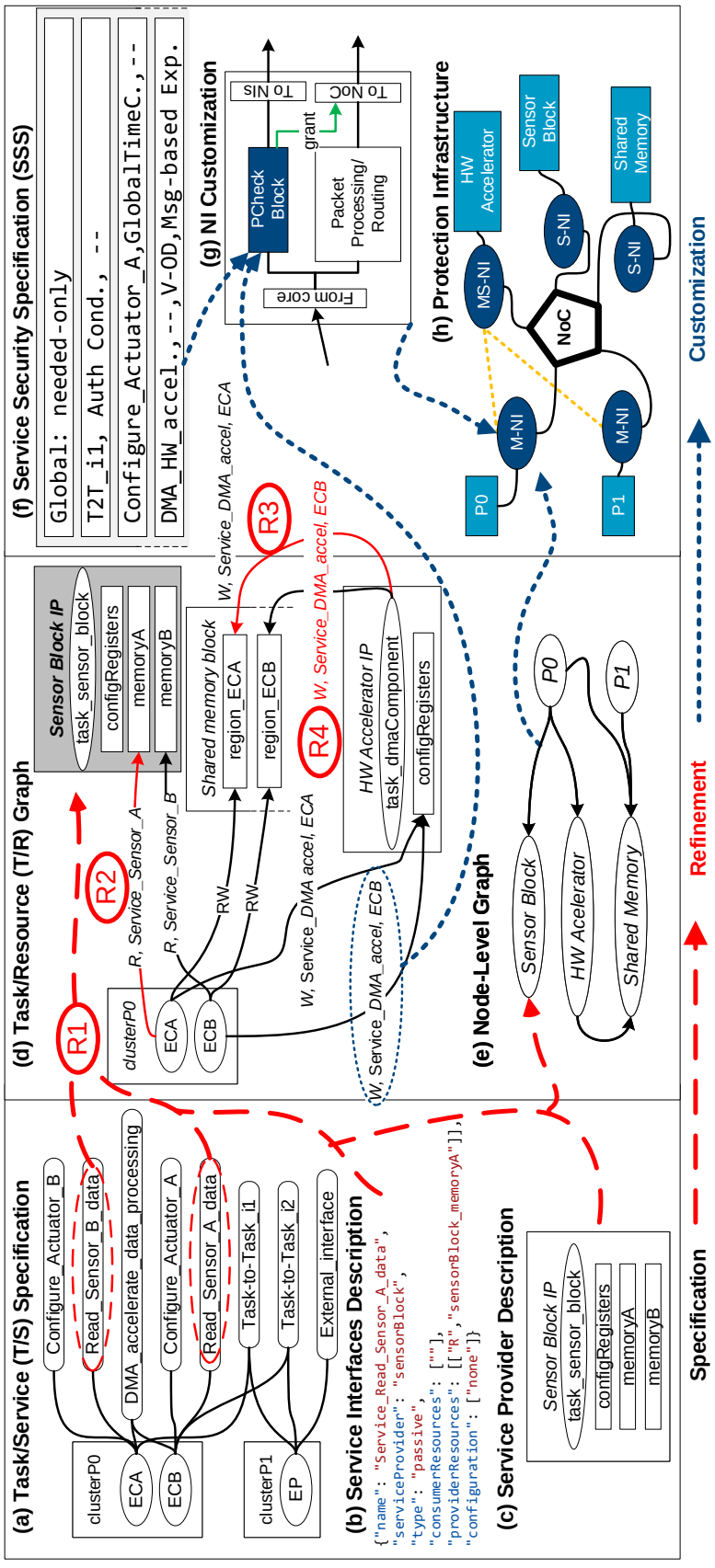


Figure 6.8: A detailed overview: wide-dashed arrows represent *Refinement*, short-dashed arrows represent *Customisation*

6.4.2 Refinement

Using the T/S specification, we perform *refinement* to create a T/R graph, which represents the read/write relationships between *application tasks*, *IP tasks*, and memory-mapped *resources* in the system.

Refinement has two purposes: the first is to identify the different service providers that are needed for a given design, and the second is to identify which resources are accessed in the context of using/performing a certain service. To create the T/R graph, we perform the following steps (labels correspond to Figure 6.8):

R1: Replace each service node in the T/S specification with a cluster containing the service provider's task and resource nodes

R2: Connect each application task with the resource nodes of the service provider, as specified by the service interface

R3: If an *active* service is used, connect the IP task in the service provider to the resources provided by the requesting task

R4: Label each edge with the nature of the relationship between task and resource (read/write), the service name, and service user

These steps are outlined in pseudocode as Algorithm 4.

In the case where there are multiple options of service provider, designers choose their own criteria for deciding which provider to employ (using criteria such as resource consumption or power consumption to perform design trade-off). The resulting T/R graph can also be analysed to see where security domains overlap. Designers can identify where resources are shared by performing reachability analysis (in other words, Impact Analysis §3.3.2) between application tasks, and if necessary, use multiple instances of a service provider to reduce sharing (and increase isolation).

To complete the refinement process, we also perform some further housekeeping. In our prototype realization of the refinement process, we add a resource node to represent the private memory area for each application task. Application tasks "own" this memory region, and resources provided to service providers for services can be drawn from here. Resource nodes are also added to allow application tasks to communicate; these buffers are then grouped into a shared memory cluster. A node-level graph is generated by examining the connectivity between the clusters in the T/R graph (Figure 6.8(e)).

Algorithm 4: Refinement Algorithm

```
Input: List of Tasks: taskList[], Task-Service Specification: tss[][], Service
       Interfaces Description: sid[], Library of Service Provider
       Descriptions lib_spd[]
Output: Task-Resource Matrix trm[][]
Current Service Provider List csp[];
/* First add Service Providers to the trm                          */
for service in tss do
    service_provider servpro = sid.findProvider(service);
    /* if there are multiple options for providers, designers
       use their own selection algorithm and procedures for
       replicating providers                                       */
    if servpro not in csp then
        csp.add(servpro.name);     // keep track of added providers
        /* Add the service provider's T/R nodes to the
           task/resource graph                                     */
        for resource in lib_spd[servpro.name] do
            trm.addColumn(resource.name);
        end
        if servpro is active then
            for IP_task in lib_spd[servpro.name] do
                trm.addRow(IP_task.name);
            end
        end
    end
end
/* Now connect consumers and providers                             */
for task in taskList do
    for service in tss[task.name][] do
        find service in sid to get service provider;
        for resource in sid[service.name] do
            trm.addConnection(task,resource);
        end
        if service is active then
            trm.addConnection(IP_task, resources provided by consumer);
        end
    end
end
```

Refinement reveals where there are IP tasks that are unused. Notice how the Sensor Block (shaded box) in Figure 6.8(d) contains a task node and a resource node that are disconnected. Even though these components are not used in the controller application, they are still present when the IP is integrated in the NoC.

6.4.3 NI Customisation

The final piece in our design flow is *customisation* of the protection infrastructure. First, we look at the Node-level graph (Figure 6.8(e)) to identify which IPs communicate, and customise each NI to prevent communication between nodes that do not need to communicate (such as by removing routing table entries). Where nodes have outgoing edges, we add a master-side NI (which supports transaction initiators), and add slave-side NIs where nodes have incoming edges. This prevents IPs that only offer passive services from *initiating* malicious memory accesses, as in the AS3-type attack. Each master-side NI is then configured with a custom PCheck based on the application's security specification.

The labelled edges of the T/R graph form the basis for populating the permission blocks of each PCheck. Edges to each task's private memory area are used to create the *static base permissions*, and edges that are associated with services that have no access conditions generate *static service permissions*.

Then, we customise NIs to support dynamic permissions. For each edge in the T/R graph that is labelled with a service that has access conditions, we add a permission to the *conditional service permissions* block, and expose an interface to receive external conditions (such as the current time-stamp) as required. If an application task uses an active service, we also create a *Configuration Monitor* to check that the service configuration is as expected.

6.5 Experimental Evaluation and Discussion

To investigate the feasibility of our approach, we implemented prototype tools for refinement as well as synthesisable master-side NIs for a NoC based on a multi-stage butterfly architecture (such as that described in [107]). We described the T/S specification of the smart controller example using the DOT language, described IP service interfaces in JSON, and implemented automated refinement in a python script with GraphViz [37]. Examples of how we represent these can be found in Appendix C. The generated T/R graph was used to create the different customised NIs, and these were synthesised with Altera Quartus 15.0 for a Cyclone IV target to

Table 6.1: Synthesis Results for various components (NI at an application task cluster (P0), and an NI at a service provider (DMA-IP) of varying complexities, and a Nios II processor)

NI variant	Resource Use/LEs	% overhead c.f. NoC	Max. freq/MHz
NI@P0: Basic	220	13.4	328.7
NI@P0: No access condition	242	14.8	193.8
NI@P0: F-OD	427	26.1	150.3
NI@P0: V-OD	584	35.7	117.5
NI@DMA-IP: F-AA	216	13.2	243.7
NI@DMA-IP: F-OD	216	13.2	240.7
NI@DMA-IP: V-OD	481	29.4	141.3
Nios II/f	2053 (ALUTs)	125.6	150 [53]
Nios II/f with MPU	2603 (ALUTs)	159.2	150 [53]

get estimates of resource consumption and maximum frequency. Some synthesis results are presented in Table 6.1.

To get a sense of the relative complexity of the NIs, we also present the synthesis results of a Nios II/f processor (generation 2) from the MPU design example [51], with a 4K instruction cache, 2K data cache, and MPU with 4 regions. We also add a performance counter to the Nios II to measure the cycle overhead for MPU-related operations. These synthesis results are compared against an 8-node NoC consisting of twelve 2-port switches, which requires 1635 LEs.

To evaluate the security efficacy, we performed cycle-accurate simulation of the RTL using Modelsim 10.4 to validate that no unexpected memory accesses were possible despite spurious memory accesses. In a generic NoC with no security customisations, all nodes can access all other nodes. When the F-AA scheme is used for the NI@DMA-IP, the DMA-IP can be commandeered to manipulate all regions of memory it has access to (AS1-type attack). The window of exploitation is reduced by using an On-Demand scheme, thus preventing crossover of security domains through the abused IP.

The basic NI contains a routing table and a simple controller for injecting packets into the NoC. The NI@P0 contains nine permissions for access to various services. The NI@DMA-IP contains four permissions in the F-AA case, and two dynamic permissions in the OD variants. As expected, increasing complexity results in greater resource use, especially when a configuration monitor is introduced for the on-demand activation scheme. As soon as permission checks are introduced for

Table 6.2: Typical Overheads of Check/Reconfiguration Operations

Operation	Typical Overhead (Clock Cycles)
Software Permission Check, Nios II/f	~32 per region
MPU Enable/Disable, Nios II/f	9
Permission Reconfiguration, Nios II/f	~50–90 per region
Permission Check, Enhanced NI	1
Permission Reconfiguration, NI	1

P0, the estimated maximum frequency drops significantly due to several combinational checks being added to the critical path, and 10% additional logic elements are required. In the fixed-scheme cases for both the NI@P0 and NI@DMA-IP, statically configured regions allow optimization for memory address boundary comparisons; hence the V-OD scheme for the NI@DMA-IP incurs a greater relative resource cost because comparators for an entire 32-bit address are required.

Because the various permissions are implemented directly in the PCheck hardware, and all permissions are checked in parallel, permission checks incur only a single cycle overhead, which is significantly lower than the ~32 cycles required to implement comparable permission checks in software. When using a point-to-point connection between NIs that provide shared permissions, only a single cycle is required at run-time for activating the temporary permissions. This will vary depending on how designers choose to interconnect the NIs. As a comparison, the Nios II MPU requires that the MPU first be disabled, new permission regions then loaded, before the MPU is re-enabled. This is *on top* of any context switching overhead required when an OS is used. Of course, general MPUs provide a lot of flexibility, as our NIs are mostly pre-configured with static permissions, but we show that application-specific customisation is possible.

These results also indicate that greater flexibility in the NIs entails greater resource cost; in order to support variable memory regions, or fully dynamic permissions (as opposed to simply disabling/enabling statically determined ones), PChecks need to become more complex. However, this provides opportunities for design space exploration. In our proposed flow, all tasks and IPs are known in advance. One way in which we can "measure" security improvement is to perform reachability analysis in the T/R graph; with our flow, tasks should only be able to affect resources that are needed as part of the application logic. If we can completely trust different components, we may be able to relax the global policy, and

have some un-regulated paths in the design to save on resource costs. This type of design space exploration presents some potential future work.

6.6 Related Works

Security in embedded system design is a fast evolving arena, and there are numerous approaches for improving the security of MPSoC designs. Access controls as part of a security foundation have long been proposed [76], and many recent works have looked at firewalls or memory protection units (MPU) as the main feature of a protection infrastructure. Another related approach includes ARM's TrustZone [75], which provides physical isolation by coarse separation of the SoC into secure and non-secure worlds, with limitations in protection granularity. The recently proposed work in [13] provides a security architecture for embedded devices centred around an MPU with enforcement linked to a processor's program counter, but it is not extended to a heterogeneous multicore context.

Permission checking at the NI has been a popular area for exploration, and various solutions have been proposed [31, 36, 73, 97, 113]. Interestingly, several of these approaches rely on some central entity for run-time configuration. The Data Protection Units of [31] are managed by a central Network Security Manager IP, where cores that wish to activate or release protection regions need to be in a *privileged* state. A centralised *integrity core* stores access rules in [73], and updates local firewalls as required, such as when an access request is initially denied. Firewalls in [36] take a more distributed approach, where each firewall has an Operating Mode Controller to update rules locally, based on requests from the local CPU. Several Local Trusted Agents and a Global Trusted Agent (GTA) are used in [97] to manage MPU configurations; these software entities are implemented by privileged software, such as hypervisors or operating systems, with the GTA executing on its own *dedicated* core in the NoC. The approach in [113] features a security manager task running on a secure processor to compute the rules for populating the security tables of each firewall, hence facilitating risk aware routing to avoid nodes which may have been compromised.

A common thread in these works is the requirement that there are privileged entities in the system, or that IPs are explicitly aware of security (so that they are able to appeal to a central manager for run-time permission modifications). In contrast to these works, we propose an approach that accommodates software and hardware that are not necessarily security-enabled. Unlike the aforementioned works, we do

not need a hierarchical notion of privilege, and accommodate DMA-capable IPs, as well as decentralised run-time permission enabling/disabling. While the work presented in [124] proposes a decentralised approach to isolate different security domains, it relies on individual tasks to be explicitly security-aware. Our proposed security infrastructure is able to manage the permissions in a transparent fashion.

Furthermore, these aforementioned works also present a general architecture without much detail on how designers might align the security approach to their specific application in terms of specifying security rules or customizing the architecture. We aim to address this gap by proposing a design flow that starts with a high-level specification inspired by the emerging service-oriented paradigm in an MPSoC context (recently explored in [131] as part of supporting run-time FPGA dynamic reconfiguration). The specification is then systematically used in customisation of NIs to form the protection infrastructure.

Another related work is presented in [47], where a high-level specification is used to generate the firewalls for a specific application. The aim of this work is to optimise resource usage by deciding which levels of firewall complexity is needed for each link. Similarly, [123] provides an automated approach to generate potential protection options, but does not deal with permissions that can change at runtime. A type of reachability analysis is used to compare the relative security of each option, and this could be applied to our proposed approach. Another high-level approach is proposed in [101], which explores detection and isolation of malicious third party IPs, as opposed to customizing access controls. Our proposed approach also resembles control-flow checking, in that we monitor the sequence of memory accesses during service configuration. In MPSoCs, the approach proposed in [91] adds a specialised processor, and requires instrumentation of executing code to determine checkpoints for monitoring purposes. Our approach has considerably lower granularity, but carries the benefit of compatibility where custom instructions cannot be added to processors, or where task program code is not easily modified.

6.7 Summary

In this chapter, we have presented an approach for customizing the protection architecture of an MPSoC for improved security. As future embedded systems tend towards integration of many different behaviours into a single NoC-based design, we need security approaches that can be used even when cores are heterogeneous, or

not security enabled. To improve security, we presented the notion of context into hardware-enforced permissions, allowing designers to check that memory accesses happen when expected. We specify security constraints as enforcing access conditions for services and adding restrictions to the capabilities of service providers, and use this specification to add customised support for dynamic permissions. With this approach, we can reduce the risk from integrating heterogeneous IPs into a NoC-based design. In adopting a novel service-level abstraction to capture the capabilities of IP blocks, we proposed a top-down design approach to accommodate complex IPs.

CHAPTER 7

Conclusions and Future Work

7.1 Summary and Contributions

Security is a lofty, but worthy aim. In embedded systems there can be many different potential attack vectors, and many different security requirements. Every application has its own set of assets, and its own threat vectors, and as such, there is a need for strategies that can better match security mechanisms with the specific needs of a design.

In this thesis, we have examined and proposed solutions to security issues in heterogeneous multiprocessor systems on chip. Challenges in the ever evolving field of MPSoC design include dealing with risks from integrating concurrent behaviours in a single platform, integration of security into the design flow, and attack mitigation given different security domains. From our examination of the literature in Chapter 2, we identified opportunities for building security into the hardware platform. Our novel security techniques moved away from traditional hierarchical approaches for security enforcement, and we developed a system-level view of security to better facilitate the specification of security rules, and automated security infrastructure exploration. Over the course of this thesis, we presented the following contributions:

A system-level security-aware approach for MPSoC design

In Chapter 3 we presented a novel security model for MPSoCs focussed on the relationship between application tasks and resources in a design. After considering several different access control models, and their relevance to complex MPSoCs,

we identified features useful for supporting isolation of different tasks in a design. This led us towards the need for static and dynamic permissions as expressions of task/resource relationships. We proposed a decentralised approach for permission management, where tasks "own" parts of the address space. Instead of a privileged entity with access to, and control of, all resources in a system, we examined a design by considering each task and its relationships in turn. Tasks then share parts of their own regions with one another as needed. This led us to define *Impact Analysis*, which gives designers a measure of attack impact. Impact Analysis was then used in Chapter 5 as a means to evaluate whether a given design satisfies a given set of rules. Our security approach was then applied to the higher service-level abstraction in Chapter 6, where we formed permissions based on the idea of using services provided by IPs. We proposed the use of a mix of static and on-demand schemes for managing access to services, and applied our security model to allow safer sharing of DMA-capable IP blocks.

Hardware support for decentralised and dynamic access control

Once we presented our security approach, we set about developing the security infrastructure. As part of our security approach, we developed mechanisms for sharing libraries, memory regions, and shared IP blocks with direct memory access capabilities. In Chapter 4 we described the *Isolation Unit* (IU) as a dedicated memory protection block that does not require an OS for run-time customisation. We looked at different ways of interconnecting IUs to allow dynamic access permissions between different tasks. We then packaged IU functionality in a Permission Checking (PCheck) block in Chapter 6, which can be integrated into a Network Interface (NI). The PCheck block contained several optional hardware-based monitors for checking that IPs were used correctly, and that IPs were provisioned with the necessary shared permissions, even if the application task was not security-aware. We characterised our hardware implementations by finding their resource cost when synthesised for an FPGA-based execution platform, and examined, in simulation, the possible run-time overhead when used in a TDMA-MIN NoC design.

Systematic and automated design methodologies

Using our hardware-based mechanisms as building blocks, we presented novel system-level design flows to quantify and improve security in Chapter 5. We proposed a systematic approach to generate several MPSoC architecture configura-

tions, and then evaluate these options against designer-defined security rules. We then adopted a service-level abstraction in Chapter 6, which allowed us to better encapsulate complex DMA-capable IP clusters. Using this service-level approach we created a design flow that involves conversion of a Task/Service Specification into a Task/Resource graph, and then enhancement of each NI as required to generate a customised protection infrastructure.

With the conceptual work we presented in this thesis, designers are able to more systematically design MPSoCs with security as a design goal. We demonstrated that hardware-supported decentralised dynamic controls were possible, incurring modest resource overhead, especially when customised at design-time to the specific application.

7.2 Future Work

One exciting aspect about the research presented in this thesis are the opportunities that have been revealed for future work. Some areas for future research could include:

- Design exploration – Additional metrics, such as power consumption or performance, could be integrated with our security-aware design approaches. Future research would involve quantifying the trade-off between security and other design objectives, such as examining how varying the amount of resource duplication (to decrease Impact Profiles) affects these other objectives. Part of this future work would involve mathematical formalisation of the design problem, and this can be built on the foundations laid by the task/resource relationship model.

- Integration and interaction with other security approaches – An interesting area of research is in new techniques for integrating a variety of security mechanisms, such as how we could use our isolation approaches with other detection and recovery strategies. This also includes management of potential side-channel issues, which were not directly addressed in this work.

- Design flows – High-level synthesis (HLS) techniques provide a pathway from high-level application specification to hardware implementation, and further work could involve investigating how our service-level approach could be used to protect MPSoCs that are generated though HLS. Furthermore, integration of explicit security specifications alongside system-level

design in a language-based approach could be explored as a means to improve the overall design flow.

- Dynamic tasks – Further hardware support is needed for accommodating dynamic tasks, and more work could be done in safely allowing service discovery and sharing alongside the static tasks in a design.

Appendices

APPENDIX A

List of Publications

Published:

- Benjamin Tan, Morteza Biglari-Abhari, and Zoran Salcic, "A system-level security approach for heterogeneous MPSoCs," 2016 Conference on Design and Architectures for Signal and Image Processing (DASIP), Rennes, 2016, pp. 74-81. doi: 10.1109/DASIP.2016.7853800
- Benjamin Tan, Morteza Biglari-Abhari, and Zoran Salcic, "Towards decentralized system-level security for MPSoC-based embedded applications", In Journal of Systems Architecture, Volume 80, 2017, Pages 41-55, ISSN 1383-7621 doi: 10.1016/j.sysarc.2017.09.001.
- Benjamin Tan, Morteza Biglari-Abhari, and Zoran Salcic. 2017. "An Automated Security-Aware Approach for Design of Embedded Systems on MPSoC". ACM Trans. Embed. Comput. Syst. 16, 5s, Article 143 (September 2017), 20 pages. doi: 10.1145/3126553

Under preparation:

- Benjamin Tan, Morteza Biglari-Abhari, and Zoran Salcic, "Towards Context-Aware Service Protection for NoC-based Heterogeneous MPSoCs"

APPENDIX B

SHCS Representations

To process the Task/Resource representations of our applications, we can describe them as a matrix or the equivalent bipartite graph. In Chapter 5 we described the Smart Home Control System (SHCS) scenario (§5.4), and the task/resource relationship is described in a matrix (Table 5.1). The matrix is easily stored as a *comma separated values* (.csv) file, shown in Figure B.1.

For processing using the GraphViz tool [37] (which is useful for things like generating diagrams), we can represent the application using the DOT language. We wrote a python script to convert between the matrix and graph representations, and an example of the output produced is shown in Figure B.3. This example features the resources re-named following Cluster Method 1 (§5.5.2).

We also describe the rules used in Rule Checking (§5.6) in a .csv, and the Optimistic Rule-set is shown in Figure B.2.

```
1  ,,,light_sensor,light_data,light_cfg,lights,temp_sensor,
temp_data,temp_cfg,temp_ctrl,display_data,ethernet,password,
control_panel_ss,fire_sensor,alarm,photos,media_data,
accelerator_cfg,audio_system,request
2  ,,,A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,REQ
3  ,,,p,,h,ph,p,,h,ph,,p,c,p,cph,cph,,,p,p,
4  light_sense,T0,,RW,W,,,,,,,,,,,,,,,,
5  light_manage,T1,,,R,R,RW,,,,,,,,,,,,,,
6  temp_sense,T2,,,,,,RW,W,,,,,,,,,,,,
7  temp_manage,T3,,,,,,,R,R,RW,,,,,,,,,,
8  user_remote_status,T4,o,,Rd,,,,Rd,,,RW,RW,,,,,,,,W
9  user_local_manage,T5,o,,,RWd,,,,RWd,,,,,RW,,,,,,,W
10 fire_detect,T6,c,,,,,,,,,,,,RW,RW,,,,,
11 media_player,T7,,,,,,,,,,,,RW,,,RW,RW,RW,,
12 media_accelerator,T8,,,,,,,,,,,,,,,,,RW,RW,RW,
13 helper,T9,,,,,,,,,,,,R,,,,,,,R
```

Figure B.1: the SHCS example represented in a CSV file

```
1  1,user_remote_status,media_accelerator
2  1,fire_detect,media_accelerator
3  1,fire_detect,light_sense
4  1,fire_detect,light_manage
5  1,fire_detect,user_remote_status
6  1,fire_detect,temp_sense
7  1,fire_detect,temp_manage
8  1,fire_detect,helper
9  1,fire_detect,user_local_manage
10 1,fire_detect,media_player
11 2,user_local_manage,ph
12 2,user_remote_status,h
13 1,light_manage,user_remote_status
14 2,helper,h
15 3,temp_manage,display_data
16 4,media_player,xxxx
17 5,password,helper
```

Figure B.2: Optimistic Rules represented in a CSV file

```
digraph SHCS {
    rankdir = LR
    ratio="3:4"
    light_sense;
    light_manage;
    temp_sense;
    temp_manage;
    user_remote_status;
    user_local_manage;
    fire_detect;
    media_player;
    media_accelerator;
    helper;
    physicalNode_A[shape=box];
    physicalNode_D[shape=box];
    physicalNode_E[shape=box];
    physicalNode_H[shape=box];
    physicalNode_J[shape=box];
    physicalNode_L[shape=box];
    physicalNode_M[shape=box];
    physicalNode_N[shape=box];
    physicalNode_Q[shape=box];
    physicalNode_R[shape=box];
    cluster1[shape=box];
    cluster2[shape=box];
    cluster5[shape=box];
    cluster6[shape=box];
    cluster8[shape=box];
    cluster10[shape=box];
    cluster14[shape=box];
    cluster15[shape=box];
    cluster18[shape=box];
    light_sense -> physicalNode_A[label="RW"];
    light_sense -> cluster1[label="RW"];
    light_manage -> physicalNode_D[label="RW"];
    light_manage -> cluster1[label="RW"];
    light_manage -> cluster2[label="RW"];
    temp_sense -> physicalNode_E[label="RW"];
    temp_sense -> cluster5[label="RW"];
    temp_manage -> physicalNode_H[label="RW"];
    temp_manage -> cluster5[label="RW"];
    temp_manage -> cluster6[label="RW"];
    user_remote_status -> physicalNode_J[label="RW"];
    user_remote_status -> cluster1[label="RW"];
    user_remote_status -> cluster5[label="RW"];
    user_remote_status -> cluster8[label="RW"];
    user_remote_status -> cluster18[label="RW"];
    user_local_manage -> physicalNode_L[label="RW"];
    user_local_manage -> cluster2[label="RW"];
    user_local_manage -> cluster6[label="RW"];
    user_local_manage -> cluster18[label="RW"];
    fire_detect -> physicalNode_M[label="RW"];
    fire_detect -> physicalNode_N[label="RW"];
    media_player -> physicalNode_L[label="RW"];
    media_player -> physicalNode_Q[label="RW"];
    media_player -> cluster14[label="RW"];
// (continued)
    media_player -> cluster15[label="RW"];
    media_accelerator -> physicalNode_Q[label="RW"];
    media_accelerator -> physicalNode_R[label="RW"];
    media_accelerator -> cluster15[label="RW"];
    helper -> cluster10[label="RW"];
    helper -> cluster18[label="RW"];
}
```

Allocations Generated by our tool (Option 0)

```
physicalNode_A, connectivity: 1, contents: 0(light_sensor),
physicalNode_D, connectivity: 1, contents: 3(lights),
physicalNode_E, connectivity: 1, contents: 4(temp_sensor),
physicalNode_H, connectivity: 1, contents: 7(temp_ctrl),
physicalNode_J, connectivity: 1, contents: 9(ethernet),
physicalNode_L, connectivity: 2, contents:
11(control_panel_ss),
physicalNode_M, connectivity: 1, contents: 12(fire_sensor),
physicalNode_N, connectivity: 1, contents: 13(alarm),
physicalNode_Q, connectivity: 2, contents:
16(accelerator_cfg),
physicalNode_R, connectivity: 1, contents:
17(audio_system),
cluster1, connectivity: 3, contents: 1(light_data),
cluster2, connectivity: 2, contents: 2(light_cfg),
cluster5, connectivity: 3, contents: 5(temp_data),
cluster6, connectivity: 2, contents: 6(temp_cfg),
cluster8, connectivity: 1, contents: 8(display_data),
cluster10, connectivity: 1, contents: 10(password),
cluster14, connectivity: 1, contents: 14(photos),
cluster15, connectivity: 2, contents: 15(media_data),
cluster18, connectivity: 3, contents: 18(request),
```

Figure B.3: the SHCS example represented in the DOT language

APPENDIX C

Example Service Descriptions

As part of our exploration into the feasibility of automating the design flow, we came up with a means to represent the services provided by an IP block. It is our expectation that the IP vendor would provide the necessary descriptions of the IPs they create, although designers can create their own descriptions based on their expectations of how to use a given IP. In Figure C.1 we show how we can represent the sensor block's capabilities using a JSON representation (as discussed in Chapter 6).

The Service Provider Descriptions, which are used as an input in the Refinement step (§6.4.2), provide details about *visible* elements of an IP block. Such elements include the memory-mapped regions that are accessible by application software. Designers can add *task* nodes to represent DMA capabilities. An example of how we represent this in our prototype implementation is shown in Figure C.2, where we describe the Service Provider in the DOT language (a part of GraphViz [37]).

```
[{  "serviceProvider": "sensorBlock",
    "consumerResources": [""],
    "type": "passive",
    "name": "Service_Read_Sensor_A_data",
    "providerResources": [["R","resource_sensorBlock_memoryA"]],
    "configuration": ["none"]
}
,
{   "serviceProvider": "sensorBlock",
    "consumerResources": [""],
    "type": "passive",
    "name": "Service_Read_Sensor_B_data",
    "providerResources": [["R","resource_sensorBlock_memoryB"]]
}
,
{   "name": "Service_Stream_Sensor_A_data",
    "type": "active",
    "serviceProvider": "sensorBlock",
    "consumerResources": [["W","region"]],
    "providerResources": [["RW","resource_sensorBlock_configRegisters"]],
    "configuration": [["W","base","region.start"],["W","base+4","size"],["W","base+8","00000001"]]
}
,
{   "serviceProvider": "sensorBlock",
    "consumerResources": [["W","region"]],
    "type": "active",
    "name": "Service_Stream_Sensor_B_data",
    "providerResources": [["RW","resource_sensorBlock_configRegisters"]]
}
,
{   "serviceProvider": "sensorBlock",
    "consumerResources": [["W","region"]],
    "type": "active",
    "name": "Service_Stream_Sensor_A_data_processed",
    "providerResources": [["RW","resource_sensorBlock_configRegisters"]]
}
,
{   "serviceProvider": "sensorBlock",
    "consumerResources": [["W","region"]],
    "type": "active",
    "name": "Service_Stream_Sensor_B_data_processed",
    "providerResources": [["RW","resource_sensorBlock_configRegisters"]]
}
]
```

Figure C.1: Examples of Service Interface Descriptions, in JSON

```
graph sensorBlock {
    node [shape="hexagon"]
    task_sensorBlock_Processor;
    resource_sensorBlock_configRegisters;
    resource_sensorBlock_memoryA;
    resource_sensorBlock_memoryB;
}
```

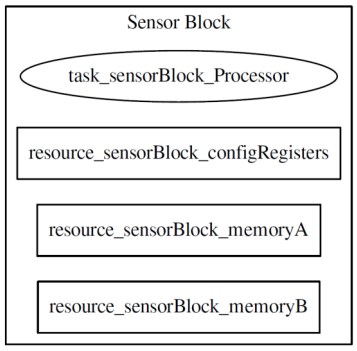


Figure C.2: An Example Service Provider Description, in DOT (left) and graphically (right)

References

[1] A. Algaith, I. Gashi, B. Sobesto, M. Cukier, S. Haxhijaha, and G. Bajrami. "Comparing Detection Capabilities of AntiVirus Products: An Empirical Study with Different Versions of Products from the Same Vendors". In: *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*. June 2016, pp. 48–53. DOI: 10.1109/DSN-W.2016.45.

[2] M. Anderson. *Black Hat 2014 Hacking the Smart Car*. URL: http://spectrum.ieee.org/cars-that-think/transportation/systems/black-hat-2014-hacking-the-smart-car.

[3] R. Anderson. *Security engineering*. 2. ed. Indianapolis, Ind: Wiley, 2008. ISBN: 9780470068526.

[4] ARM. *Memory Protection Unit (MPU) Version 1.0*. URL: https://static.docs.arm.com/100699/0100/armv8m_architecture_memory_protection_unit_100699_0100_00_en.pdf.

[5] A. W. Atamli and A. Martin. "Threat-Based Security Analysis for the Internet of Things". In: *2014 International Workshop on Secure Internet of Things*. Sept. 2014, pp. 35–43. DOI: 10.1109/SIoT.2014.10.

[6] M. B. Barcena and C. Wueest. *Insecurity in the Internet of Things*. Tech. rep. Symantec, Mar. 2015. URL: http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/insecurity-in-the-internet-of-things.pdf.

[7] L. A. D. Bathen and N. Dutt. "PoliMakE: A Policy Making Engine for Secure Embedded Software Execution on Chip-multiprocessors". In: *Proceedings of the 5th Workshop on Embedded Systems Security*. WESS '10. Scottsdale, Arizona: ACM, 2010, 2:1–2:10. ISBN: 978-1-4503-0078-0. DOI: 10.1145/1873548.1873550.

[8] L. A. D. Bathen and N. Dutt. "TrustGeM: Dynamic trusted environment generation for chip-multiprocessors". In: *Hardware-Oriented Security and Trust (HOST), 2011 IEEE International Symposium on*. 2011, pp. 47–50. DOI: 10.1109/HST.2011.5954994.

[9] M. Ben-Yehuda, J. Xenidis, M. Ostrowski, K. Rister, A. Bruemmer, and L. v. Doorn. "The price of safety: Evaluating IOMMU performance". In: *Ottawa Linux Symposium (OLS)*. 2007, pp. 9–20.

[10] M. Bishop. "What Is Computer Security?" In: *IEEE Security and Privacy* 99.1 (Jan. 2003), pp. 67–69. DOI: 10.1109/MSECP.2003.1176998.

[11] J. Biskup. *Security in Computing Systems: Challenges, Approaches and Solutions*. 1st Edition. Berlin: Springer, 2009. ISBN: 9783540784425.

[12] J. Blythe, R. Koppel, and S. W. Smith. "Circumvention of Security: Good Users Do Bad Things". In: *Security Privacy, IEEE* 11.5 (2013), pp. 80–83. DOI: 10.1109/MSP.2013.110.

[13] F. Brasser, B. E. Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl. "TyTAN: Tiny Trust Anchor for Tiny Devices". In: *Proceedings of the 52Nd Annual Design Automation Conference*. DAC '15. San Francisco, California: ACM, 2015, 34:6. ISBN: 9781-450335201. DOI: 10.1145/2744769.2744922.

[14] D. F. C. Brewer and M. J. Nash. "The Chinese Wall security policy". In: *Proceedings. 1989 IEEE Symposium on Security and Privacy*. May 1989, pp. 206–214. DOI: 10.1109/SECPRI.1989.36295.

[15] J. Brunel, R. Pacalet, S. Ouaarab, and G. Duc. "SecBus, a Software/Hardware Architecture for Securing External Memories". In: *2014 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*. Apr. 2014, pp. 277–282. DOI: 10.1109/MobileCloud.2014.49.

[16] *Chapter 15. Jails*. URL: https://www.freebsd.org/doc/handbook/jails.html.

[17] S. Checkoway, D. Mccoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno. "Comprehensive Experimental Analyses of Automotive Attack Surfaces". In: *USENIX SECURITY*. USENIX, 2011.

[18] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos. "Flexible Hardware Acceleration for Instruction-Grain Program Monitoring". In: *Proceedings of the 35th Annual International Symposium on Computer Architecture*. ISCA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 377–388. ISBN: 978-0-7695-3174-8. DOI: 10.1109/ISCA.2008.20.

[19] *Computer Laboratory: Bluespec Extensible RISC Implementation (BERI)*. URL: http://www.cl.cam.ac.uk/research/security/ctsrd/beri/.

[20] P. Cotret, G. Gogniat, J. P. Diguet, and J. Crenne. "Lightweight reconfiguration security services for AXI-based MPSoCs". In: *22nd International Conference on Field Programmable Logic and Applications (FPL)*. 2012, pp. 655–658. ISBN: 1946-147X. DOI: 10.1109/FPL.2012.6339233.

[21] cybellum. *DoubleAgent: Taking Full Control Over Your Antivirus*. en-US. Mar. 2017. URL: https://cybellum.com/doubleagent-taking-full-control-antivirus/ (visited on 01/31/2018).

[22] W. J. Dally and B. Towles. *Principles and practices of interconnection networks*. Amsterdam: Morgan Kaufmann, 2004. ISBN: 0122007514.

[23] M. Dalton, H. Kannan, and C. Kozyrakis. "Raksha: A Flexible Information Flow Architecture for Software Security". In: *Proceedings of the 34th Annual International Symposium on Computer Architecture*. ISCA '07. San Diego, California, USA: ACM, 2007, pp. 482–493. ISBN: 978-1-59593-706-3. DOI: 10.1145/1250662.1250722.

[24] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo. "On the Feasibility of Online Malware Detection with Performance Counters". In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ISCA '13. Tel-Aviv, Israel: ACM, 2013, pp. 559–570. ISBN: 978-1-4503-2079-5. DOI: 10.1145/2485922.2485970.

[25] J. B. Dennis and E. C. Van Horn. "Programming Semantics for Multiprogrammed Computations". In: *Commun. ACM* 9.3 (Mar. 1966), pp. 143–155. ISSN: 0001-0782. DOI: 10.1145/365230.365252.

[26] U. Dhawan, N. Vasilakis, R. Rubin, S. Chiricescu, J. M. Smith, J. K. T. F., B. C. Pierce, and A. DeHon. "PUMP: A Programmable Unit for Metadata Processing". In: *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*. HASP '14. Minneapolis, Minnesota, USA: ACM, 2014, 8:1–8:8. ISBN: 978-1-4503-2777-0. DOI: 10.1145/2611765.2611773.

[27] G. Drescher, C. Erhardt, F. Freiling, J. Götzfried, D. Lohmann, P. Maene, T. Müller, I. Verbauwhede, A. Weichslgartner, and S. Wildermann. "Providing security on demand using invasive computing". In: *it - Information Technology* 58.6 (Sept. 2016), p. 1611. DOI: 10.1515/itit-2016-0032.

[28] D. Evtyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. A. Ghazaleh, and R. Riley. "Iso-X: A Flexible Architecture for Hardware-Managed Isolated Execution". In: *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*. 2014, pp. 190–202. ISBN: 1072-4451. DOI: 10.1109/MICRO.2014.25.

[29] A. Fehnker, R. Huuck, and W. Rodiger. "Model Checking Dataflow for Malicious Input". In: *Proceedings of the Workshop on Embedded Systems Security*. WESS '11. Taipei, Taiwan: ACM, 2011, 4:1–4:10. ISBN: 978-1-4503-0819-9. URL: http://doi.acm.org/10.1145/2072274.2072278.

[30] D. Ferraiolo and R. Kuhn. "Role-Based Access Controls". In: *In 15th NIST-NCSC National Computer Security Conference*. 1992, pp. 554–563.

[31] L. Fiorin, G. Palermo, S. Lukovic, V. Catalano, and C. Silvano. "Secure Memory Accesses on Networks-on-Chip". In: *Computers, IEEE Transactions on* 57.9 (2008), pp. 1216–1229. DOI: 10.1109/TC.2008.69.

[32] Google. *Go Programming Language*. URL: https://golang.org/.

[33] B. Gorenc and J. Spelman. *Java Every-Days Exploiting Software Running on 3 Billion Devices*. Tech. rep. HP Security Research Zero Day Initiative, 2013. URL: https://media.blackhat.com/us-13/US-13-Gorenc-Java-Every-Days-Exploiting-Software-Running-on-3-Billion-Devices-WP.pdf.

[34] J. Gosling and H. McGilton. *The Java language environment*. English. May 1996. URL: http://www.oracle.com/technetwork/java/langenv-140151.html.

[35] M. D. Grammatikakis, K. Papadimitriou, P. Petrakis, A. Papagrigoriou, G. Kornaros, I. Christoforakis, and M. Coppola. "Security Effectiveness and a Hardware Firewall for MPSoCs". In: *High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS), 2014 IEEE Intl Conf on*. 2014, pp. 1032–1039. DOI: 10.1109/HPCC.2014.173.

[36] M. D. Grammatikakis, K. Papadimitriou, P. Petrakis, A. Papagrigoriou, G. Kornaros, I. Christoforakis, O. Tomoutzoglou, G. Tsamis, and M. Coppola. "Security in MPSoCs: A NoC Firewall and an Evaluation Framework". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.8 (2015), pp. 1344–1357. DOI: 10.1109/TCAD.2015.2448684.

[37] GraphViz. *The DOT Language*. URL: https://graphviz.gitlab.io/_pages/doc/info/lang.html.

[38] A. Grau. "Can you trust your fridge?" In: *IEEE Spectrum* 52.3 (Mar. 2015), pp. 50–56. ISSN: 0018-9235. DOI: 10.1109/MSPEC.2015.7049440.

[39] J. Greene. *Intel® Trusted Execution Technology - Hardware-based Technology for Enhancing Server Platform Security*. Tech. rep. Intel Corporation, 2012. URL: http://www.intel.com/content/www/us/en/architecture-and-technology/trusted-execution-technology/trusted-execution-technology-security-paper.html.

[40] N. Hardy. "The Confused Deputy: (or Why Capabilities Might Have Been Invented)". In: *SIGOPS Oper. Syst. Rev.* 22.4 (Oct. 1988), pp. 36–38. ISSN: 0163-5980. URL: http://dl.acm.org/citation.cfm?id=54289.871709 (visited on 03/25/2018).

[41] E. A. Harris and N. Perlroth. "For Target, the Breach Numbers Grow". In: *The New York Times* (Jan. 2014). URL: http://www.nytimes.com/2014/01/11/business/target-breach-affected-70-million-customers.html?_r=0.

[42] A. Hattendorf, A. Raabe, and A. Knoll. "Shared memory protection for spatial separation in multicore architectures". In: *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*. 2012, pp. 299–302. ISBN: 2150-3109. DOI: 10.1109/SIES.2012.6356601.

[43] G. Heiser. “Virtualizing embedded systems - why bother?” In: *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*. 2011, pp. 901–905. ISBN: 0738-100x.

[44] M. Henson and S. Taylor. “Memory Encryption: A Survey of Existing Techniques”. In: *ACM Comput.Surv.* 46.4 (Mar. 2014), 53:1–53:26. DOI: 10.1145/2566673.

[45] D. Hong, L. A. D. Bathen, S.-S. Lim, and N. Dutt. “DynaPoMP: Dynamic Policy-driven Memory Protection for SPM-based Embedded Systems”. In: *Proceedings of the Workshop on Embedded Systems Security*. WESS ’11. Taipei, Taiwan: ACM, 2011, 5:1–5:10. ISBN: 978-1-4503-0819-9. DOI: 10.1145/2072274.2072279.

[46] A. V. Hu (NIST), A. D. Ferraiolo (NIST), A. R. Kuhn (NIST), A. A. Schnitzer (BAH), A. K. Sandlin (MITRE), A. R. Miller (MITRE), and A. K. S. (Cybersecurity). *SP 800-162, Guide to Attribute Based Access Control (ABAC) Definition and Considerations*. EN-US. URL: https://csrc.nist.gov/publications/detail/sp/800-162/final (visited on 02/08/2018).

[47] Y. Hu, D. Muller-Gritschneder, M. J. Sepulveda, G. Gogniat, and U. Schlichtmann. “Automatic ILP-based Firewall Insertion for Secure Application-Specific Networks-on-Chip”. In: *2015 Ninth International Workshop on Interconnection Network Architectures: On-Chip, Multi-Chip*. Jan. 2015, pp. 9–12. DOI: 10.1109/INA-OCMC.2015.9.

[48] T. Huffmire, C. Irvine, R. Kastner, T. Levin, T. D. Nguyen, and T. Sherwood. *Handbook of FPGA Design Security*. 1., Edition. Berlin: Springer Netherland, 2010. ISBN: 9048191564.

[49] Intel. *iAPX 286 Programmer’s Reference Manual*. Intel Corporation, 1983. URL: http://bitsavers.org/components/intel/80286/210498-001_iAPX_286_Programmers_Reference_1983.pdf.

[50] *Intel® Virtualization Technology*. URL: http://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html.

[51] intelFPGA. *AN-540: Nios II MPU Usage*. URL: https://www.altera.com/en_US/pdfs/literature/an/an540.pdf (visited on 03/29/2018).

[52] intelFPGA. *Embedded Peripherals IP User Guide*. URL: https://www.altera.com/documentation/sfo1400787952932.html (visited on 03/25/2018).

[53] intelFPGA. *Nios II Performance Benchmarks*. en. URL: https://www.altera.com/en_US/pdfs/literature/ds/ds_nios2_perf.pdf.

[54] *ISO/IEC JTC1/SC22/WG14 - C: Approved standards*. URL: http://www.open-std.org/JTC1/SC22/WG14/www/standards (visited on 01/24/2018).

[55] K. Schaffer and J. Voas. "What Happened to Formal Methods for Security?" In: *Computer* 49.8 (Aug. 2016), pp. 70–79. ISSN: 0018-9162. DOI: 10.1109/MC.2016.228.

[56] E. Kang. "Design Space Exploration for Security". In: *2016 IEEE Cybersecurity Development (SecDev)*. 2016, pp. 30–36. DOI: 10.1109/SecDev.2016.017.

[57] H. Kannan, M. Dalton, and C. Kozyrakis. "Decoupling Dynamic Information Flow Tracking with a dedicated coprocessor". In: *Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on*. June 2009, pp. 105–114. DOI: 10.1109/DSN.2009.5270347.

[58] E. Keller, J. Szefer, J. Rexford, and R. B. Lee. "NoHype: Virtualized Cloud Infrastructure Without the Virtualization". In: *Proceedings of the 37th Annual International Symposium on Computer Architecture*. ISCA '10. Saint-Malo, France: ACM, 2010, pp. 350–361. ISBN: 978-1-4503-0053-7. DOI: 10.1145/1815961.1816010. URL: http://doi.acm.org/10.1145/1815961.1816010.

[59] D. Kleidermacher and M. Kleidermacher. *Embedded systems security*. 1. publ. Amsterdam [u.a.]: Elsevier, 2012. ISBN: 0123868866.

[60] G. Klein. "Operating System Verification — An Overview". In: *Sādhanā* 34.1 (Feb. 2009), pp. 27–69.

[61] P. C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *Advances in Cryptology — CRYPTO '96*. Ed. by N. Koblitz. Springer Berlin Heidelberg, 1996, pp. 104–113. ISBN: 978-3-540-68697-2.

[62] P. Kocher, J. Jaffe, B. Jun, and P. Rohatgi. "Introduction to differential power analysis". In: *Journal of Cryptographic Engineering* 1.1 (Apr. 2011), pp. 5–27. ISSN: 2190-8516. DOI: 10.1007/s13389-011-0006-y.

[63] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. "TrustLite: A Security Architecture for Tiny Embedded Devices". In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys '14. Amsterdam, The Netherlands: ACM, 2014, 10:14. ISBN: 9781-450327046. DOI: 10.1145/2592798.2592824.

[64] D. R. Kuhn, M. S. Raunak, and R. Kacker. "An Analysis of Vulnerability Trends, 2008-2016". English. In: IEEE, 2017, pp. 587–588. DOI: 10.1109/QRS-C.2017.106.

[65] D. Kushner. "The real story of stuxnet". In: *IEEE Spectrum* 50.3 (2013), pp. 48–53. DOI: 10.1109/MSPEC.2013.6471059.

[66] S. Kutzner, A. Y. Poschmann, and M. Stottinger. "Hardware Trojan Design and Detection: A Practical Evaluation". In: *Proceedings of the Workshop on Embedded Systems Security*. WESS '13. Montreal, Quebec, Canada: ACM, 2013, 1:1–1:9. ISBN: 978-1-4503-2145-7. DOI: 10.1145/2527317.2527318.

[67] E. Ladakis, L. Koromilas, G. Vasiliadis, M. Polychronakis, and S. Ioannidis. "You Can Type, but You Can't Hide: A Stealthy GPU-based Keylogger". In: *EuroSec'13*. Prague, Czech Republic, Apr. 2013.

[68] C. Landwehr. "Formal Models for Computer Security". English. In: *ACM Computing Surveys (CSUR)* 13.3 (Sept. 1981), pp. 247–278. DOI: 10.1145/356850.356852. URL: http://dl.acm.org/citation.cfm?id=356852.

[69] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi. "A Taxonomy of Computer Program Security Flaws". In: *ACM Comput. Surv.* 26.3 (Sept. 1994), pp. 211–254. ISSN: 0360-0300. DOI: 10.1145/185403.185412. (Visited on 01/23/2018).

[70] R. Langner. *To Kill a Centrifuge - A Technical Analysis of What Stuxnet's Creators Tried to Achieve*. Tech. rep. The Langner Group, 2013. URL: http://www.langner.com/en/wp-content/uploads/2013/11/To-kill-a-centrifuge.pdf.

[71] C. Lee, L. Zappaterra, K. Choi, and H.-A. Choi. "Securing smart home: Technologies, security challenges, and security requirements". In: *2014 IEEE Conference on Communications and Network Security*. Oct. 2014, pp. 67–72. DOI: 10.1109/CNS.2014.6997467.

[72] R. B. Lee. "Rethinking Computers for Cybersecurity". In: *Computer* 48.4 (2015), pp. 16–25. ISSN: 1558-0814. DOI: 10.1109/MC.2015.118.

[73] M. LeMay and C. A. Gunter. "Network-on-Chip Firewall: Countering Defective and Malicious System-on-Chip Hardware". In: *Logic, Rewriting, and Concurrency: Essays Dedicated to José Meseguer on the Occasion of His 65th Birthday*. Ed. by N. Martí-Oliet, P. C. Ölveczky, and C. Talcott. Cham: Springer International Publishing, 2015, pp. 404–426. ISBN: 978-3-319-23165-5. DOI: 10.1007/978-3-319-23165-5_19".

[74] F. L. Levesque, A. Somayaji, D. Batchelder, and J. M. Fernandez. "Measuring the health of antivirus ecosystems". In: *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*. Oct. 2015, pp. 101–109. DOI: 10.1109/MALWARE.2015.7413690.

[75] A. Limited. *ARM Security Technology - Building a Secure System using TrustZone® Technology*. Tech. rep. PRD29-GENC-009492C. ARM, 2009. URL: http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

[76] T. A. Linden. "Operating System Structures to Support Security and Reliable Software". In: *ACM Comput.Surv.* 8.4 (Dec. 1976), pp. 409–445. DOI: 10.1145/356678.356682.

[77] C. Liu, J. Rajendran, C. Yang, and R. Karri. "Shielding Heterogeneous MPSoCs From Untrustworthy 3PIPs Through Security-Driven Task Scheduling". In: *IEEE Transactions on Emerging Topics in Computing* 2.4 (2014), pp. 461–472. DOI: 10.1109/TETC.2014.2348182.

[78] S. W. Lodin and C. L. Schuba. “Firewalls fend off invasions from the Net”. In: *IEEE Spectrum* 35.2 (Feb. 1998), pp. 26–34. ISSN: 0018-9235. DOI: 10.1109/6.648669.

[79] J. Loinig, C. Steger, R. Weiss, and E. Haselsteiner. “Towards formal system-level verification of security requirements during hardware/software codesign”. In: *2010 IEEE International SOC Conference (SOCC)*. Sept. 2010, pp. 388–391. ISBN: 978-1-4244-6683-2. DOI: 10.1109/SOCC.2010.5784702.

[80] A. Ltd. *Cortex-M23*. en. URL: https://developer.arm.com/products/processors/cortex-m/cortex-m23 (visited on 02/10/2018).

[81] S. Lukacs, A. Lutas, D. H. Lutas, and G. Sebestyen. “Hardware virtualization based security solution for embedded systems”. In: *2014 IEEE International Conference on Automation, Quality and Testing, Robotics*. 2014, pp. 1–6. DOI: 10.1109/AQTR.2014.6857879.

[82] J. Manyika, M. Chui, P. Bisson, J. Woet, R. Dobbs, J. Bughin, and D. Aharon. *The Internet of Things : mapping the value beyond the hype*. English. Tech. rep. McKinsey Global Institute, June 2015. URL: https://www.mckinsey.com/business-functions/digital-mckinsey/our-insights/the-internet-of-things-the-value-of-digitizing-the-physical-world.

[83] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. “Innovative Instructions and Software Model for Isolated Execution”. In: *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP ’13. New York, NY, USA: ACM, 2013, 10:1–10:1. ISBN: 978-1-4503-2118-1. DOI: 10.1145/2487726.2488368. (Visited on 02/04/2018).

[84] *MISRA - The Motor Industry Software Reliability Association*. URL: https://www.misra.org.uk/ (visited on 01/24/2018).

[85] R. Mitchell and I.-R. Chen. “A Survey of Intrusion Detection Techniques for Cyber-physical Systems”. In: *ACM Comput.Surv.* 46.4 (Mar. 2014), 55:29. DOI: 10.1145/2542049.

[86] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. “CCured: Type-safe Retrofitting of Legacy Software”. In: *ACM Trans.Program.Lang.Syst.* 27.3 (May 2005), pp. 477–526. DOI: 10.1145/1065887.1065892.

[87] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. “How Amazon Web Services Uses Formal Methods”. In: *Commun. ACM* 58.4 (Mar. 2015), pp. 66–73. ISSN: 0001-0782. DOI: 10.1145/2699417. (Visited on 01/25/2018).

[88] J. Newsome and D. Song. "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software". In: *12th Annual Network and Distributed System Security Symposium*. San Diego, California, 2005. URL: http://valgrind.org/docs/newsome2005.pdf.

[89] A. Nordrum. *Popular Internet of Things Forecast of 50 Billion Devices by 2020 Is Outdated*. 18 Aug 2016. URL: https://spectrum.ieee.org/tech-talk/telecom/internet/popular-internet-of-things-forecast-of-50-billion-devices-by-2020-is-outdated.

[90] M. Oh. *An evolution of BlackPOS malware*. URL: https://web.archive.org/web/20140302015627/http://h30499.www3.hp.com/t5/HP-Security-Research-Blog/An-evolution-of-BlackPOS-malware/ba-p/6359149.

[91] K. Patel and S. Parameswaran. "SHIELD: A software hardware design methodology for security and reliability of MPSoCs". In: *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*. June 2008, pp. 858–861. DOI: 10.1145/1391469.1391686.

[92] K. Patel, S. Parameswaran, and R. G. Ragel. "Architectural Frameworks for Security and Reliability of MPSoCs". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 19.9 (Sept. 2011), pp. 1641–1654. DOI: 10.1109/TVLSI.2010.2053856.

[93] M. Paulitsch, O. M. Duarte, H. Karray, K. Mueller, D. Muench, and J. Nowotsch. "Mixed-Criticality Embedded Systems – A Balance Ensuring Partitioning and Performance". In: *Digital System Design (DSD), 2015 Euromicro Conference on*. 2015, pp. 453–461. DOI: 10.1109/DSD.2015.100.

[94] S. Pfleeger and R. Cunningham. "Why Measuring Security Is Hard". In: *IEEE Security & Privacy* 8.4 (2010), pp. 46–54. DOI: 10.1109/MSP.2010.60.

[95] L. Piccolboni, A. Menon, and G. Pravadelli. "Efficient Control-Flow Subgraph Matching for Detecting Hardware Trojans in RTL Models". In: *ACM Trans. Embed. Comput. Syst.* 16.5s (Sept. 2017), 137:1–137:19. ISSN: 1539-9087. DOI: 10.1145/3126552.

[96] A. D. Pimentel, C. Erbas, and S. Polstra. "A systematic approach to exploring embedded system architectures at multiple abstraction levels". English. In: *IEEE Transactions on Computers* 55.2 (2006), pp. 99–112. DOI: 10.1109/TC.2006.16.

[97] J. Porquet, A. Greiner, and C. Schwarz. "NoC-MPU: A secure architecture for flexible co-hosting on shared memory MPSoCs". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*. 2011, pp. 1–4. ISBN: 1530-1591. DOI: 10.1109/DATE.2011.5763291.

[98] J. Porquet and S. Sethumadhavan. “WHISK: An uncore architecture for Dynamic Information Flow Tracking in heterogeneous embedded SoCs”. In: *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013 International Conference on*. 2013, pp. 1–9. DOI: 10.1109/CODES-ISSS.2013.6658991.

[99] S. L. Project. *SE Linux Project*. URL: http://selinuxproject.org/page/Main_Page.

[100] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu. “LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks”. In: *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 39. Washington, DC, USA: IEEE Computer Society, 2006, pp. 135–148. ISBN: 0-7695-2732-9. DOI: 10.1109/MICRO.2006.29.

[101] J. J. Rajendran, O. Sinanoglu, and R. Karri. “Building Trustworthy Systems Using Untrusted Components: A High-Level Synthesis Approach”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24.9 (2016), pp. 2946–2959. DOI: 10.1109/TVLSI.2016.2530092.

[102] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady. “Security in embedded systems”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 3.3 (Aug. 2004), pp. 461–491. DOI: 10.1145/1015047.1015049.

[103] S. Ray, E. Peeters, M. M. Tehranipoor, and S. Bhunia. “System-on-Chip Platform Security Assurance: Architecture and Validation”. English. In: *Proceedings of the IEEE* PP.99 (), pp. 1–17. DOI: 10.1109/JPROC.2017.2714641.

[104] R. Rivest, A. Shamir, and L. Adleman. *A method for obtaining digital signatures and public-key cryptosystems*. Jan. 1983. DOI: 10.1145/357980.358017.

[105] Z. Salcic, M. Nadeem, H. Park, and J. Teich. “A heterogeneous multi-core SoC for mixed criticality industrial automation systems”. In: *21st IEEE International Conference on Emerging Technologies and Factory Automation*. ETFA 2016. Berlin, Germany, Sept. 2016. DOI: 10.1109/ETFA.2016.7733519.

[106] Z. Salcic, M. Nadeem, H. Park, and J. Teich. “Optimizing Latencies and Customizing NoC of Time-Predictable Heterogeneous Multi-Core Processor”. In: *IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC-16)*. Lyon, France, 2016. DOI: 10.1109/MCSoC.2016.34.

[107] Z. Salcic, H. Park, J. Teich, A. Malik, and M. Nadeem. “Noc-HMP: A Heterogeneous Multicore Processor for Embedded Systems Designed in SystemJ”. In: *ACM Trans. Des. Autom. Electron. Syst.* 22.4 (June 2017), 73:1–73:25. ISSN: 1084-4309. DOI: 10.1145/3073416.

[108] R. Saleh, S. Wilton, S. Mirabbasi, A. Hu, M. Greenstreet, G. Lemieux, P. P. Pande, C. Grecu, and A. Ivanov. "System-on-Chip: Reuse and Integration". In: *Proceedings of the IEEE* 94.6 (June 2006), pp. 1050–1069. ISSN: 0018-9219. DOI: 10.1109/JPROC.2006.873611.

[109] J. H. Saltzer and M. D. Schroeder. "The protection of information in computer systems". In: *Proceedings of the IEEE* 63.9 (Sept. 1975), pp. 1278–1308. ISSN: 0018-9219. DOI: 10.1109/PROC.1975.9939.

[110] J. H. Saltzer. "Protection and the Control of Information Sharing in Multics". In: *Commun.ACM* 17.7 (July 1974), pp. 388–402. DOI: 10.1145/361011.361067.

[111] R. S. Sandhu and P. Samarati. "Access control: principle and practice". In: *IEEE Communications Magazine* 32.9 (1994), pp. 40–48. DOI: 10.1109/35.312842.

[112] M. D. Schroeder and J. H. Saltzer. "A Hardware Architecture for Implementing Protection Rings". In: *Commun. ACM* 15.3 (Mar. 1972), pp. 157–170. ISSN: 0001-0782. DOI: 10.1145/361268.361275. (Visited on 02/02/2018).

[113] J. Sepulveda, D. Florez, R. Fernandes, C. Marcon, G. Gogniat, and G. Sigl. "Towards risk aware NoCs for data protection in MPSoCs". In: *2016 11th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*. June 2016, pp. 1–8. DOI: 10.1109/ReCoSoC.2016.7533898.

[114] A. Shostack. *Threat modeling*. Indianapolis, Ind: Wiley, 2014. ISBN: 9781118809990.

[115] R. Slayton. "Measuring Risk: Computer Security Metrics, Automation, and Learning". In: *IEEE Annals of the History of Computing* 37.2 (Apr. 2015), pp. 32–45. ISSN: 1058-6180. DOI: 10.1109/MAHC.2015.30.

[116] T. Stapko. *Practical Embedded Security*. English. 1st ed. Newnes, 2011. ISBN: 9780750682152. URL: http://lib.myilibrary.com?ID=103931.

[117] N. Statt. *How an army of vulnerable gadgets took down the web today*. Oct. 2016. URL: https://www.theverge.com/2016/10/21/13362354/dyn-dns-ddos-attack-cause-outage-status-explained.

[118] I. Stierand, S. Malipatlolla, S. Frschle, A. Sthring, and S. Henkler. "Integrating the Security Aspect into Design Space Exploration of Embedded Systems". In: *2014 IEEE International Symposium on Software Reliability Engineering Workshops*. 2014, pp. 371–376. DOI: 10.1109/ISSREW.2014.29.

[119] S. Stolfo, S. M. Bellovin, and D. Evans. "Measuring Security". In: *IEEE Security & Privacy* 9.3 (2011), pp. 60–65. DOI: 10.1109/MSP.2011.56.

[120] R. Strackx, F. Piessens, and B. Preneel. "Efficient Isolation of Trusted Subsystems in Embedded Systems". In: *Security and Privacy in Communication Networks*. Ed. by S. Jajodia and J. Zhou. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 344–361. ISBN: 978-3-642-16161-2. DOI: 10.1007/978-3-642-16161-2_20.

[121] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas. "Design and implementation of the AEGIS single-chip secure processor using physical random functions". In: *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*. 2005, pp. 25–36. ISBN: 1063-6897. DOI: 10.1109/ISCA.2005.22.

[122] L. Szekeres, M. Payer, L. T. Wei, and R. Sekar. "Eternal War in Memory". In: *IEEE Security and Privacy* 12.3 (2014), pp. 45–53. DOI: 10.1109/MSP.2014.44.

[123] B. Tan, M. Biglari-Abhari, and Z. Salcic. "An Automated Security-Aware Approach for Design of Embedded Systems on MPSoC". In: *ACM Trans.Embed.Comput.Syst.* 16.5s (Sept. 2017), 143:20. DOI: 10.1145/3126553.

[124] B. Tan, M. Biglari-Abhari, and Z. Salcic. "Towards decentralized system-level security for MPSoC-based embedded applications". In: *Journal of Systems Architecture* 80 (Oct. 2017), pp. 41–55. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2017.09.001. (Visited on 03/29/2018).

[125] N. Thamsirarak, T. Seethongchuen, and P. Ratanaworabhan. "A case for malware that make antivirus irrelevant". In: *2015 12th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*. June 2015, pp. 1–6. DOI: 10.1109/ECTICon.2015.7206972.

[126] D. L. C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. "Architectural Support for Copy and Tamper Resistant Software". In: *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS IX. Cambridge, Massachusetts, USA: ACM, 2000, pp. 168–177. ISBN: 1-58113-317-0. DOI: 10.1145/378993.379237.

[127] *Trusted Computing Group*. URL: http://www.trustedcomputinggroup.org/.

[128] F. Vahid and T. Givargis. *Embedded system design*. New York, NY: Wiley, 2002. ISBN: 9780471386780.

[129] G. Venkataramani, I. Doudalis, D. Solihin, and M. Prvulovic. "FlexiTaint: A programmable accelerator for dynamic taint propagation". In: *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*. Feb. 2008, pp. 173–184. ISBN: 1530-0897. DOI: 10.1109/HPCA.2008.4658637.

[130] E. I. Vrt. *Broadpwn: Remotely Compromising Android and iOS via a Bug in Broadcom's Wi-Fi Chipsets*. June 2017. URL: https://blog.exodusintel.com/2017/07/26/broadpwn/.

[131] C. Wang, X. Li, Y. Chen, Y. Zhang, O. Diessel, and X. Zhou. "Service-Oriented Architecture on FPGA-Based MPSoC". In: *IEEE Transactions on Parallel and Distributed Systems* 28.10 (Oct. 2017), pp. 2993–3006. DOI: 10.1109/TPDS.2017.2701828.

[132] A. Wasicek, C. El-Salloum, and H. Kopetz. "A System-on-a-Chip Platform for Mixed-Criticality Applications". In: *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2010 13th IEEE International Symposium on*. May 2010, pp. 210–216. ISBN: 1555-0885. DOI: 10.1109/ISORC.2010.43.

[133] J. Williams. *A Guide to Understanding Security Modeling in Trusted Systems*. English. Tech. rep. NCSC, GPO, Washington, DC, Oct. 1992. URL: http://www.dtic.mil/docs/citations/ADA392777.

[134] R. Wilson. *Security for Embedded Systems: Today's Issue, Not Tomorrow's*. URL: http://www.altera.com/technology/system-design/articles/2012/embedded-security.html.

[135] J. D. Woodruff. *CHERI: A RISC capability machine for practical memory safety*. Tech. rep. University of Cambridge, Computer Laboratory, July 2014. URL: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-858.pdf.

[136] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. "The CHERI Capability Model: Revisiting RISC in an Age of Risk". In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. ISCA '14. Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 457–468. ISBN: 978-1-4799-4394-4. DOI: 10.1109/ISCA.2014.6853201.

[137] *Xen Project*. en-gb. URL: https://www.xenproject.org/ (visited on 04/11/2018).