Formal Verification and Modelling of the Gen-Z Specification

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

Master's Thesis Nr. 311

Systems Group, Department of Computer Science, ETH Zurich

Formal Verification and Modelling of the Gen-Z Specification

by

Roman Brunner

Supervised by

Prof. Timothy Roscoe, Dr. David Cock, Nore Hossle

April 2020–October 2020

inf | Informatik
Computer Science

Abstract

In this thesis, I will explore different approaches to modelling the Gen-Z specifications [60]. The Gen-Z interconnect enables new system topologies, such as memory-centric systems, based on the disaggregated system design, leveraging network based system interconnects. The flexibility of Gen-Z concerning system design and topologies makes it attractive to industry and research alike. We have the rare opportunity to influence the design of Gen-Z in an early stage, as only recently the first final specification for Gen-Z has been released, and as of today no commercial hardware is available. This makes it interesting to formally model the Gen-Z specification in order to enable proofs of the different properties provided by the Gen-Z interconnect, as currently changes can be easily implemented. As the abstraction level of the Gen-Z specification ranges from the physical layer up to the application layer, including abstract security protocols, the types of proofs, the requirements for a formal specification and modelling languages, as well as the toolchain to implement such a model, vary significantly across the different layers. I evaluated different approaches to modelling Gen-Z by comparing them with an newly defined ideal for formal languages and their toolchain. To evaluate the different modelling approaches, I wrote several proofs-of-concept models and tools. I also applied these models to prove different properties of the security protocols of Gen-Z. Using the results of the proofs, I was able to identify two protocols within Gen-Z that do not provide additional guarantees and therefore could be removed and additional assumptions that are required in order for the security protocols to guarantee the desired properties.

Acknowledgements

First of all, I want to express my gratitude for the support and guidance I received from my supervisors, Timothy Roscoe, David Cock and especially Nora Hossle. Not only did they provide me with feedback to the work I did but motivated me to ask more and more in-depth questions, which added a lot of facets to this thesis with which I would not have come up on my own.

My thanks also go to the rest of the Systems Group, which also provided feedback, discussed issues as well as questions, and provided support whenever necessary.

I want to thank my family for supporting me throughout my studies at ETH. A special thanks also goes out to my friends that made studying at ETH a lot more fun. Last but not least I want to thank my girlfriend that often had to stand back such that I could work for ETH.

Contents

1 Introduction

Real hardware systems often do not look like the ones that we are taught in basic systems courses or are described in many textbooks. These are merely abstractions, a sample of which can be seen in Figure 1, of a complex interconnect, such as in Figure 2, of different hardware parts, that have different behaviours and use a wide variety of protocols to talk with each other. This has been the status quo for several decades now, and as new even complexer hardware and hardware-based protocols are released, the gap between the mental model that programmers have of hardware and the actual hardware present in a system gets wider and wider.

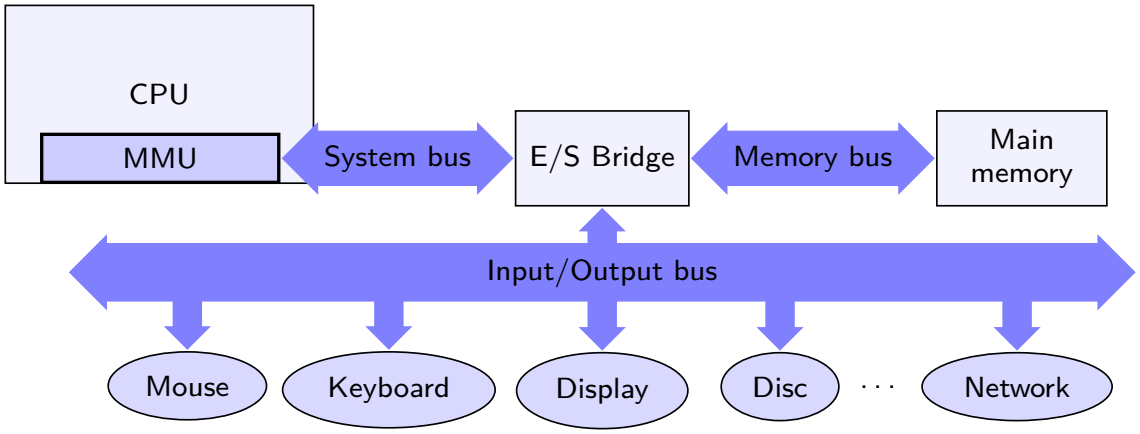


Figure 1: Simplified representation of a system as often taught in introductory systems programming or operating systems courses, adapted from [19]

.

Such gaps lead, in the best case, to inefficient implementations of system code, such as shown in [40], but more often than not, the implementations are faulty and do not provide the guarantees that the systems programmer think they do, such as shown in [11]. These types of missing guarantees are up to date a severe problem, as we rely on operating systems to provide different security guarantees, such as isolation of processes, protecting memory and enforcing organisational security guidelines. Aside from these tasks, the operating system and lower-level code needs to ensure that its execution does not damage the hardware.

In order to close the gap between software implementation and actual hardware reality, the Systems Group at ETH has been working on formally modelling the software-hardware boundary for several years now. Gen-Z, which is a new specification for an open system interconnect (OSI) structured bus replacement supporting arbitrary hardware components, could increase the complexity of hardware systems even further. Not only does Gen-Z shift more logic and therefore responsibility to hardware but it also allows to reconfigure how different components are connected dynamically. Even though Gen-Z tries to hide these challenges from the OS, it still changes the behaviour of the system. Hence, differences in the topology can influence which resource allocation and process scheduling decisions are optimal. In this thesis, I take the recently released initial stable specification of Gen-Z and experiment with several different formal modelling and specification languages to find out if Gen-Z does hold up to its promises and how such a complex interconnect is best modelled. As the Gen-Z specification defines protocols on several different levels of abstraction, ranging from the physical layer up to high-level security protocols, I will explore different approaches for modelling different parts of the specification. I will discuss the advantages and disadvantages of combining and splitting different levels of abstraction and provide a proof of concept for the different approaches I explored.

As of now, Gen-Z-compliant hardware is not generally available to the public market. Thus I argue

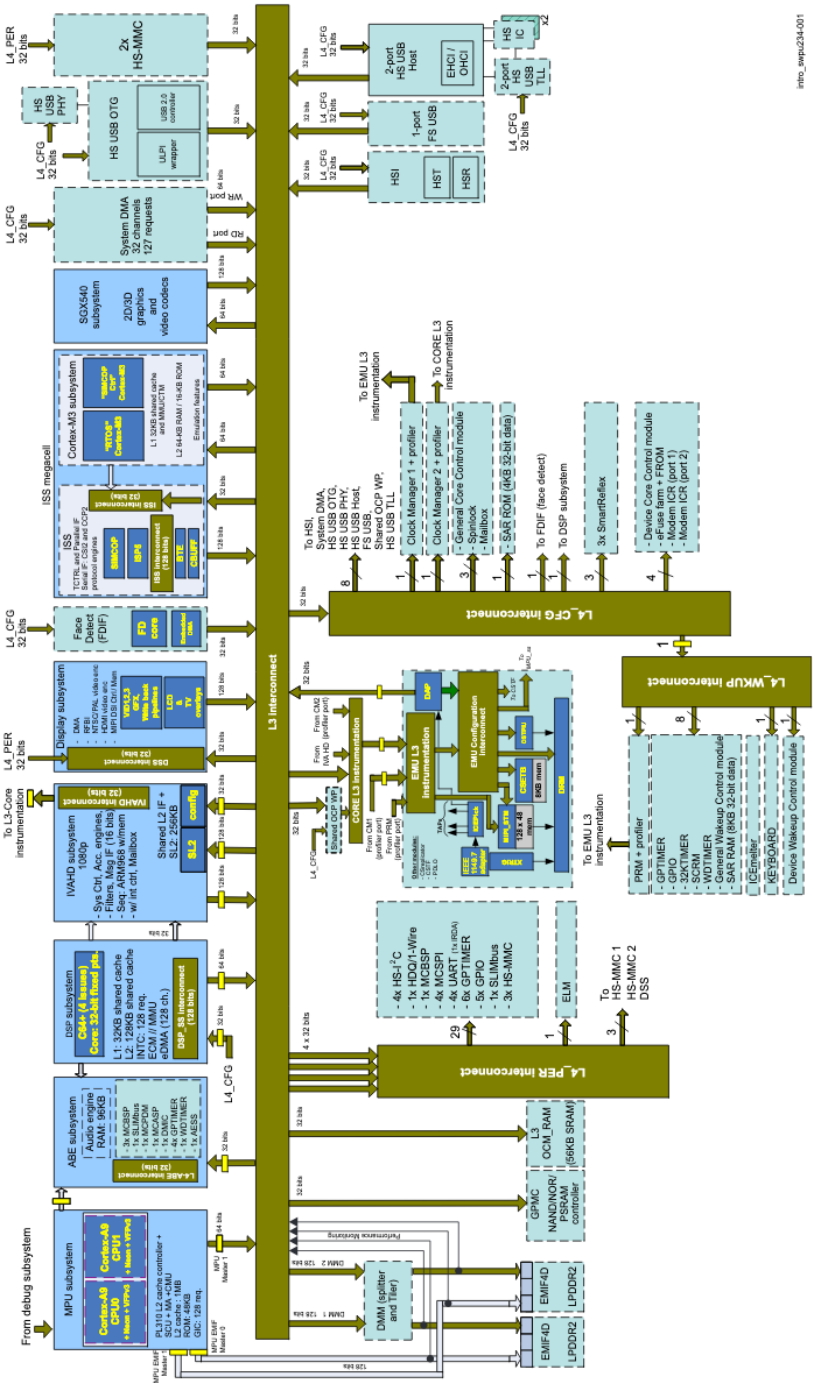


Figure 2: This block diagram represents a system on a chip which is already almost ten years old (Texas Instruments OMAP 4460). We can clearly see that there is way more going on than just a simple single interconnect. It is totally different to program such a chip than compared to the system in Figure 1

.

that it is the right time to formally model the different parts and check if the claims by Gen-Z hold or which additional assumptions are required by that the protocols provide the promised guarantees. At this point, feedback with respect to the specification can be implemented reasonably easily, as there is no hardware yet shipped, hence no costly upgrades are required. Additionally, software development for Gen-Z systems can be supported by early findings and simulators in order to test systems software before general hardware availability.

It is not a coincidence that several initiatives are currently pushing forward different specifications that try to achieve a universal interconnect between vastly different hardware components of a system. The need of the industry, especially in the cloud computing and infrastructure-as-a-service (IAAS) domain, for flexible and dynamically reconfigurable systems has driven this development. As these systems are usually multi-tenant systems, in the sense that parts of the hardware resources, such as the interconnect itself, are shared among multiple customers, isolating and protecting the data from unauthorized access, modification or destruction is of uttermost importance. All interconnects thus provide some mechanism to protect against wide varieties of attacks in hardware. Protection that is enforced through the hardware itself should make it considerably more challenging for an attacker to circumvent security measures. Nevertheless, recent media coverage [52] raised awareness that an attack against a system could be rooted in the hardware itself. Gen-Z also tries to protect against these kinds of attacks, and I will model and prove the different security protocols that should achieve a secure system, even under the assumption of malicious hardware.

Even though formal models help us in proving properties and generating code, they still have one weakness: we cannot prove the equivalence of the specification that we are modelling and the model itself. This discrepancy could only be overcome if the specification would have been written in a formal specification language itself, which would allow us to write proofs over the specification itself and refine them in arbitrary directions. Such specification languages should be easy to write, as otherwise they will not be used by the designers and can provide additional benefits, such as proving or providing simulators. In addition, they also should be easily readable as it should serve as a specification. I will discuss different approaches to such formal specification languages and their tradeoffs and the reason why I decided to use one or the other approach.

An ideal model would be one that captures the full Gen-Z specification and provides us with proofs about different types of properties, an executable simulator, and transpilers to different targets, ranging from software interface descriptions down to hardware specification languages. As my resources and time are constrained, I was not able to explore all the different details and also not able to write proofs-of-concepts for all different possible approaches. Nevertheless, with this thesis, I contribute to

1. A formal model of a Gen-Z memory component, implementing the end-to-end protocol for memory semantic accesses
2. A transpiler from the formal model in Arm ASL to Isabelle/HOL for proving purposes
3. A simulator based on the formal model of the Gen-Z memory component
4. A formal symbolic model based on Tamarin of the authentication and secure session establishment protocols
5. Several Proofs of the security protocols' authenticity and secrecy properties over Tamarin models
6. A proof that the in-situ prevention protocol and the bump-in-the-wire detection protocol of Gen-Z do not provide additional security guarantees
7. The discussion of different approaches to formally specifying and modelling complex hardware-software boundary specifications

In section 2, I will go through previous work, that forms the basics for understanding Gen-Z and how I approached the modelling of the specification.

After the overview of previous work, I will explain why we should model complex hardware and software systems formally and which tradeoffs there are in section 3. It contains several subsections with more narrowed down explanations on formal modelling different aspects of Gen-Z.

After the motivation and explication of formal models for different aspects of Gen-Z, I detail the different proofs-of-concepts I wrote in section 4, give implementation details on the Arm ASL Gen-Z Memory Component Model in subsection 4.1 as well as on the theories developed with Tamarin in subsection 4.3 and evaluate their performance within the respective subsections.

This brings me to the section 5, which contains my findings concerning the Gen-Z specification, but also for the different approaches of formally specifying and modelling hardware-software boundaries. In subsection 5.1 I describe the results achieved through the hardware-component models. In subsection 5.2 I will go through the proofs achieved with the Tamarin models. In subsubsection 5.2.5 I will discuss further implications of the proven properties and detail the findings with respect to the in-situ and bump-in-the-wire protocols.

Pulling from the results, I will then give an overview of potential future research, approaches to hardware-software interface modelling and how to contend with the complexity of future systems, that could be implemented based on the findings in this thesis in section 6. Finally, I conclude the thesis in section 7 and summarize my findings and implications thereof for Gen-Z and similar complex system specifications.

2 Previous Work/A bit of History

In this section, I am going to look at previous work that leads up to the different parts of this thesis. Further, the section contributes to the understanding of why the current development of computer systems requires specifications such as Gen-Z. Finally, this section will also introduce and compare several tools that have been used throughout the project.

2.1 Formal Hardware Models

In this section, I am going to present previous work that is concerned with the formal modelling of hardware and proving different properties about the hardware itself. As I am mostly interested in models of the hardware-software interface, I cover these aproaches more in-depth than the work that is concerned with hardware correctness, for which I only give a short and rough overview.

Hardware is inherently complex. Even though system engineers continuously extend their knowledge about how the hardware behaves, keeping up with the changes of the hardware is close to impossible. As hardware continues to apply more sophisticated techniques to increase system's speed and take over more versatile functionalities, complexity continues to increase fast. In order to cope with the increasing complexity, formal hardware specifications and models can support the work of system engineers by enabling software development through refinement of the formal specifications to actual implementations, proving properties of hardware and therefore also enabling proofs of software-correctness based on the proven properties of the hardware as well as enabling simulators and automated test generation to support correct by design systems. While this challenge already exists for classical hardware [41, 47, 20], the rise of FPGAs and flexible system architectures additionally contribute to the need for fast, adaptable operating systems. Formal hardware models can support the OS in achieving the flexibility required to switch seamlessly between vastly different systems by automatically generating glue code, e.g. by providing a scheduler strategy that is optimal with respect to the current system or by marshalling communication with a single hardware component according to the interface specification.

The most crucial goal of formal hardware models is to prove correctness concerning certain aspects of the hardware. In their survey, Kern and Greenstreet [7] describe a wide variety of approaches to formally modelling hardware and how these approaches lead to correct hardware. In their summary, they focus on techniques that are tractable and can be applied in practice to actual systems. They categorize the different approaches into two categories: frameworks for the formalization of specifications and implementation of hardware designs as well as tools for formal verification of hardware and properties thereof.

Armstrong et al., Gray et al. (Sail) [51, 45] and Reid et al. (Arm ASL) [41, 47], both describe approaches on how to prove correctness properties by formally specifying and describing the instruction set architecture (ISA) through a formal specification language. Different transpilers to different provable languages such as Isabelle/HOL, Coq or satisfiability modulo theories (SMT) solver input allows applying sophisticated provers to lemmata describing hardware properties. While the transpilers for the Sail language are publicly available, the transpilers for Arm ASL have not been publicly released. Additionally, through the use of interpreters, formal models in either language can be turned into a simulators of the described hardware, enabling features such as auto-test generation, easier debugging of misbehaviours and further analysis using other approaches such as information flow analysis. In this thesis, I am going to compare these tools, apply them to the Gen-Z specification and evaluate them compared to an ideal formal language.

While the previously mentioned applications of formal specifications to hardware pivoted around the correctness and the interface of single hardware components, the work by Achermann et al. and Humbel et al. [42, 50, 46] concentrates on a complete system. They model the system as a hardware decoding network and focus on how it routes information through different levels and layers of interconnects that connect the single hardware components. Through the representation of hardware decoding networks in Isabelle/HOL, they can derive proofs over the structure of a hardware system. A core focus lies on

the routing of requests based on addresses and of interrupts. Sockeye is the corresponding modelling language which offers tool support for transpilation to several targets, including Isabelle/HOL and Prolog, which then also can directly be used by an OS to run on the modelled hardware. The language and tools are described in great detail by Daniel Schwyn in [49, 56].

The above-mentioned approaches to formal hardware modelling have shown that hardware can be abstracted and modelled on different levels. In this thesis, I am mostly interested in the hardware-software boundary and hardware-based protocols and hence focus on ASL/Sail for modelling the behaviour and interface of single hardware components and Sockeye for modelling the interface and behaviour of more complex systems, consisting of multiple hardware components.

2.2 Hardware Security

In this section, we are going to compare the assumptions we usually have to the assumptions we can prove, concerning the security of hardware. In software engineering, we usually assume the hardware we run on to be untampered and therefore behaves as specified by the manufacturer modulo potential bugs that are only found after shipping. However, while research has been looking for a way to detect hardware trojans (HT) for quite some time, only recently an article in Bloomberg Businessweek [52] about a potential hardware-based attack on multiple American companies, including Apple and Amazon, though highly disputable if the attack actually took place, has brought this issue to the awareness of a broader public.

Detecting hardware attacks from software can be difficult or even impossible if there is only a single component available that offers a given service to the system. In systems code, we have a particular bootstrapping problem if we want to detect a HT in the chip that the software should be initially started on. The software requires the chip to be executed, but if the chip is not trusted, the outputs of the software might have been modified by some adversary calculation of the chip itself.

Many of the potential approaches to detect such HTs, e.g. the one used by Banga et al. [14], compare the unit under test (UUT) to a gold model of the unit. For a complete overview of these techniques, see [23], which compares the different approaches that use a gold standard as the starting point. As these methods are all test-based and the adversary might also has access to a gold-standard version (unmodified chip), no guarantees with respect to the absence of HTs can be derived. More importantly, this approach cannot defend against malicious circuits that have been added during the design phase, as these circuits are also present in the gold-standard.

More recent efforts on detecting HTs attempt to use a more structured approach, by clearly defining threat model classes [53] and apply automated classification on different measurements of the chip for varying inputs [48, 37]. While these techniques might be more difficult to subvert by an attacker, they also have their limits. Given the size of modern chips, it is challenging to achieve a good coverage of configurations of a chip, such that potential malicious behaviour has been triggered. Also such automated approaches can only classify behaviour that they have seen, and therefore identifying the activation configuration of adversarial of bug-introduced HTs is an important, yet unsolved problem. For a more detailed overview, see [39].

Gen-Z suggests a defending mechanism that tries to detect HTs in subcomponents before integrating them into a component. The detection of HTs has to be performed by the integrator or the manufacturer of the single sub-components. For this, they will most likely apply one of the previously mentioned approaches. In order to prove that this check has been completed, the attestation should be signed with a hardware-specific trusted key, that from then on guarantees that the component has been untampered with during the whole manufacturing process. Such testing requirements before the assembly of sub-components into larger components raise the bar for a compromise during the production or even the design of a component. However, it still is an open research question, as none of the HT detection mechanisms can formally prove that it prevents all types of HTs. Further, this approach does not prevent a compromise after production, e.g. during shipping.

While these additional steps during manufacturing might make it harder to hide a maliciously planted HT, no additional guarantees are achieved so far using these approaches. In the end, we still

have to trust that the hardware is untampered and runs correctly.

2.3 Hardware Roots of Trust

In this section, we are going to discuss the guarantees and approaches that enable running software in an untrusted environment, only trusting the hardware that the software is run on. These techniques provide functionality that is equally required for attestation and hence are required in the authentication protocols that will be discussed in subsection 4.3 and subsubsection 5.2.1. Hardware that enables such guarantees usually aims at bypassing any intermediary layers of software that are not trusted and drop the software to be executed right onto the hardware without the need to trust any other software and therefore minimizing the trusted computing base (TCB) or enable proving that the software layers below are trustworthy through attestation. Hence the root of trust resides within the hardware, and as I have described in subsection 2.2 the hardware has to be trusted anyway.

Any such security architecture usually relies on some form of a secure co-processor, that has a substantially reduced set of functionality, that only allows the platform to extend hashes of executed software into a register of the chip. The reduced set of functionality is intentional, as less exposed functionality equals to a smaller attack vector, which for a piece of hardware that is crucial for the systems security is a highly desirable property. While the data, that the secure co-processor works over, is usually stored in untrusted storage, the extended hash that depends on the complete history of executed software is stored in a register within the trusted security chip and can only be changed by the secure co-processor itself. This allows an external validator to check for the full history of executed software layers and check that the history has been untampered with by verifying the signed summary hash from the secure co-processor.

One of the best-known security co-processors that achieves these goals is the Trusted Platform Module [32]. This chip can perform several cryptographic services and protect measurements of software through a hash chain that is sealed by a summary hash. Further, the TPM chip offers attestation by signing the summary hash by a key that can be linked to a TPM chip and at the same time can guarantee privacy. The functionality of the TPM chip is exposed through an interface that only allows particular functionality, such as extending hashes of platform measurements and signing of predefined messages. Arbitrary content can be signed but is distinguishable from signed states of the chip itself, such as the signed summary hash.

Nevertheless, depending on the version and the application, several attacks have been found against TPM. One of the most famous attacks is the cuckoo attack [18] that makes use of one platform that has been tampered with and another honest one. It uses the honest platform as an oracle to provide the expected responses to the verifier, such that the verifier trusts the system, even though it is not the same system that produced the measurements and the responses. Under these premises, every protocol that makes use of such trusted co-processors must be checked against potential misuse by an adversary.

Trusted execution environments (TEE), such as Intel Software Guard Extension (SGX) [66, 30, 28] and Arm TrustZone [65], provide similar possibilities with extended functionality over the TPM. While the TPM chip only allowed the establishment of a static root of trust, that verifies if the software loaded is trustworthy, these TEEs allow to dynamically establish a root of trust through enabling the software to be run directly on the hardware. Hence, this approach bypasses any potential software layers between the hardware and the application. However, this requires support for special instructions by the CPU. The software that is loaded into a TEE then can be externally verified through remote attestation, which guarantees the user that the correct software will be executed within the TEE. This approach minimizes the TCB, by not including other software.

These techniques allow providing additional guarantees with regards to the trust that can be placed into the software executed. The user receives guarantees either concerning the state of the system that runs its software and that it was his software that produced the returned output (TPM) or enable the software to be provably run directly on the hardware and hence only the hardware has to be trusted (Intel SGX, Arm TrustZone).

2.4 Software Roots of Trust

While the previously discussed hardware roots of trust allow a multitude of applications and provide some necessary guarantees with respect to the state of a system and the executed software, it requires hardware support. In this section, I am going to discuss which approaches there exist for software-based roots of trust.

There are cases, where it is not possible to use trusted hardware, e.g. if no such chip is available or if we do not trust the built-in chip. In these cases, Seshadri et al. [10, 9] have shown that it is still possible to prove to a verifier that the system is executing the correct software stack. This type of attestation uses the timing of measuring random memory locations, including several pointers. If an attacker wants to execute any modified code, he would have to hide this code from the measurements, which would introduce overhead during measuring the memory. To avoid detection through this mechanism, the attacker would need to check each memory address if it lies within the malicious code segment. If it does so, the memory at that address has to be re-established to the expected values, and the attacker code has to be moved to some other memory location. This overhead guarantees that an attack could only respond slower than the fastest authentic measurement implementation. In order for the verifier to distinguish between a valid response time and a malicious one, it has to exactly know how long the execution of a measurement should take on the hardware that is used to execute the software.

In order for this to work reliably enough, the probability of hitting the location with the malicious code must be large enough such that the attacker bothers with checking the addresses. Hence this technique is not necessarily viable for components with vast amounts of memory. For these systems, another technique has been proposed, which initially only proves that the executing verification function is unmodified, and an untampered execution environment has been established. After this, the user only requires attestation of the software that has been loaded into the software-established TEE in order to be convinced that he obtained the correct outputs from the correct software. This approach dramatically reduces the overhead required to attest the software that is being executed, as only the verification function, the establishment of a TEE and the software to be executed have to be proven as compared to the complete memory content. Both techniques do not require any trust in other parts of the system.

These approaches could be especially helpful for low-price components that only have to attest small chunks of software, such as firmware. Even though TPMs are relatively cheap, they increase the complexity and cost of a hardware component and system considerably if every single component needs to authenticate and attest itself and hence needs to establish a root of trust. Hence the aforementioned techniques could be a reasonable way of ensuring authentication and attestation while keeping costs low.

2.5 Security Protocol Modelling

In this section, I am going to discuss several techniques and tools that allow modelling security protocols and proving properties over these models. While some of the techniques assume a fixed attacker model, other techniques allow the precise definition of the attacker.

The need for provably secure security protocols is easily explained by the fact that so many seemingly correct protocols fail to achieve the intended security goals. This is due to the vast amount of attack vectors a protocol offers, as, in the typical Dolev-Yao (DY) attacker model, the attacker can interfere with network communication at any point in time and however it wants. This attacker model enables the attacker to listen to any messages that are sent through the network, prevent messages from arriving at the destination or insert own messages. The DY attacker's capabilities are only restricted by cryptographic methods, as such a DY attacker can only perform cryptographic functions if it knows the corresponding keys. To think through every possible interleaving of a protocol under a specified attacker model is almost impossible for a human and hence leads to the development of an abundance of tools. These tools support the developer of such protocols while proving properties about them.

Scyther [15, 16] is such a prover tool, which is easy to use in the sense that one can write up the behaviour of the different roles in a protocol in a single block of code. It offers unbounded verification, attack finding and visualisation of found attacks and can prove typical properties of security protocols, such as secrecy, agreement, aliveness and synchronisation. In order to prove properties and find attacks, Scyther performs a backward search, starting at the last action of a role applied. However, it is not possible to specify custom properties, e.g. an arbitrary lemma about the view of both parties on the state at a given time. Additionally, it does not allow to specify states of the protocol execution, which is often required for ongoing communication. Gen-Z protocols make use of a global state to prove through a signed exchange, that both parties agree on the current state of execution, hence proving these state equalities would not have been possible with Scyther.

The ProVerif tool[8, 38] mainly focuses on the adversary's knowledge and hence is not able to fully model several essential protocols. For example, ProVerif is only able to partially model the Diffie Hellman key exchange protocol, imposing restrictions on the exponents being used. As the Gen-Z security protocol uses the Diffie Hellman key exchange to establish a shared secret for a secure session, ProVerif was not suitable to model these types of protocols.

Tamarin [33, 27, 31], which has been developed by the Information Security Group at ETH, the successor of Scyther, solves many of these issues. Tamarin is able to prove the DHE, allows custom channel definitions, and enables us not only to write down our own, very general attacker model, but also allows us to define a global state that is required for modelling the Gen-Z security protocols. In order for Tamarin to achieve this generality, the write-up of the protocol no longer resembles programming a single agent with sequential steps, but defining multiset rewriting rules. The global state is denoted as a multiset that is changed through the rules that are applied to the current state. In addition to that, it is custom to model the state of a single party in the execution via state facts that are handed over from rule to rule of a single role. Tamarin also supports an interactive proving mode, which is very helpful in debugging how unintended traces come into existence. This step for step proving process where one can decide on any order of rule application, supported by the tool which only shows applicable rules, helps refine a model until it fits the specification.

As Gen-Z also intends to defend against malicious components, I required a way to model an adversarial party, participating in the protocol run. Tamarin's flexible attacker model allowed me to define an attacker that has full control over the component. After the first proof-of-concept, I hence decided that Tamarin offers all the functionality required to write proofs about the Gen-Z security protocols.

2.6 Disaggregated Systems and Memory Centric Architectures

In this section, we will look at which challenges lead to the development of disaggregated systems and memory-centric architectures. Further, I will describe why these approaches can solve some of the problems that are encountered with the current computing-centred architecture.

Already in 1995 Wulf and McKee [6] coined the notion of the memory wall. They describe how memory operations will eventually account for the total execution time of software, even under optimistic assumptions. Their model in particular assumes that the cache is of infinite size; hence there are only compulsory misses. Additionally, they assume the most favourable speed improvement ratios between memory and CPU and software load where only 20% of the instructions reference memory, which according to Hennessy et al. [3] is a lower bound to memory accesses from typical software load. They were also able to show that improving the bandwidth or latency [5, 4] is not able to fundamentally solve the problem but can only affect the time to impact. Nevertheless, due to the increasing gap between memory and computing chip speeds, at some point, the execution time of software will be entirely determined by how fast the data can be delivered to the computing unit.

There have been several attempts to solve this disparity. A potential solution is the memory-centric architecture, which does not centre around the computing unit, but puts a tremendous amount of storage at the core of a system. Around this massive block of memory, any number of additional hardware components can be assembled, which share access to the massive block of memory. One

advantage of such an approach over classical computing centric designs is that fewer copies between different memory locations are required. In a classical server, each CPU has a few gigabytes of local RAM. Now a cluster of such devices should serve a database of the size of several terabytes. In this scenario, not only is the memory slow in handing over data to the CPU, but it also requires to coordinate among the different nodes to keep the database coherent. This results in multiple invalidations of cache lines or copies into the local memory of a CPU.

Additionally, the amount of CPUs per gigabyte of memory is pretty much fixed in classical systems as there exist a fixed amount of lanes connecting the memory with the CPUs. In the memory-centric architecture, all CPUs can access the whole memory pool, which could be even large enough to hold to complete database in memory. Additionally, we can multiply the connections between memory and CPUs or add more computing units as required to increase the throughput between memory and CPU.

Further, memory-centric architecture is often connected with in-memory computing and storage-class memory (SCM). The idea is that some small operations can be applied to the memory directly instead of requiring to deliver the data to the fast computing nodes. This would allow, that only the data which requires complex processing, is sent to the computing nodes which support these types of operations. Complexer operations usually also take longer to complete, which decreases the effect of the slow memory on the overall execution time.

The advantage of SCM is that no additional time is required to load the data from disk into the memory. Even though backups on other media storage might be required, this can be done lazily, as the data in SCM is not lost if the system experiences a power outage or a crash.

This approach, however, has a prerequisite in the concept of disaggregated systems. This concept describes that if there is a fast enough link in terms of low latency and high throughput, it is no longer required that all components live on the same system board but can be attached through a network of links, switches and routers. This allows arbitrary system topologies, including the memory-centric approach. Depending on the implementation of the communication protocol between different components of such a network-connected system, it also reduces the interdependencies of different technologies, such as CPU and memory. This allows for faster deployment of new technology as other components communicate with the new technology through a clearly defined interface.

For an overview of different approaches, I refer to Hazarika et al. [61]. They compiled an extensive review of available concepts and architectures concerning DRAM, including hardware accelerators, disaggregated systems, memory-centric architecture, and Processing-in-Memory (PIMs).

2.7 Gen-Z Interconnect

In this section, I will first provide a high-level overview of Gen-Z before discussing which developments were prerequisites for the development of such an interconnect. Additionally, I shortly cover address translation and concepts similar to addresses in Gen-Z.

2.7.1 Gen-Z Overview

First, I will give a short, high-level overview of how Gen-Z works and the different elements we are looking at in the following sections of this thesis. Gen-Z is an interconnect that has the potential to replace legacy bus systems. For the interconnect design, Gen-Z uses the same layering as the Open System Interconnect (OSI) model. After the first introductory overview I will give a structured overview and explain Gen-Z with the help of the OSI model.

Each piece of hardware that can be attached to a Gen-Z system is called a component and contains a Gen-Z bridge which contains one or more interfaces to connect to the Gen-Z fabric to communicate with an arbitrary number of other components. The Gen-Z bridge is specialised hardware that is required on both ends. These bridges ensure that hardware such as the processor does not need to know directly about Gen-Z, but can only issue a request to a specific address in order to read or write data, device configuration or I/O streams. Such requests then are translated within the bridge

to Gen-Z packets and vice-versa for the response packets. Additionally, bridges handle all network related logic, that is required for reliably sending and receiving packets.

Through the use of a switched interconnect between different components, Gen-Z enables a wide variety of system topologies through system disaggregation; this also includes memory-centric architectures. The switched interconnect also allows the system to evolve dynamically to adapt to the workload by reconfiguring routing between hardware components or combining parts of the available hardware to smaller independent systems. This makes Gen-Z interesting for the application in large cloud computing centres, as the available hardware can be split up however needed by the customers. Gen-Z is also of interest for systems research as a few components, links and routers allow for a fast and reconfigurable system that can take on any desired topology. The potential use in multi-tenant systems is also supported by a wide range of security protocols and mechanisms that aim to enforce isolation between tenants. The wide variety of functionality Gen-Z offers, allows it to connect arbitrary hardware components without requiring secondary bus systems. Whilst many of the protocols required to build an operational system are already defined in the Gen-Z specification it remains extensible and adaptable for corner cases by allowing vendor-defined protocols to be tunnelled through the Gen-Z fabric.

Each component additionally supports two address ranges, one is the configuration space, and the other is the address range to address the actual functionalities of the component. Reading and writing the configuration address range can happen through unspecified out-of-band protocols or the Gen-Z specified in-band messages. Additionally, access to the configuration addresses is only allowed to the component with the component ID (CID) of the primary manager, which is written into the configuration of the component. If a component does not yet have a primary manager CID configured, it sets the first CID of the component attempting a configuration write as the primary manager CID. Suppose that during a later phase, the primary manager should be changed. In that case, this change needs to be explicitly written into the configuration by the current primary manager, which changes the field to the new CID of the new primary manager. The second address range is used to address the actual functionality of the component. It is accessible to all other components that can communicate with the component, as long as the management does not restrict access to the component and the address ranges itself.

In order to enable communication over the physical links between the components, Gen-Z defines a protocol and interconnect which follow the OSI model and defines different concepts on different layers of the OSI model as depicted in Figure 3. While it is based on the OSI model, some layers are not distinguishable, as single Gen-Z concepts execute responsibilities of different layers. The communication entity that enables any component to communicate with any other connected component is called an end-to-end packet. While these packets contain the information required to be routed from component a to component b, which is information generally found on the network layer, it also contains information from the transport layer that allows ensuring reliability, such as error detection codes or acknowledgements. Hence end-to-end packets span across the network and the transport layer. I will now cover the layers of the OSI model in more detail and explain which concepts of Gen-Z are related to them.

2.7.1.1 Physical Layer

For the physical layers, Gen-Z supports either PCIe, IEEE 802.3 or optical links that they specified themselves. Their own-defined links should support error corrected data speeds of up to 47 Gbps per lane and up to 256 lanes per link. In the work, we are going to assume that a physical layer is in place, that delivers the messages as they have been sent. For the security protocols, we will assume that the attacker controls the physical layers, hence can read any message and insert arbitrary messages on the links themselves.

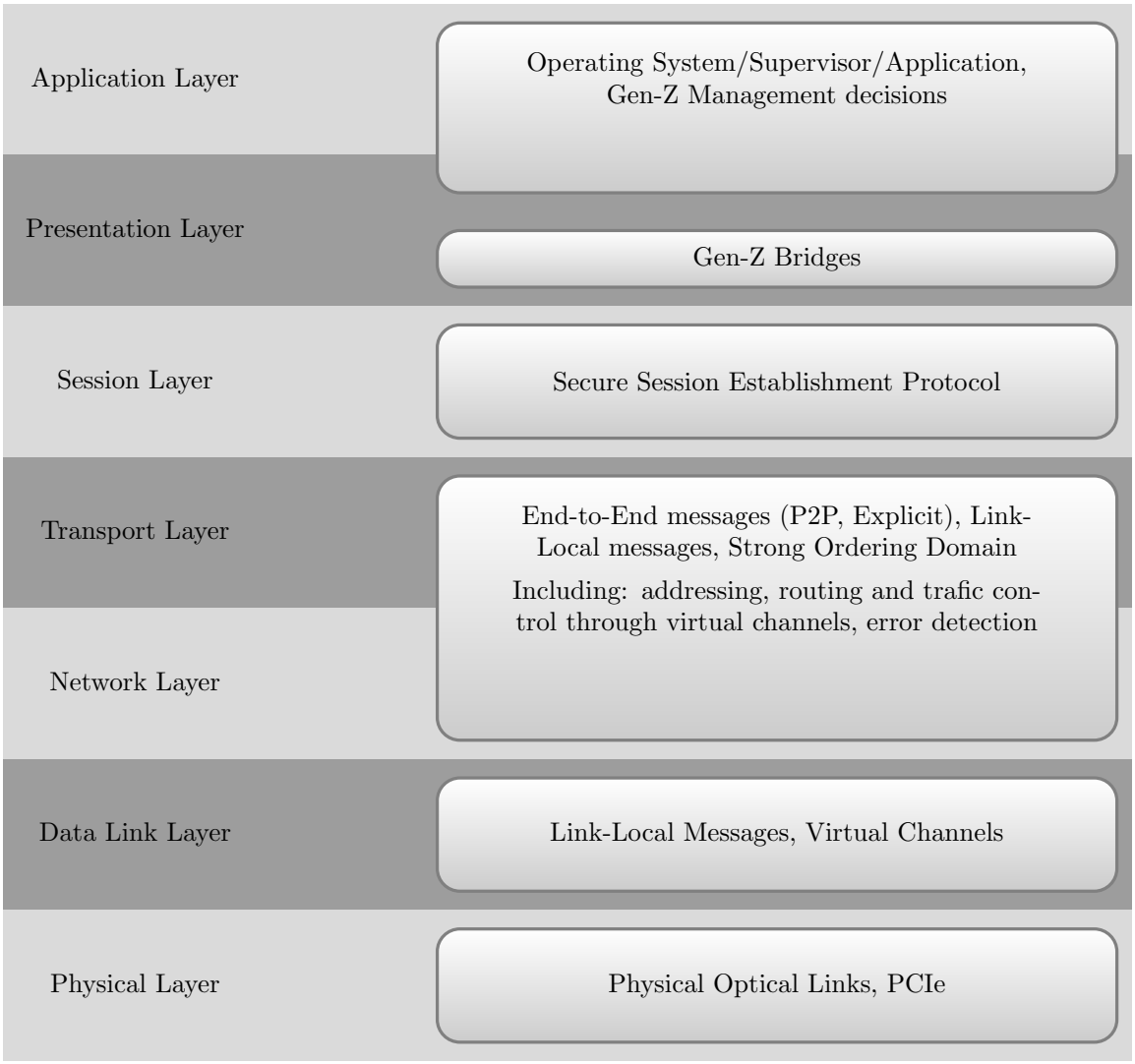


Figure 3: A rough overview of Gen-Z and the corresponding OSI layers. Note that some of the concepts of Gen-Z stretch across multiple layers of the OSI model, such as the end-to-end messages that do not only provide network layer functionality such as routing information but also transport layer functionality such as error detection codes and acknowledgements which ensure reliable transport.

2.7.1.2 Data Layer to Transport Layer

Several Gen-Z concepts extend from the data layer over the network layer up to the transpot layer. As such I am going to discuss the different concepts in these three layers in one section.

A Gen-Z interface can support an arbitrary but greater than zero number of virtual channels. These channels allow two connected components to have multiple logical streams over a single physical link. These virtual channels can be understood as multiple queues, for which we can have different configurations (e.g. enable PCIe compatible ordering, Traffic Classes) and different buffer sizes. A sender now can ensure that the link is best utilised, as blocked virtual channels do not mean that the other virtual channels cannot make progress. Note that only end-to-end packets are associated with a virtual channel, as link-local packets are so small in size (32 bits or 128 bits, respectively 1

or 4 words) that they fit into the smallest unit of 128 bits for sending and receiving data, such that they should be sent right away. Link-local packets are associated with communication between two adjacent components concerning the management of the link, while end-to-end packets are associated with communication between two endpoints, providing messaging services. For end-to-end packets, there can be all kind of different interests and reasons why a packet on top of a virtual channel might not be ready to be sent, e.g. congestion on a single virtual channel. Hence multiple virtual channels allow for better usage of the link, as a single blocked packet does not block the other packets. The link is time-multiplexed between the different virtual channels, which allows for "faster" and "slower" channels in the sense that smaller packets will complete faster than larger packets if they are not in the same virtual channel.

Link management through link-local packets is used between two directly adjacent nodes in order to keep the link alive through link idle operation packets, initiate (re)training of the link after a predefined amount of (re-)transmission failures or ensure link-level reliability as well as low-level functionality for link control and link-state transitions. Further, they do provide not only data link layer services but also network and transport layer services, such as explicit flow-control by providing feedback of the available free receiver buffer space on a specific virtual channel to the sender. This allows the sender to block sending packets on a specific virtual channel if not enough free space is available on the receiver's side. Additionally, it also offers link-specific management, e.g. bring up a link or transfer it to the down state.

For communication between two arbitrary nodes in the network, Gen-Z introduces the concept of end-to-end packets. Depending on the topology of the system, an end-to-end packet can be categorised either into point-to-point (P2P) or explicit type, where P2P packets only allow for communication between adjacent components. In contrast, explicit end-to-end packets contain addressing information which allows routing the packet through switches and routers to the denoted component in the denoted subnet. Subnets and components are identified through a configurable ID, which is given to a subnet or component during the initialisation phase. P2P packets further can be distinguished into P2P64, and P2P vendor-defined packet formats and protocols, where the P2P64 format can be used for general byte-addressable functionality.

All packet formats, link-local and all end-to-end formats have the Prelude Cyclic Redundancy Check (PCRC) and the length field in common. A sample explicit end-to-end request packet can be seen in Figure 4. The PCRC protects the length field and five additional bits that belong to the header of the packet, where the additional bits are interpreted depending on the packet type. The distinction between link-local and end-to-end packets depends on the value of the length field, the distinction between the different types of end-to-end packets depends on the configuration of the attached interfaces.

Further, both types contain an operation code (`OpCode`) which describes which operation should be executed upon receiving the packet. While link-local packets are concerned with keeping the link and virtual channels operational and working correct (explicit flow control, link resynchronisation, link control and link idle), end-to-end packet `OpCodes` actually denote functionality that serves a purpose to the system overall, e.g. read memory at the address specified in the address field of the packet. Note that the address field is distinct from the destination component (DCID) and destination subnet IDs (DSID). The IDs are purely used for routing the request to the right component, where the address is an intermediary address between CPU and component. For more details with regards to address translation see subsubsection 2.7.2.

2.7.1.3 Session Layer

In the most simple case, Gen-Z does not require a session to be established between different components. After the setup, each component can, given no additional restrictions are imposed by management nodes, talk to each component that it has an adequate connection to. In an explicit end-to-end network two components are required to reside within the same connected component, while in the P2P case the two components have to share a common link in order to be talking to each other. How-

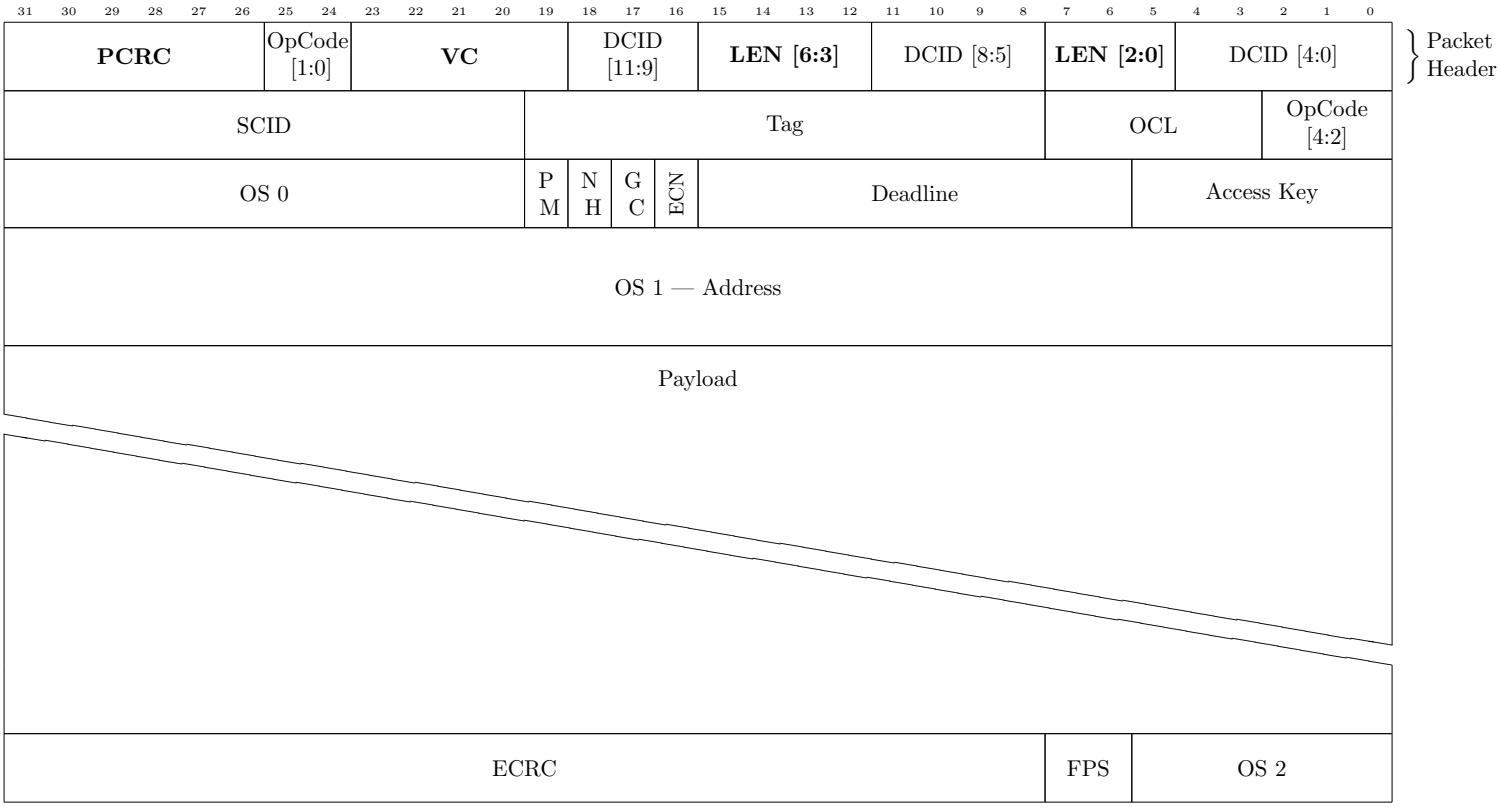


Figure 4: Format of a generic explicit end-to-end request packet. The fields that are in bold font belong to the header which is protected by the PCRC. All fields that are protected by the PCRC except the VC field are interpreted the same in all types of packets, link-local and all falvours of end-to-end. The bits in the location of the VC field has different interpretations depending on the packet type. A list describing the fields in the message can be found in Appendix B
.

ever, we might want to defend the system against a wide variety of attacks from different potential attackers, even such attackers that might have access to the hardware and the fabric itself. To achieve that, Gen-Z offers a secure session, which ensures that the communication between two components is authenticated and encrypted. For more details with regards to the establishment of secure sessions, see subsection 4.3.

2.7.1.4 Presentation Layer

In order for any hardware component to communicate over a Gen-Z fabric, it requires access to a Gen-Z bridge which is responsible for the translation of the address line and access type into a Gen-Z packet. Using this memory semantic information, it utilises its Gen-Z MMU (ZMMU) to look up additional information as required for the packet, such as the destination component ID (DCID) or the Access Key (A-Key). It then builds the packet and specifies any fields required such as the VC which depends on the configuration mandated by the manager before handing it down to the interface, which in flight computes PCRC and ECRC before sending the packet through the outgoing link. As such the bridge takes over the functionality of the presentation layer, as it translates the representation that the hardware has used for this operation, e.g. a memory read, to a Gen-Z packet representing the

same operation. Additionally, the bridge makes the process of waiting for the response transparent by handling the full communication flow before returning the result, again translated such that it is understandable to the hardware.

2.7.1.5 Application Layer

The application layer includes all the software that is running on top of the Gen-Z system. Regular user application and legacy OSs will run without modification, as Gen-Z hides the complexity introduced by such a protocol as well as possible. However, each Gen-Z system requires at least one management node, which takes over management tasks, such as including components into the fabric, enable or disable links in order to achieve the expected topology, configure single devices, configure the fabrics namespace (CID and SID planning) as well as enforcing security policies through the available mechanisms.

While it is possible that a small Gen-Z system possesses a single manager, in larger systems the role of the manager can be shared by multiple nodes, and there are also different management roles, which only take on a subpart of the whole set of tasks. While the primary manager usually controls the components that form a single physical unit, the fabric manager usually is responsible for multiple such units. There are other upper-level management entities such as the composability manager, which can be seen as the central manager, as it has to enforce the *grand plan*. The grand plan in Gen-Z is a high-level abstraction of how the system should be dynamically assembled through the use of managers. This could be some configuration file detailing how the system should look like or any other algorithm, which successfully completes the task of bringing up the system.

As such, the management is the most fundamental application that needs to be executed on top of a Gen-Z system, as it is required for the successful boot-up of such a system. The task of the management can be delegated to specialised software that runs below the OS or to the OS itself, if it is adapted to Gen-Z.

2.7.2 Gen-Z Address Translation

In this section, I will shortly cover where address translations take place in the Gen-Z communication protocol. Note that there are always two layers of addresses in Gen-Z, one that ensures that a packet is delivered to the correct component, which uses subnet IDs (SID) and component IDs (CID), and a second one, which consists of an actual address denoting some specific memory address, that might be mapped to memory, I/O or some component specific concept.

Address translation takes place across most of the layers of Gen-Z and serves different purposes in the different layers. In order for a CPU component to load a cache line from a memory component, the translation steps described in Figure 5 have to be completed. After issuing the cache line load request by the CPU, the CPU virtual address is translated using the MMU into a physical address that is specific to the CPU component. This address then is translated in the Gen-Z Bridge using the Requester Gen-Z MMU (ZMMU) from the CPU physical address to the information that is required to create a Gen-Z packet, including the DCID, potential Access and Region Keys (A-Key and R-Key) which can restrict access to devices and address regions as well as a Gen-Z address. The Gen-Z address can be of a different width than the CPU physical address, but other than that it is a regular memory address. These pieces of information then are combined by the bridge to a Gen-Z packet, and the packet is sent over the Gen-Z fabric to the destination component. The memory component may possess a responder ZMMU, that takes the Gen-Z address and translates it to an address that is specific to the component and indicates the actual location where the data is stored. Note that a component does not necessarily need another ZMMU, but can just take the address from within the packet itself.

The translation from CPU virtual to CPU physical address might not strictly be necessary if the CPU is only communicating with attached hardware through Gen-Z. However, there may are other addressable resources that are directly attached to the CPU, for which the MMU is required. On legacy systems, it could even be that the Gen-Z Bridge is treated as a memory-mapped I/O device, and hence the MMU translation is required.

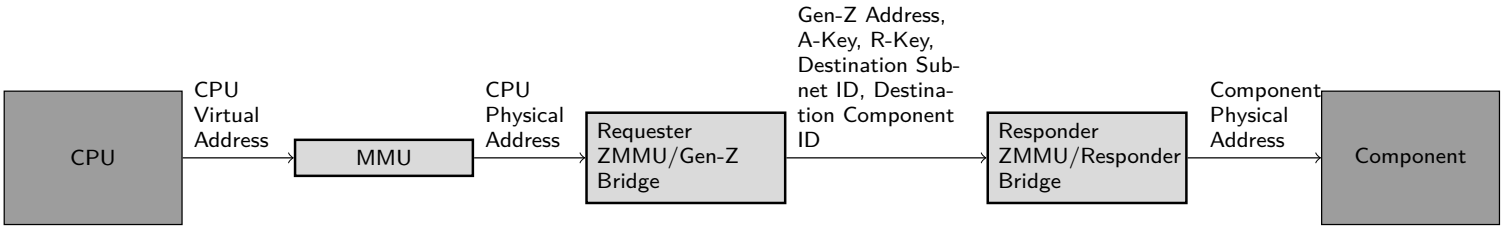


Figure 5: An overview over a complete address translation with all possible translation steps in a simple memory request.

In the translation within the Gen-Z bridge, much different information is inferred from the CPU physical address. Where the Gen-Z address is a typical address, reminiscent of physical memory addresses, the other concepts are specific to Gen-Z. The A-Key and the R-Key are responsible for restricting access to devices and memory regions within these devices. They serve a similar purpose to the MMU in a classical system but work a layer below the operating system. They are used to split up a larger system into multiple smaller logical units, where a component can only access other components if it knows the A-Key of the given component. If a component should be shared among several such smaller logical units, the R-Key can be used to provide access to a subset of the address space of this particular component. A-Keys are visible to routers and switches, such that unauthorised packets can be dropped early instead of being delivered to the component. Violations of the accesses to R-Key protected regions are regitstered within the target component itself. In case such an illegal access is detected, a security event is raised in order to bring the violation of memory protection to awareness of the systems manager or programmer.

Additionally, the bridge infers the information about the destination subnet and component. This allows the bridge to craft the packet that then is sent over the fabric to the responder interface. There, the responder bridge takes the Gen-Z address and translates through the ZMMU to a component physical address, that then is used to complete the memory request. For the response, the process requires less translations, as the request packet already contains all the information required to send a response.

Aside from these translations, there might be additional translations in the fabric, if a packet travels over subnet boundaries, as the SCID is defined as the ID of the component that inserted a packet into the subnet, hence components that connect multiple subnets need to keep track of different mappings they established between the subnets. A last potential exception to the denoted translation process are transparent accelarators. A request to a component may be answered by an accelerator, which can provide the response faster than the actual component. This approach can be seen as an intended man-in-the-middle that is built-in to the fabric.

2.7.2.1 Address Translation Services (ATS)

Aside from directly translating addresses, Gen-Z also specifies packet formats that could be exchanged with an unspecified address translation service. In that case, a component that wants to make a request can ask the ATS to translate one or multiple addresses for it. The ATS answers with one or multiple translated addresses, which in turn the requesting component can cache for future requests, which then are directly sent to the destination component.

If any of the information with regards to the translated addresses changes, the ATS has to send a Translation Invalidate Request to the component which received such a translated address. Any component receiving such a request should acknowledge the invalidation request once it has stopped using the translated address.

2.7.3 Technical Prerequisites

The development of Gen-Z pulls from different branches of research. In order to be able to provide comparable speeds to legacy systems, it requires low latency, and high throughput links to make access to memory and other devices fast enough. The latency and throughput requirements of different typical workloads, such as GCC, Memcached, indexer or pgbench, for a network of a disaggregated system have been measured and specified by Lim et al.[21, 26] and Han et al.[29]. As Gen-Z hardware is not yet publicly available, I do not yet have any numbers with regards to the latency of Gen-Z links, switches and routers, hence I cannot estimate how Gen-Z will perform compared to the theoretical requirements.

Fast enough network equipment is a prerequisite for disaggregated and memory-centric systems. These types of systems are potential solutions to the memory wall problem and therefore have been heavily researched. Even though Gen-Z enables a wide variety of system designs, it is specifically well suited for memory-centric and disaggregated systems.

An additional benefit of Gen-Z over legacy systems is that the memory controller no longer resides entirely within the CPU, but is split up between CPU and memory component. The generic memory management parts reside within the CPU. In contrast, the device-specific circuits, that control the actual memory accesses, the memory refresh and similar aspects, are shifted to the media controller within the memory component. This approach breaks the close interlock between CPU and memory technology, which so far required that a new memory technology also required support by the CPU. Through the decoupling of these two components, new approaches to memory can be deployed faster without requiring to exchange the CPU in order to support the new technology. As such, CPU and memory can now be developed independently of each other.

Such a structure also allows for greater flexibility in terms of types of services that a component can offer, as it does not require specialised circuits within the CPU or a driver in software, but can ship with its own media controller, that performs the actions requested by the CPU. Gen-Z enables memory semantic accesses as well as block-device accesses. By giving different addresses distinguished semantic meaning, the CPU can now directly talk to specialised hardware without requiring additional driver software to do so. Any computing, memory or I/O component can be directly connected to each other through Gen-Z.

This flexibility in setting up a Gen-Z system allows that a large pool of hardware can be dynamically reconfigured in different systems, such that we achieve a multi-tenant system without traditional virtualisation. This, however, shifts responsibility to the hardware in terms of security requirements. The hardware now has to offer more flexible ways of partitioning the system and protecting different address ranges from unauthorised access. As all the components have to work together to achieve the security goals, it is essential also to detect configuration errors or malicious activities and prevent such rogue components from infiltrating the whole system.

While Gen-Z arguably is the most versatile proposal for an interconnect, there exist several others, which overlap in functionality with Gen-Z. Benton [43] compares in his presentation CCIX, Gen-Z and OpenCAPI on a high level, providing a rough overview of functional ranges of the different proposed technologies and communication range. While CCIX and OpenCAPI focus on connecting hardware that is close to the computing unit or on the same SoC, Gen-Z enables connection across a wide range, starting at connecting close-range devices, such as the CPU to DRAM but also enabling connections across a whole data centre through the use of switched networks. Further, while Gen-Z breaks up the close interlock between CPU and memory, OpenCAPI works on virtual addresses and relies on the CPU for memory accesses. Hence Gen-Z is suited as an interconnect between any two entities in an extensive system, starting on connecting hardware on the same system board up to connecting multiple subnets, each representing a single cabinet.

As these types of interconnects shift a considerable amount of memory protection control to the hardware, the question of how well hardware can enforce security policies has been risen by Tsai and Zhang [57]. They describe several attacks but also potentially beneficial applications for what they call *one-sided network communication*. They coin this term for any communication where only on

one side has a general-purpose computing unit that runs software in order to enforce the security policies. They show several attacks on specific RDMA Network Interface Cards, such as Denial of Service attacks enabled through missing accountability, attacks on the hardware generated keys used in the respective security protocols or the fact that these devices expose physical addresses to remote machines. These cards are not Gen-Z compliant, but the authors still link to Gen-Z as a one-sided network communication hardware protocol. This implies that also Gen-Z devices might be subject to such attacks. Although Gen-Z does not employ software-controlled access control directly in all components, the interconnect enrolls several techniques that do prevent the attacks found by Tsai et al. Some of their attacks also heavily depend on the actual implementation of the hardware itself. As we do not have access to Gen-Z hardware at the time of writing, we cannot confirm or refute the found attacks on the key generation. However, Gen-Z does not have the missing accountability problem as the source component ID, and a potential source subnet ID has to be present in any packet and can be verified through other components that route the packet or in the case of a point-to-point communication by the receiving component. This does not necessarily prevent a denial of service attack, but the component that is causing the attack can be identified. Additionally, the congestion control that is employed by Gen-Z also protects against Denial-of-Service attacks through early detection of congestion and queueing up packets on the sender side rather than on the receiver side.

The missing hardware also lead to work by Hong et al. [62], in which they build an experimental Gen-Z system with Gen-Z specific hardware implemented on an FPGA. This allows them to run initial timing measurements and compare them to legacy access to different types of storage. Even though their implementation is experimental, they could show that in some instances Gen-Z type systems could outperform even local memory. They also found similar effects when using Gen-Z in conjunction with block devices. However, this does not mean that it is faster for all applications, as Gen-Z cannot directly leverage the CPU cache.

To conclude, in this section, I have introduced the different concepts specified by Gen-Z and explained how these might contribute to solving issues concerning the memory wall. Following the OSI layers, I outlined the different functionality provided by individual parts of a Gen-Z system on different layers of abstraction. Further, I highlight the different addressing concepts a Gen-Z system needs to handle and at which boundaries address translations take place. Finally, I provided an overview of related concepts to Gen-Z, similar specifications, research that has been prerequisite for Gen-Z and recent research based on the Gen-Z specification.

3 Formal Languages

In this section, I am going to talk about formal models and specifications as well as their application to different parts of a system. First, I will motivate the benefits of formally modelling different aspects of a system and define how an optimal formal language should look like and which guarantees it should achieve. After this, I will introduce multiple different modelling languages and tools that are applied to hardware and security protocols.

Models and specifications are related but not entirely the same. While formal models are not required to have specified the complete system as long as the missing parts can be replaced by either proven or assumed properties of the missing part, the formal specifications also have to fulfil a purpose in communicating the underlying concepts and ideas of a to-be implemented system. As such, formal specification languages also should have the ability to deliberately leave certain decisions to the implementation. Nevertheless, the specification language should provide minimal requirements with respect to any potential implementation, such that the specification does work as intended with any conforming implementation. Often the boundaries between formal model and formal specification are not as clear, and hence in the following, I am not going to distinguish them.

The ideal formal specification languages would be to have one specification in this given language and then to be able to transpile it to several targets, firstly a human-readable version (e.g., a PDF, with tables, flow charts, and text), describing how the product behaves and how to use its interfaces, secondly a formal and provable model, such that we can prove the claims that are made in the specification part of the language, thirdly an executable model, which would allow the automated generation of traces for a prover or even automate test generation for any implementation based on the specification. Of course, it would also be neat to produce an actual implementation directly, such that we could test the specification in a real environment, but let us postpone this for now, as we are currently not nearly close to such a target.

Formal languages can not only provide us with proofs, test implementations and functional models but could also result in improved, more complete specifications. This is a result of the fact that a formal model requires either to specify into great detail how certain aspects influence the intended behaviour or which assumptions and guarantees the model requires from any parts that have been deliberately abstracted away. One of the target outputs are proofs over the model. If essential parts of the specification were forgotten in a formal model implementation, one would notice as soon as a proof is attempted over the specification. Such proofs could provide the designer with insights and guarantee correctness with respect to the specifications and the desired properties. Once every claimed property has been proven correct through the formal model, we can be reasonably confident that the specification that is generated from such a model is unambiguous and complete with respect to the specified parts and properties. Furthermore, we can provide the implementer with additional resources, such as the executable model to poke around with and get a feel for how the specified piece of hardware or interface should behave.

There are several solutions for all the parts mentioned above, but none of them achieves all goals simultaneously. So clearly, this must be a hard problem to solve. When we go through the list of targets, we find that all of them live on different abstraction levels, which is the first explanation why we always have to decide for a modelling language according to level of abstraction on which it should be applied. We could write down all the details needed for all the different models in a single file; we only need to ask how we can do this such that the content of the file stays manageable for the authors. According to Robert C. Martin, "...the ratio of time spent reading versus writing is well over 10 to 1. We are constantly reading old code as part of the effort to write new code. ...[Therefore,] making it easy to read makes it easier to write" [17]. We could now argue that our ideal specification language can generate a PDF output for reading, so we do not need to make the code that readable, as long as it achieves all the outputs.

Nevertheless, this kind of specification implementation will contain bugs, see changes, and evolve. So even if only a single person is writing on the specification, it will take them more time reading through what they have written than actually writing it, trying to understand where a particular

part goes rogue. We should consider this in the language design, resulting in our first target for any language, simplicity. If we do not achieve simplicity in reading and writing, no one will ever write a specification in this language. Even though there exist proxy measurements for the simplicity of a language [2], I would claim that it is a subjective measure. Additionally, I would argue that many available tools for specification verification, automation, and model generation already fail to provide simplicity. They are not comfortable enough to use, especially as specifications evolve during the design or implementation process, changing something in the design, and hence in the code file, must be easy.

An additional goal for all these languages is to add as little target-specific information as possible. If we try to achieve all goals simultaneously, there will be plenty of code that is only there to achieve one of the compilation targets, not all of them. That has several disadvantages, the first being that a bug can be present only in one single output of the compilation, which renders the unification of all models in one file useless. We do this because we want coherency between specification, specification model, formal model, and ideally the executable model for testing purposes. We only can guarantee coherency between the different transpilation targets, modulo bugs in the transpilers used to generate the targets, if we start from a single source of truth. A second disadvantage is again coupled with simplicity. We need to push the writer into noting down all the required information, even though it has not yet finished formalising the content. If we allow for missing information, we end up with a model that cannot produce all compilation outputs, which defeats the purpose of the sought tool.

A problem with regards to a single language for all purposes is the fact that different concepts are best modelled differently. If we are modelling a protocol based specification, it is easier to follow the path of time, compared to when we are modelling hardware, where it is easier to follow the data. To resolve this, we would need a very versatile language, which allows us to model both easily and understandably. One could also argue in favour of multiple languages, that each for themself produces all the required outputs. In such a case, we would need to specify some interface, such as for the executable model, such that different models can easily interact with each other. Imagine we have a hardware component that does some calculation while having a network interface which runs some protocol with a requester. These two components would have to work together closely; probably, they depend on each other's input. So they need a way of sharing this information, which again requires some interface. Such an interface should not need to be specified by the writer but should result from the compiled targets and should naturally follow the specification written down. The same does hold true for proofs over multiple source models. In such a case proofs of one must be able to be used in the other, such that also issues can be found that require an interaction between the two different parts of the model.

We could go on, and probably find more issues with such a concerted attempt. We see that all the goals have different requirements with respect to the level of detail and hence also the language that describes the properties. Sometimes it might be possible to match some of the requirements up, e.g. Klein et al. [20] described such a scenario with the seL4 kernel where they combined a provable model, an executable model and a complete implementation. They were able to prove the kernel to be secure by starting off at a high-level specification over which initial properties have been proven. Then they iteratively applied refinement proofs until they ended up with a proven implementation of a kernel, which through the refinement proofs is provably equivalent to the specification. Nevertheless, also in this case, they required different languages to describe different levels of abstraction, and they had to be very careful to ensure coherency between the different layers, which places this whole process far from simple writing and compiling.

If we think about designing another specification language, we should first think about simplicity. I would argue that we should look at languages that combine the specification output target with the formal model target, as these two seem to be closer in terms of abstraction than the rest of the targets. Further, it would enable proving a specification to be correct or not, without having an actual implementation available. A helpful feature of any proof is that we can specify assumptions, which allow us to work with a not fully specified system, stating that there will be something that guarantees us specific attributes. For example, Klein et al. abstracted away from specific hardware to general

assumptions about the hardware in their sel4 proof. Once a specification reaches its final state, the only assumptions that should be left in the specification are the ones specifying the interface to a layer below or above the specification. Implementation details should be left out, but the specification should mention the properties that a compliant implementation must fulfil. If the other layers we talk to are also formally specified, we even can integrate the proofs about the interface into our model.

The ideal language would provide us with a single implementation for all possible transpilation outputs. As such, this would guarantee us a single source of truth, which is another desirable property for a formal language. While I have explained above that achieving this is difficult, we might be fine with a compromise between different outputs. We might not need a full executable model from a formal specification, in order to be helpful for the implementer of a specification. As we will see, simple decoding of messages and translating input into semantic information, that is easier understood by the implementer and also more comparable to the information in the specification, e.g. by matching input to concepts, already provides a lot of additional information to an implementer over a purely textual representation of the input to concept mapping, such as Figure 4. Instead of having to deal with a long bitstring, an implementer is already presented with a set of flags and their values. This is not a fully executable model, but has value, as misinterpreting the bitstring is already excluded as a source of errors.

Such applications already have been explored in previous work, e.g. for the specification and simulation of the Enzian Coherent Interconnect (ECI) by Jakob Meier and Alexander Hedges [63, 54] using the ETH flavour of the Arm Architecture Specification Language (ETH ASL), discussed below in paragraph 3.1.1.1. In this work, we try to extend this concept over the boundaries of a single language and combine different levels of abstraction meaningfully, such that we can support the development of the Gen-Z specification, as well as the development of systems software that should configure and run on top of Gen-Z systems.

If we do not achieve a strong coherency between different compilation targets or different implementations, we have to resort to weaker ways of arguing that two implementations are functionally or semantically equivalent. As in this case we are dealing with a provided textual description of a specification, there is no formal way of proving equivalence between specification and our models. A potential solution is a technique also applied by Cremers et al. in [44], where they annotated the specifications with their model decisions. This helps to communicate the thought process that went into the model and helps to convey the precise information to others that wish to validate the model against the specification. Unfortunately, this does also not allow for automated comparison, but at least it makes it as easy as possible for fellow researchers and developers while forcing the model designers to state any assumptions made.

While such an ideal formal specification language would be beneficial for many reasons, I showed there are some fundamental issues, which prevent us from achieving such an ideal specification language. However, by combining several formal languages for specifying and modelling, we are able to achieve a wide range of functionality. In the following I am going to discuss several examples of these languages.

3.1 Hardware Specification Modelling

There exist several hardware modelling languages that capture the behaviour and functionality of parts of a more extensive hardware system. We evaluated them with regards to the different abstraction levels described above and how they can be applied to the Gen-Z specification. The two main groups of languages we looked at either describe the instruction set architecture of a single hardware component or describe a more extensive system with multiple components and how the information is routed between several components, potentially including multiple buses. Both groups are required to model the Gen-Z specification, the former by describing how a single component behaves when presented with particular messages on specific input or output interfaces, and the latter by describing how information should be routed and which nodes might influence the path of a message through the fabric.

3.1.1 Instruction Set Architecture Specification Languages

In this section I am going to describe ISA specification languages, that can be applied to model any hardware component that supports the concept of an ISA. In this group, I will describe the Arm Architecture Specification Language (Arm ASL) and Sail. Both languages are used to describe how a processor or more generally any piece of hardware behaves when presented with a specific input. Both support some form of decoding an input into a semantic concept, that then can be executed. The advantage of this approach is that one could quickly generate a human-readable version out of this, that only requires a few additional annotations to be understood on its own, as all the information is available to describe the functionality invoked.

Further, both languages support the concept of an encoding, which describes which function will be executed by a specific decoding. This information can also be compiled into a simple yet complete specification of the platform. Aside from the human-readable target, both languages are capable of executing software over a runnable hardware simulator. These prototypes could help in finding issues in specific implementations that should follow the specification in terms of functionality and completeness. Using the prototypes can also fuel the automated generation of test cases for actual hardware implementations and also serve as a point of reference for system software developers, that can early on test their implementation without actually having access to hardware. Further, as the execution is emulated, it allows a more convenient access to the full state of the system, which helps assist when debugging issues along the software-hardware boundary.

Additionally, Sail provides a transpiler to Isabelle/HOL. This kind of functionality is not publicly available for Arm ASL, but as described by Reid et al. [41], it is also possible to generate a provable model out of the Arm ASL code. Such provable models can be used to prove functional, and correctness properties of the hardware pieces and hence can be useful to verify that the desired properties are achieved.

So overall, we can state that Sail and Arm ASL both achieve the same wide variety of goals, including the specification, simulator and provable model targets I described above. They do that by reducing the scope of the language to a concrete instantiation of hardware, namely the hardware that provides an instruction set architecture. However, it is not easily proven that such a formal implementation is equivalent to any candidate implementation in hardware, but as the models can be used to generate at least parts of the specification, the equivalence to the specification is partially given. As such the models can therefore be valuable for auto-testing candidate implementations.

3.1.1.1 Flavours of ASL

During the initial phase of the project, I discovered that there exist two different implementations of ASL, which I dubbed ETH ASL and Arm ASL to distinguish the two. As Arm did not release tools for ASL early on, previous projects at ETH developed their own tools to parse and process ASL code into executable code. As in the meantime the Arm ASL tools became available, I noticed that there are substantial differences between the two flavours. As both flavours had additional features over the other, I could not just simply build a cross compiler in order to transfer the ETH flavour code files to Arm ASL. The differences I found so far are listed and described in greater detail in Table 1.

Additionally, Arm is reluctant in releasing more tools that would provide a greater variety of functionality to ASL. Some language features are not described in the grammar that has been published with their tools, and hence we could only speculate what these features are used for. For some of the features, we were able to extract their meaning from the source code of the ASL interpreter (ASLi) tool [55]. The hidden features we uncovered are listed below.

3.1.1.1.1 Arm ASL Hidden Features List

- `__operator1 <unop>`: Used to define unary operators in terms of functions, e.g. `__operator1 - = neg_int, neg_real;`, where `neg_int` and `neg_real` are the built-in negation functions. We

Feature Name	Description	ETH Flavour	Arm Flavour
`__postdecode`	Different meaning in both flavours. Arm: Code that is executed after the decoding step within the encoding primitive ETH: Establish state after decoding of the instruction	Different Interpretation	Different Interpretation
`__postencode`	Executed after the execution of the instruction	Supported	Unsupported
`__encoding` →`__decode`	The deocding block within the encoding block is used to decode instruction specific fields	Unsupported	Supported
`__instruction_set`	Specify to which instruction set this instruction belongs to	Unsupported	Supported
`__opcode <bit mask>`	Serves as a guard to check if the instruction that arrives at this encoding block actually matches the expected instruction. Helps to early detect compatibility issues of the decoding tree with the instruction set encoding	Unsupported	Supported

Table 1: Differing features in the different flavours of ASL

also note that Arm ASL supports overloading, as the `-` operator can be used with integers and real numbers

- `__operator2 <binop>`: Used to define binary operators or concatenation operators, e.g. `__operator2 - = sub_int, sub_real, sub_bits, sub_bits_int;`. Note that the `-` operator is now overloaded twice, once with respect to the types and once with respect to the number of arguments.
- Features of unknown functionality:
  - `__newevent`
  - `__event`
  - `__newmap`
  - `__map`
  - `__config`

3.1.1.2 Arm ASL Structure

For any ISA modelled using Arm ASL, the following structure is usually used for the different parts. The samples are given for an imaginative model of a single processor core.

- `core_decode.asl`: Contains the decoding tree that maps a bitstring to an instruction to be executed. In our sample, this would correspond to the decoding logic that is applied to a newly fetched instruction
- `core_instrs.asl`: Contains the code for the different instructions and instruction specific decoding, such as interpretation of arguments. In our sample, this would correspond to checking if the instruction should be executed by comparing it to a guard, then decoding the arguments that this instruction expects and finally executing the instruction

- `core.asl`: This file usually contains auxiliary functions and helpers that are specific to the concept that we are modelling. In our sample, this file would contain the logic for advancing the instruction pointer and fetching the next instruction

The `<component_name>_decode.asl` file contains one or more `__decode <ISA identifier>` primitives at the root of the file, each of which then contain `case(bitslice$_1$, ..., bitslice$_n$)` statements, which allow the model developer to pattern match to the bitslices, where `X` denotes that either 0 or 1 are acceptable in this position. The bitslices can be the full input or only parts of it, using the bitslicing operator of Arm ASL. The bitslicing of Arm ASL is `a +: b`, which denotes a bitslice that starts at `a` and ends at `a+b-1`. I will use the same bitslice operator throughout this document. The decoding of an instruction can complete either successfully when it finds the correct instruction encoding to be executed, or with an error, e.g. `__UNPREDICTABLE`. In Code 1, a small part of a `__decode` tree taken from the official Arm specifications is provided as a sample.

```
__decode A64
case (29 +: 3, 24 +: 5, 0 +: 24) of
    when (_, '0000x', _) =>
        // reserved
        case (29 +: 3, 25 +: 4, 16 +: 9, 0 +: 16) of
            when ('000', _, '000000000', _) => // perm_undef
                __field imm16 0 +: 16
                case () of
                    when () => __encoding aarch64_udf // UDF_only_perm_undef
            when (_, _, !'000000000', _) => __UNPREDICTABLE
            when (!'000', _, _, _) => __UNPREDICTABLE
```

Code 1: Decoding instructions of the Arm A64 instruction set architecture [59]

The `<component_name>_instrs.asl` file contains multiple `__instruction <name>` primitives, which in turn can define one or more `__encoding <encoding identifier>` primitives for the containing instruction. These encodings then can splice up the input into multiple fields that are specific to this encoding, e.g. mapping bit positions to semantic argument information for the instruction. Additionally, the encodings contain an opcode mask which serves as a guard by comparing the input to the mask and throw an error if the input does not match the mask. Further, an encoding contains an additional `__decode` block, which does the final, encoding specific decoding, such as extracting arguments, translating bits to datatypes and establishing default values.

Decoding that has to take place at the instruction level but is common to all the different possible encodings of the instruction is best put into the `__postdecod` primitive. This block of code is executed after the `__encoding` specific decodings have been executed, but before the instructions `__execute` block of code gets executed. The concept of the `__postdecode` could also be left out by putting the contained code in each `__decode` block of each encoding. However, this would introduce code duplication and therefore the `__postdecode` is better suited for decoding code that is common to all encodings of an instruction.

The execution of the instruction is described in a single `__execute` block of code. As such the execution should be the same for all encodings of the instruction. Differences between the different encodings of an instruction should be resolved within the encoding-specific `__decode` blocks and not influence the code in the `execute` block. If there are substantial differences in how different encodings execute, then they should probably be different instructions altogether. Differences in the encoding are ok, as these can be resolved in the different blocks of `__decode`. Conditional execution and branching are available in the `__execute` block, but this should only be used to branch depending on the input and the state of the modelled device, not depending on the encoding.

In Code 2, a small sample of instruction definitions extracted from the official Arm specification is given as a sample.

```

__instruction aarch64_memory_single_general_immediate_signed_post_idx
    __encoding aarch64_memory_single_general_immediate_signed_post_idx
        __instruction_set A64
```

```
        __field size 30 +: 2
        __field opc 22 +: 2
        __field imm9 12 +: 9
        __field Rn 5 +: 5
        __field Rt 0 +: 5
        __opcode 'xx111000 xx0xxxxx xxxx01xx xxxxxxxx'
        __guard TRUE
        __decode
            boolean wback = TRUE;
            boolean postindex = TRUE;
            integer scale = UInt(size);
            bits(64) offset = SignExtend(imm9, 64);

    __encoding aarch64_memory_single_general_immediate_signed_pre_idx
        /* omitted */

    __encoding aarch64_memory_single_general_immediate_unsigned
        /* omitted */

    __postdecode
        integer n = UInt(Rn);
        integer t = UInt(Rt);
        AccType acctype = AccType_NORMAL;
        MemOp memop;
        boolean signed;
        integer regsize;

        if opc[1] == '0' then
            /* omitted */
        else
            /* omitted */

        integer datasize = 8 << scale;
        boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);
    __execute
        if HaveMTEExt() then
            SetTagCheckedInstruction(tag_checked);

        bits(64) address;
        bits(datasize) data;

        boolean wb_unknown = FALSE;
        boolean rt_unknown = FALSE;

        if memop == MemOp_LOAD && wback && n == t && n != 31 then
            c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
            assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF,
    Constraint_NOP};
            case c of
                when Constraint_WBSUPPRESS wback = FALSE;       // writeback is
    suppressed
                when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is
    UNKNOWN
                when Constraint_UNDEF      UNDEFINED;
                when Constraint_NOP        EndOfInstruction();

        /* omitted */

        if wback then
            if wb_unknown then
                address = bits(64) UNKNOWN;
            elsif postindex then
                address = address + offset;
            if n == 31 then
                SP[] = address;
```

```
67                 else
68                     X[n] = address;
```

Code 2: Instruction definitions of the Arm A64 instruction set [59]

In this subsection, I introduced the instruction set architecture specification languages Arm ASL and Sail. As Sail and ASL provide the same expressiveness, for the remainder of the subsection and the thesis I focus on ASL. I highlighted that there exist two different flavours of ASL and explained known differences and missing features. Finally, I introduced the standard project structure for Arm ASL and the ASL syntax. Note that the project structure and syntax is the same for ETH ASL.

3.1.2 Information Routing in Hardware Systems

In this section I am going to discuss the application of Sockeye as a specification language for a Gen-Z specific simulator. While Sockeye is currently used for describing existing hardware, in this scenario Sockeye would be used to define the hardware and as a result of that also ensuring equivalence between the simulated hardware the OS runs on and the view the OS has on the hardware.

With regards to address translation and routing in hardware systems, I only looked at Sockeye, which is developed by the Systems Group at ETH. It is currently used to describe hardware as a hardware decoding network (HDN) and hence is an interface specification language, describing the hardware and its addressing and interrupt routing schemes for the software that needs to manage it. Additionally, it allows to prove high-level properties of the hardware and hence can be used to prove correctness and security properties of particular hardware specification. It does, however, not catch hardware implementation bugs, as it works on a symbolic level when looking at it from the hardware perspective. Additionally, Sockeye compiles into the Software Knowledge Base, which is used by the Barrelfish OS at runtime to configure and address hardware correctly.

For an implementation of the Gen-Z specification, Sockeye would serve as the specification of an instantiation a Gen-Z compliant fabric. Instead of describing existing hardware, Sockeye would, in this case, serve as a specification language rather than a modelling language by specifying how the system is structured. As such, it could be used to describe the systems hardware topology as well as serve as a point of reference for the system software. Also, it could be used to generate management code directly out of the specification, that achieves such a system state.

Without further additions to the language and its toolset, Sockeye would tick three goals. Firstly it would serve as a specification, that could be translated to a human-readable description of a Gen-Z system. Secondly, it would allow us to prove properties of such an instance of a system, and lastly, it would provide us with an implementation that allows a system to run on top of the specified instance through the SKB.

With only minor additions, we would also be able to provide a simulator, which in return would require simulators for single hardware components. This would be the first instance of the above-described interoperability of several formally specified models. In this case, several formally specified components, e.g. using Arm ASL, would be instantiated and connected through links described by Sockeye. Address ranges would be assigned to different ZMMUs and routers available in the network according to the Sockeye specification of the system. Such a system would be able to fully virtualise a Gen-Z system, enabling emulation of software over such a simulator. This approach would provide several benefits, including a combination of proofs over several layers of abstraction in Isabelle/HOL and auto-generated OS code, which simplifies porting the OS to different instances of a Gen-Z system.

Hence, Sockeye is a perfect candidate for modelling and specifying Gen-Z system topologies. If used in conjunction with a formal ISA architecture model language, it could provide a complete executable simulator of a Gen-Z system.

3.2 Hardware (Protocol) Security Modelling

In this section, I will motivate the application of security protocol modelling to hardware specifications. Further, I will discuss the tools used in this thesis to model and prove security protocols, describe

security properties or find potential attacks on such protocols. For an overview on the different available tools, see subsection 2.5.

There exist several interconnects, bus protocols and fabric specifications out there. Depending on their application and assumptions on the environment a system will be run in, they require some form of protection against malicious users, operators or even faulty configuration, to prevent interfering with the system's execution and the protection of the data that is processed.

We can usually distinguish between two different types of assumptions concerning the environment. The first assumption is that the hardware is fixed and will not be tampered with during execution. This assumption usually is made for small, enclosed devices, such as end-user devices, as it is cheaper to not protect against hardware attacks, and often we assume that a user would have observed any tampering attempts.

The second assumption hence assumes that hardware is untampered before being inserted into a more extensive system, but could be attacked at any point after installation. This is usually done for larger cloud computing facilities, as the whole system cannot easily be overlooked by a single security guard and further, the cloud provider often wants to prove that their system is secure and working as intended to prospective users. In subsubsection 5.2.5 I will show why this assumption is not entirely realistic.

An additional distinction needs to be drawn between different mechanisms of securing larger systems. On the one hand, we have hardware support for OS functionality, such as isolation of processes or emulated guest operating systems, on the other hand, there are cryptographic protocols, which aim to ensure confidentiality, authenticity and integrity across the whole system. In this section, we are going to look at how one can model the latter form of security mechanisms.

Attacks in a large cloud computing facility can take a wide variety of forms. First of all, we want to prevent the insertion of malicious components into the system in the first place. This requires some form of detection of tampered hardware. Overall, this is still an open research question, as explained in subsection 2.2, and goes way beyond what is specifiable. In general, we can state that this is a cat and mouse game and so far there is no known approach that is formally provable to detect all or a formally defined subset of attacks. Gen-Z tries to prevent malicious components from being integrated into larger systems by testing the sub-component throughout the manufacturing process and by requiring a signed attestation of the state of any sub-component at each integration step into a larger component. As such, these attestations form a chain that claims that the hardware has been untampered with at the time of signing. However, as formal approaches are not yet known, and the problem is not necessarily specific to Gen-Z components, I omit further discussions.

We are more concerned with secondary attacks on the system, once it is up and operational. This includes data theft, imposter hardware components, stealing secrets from memory, fabric or bus-eavesdropping. Further attacks might be similar to the cold-boot attack [13], where an attacker with access to the hardware might remove a piece of hardware at any time. The cold-boot attack makes use of the fact that data stored in memory decays slower, the colder the component is. A component under attack is cooled down as low as possible, before being removed from the socket. Now the component can be inserted into the socket of an attacker system, and a full copy of the memory content can be created. Such an attack cannot be prevented by software or any protocol, but it can be detected early and through in-memory encryption render the effects of the attack to be minimal.

The in the previous section described formal modelling mechanisms for hardware are not entirely suitable to model these kind of protocols and properties, as the output of the models only describes how the component and the network behaves, but one would have to model an adversary to interfere at any time with arbitrary messages and actions. It could be done by augmenting the outputs of said models with additional Isabelle/HOL functions, that model the behaviour of the adversary. However, it would be much nicer if we were able to specify the adversary with all the assumptions made on a better understandable level.

For this purpose, there exist several languages, including Tamarin, developed at ETH. Tamarin is a protocol prover that allows specifying arbitrary attacker models, interconnect models and protocols. On top of the specified protocols, which follow a state transition approach, one can define different

lemmata. An automated prover, that can be guided manually, helps to prove the such specified lemmata. This allows for a rapid description of a protocol and the assumptions, as well as relatively fast proving of certain properties.

The graphical output of the traces found help enormously in conveying the information and compare it to the specification, which makes writing lemmata and proofs a lot easier. However, there are also some drawbacks to Tamarin, first and foremost, Tamarin operates on a symbolic level. Hence we cannot easily connect the specified model with an implementation or execute the different roles as a simulator. This would be necessary to prove that a specific implementation achieves the proven security properties by the model.

Aside from that, we are able to generate some specification using the traces and specifying lemmata that denote the default execution. Since not all information is necessary to be drawn in a documenting trace, Tamarin offers functionality to reduce the complexity of the generated output graphs. This might be helpful for specification purposes, but when proving, this feature might hide important information from the developer. Hence it should only be used cautiously when proving.

Tamarin is a great match for any protocol based description that makes use of cryptographic primitives to achieve its goals. It is less suited for any hardware-enforced mechanisms, as these usually rely on not being intercepted by an attacker and that only trustworthy parties learn this information. It is possible to model such kind of protocols using Tamarin, but the included adversary model is not of great help to prove achieved goals when doing so. Hence modelling of process separation through address translation or capabilities can be modelled using Tamarin, but proving isolation requires us to encode all possible ways how one could interfere with the protocol. In such a scenario, Tamarin does not necessarily support the proving process; hence I would suggest using Sockeye or plain Isabelle/HOL for such purposes.

In this section, I detailed the different aspects of hardware protocol security models. First, I motivated the application of protocol modelling to hardware by noting that increasingly complex hardware takes over more responsibility for the system's security. Then I introduced some basic assumptions and distinctions concerning security claims based on hardware. Finally, I combine the assumptions with Gen-Z and the protocol modelling tool Tamarin and explain why it is well suited for modelling the protocol specified by Gen-Z.

3.2.1 Tamarin Project Structure and Syntax

When reading Tamarin theory files, it is important to note that it uses a set-theoretic notation. In Code 3 we can see a sample of a Tamarin rule. To each rule, there are mainly three parts, where the first bracket pair encloses the pre-conditions that the current state set needs to fulfil in order for the rule to be applicable. In this specific case, the pre-condition only states that a fresh value is required. A fresh value can always be generated, and hence it is always part of the current state, which leads to the fact that this rule can be applied at any time during the prove.

The arrow-brackets then enclose something that Tamarin calls actions. These actions are beneficial for writing lemmata and prove them, as within lemmata we do not have direct access to the state, but only to the actions that take place at a specific time of protocol execution. The `GenLtk($A)` action introduces the concept of a variable; in this case, we use `$A` to denote the party on which behalf the key-pair is generated.

The output facts are defined, which will then be added to the global state set. Here we see another modifier of facts, the `!`, which denotes a persistent fact. A persistent fact is one that is not consumed by the application of a rule in contrast to a normal fact which is removed from the global state set after an application of a rule that requires this fact in the pre-condition, the persistent fact can be used an arbitrary number of times by any rule.

```
1 rule Register_pk:
2   [ Fr(~ltkA) ]
3   --[ GenLtk($A), HonestUse(~ltkA) ]->
4   [
5     !ToCA(~ltkA, $A),
```

```
    !Ltk($A, ~ltkA),
    Out(pk(~ltkA))
  ]
```

Code 3: Sample of a Tamarin rule that uses a newly created fresh value in order to generate a secret/public key pair

The last concept that Tamarin offers are functional equations. While these are called functions, they cannot be defined arbitrarily, but only can establish an equation that relates different concepts to each other. For example in the sample given in Code 3, `pk` is such a function denoting the public key derrivation of a secret key, which could be defined together with the `aenc` (asymmetric encryption) and `adec` (asymmetric decryption) functions as such: `x.1 = adec(aenc(x.1, pk(x.2)), x.2)`.

4 Gen-Z Formal Models

In this section I describe the different models I implemented, give implementation detail and evaluate how the models perform in terms of speed, what guarantees they can give us and with respect to their versatility in fields of applications. I evaluated several tools (ASLi, Tamarin, ProVerif, Scyther, Sockeye Compiler) and wrote proofs-of-concept in different specification languages (Arm ASL, Sail, Tamarin). For tools that were missing, I implemented my own tools that took the model and transpiled them into other languages for better tool support. In the following, I describe the different partial models of Gen-Z that I wrote and my findings with regards to the specification languages.

4.1 Arm ASL Gen-Z Memory Component Model

This section is concerned with the implementation details of the Arm ASL model of Gen-Z. It provides some detailed views into the implementation and shows how I overcame some difficulties. The evaluation contains a short discussion of what we can gain from this specific model and how we can get the desired outputs. In particular I am going to discuss the executable simulator and potential proofs over the ASL model.

Before we go into the details, I want to name a few challenges I faced during the implementation. This should help to understand some of some of the design decisions made.

Arm ASL was originally designed to model and specify fixed-width instruction set architectures. Given that Gen-Z is a message based protocol, where packets can be of different sizes within certain boundaries, some of the Arm ASL structures have not been well suited for this particular application. In order to overcome these limitations, I had to adapt the ASLi toolset by adding additional instruction set architecture definitions, which covered the allowed range of packet sizes.

Additionally, the tools that were provided by Arm seem not to cover the full Arm ASL language, as several stubs in the implementation indicate. For a list of missing concepts see subparagraph 3.1.1.1.1. Notably, some tools were not publicly released, such as a transpiler to a not closer specified Satisfiability Modulo Theories (SMT) solver input language. This lead me to develop a proof of concept of a transpiler from Arm ASL to Isabelle/HOL. This transpiler is far from finished, but the static type declaration from Arm ASL makes it easy to transpile the types to Isabelle/HOL types. For the code, we would have to go greater lengths, as Arm ASL uses a procedural language to describe the behaviour. This would have to be translated using the monadic functions from Isabelle/HOL. The compiler stub in the project is embedded in the ASLi tool and could be extended to a full-blown transpiler if required to write proofs on top of the Arm ASL model.

As an example, I will describe a simple memory component that handles incoming messages for storing and retrieving memory at a given address. This model does only look at the simple functionality of storing and retrieving data, but it is not concerned with more advanced features of the Gen-Z protocol.

4.1.1 Implementation

In this subsection, we shortly discuss different aspects of the implementation. It is tightly coupled to the code that can be found in the code repository[1]. The main directories of interest are AslDesc and the asl-interpreter-sync directory.

Before we can start with the model implementation in ASL, we need to ensure that the ASL interpreter and compiler knows the width of the instructions that it should handle. That the ASLi can read and interpret the Gen-Z model, I added the instruction sets as denoted in Table 2. The format I chose for the ISA names is `<standard name><short version><message type>`, such that it is easier to understand which type is expected and also to follow along with the specifications themselves. In the following I use the convention introduced in paragraph 3.1.1.2 for bitslicing. The models are

[1] https://gitlab.inf.ethz.ch/OU-ROSCOE/Students/2020-msc-robrunne/2020-msc-robrunne-code

implemented in Arm ASL, which is roughly introduced in subsubsection 3.1.1, for more details see the grammar description in Appendix A.

ISA Name	Width [bits]	Description
`GenZ11Header`	32	Any correctly encoded Gen-Z message contains at least 32 bits, which I interpret as the header. From this header we can derive whether a message is link-local or a end-to-end message and how long the message is.
`GenZ11LL32`	32	The shortest link-local message, that consists only of the header itself.
`GenZ11LL128`	128	The second size of link-local messages
`GenZ11EEP2PV`	4032	This is required for end-to-end messages between directly attached components with a vendor defined protocol and message format
`GenZ11EEP2P64`	4032	The end-to-end message format specified by Gen-Z for directly attached components
`GenZ11EEE`	4032	The end-to-end message format for fabric attached components, which require an explicit operation class (OP class). Most of the OP classes are specified by Gen-Z, but some allow vendor defined protocols and formats

Table 2: Different Gen-Z specific instruction sets and widths. For all end-to-end packets, 4032 bits denotes the maximal size allowed by the Gen-Z specification. In order to satisfy the ASLi type checker, every end-to-end packet is zero extended at the of the message to match the expected width of 4032 bits.

Not all steps of the decoding and interpretation of an ingress packet depend solely on the packet, but on the state of the component interface itself. The first step in bringing the component model interface up is to establish a known state. This model only requires the Operation Class Selector (OCLSelect) to be configured, which corresponds to the bits [6 +: 8] in the field Interface I-CAP 1 Control within the interface's configuration space according to the Gen-Z specification. The OCLSelect specifies which of the different end-to-end packet formats shown in Table 2 is used by the interface. If an end-to-end packet arrives at this interface, it is interpreted according to the setting in the OCLSelect field. So far, no additional settings are supported, as with this information, the model is able to decode and interpret simple memory read and write packets.

A packet is received through the `ReceiveMessage(integer N, bits(N) message)` function. Note that in the code `message` has been used instead of `packet` but that in this section both denote the same concept of information that has been sent over a Gen-Z fabric. The integer argument `N` denotes the size of the input message, which does give an upper boundary on the size of the packet that is contained within the `message` argument. Before the processing of the packet starts, some sanity checking on the input is done by checking the size range. These sanity checks are only implemented for early error detection within the model and are not present in the Gen-Z protocol.

```
1  __decode GenZ11Header
2      case (16 +: 16, 12 +: 4, 8 +: 4, 5 +: 3, 0 +: 5) of  // length=[12 +: 4] :: [5
     +: 3]
3          when (_, '1111', _, '111', _) =>
4              // link_local
5              __field OPC 19 +: 5  // OPeration Code
6              case (OPC) of
7                  when ('00xxx') =>
8                      __field unpred 19 +: 2
9                      case (unpred) of
```

```
                        when ('11') => __UNPREDICTABLE
                        when (_) => __encoding _h_link_local_128
                when ('1110x') => __encoding _h_link_local_32
                when ('11111') => __encoding _h_link_local_128
                when (_) => __UNPREDICTABLE
        when (_, 'xxxx', _, 'xxx', _) => __encoding _h_end_to_end


```

Code 4: Decoding instructions of the GenZ11Header instruction set

If the input passes the sanity checks, the first step is to decode first 32 bits of the message, containing the header of the packet as seen in Figure 4. This is done by using a particular ISA type `GenZ11Header` that I added to the ASLi tool. The decoding tree of the header is primarily used to distinguish between link-local 32 bit wide, link-local 128 bit wide and end-to-end packets of any type and a width of up to 4032 bits. The detailed decoding tree for the header can be seen in Code 4. The `header_encoding` instruction then reads the header, with different encoding for each of the three types distinguished by the decoding tree and mapping the bits on different locations to the actual concepts, such as the PCRC, the length and the Operation Code (OPCode)/Virtual Channel (VC) field. As the state is global in ASL, I added a `DecodedMessage` type as shown in Code 5, which provides a structured way of storing the decoded information within the global state. Any additional type definitions can be found in the repository under AslDesc/types.asl[2].

```
type DecodedMessage is (
  integer length,  // actual message length in bits
  bits(7) LEN,  // value of the len1 and len2 fields combined
  bits(5) OpCode,
  // Type of the message GenZ11*(LL32, LL128, EEP2P64, EEP2PV, EEE)
  MessageType messageType,
  // Prelude CRC, only for a few bits in the first 32 bits (minimum packet size)
  bits(6) PCRC,
  bits(24) ECRC,  // Lives in the last DW of any message
  // If MessageType is an end to end Message type, the more detailed fields are
  contained here
  EndToEndMessage eeMessage,
  LinkLocalMessage llMessage,
  Error errorField
)
```

Code 5: Denotes the current state of the decoding and allows easy access to the information that has already been encoded

After the header is completely decoded and the required information for the next decoding steps is available, the `ReceiveMessage` function proceeds by initiating the decoding of the specific packet types. The link-local, explicit end-to-end and vendor-defined P2P packets are currently not further implemented than the decoding step. In the case of vendor-defined packet formats, further decoding is not even possible, as this depends on vendor-specific specifications. The encodings and instructions provided through explicit end-to-end packets are similar to the P2P64 end-to-end packets, but the explicit end-to-end packets exhibit higher complexity as routing-related information is added to the packet. As in the current stage of the project, I am not able to simulate a full fabric with switches and routers, only the P2P64 packet decoding and instruction is implemented in greater detail.

The primary step in decoding a P2P64 packet is to distinguish different operation codes (OpCodes). There are some minor flags which can have an influence on the handling of a packet, but for the simple proof of concept I describe here, the OpCodes provided enough information. As for this proof-of-concept, I modelled a memory component, the instructions revolve around reads and writes. As Gen-Z does in general not enforce cache coherency, but shifts that to specialized instructions, the cache-coherent instructions were ignored for now.

[2]https://gitlab.inf.ethz.ch/OU-ROSCOE/Students/2020-msc-robrunne/2020-msc-robrunne-code/-/blob/master/AslDesc/types.asl

In order to be able to read and write memory, some representation of memory is required within the model. I used a similar implementation as Arm used for their processor models[59], which is also supported by the ASLi tool. Within ASL, RAM is a special type that takes an address width and a default value for addresses that have not yet been written. Within the ASLi, RAM is implemented as a mapping between addresses and values that are stored at these addresses. If an address has never been written before a read to this address is issued, then the specified default value is returned. If an address is written the value is inserted at the specified address, potentially overwriting previous values.

4.1.1.1 Arm ASL to Isablle/HOL Transpiler

As Arm does not publicly release additional tools, I started working on a simple proof-of-concept for a transpiler from ASL to Isabelle/HOL in a first attempt to prove certain simple properties about the ASL model. Since ASLi is very well structured, it is straightforward to add additional backends for other compilation targets.

Overall the structure of the ASLi follows that of a simple compiler, with a simple frontend, no notable performance optimization in between and in the unmodified version a backend that interprets the abstract syntax tree (AST) generated from the frontend. To execute the AST, it follows the visitor pattern, for which a simple base class for the visitor (`nopAslVisitor`) is already baked into the project.

For the transpiler to Isabelle/HOL, I only required an additional backend that inherited from the `nopAslVisitor` and some additional command-line flags to configure the behaviour of the transpiler as well as specify Isabell/HOL as a target output.

The backend for Isabelle/HOL is not yet completely implemented, especially the executable code within the instruction is not always easy to efficiently translate to Isabelle/HOL. So far, the main aspects that are available in the transpiler are the built-in types and types that are derived from built-in types. As such, the typing system of ASL is available in the transpiled output; the definitions of the executed functions are not yet functional.

For further reference, the Isablle/HOL backend is available in the repository under asl-interpreter-sync/libASL/isabelle_generator.ml[3].

4.1.2 Evaluation

In this section, I am going to discuss how the ASL model performed in terms of achieving the goals of a modelling or specification language and other design decisions that could improve the modelling power.

4.1.2.1 Formal Language Performance

Modelling Gen-Z components with ASL as described above only required minor adjustments of the available toolset in order to arrive at an executable model of a memory component. The approach can be easily adapted for other operations, especially as ASL allows arbitrary code to be executed.

In terms of conveying information to the reader of the model, especially the split into decoding and instruction part seems to work well, and even though I did not attempt to generate a specification PDF from the ASL model, the decoding part could be easily translated to be human-readable, probably even in the form of a flow diagram. This does not convey all the information required in order to implement software on top of Gen-Z but ensures that the decoding in the specification is the same as in the model itself. A possible specification output could be further augmented by incorporating the comments in the source file into the output.

As it is not possible to prove properties directly over an ASL model, a transpilation to another language, which is better suited for proving, is required. My proof-of-concept of an ASL to Isabelle/HOL transpiler shows that this is possible; however, I did not yet start proving properties over the transpiled

[3] https://gitlab.inf.ethz.ch/OU-ROSCOE/Students/2020-msc-robrunne/2020-msc-robrunne-code/-/blob/master/asl-interpreter-sync/libASL/isabelle_generator.ml

output. Depending on the model choices and potential transpiler optimizations, the proving process can become complicated.

4.1.2.2 ASL Memory

The Memory implementation of ASL only allows for a single default value, which in the case of this model is zero. This would, in reality, only happen, if the system would zero-out the memory during initialization, which is unlikely. To better model the reality of memory, either random values or a range of default values for different addresses could be used. For the current small proof of concept, a zeroed-out memory is good enough, but in case of more complex models, which could also be used to test software, more complex default values could be helpful, especially for test case generation and testing of software. A static default value could lead to false assumptions about the state of the system and as such to faulty implementations and undetected bugs in implementations.

4.1.2.3 Missing Concepts

In the model described above, multiple concepts are missing, such as the ZMMU of the component, in-band management communication, a complete interface description, including multiple virtual channels or the control logic for the link itself. These concepts were left out deliberately in order to keep the proof-of-concept model easy and simple. Nevertheless, let us discuss if adding them to the current model would be a problem.

Adding the ZMMU to the current model should not be an issue but merely an additional level of indirection, which would allow us to model more advanced concepts of Gen-Z such as support for R-Keys and A-Keys. The fact that the Arm A64 model already implements an MMU, adding an MMU should not be an issue as it would merely be another mapping from a key, consisting of the address, the R-Key and the A-Key to the actual physical address. To set up R-Keys and A-Keys, the config space and the logic to communicate with the management need to be implemented, as these keys need to be provided to the component by the management at some point. As communication with the management is again solved through memory semantic access, this in principle is the same as reading and writing memory as we already do, but in another address space and with an additional check of the primary manager CID in the control space. This can be implemented easily through the usage of another memory element and defining the address space according to the specification.

Support for link management would require support for link-local packets, which can be easily added; the stubs are already provided in the current model. However, if these packets should actually have an effect, we require a model of a link itself. While we could try an implement links through ASL, e.g. through shared memory elements, I would argue that this is far away from the physical reality of a link in Gen-Z. As such I would argue that the best fit would be a Sockeye model of the different components and links within the system and a Sockeye interpreter, which then actually delivers messages that are posted to these links. To achieve this, a substantial amount of work would be needed on the Sockeye language and toolset. The link itself could be implemented as a pure information delivery system, which itself does not have any state. The only thing that is important there is that both endpoints of a link have the same understanding on the state of the link. In terms of the interpreter, a straightforward interface would be required that can be accessed by the ASL executable models. I would suggest that Sockeye sets up a number of sockets or Unix pipes and then starts the different ASL models for the components. In terms of matching the Gen-Z specification with regards to lanes in links, Unix pipes are closer, as they only enable one-way communication. This allows adding arbitrary combinations of forward and backward lanes between any two components, which would allow the modelling of different widths of Gen-Z links. As such, link-local packets and the actual implementation of it is less an issue on the ASL model side, but rather much work on the Sockeye toolchain. As such, they were left out for this proof-of-concept.

To summarize, the model provides a simulator with the minimal required functionality of a Gen-Z component, hence keeping complexity low. The model deliberately leaves out certain concepts, as for many of these concepts, tool support is missing, e.g. definition of specific interfaces and link-

local communication only makes sense if it can be applied to a fabric. As such parts are missing, I deliberately left out such concepts in the model.

4.1.2.4 Different Approaches to Receive Packets

If we think about how a packet would be processed on hardware, usually it would be stored in some buffer first, and the handler (hardware or software-based) would be notified with some kind of signalling mechanism. In the model described above, I skip the memory when receiving a packet, by requiring the packet to be already fully known, such that it can be passed to the `ReceiveMessage` function. The `ReceiveMessage` function then, however, proceeds with first decoding the header, which is how such a packet should be interpreted, as it initially needs to know how many additional bits to read from memory. As in the case of my implementation, the model needs only to know which type of message it is looking at, such that it can continue decoding the rest of the packet. For an end-to-end packet, the packet that has been given to the receiver will always be extended in size to the maximal packet size of 4032 bits. This is done in order to allow for a more straightforward structure of the decoding tree, as a single tree per packet type suffices, as a packet can now be fully decoded and split up into different fields by a single decode tree.

Another option would have been actually to put the packet into a seperate memory region, that is known to the receiver function. From there, the receiver function could read the header in order to find how many more lines of 32 bits need to be read and interpreted. However, in order to follow the decode-instruction model of ASL, this would require that each of the 32-bit lines can be interpreted independently, as the decode tree does not have access to the devices state directly, as this is not required when decoding a single instruction. As the interpretation of later words of a packet however depend on the previously decoded words, the only workaround would be to introduce a substantial amount of ISAs and decoding trees and shift a lot of the decoding logic to the receive function. This approach is closer to what the hardware component would actually do, but would not differ in terms of potential states it can express while diminishing explanatory power of the model, making it harder to produce an understandable specification output and increase the complexity of proofs in the Isabelle/HOL output.

4.1.2.5 Translation to Hardware

Alastair Reid noted that he wrote a compiler to procedural Verilog. While this could be interesting for tiny models, it is most likely unfeasible to translate a larger hardware component to Verilog such that it could be runnable on some FPGA. The ASL language does not try to be as close as possible to actual hardware in terms of how executions are described, which would make a generated output inefficient. As Alastair Reid detailed, his compiler actually translated every bit to a single lane, which renders the output of such a compiler pretty inefficient.

4.2 Sail Gen-Z Message Decoding Model

In this section, I discuss the thinking that went into how a Sail model of Gen-Z should look like. Sail has some features that are closer to actual hardware compared to ASL, which could enable automated model-to-hardware translation.

4.2.1 General Remarks

As the Sail model would mainly have copied the functionality of the ASL model, I did not complete a full proof-of-concept model after verifying that such a model would be possible. As a copy of the Arm A64 model is available in Sail, I compared this implementation to the ASL implementation and found that all required concepts are available. However, Sail allows for statically typed widths of vector types, that can be passed directly to any Sail function. As such the workaround for the constant widths of instructions required for the decoding in ASL would not be needed in Sail.

Additionally, Sail offers many concepts, such as registers and vectors, that are close to how hardware is described in hardware description languages. If the model would be restricted to only use concepts that are closely related to hardware concepts, it would potentially enable a model that is able to be executed in a simulation, that allows proofs to be written over it and lastly to be actually translated to hardware.

However, also for Sail, it does hold true that the instructions are implemented in code that is difficult to translate to an efficient hardware output. Hence much thought would need to be put into a model that should achieve a hardware implementation in the end.

4.3 Component Authentication and Secure Session Protocol in Tamarin

This section is concerned with the details of the Tamarin model implementations for the component authentication protocols as well as the secure session establishment protocol. Please note that in the following, I will be talking about theories, which is the notion Tamarin uses for a model. Before we go into more details of the implementation, I want to highlight a few challenges and restrictions that might explain one or the other design decision.

Tamarin theories describe multi-set rewriting rules in a symbolic model. As such, it is not required to map the concepts to the actual bitstrings. Even though I initially wanted to keep things as close to the actual encoding, it made the theory very unreadable and barely understandable. This mainly because Tamarin offers no concept of giving a constant another name. As this approach also does not help with the modelling power, e.g. the adversary model does not try to send other values for the constants, this approach is very error-prone. Hence I reverted more understandable representations of the constants by naming them accordingly.

An additional problem to keep in mind is theory complexity. The complexity is not necessarily a problem for the author of the theory, but for the possible states that the tool has to explore during the loading and proving phase. When loading a theory, Tamarin deconstructs rules, which means that it derives all potential state transitions that can be performed with a given theory. If a theory offers an abundance of potential branches, already this process can lead to a massive blowup in memory requirements and take a substantial amount of time. In some cases, the tool uses up all memory resources before the deconstruction can finish, which I dubbed a *practical non-terminism*, as the tool could potentially terminate with a vast but fixed amount of memory, that we cannot provide. Additionally, even when using 48 CPU cores, the deconstruction can take several hours, before the tool fails.

If the deconstruction already takes a while to finish, proving the lemmata does in most cases also take a while, except for the most simple ones. Hence, I started with an extensive theory that included all the details of both protocols (component authentication and secure session establishment), only to notice that even with reducing the scope of the model as far as possible, it is not possible to control the state explosion. As such issues arose, I decided to split up the two protocols, even though certain parts of the secure session establishment protocol depend on the state of the component authentication.

4.3.0.1 Assumptions Concerning the Extraction of Private Keys

Since Gen-Z leaves the implementation completely open, it would be perfectly fine for a manufacturer to store the secret key in some part of memory that is accessible to the firmware and let the firmware to handle the security protocols. However, as the authentication tries to protect against firmware level attacks, this would defeat the purpose of these protocols, as using this approach, the attacker could potentially learn the secret key and hence sign whatever message he wants.

In such a scenario, defending against the attacker using trust in the secret key would not work in any case. Hence I am assuming in the following, that some kind of secure co-processor is present in the component, that we trust it is difficult to extract secret keys and has the capability of signing as well as encrypting and decrypting messages. These types of chips exist and are widely distributed, e.g. the trusted platform module (TPM)[32] would implement such a mechanism. The protocol that

one can talk with the TPM is well-studied[34, 24, 36], the chip itself is less covered. This is not of missing interest in it, but as possession of such a chip alone could open the potential for certain attacks[25, 22, 18], vendors of TPM chips are reluctant to share them with a large circle of people openly. Unfortunately, the design of these chips is also not shared, such that no easy way of analysing the chip is available.

Additionally, as of today, no information about the design of said chips has been shared with the public. Hence I am going to assume that such a chip is deployed with the component and that it provides the guarantees that are specified for the TPM chips. For more information on TPM, I suggest the book "A Practical Guide to TPM 2.0" by Arthur et al. [35].

Note that a TPM chip is not necessarily needed to attest the state of a system, Seshadri et al. found that if we can measure timing precisely, we can get away with an efficient software implementation to take the measurements and sign them [9, 10, 12]. Note that the following proofs were executed with a theory that has been implemented under the assumption that a hardware chip with similar guarantees to a TPM is present and I did not check if the protocols provide the same guarantees if we use the software-based attestation protocols.

4.3.1 Implementation

We now will go through the several implementations developed for proving the component authentication and secure session establishment protocols.

4.3.1.1 Gen-Z Component Authentication Protocol

Gen-Z Specification Authentication Protocol

Initiator		**Component**
		$cert(K_C)_{K_{CA}}$
	Request Certificates $\longrightarrow$	
Verify Certificates	$\longleftarrow$ [Certificate1, ..., CertificateN]	
	Challenge($Nonce_1$) $\longrightarrow$	$s_1 = sign(Nonce_1, K_C^{-1})$
	$\longleftarrow$ Response(s_1)	
Verify Component is genuine		
	Request firmware measurements($Nonce_2$) $\longrightarrow$	$m = (h(\text{firmware}), h(\text{config}), Nonce_2)$ $s_2 = sign(m, K_C^{-1})$
	$\longleftarrow$ Firmware measurements(m, s_2)	
Verify firmware is correct		

Figure 6: The high-level protocol overview given by Gen-Z. Note that the component can possess up to eight certificates that can be used for authentication at different stages (e.g. during production, integration, boot-up, runtime)

To prevent compromisation of the complete system, each component that is added to a Gen-

Z system needs to authenticate towards any potential initiator. Adding a component can happen during the bring-up phase of a system or during the runtime. Further, this protocol needs also to be executed if a component wakes up from a deep low power state or is power cycled, as during this time, the component could have been swapped out. Additionally, any other component can ask for authentication at any point during runtime.

The management initially does shut down any communication to and from the component and only allows for the authentication messages to be passed to and from the component. This protocol tries to prevent attacks through counterfeit component substitution, compromised firmware and faulty configuration of the device itself. The Gen-Z specification only gives a rough overview of how the protocol works, but refers to the DMTF Security Protocol and Data Model Specification (DMTF SPDM DSP0274) [64] for the detailed protocol. In Figure 6 we see the overview that Gen-Z gives and in Figure 7 we see the actual detailed protocol as defined in the SPDM specification. Comparing the two protocols, we see that some stages in the SPDM specification are only executed if necessary, where Gen-Z seems to enforce the exchange of the certificates. This is not a difference when it comes to security, but when it comes to performance, it is beneficial to save as many round-trips as possible.

The initial theory is based on the protocol as specified in the Gen-Z documentation and only uses the SPDM message format as close as possible. It has not been entirely possible to adhere to both standards simultaneously in the version that has been available at the time of writing. While Gen-Z specifies using the data objects, it specifies a protocol that leaves out several stages from the SPDM protocol, which are required by some of the data objects as a state to be signed over. The Gen-Z component authentication theory hence always compromises on the side of the data model, such that it fits the Gen-Z protocol.

The most crucial deviation in the data models of the Gen-Z authentication theory happens in the model for the challenge-response. According to the SPDM specification, the message should contain a signature over the state of the protocol execution. This state should contain all messages starting with `GET_VERSION` up to the `GET_CERTIFICATE` if this information is not already cached on the initiator's side. Mainly, it is not possible in the SPDM data model that only the certificate exchange is recorded while the digest exchange has not taken place. However, in the Gen-Z specification, this is exactly what happens. The closest we can do without violating Gen-Z specifications and only as little as possible the SPDM specification is only taking the challenge messages into account for the signed state, as is depicted in Code 6, which describes the rule how the component does answer a `CHALLENGE` request.

```
rule component1:
    let
      SPDMV='0x10'  // Version number
      CHALLENGE='0x83'   // Code indicating a challenge request
      CHALLENGE_AUTH='0x03'  // Code indicating a challenge response
      SLOT_NUMBER=$slot_number  // Slot of target certificate
      SLOT_MASK='0b00000001'
      // Incomming message
      CHALLENGE_MSG=<SPDMV, CHALLENGE, SLOT_NUMBER, '0x0', challenge_nonce>
      // Outgoing message
      CHALLENGE_AUTH_MSG=<SPDMV, CHALLENGE_AUTH, SLOT_NUMBER, SLOT_MASK,
        h(CERT), ~responder_nonce, '0'>
    in
    [
      State_COMP_FULL_CERT(SPDMV, CERTIFICATE, SLOT_NUMBER, RESERVED,
        '0x1000', '0', CERT, firmware, ltkC),
      In(CHALLENGE_MSG),
      Fr(~responder_nonce)
    ]
    -->
    [
      State_COMP_CHALLENGED(SPDMV, challenge_nonce, ~responder_nonce, CERT,
        firmware, ltkC),
      Out(<
        SPDMV,
        CHALLENGE_AUTH,
```

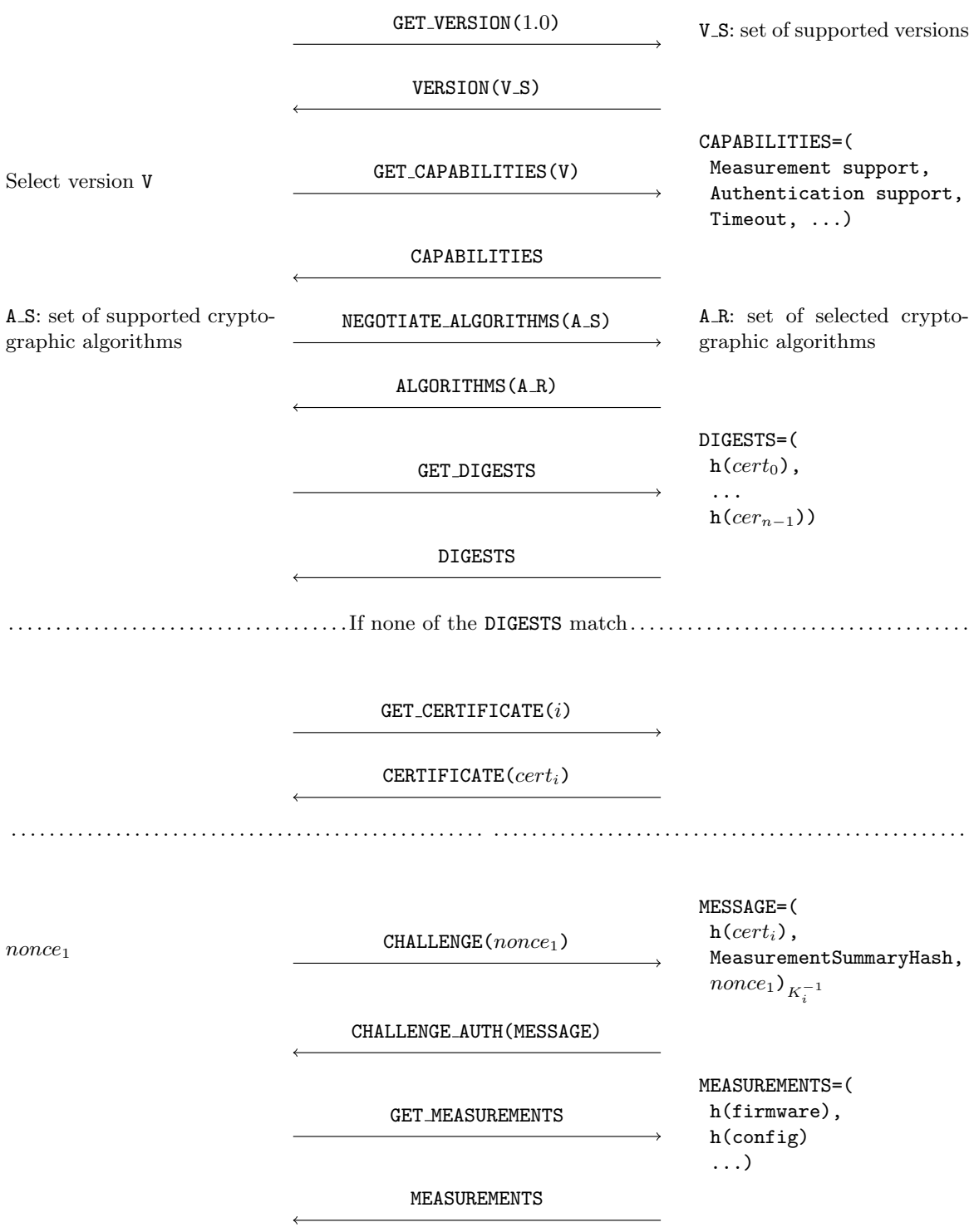


Figure 7: The SPDM specifications are a lot more specific. Note that all the stages of the protocol contribute to the state of both actors and this state is regularly included in the signed messages, such as the challenge response.

```
        SLOT_NUMBER,
        SLOT_MASK,
        h(CERT),
        ~responder_nonce,
        '0',
        sign(concat(CHALLENGE_MSG, CHALLENGE_AUTH_MSG), ltkC)
      >)
    ]
```

Code 6: Component answering a `CHALLENGE` request from the initiator. Note that the state that is signed over only contains the challenge request message and response

Aside from that, I also strictly followed the specification when it came to checking the certificates and the provided guarantees of it. In the Tamarin theory, any certificate that has been signed by a trusted CA is accepted, as the specification states that the initiator should check if the certificate chain is valid. Moreover, if the component can answer a challenge nonce with a signed response which contains the nonce, the initiator trusts that the component knows the secret key and is a genuine and trustworthy component.

```
    rule Register_trusted_hashes:
        [
        InSec($M, $R, ~firmware)
        ]  // a nonce as a placeholder for any memory that might change
    --[ GenTrustMeasurement(h(~firmware)) ]->
        [
        !TrustHash(h(~firmware)),
        Out(h(~firmware))
        ]
```

Code 7: Implementation of the database required to compare attestation hashes. The `!TrustHash` fact is used to compare to the measurements received by the component

After that, the only thing left is to check if the component also can provide the correct measurements. Here the Gen-Z specification details that a nonce must be included with the measurements, and the measurements must be signed by the private key in which trust has been established with the previous steps. The initiator then goes through the list of measurements it received and compares them to a database which lists the trusted hashes. In Code 7, the implementation of the attestation database is shown. The rule creates a persistent fact for each measurable part in the component that the initiator can consume to compare it to the measurements he received.

4.3.1.2 SPDM Component Authentication

While in the previous theory, I always chose to stick to exactly what has been written down in the Gen-Z specification as depicted in Figure 6, I also wanted to model the case where the protocol follows exactly the steps of the SPDM protocol as depicted in Figure 7.

This theory became a bit more complicated, as it has a few additional stages. However, the most complexity has been added because version, capability and algorithm negotiations would allow for potentially many different paths during the execution. I started by adding all potential branches to the model but soon had to recognise that with such a complex model, the prover would not be able to complete any proof. Hence, I chose to fix the configuration to the one that completes and includes all parts of the protocol. Especially when looking at the capability negotiation, we see that many of the capabilities are exclusive to one part of the protocol which provides its unique set of guarantees, such as the mutual authentication, where not only the component authenticates towards the initiator, but also vice versa. This allows both parties to gain trust in the party they are talking to and enables an encrypted and authenticated channel in both directions. If a component does not support mutual authentication, then the protocol steps are left out entirely, and the guarantees promised are weakened. For these aspects of the protocol, I always went with the strongest guarantees, such that I can model as much of the protocol to the test as I could and check if these additional properties also hold.

Additionally, the negotiation steps introduced many constants, that are exchanged, which made it harder for the tool to deconstruct the rules into a set of state transitions automatically. This resulted in partial deconstructions, which can be solved by either source lemmata or forcing the knowledge to the adversary. Partial deconstructions (PD) are especially harmful to the proving process, as the tool does not know which information the adversary can deduce from a not deconstructed rule that outputs a message to the network. As such, the tool assumes that the adversary can deduce arbitrary information. In order to resolve the PD, I did use both methods, in order to state that the adversary already knew the constants before they were used, I added inputs to the constant defining rules, which received the constants from the network. As no other rule established an output fact with these constants, during the proof, we let the adversary produce these outputs. During the later stages of the protocol, we then can use the fact that the adversary knows the constants and hence proof that any value that the adversary might deduce from these constants he must have already known before. This then finally resolved all the partial deconstructions in the theories.

```
    rule digest_rsp:
      let
        /* definitions omitted */
      in
      [
        ADVERSARY_UPDATES(firmware, config),  // adversary can update the soft state
        COMP_STATE_2(
          OLD_STATE, KEY_EXCHANGE_STATE, SPDMV, ltkC, cert, FLAGS, HASH_SEL,
          MEASUREMENT_SPEC
        ),
        In(IN_MESSAGE)
      ]
      --[
        Instance(C, 'Component'),
        DIGEST_RSP(STATE, SPDMV, firmware, config, ltkC, cert, FLAGS, HASH_SEL)
      ]->
      [
        // we do not need to include the digests in the state, as this is the same
        // as the h(cert) and cert is part of the state
        COMP_STATE_3(
          NEW_STATE, KEY_EXCHANGE_STATE, SPDMV, ltkC, cert, FLAGS, HASH_SEL,
          MEASUREMENT_SPEC
        ),
        COMP_SOFT_STATE(firmware, config),  // loop soft state back to the adversary
        Out(MESSAGE),  // Response to the initiator
        Out(<NEW_STATE, KEY_EXCHANGE_STATE, SPDMV, firmware, config, cert, FLAGS,
    HASH_SEL,
          MEASUREMENT_SPEC>)  // output the full state of the component for the
    adversary
      ]
```

Code 8: Sample rule of a dishonest component, that allows the adversary to update its state and gives away the state to the adversary

Further, this type of protocol should be able to protect the system from arbitrary misbehaving components. So far, we only looked at genuine and protocol compliant components. Since the adversary can inject arbitrary messages at any time during the execution, the only additional power the adversary has if the component is malicious is that he can tamper with the state of the component at any time and get additional information on the state of the protocol. To model this, I started with a copy of the theory modelling the honest component case. I added in- and outputs to the component rules as seen in Code 8, such that the adversary always learns all of the states of the component and also can inject a different state at any time. Note that the only thing that is kept secret by the component is the secret key, as we assume this key resides in a piece of hardware from which extraction of the key is difficult.

```
  rule adversary_modifies_state:
    [
```

```
    COMP_SOFT_STATE(genuine_firmware, genuine_config),
    Fr(~adv_firmware),  // we need to choose fresh values to prevent the adversary of reinserting the same values again
    Fr(~adv_config)
  ]
  --[
    ADVERSARY_COMPROMISED(),  // We use this action to refer the point where the component became malicious
    DISHONEST_COMPONENT()
  ]->
  [
    ADVERSARY_UPDATES(~adv_firmware, ~adv_config),
    Out(~adv_firmware),
    Out(~adv_config)
  ]
```

Code 9: Adversary rule that updates the state of a component to the adversarial configuration and firmware

In order to pinpoint the exact point in execution when the adversary modifies the firmware and hence renders the component malicious, I added an explicit rule (Code 9) that inserts the adversarial firmware or configuration into the component. One could argue that we should only defend against components that already contain the malicious firmware upon insertion into the system. Even though this is the easiest attack vector as an attacker could update firmware and configuration during the shipping of a component, I argue that the protocol should also protect against more sophisticated attacks, which inserts the adversary's payload during the execution of the protocol. With this rule, I allow the attacker to insert a malicious payload right until the last message exchange before the initiator deems the component trustworthy. If I were to allow injection after this exchange, the adversary would win the game every time, as it could just wait until we trust the component and then modify the state.

4.3.1.3 SPDM Secure Session Establishment

For the secure session establishment (SSE), there was only a hint within the Gen-Z specifications; hence only a single source of truth had been given in the SPDM specification. This section hence describes the Tamarin theory, modelling the SSE according to the SPDM documents.

When a Gen-Z component successfully passed the authentication protocol, it is taken into operation by the management. However, normal operation does not encrypt and authenticate packets that are transmitted through the fabric, which would allow an attacker to mount multiple attacks, starting with passive attacks such as eavesdropping on the fabric over simple injection attacks up to more elaborate attacks such as faking management commands and taking over parts of the system or replacing components entirely. Note that at this point, we assume that the guarantees provided by the authentication protocol holds, such as that none of the components on the network are malicious. We will discuss such assumptions later in section 5 and subsubsection 5.2.5.

The exchange used for the SSE is comprised of a Diffie-Hellman key exchange (DHE) [1], the exchange of the measurement summary hash of the component and an optional step in which the requester also has to authenticate towards the component. Instead of a DHE, a pre-shared secret could be used to establish a shared secret between the two parties, which then can be used to derive keys for the actual encryption of packets. An abstracted overview of the protocol can be seen in Figure 8

Altogether, multiple messages in the execution of the SSE protocol contain signed objects, containing the view on the state of the protocol by the respective party. This allows both parties to verify that they share an identical view on the execution of the protocol as well as that the other party has knowledge of the secret key that they claim to possess.

```
  rule complete_component_authentication_KEY_EX:
    let
      mgmt_cert=<<M, pkM>, CA_SIG>
      cert=<<C, pk(ltkC)>, COMP_CA_SIG>
```

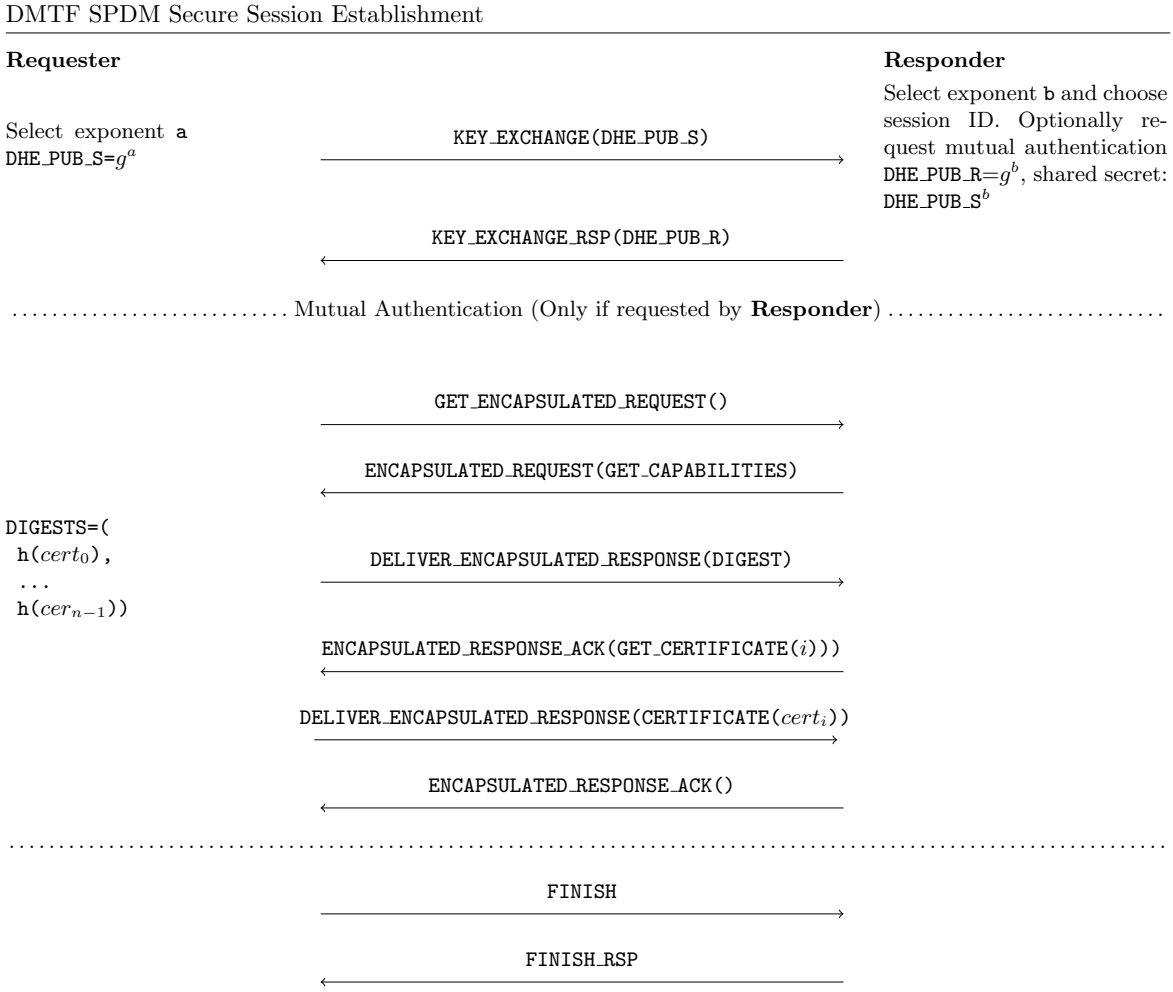


Figure 8: Overview over the Secure Session Establishment (SSE) protocol. Note that depending on the negotiated settings and the behaviour of the **Responder**, the mutual authentication steps do not place. Further, please note that the prove of posession of the secret key by the **Requester** happens with a signed state in the finish message, which also does include randomness selected by the **Responder**

```
        CERT_CAP='0b1'  // Support: DIGEST; CERTIFICATE msgs
        CHAL_CAP='0b1'  // Support: CHALLENGE_AUTH msg
        MEAS_CAP='0b10' // Support: MEASUREMENT & Signature generation
        MEAS_FRESH_CAP='0b1'  // Support: Everytime fresh measurements
        ENCRYPT_CAP='0b1'  // Support: MSG Encryption (either PSK or KEY_EXCHANGE
        MAC_CAP='0b1'  //  Support: MSG Auth, PSK or KEY_EXCHANGE
        MUT_AUTH_CAP='0b0'  // Support: mutual authentication
        KEY_EX_CAP='0b1'    // Support: KEY_EXCHANGE msgs
        PSK_CAP='0b00'       // Support: Preshared keys
        ENCAP_CAP='0b1'  // Support: *_ENCAPSULATED_REQUEST (Required for MUT_AUTH_CAP
    )
        HBEAT_CAP='0b0'  // Support: HEARTBEAT messages
        KEY_UPD_CAP='0b1'  // Support: KEY_UPDATE msgs
        HANDSHAKE_IN_THE_CLEAR_CAP='0b0' // Support: Delayed verification of data -
    only when KEY_EXCHANGE is active
        FLAGS=<CERT_CAP, CHAL_CAP, MEAS_CAP, MEAS_FRESH_CAP, ENCRYPT_CAP,
          MAC_CAP, MUT_AUTH_CAP,KEY_EX_CAP, PSK_CAP, ENCAP_CAP, HBEAT_CAP,
          KEY_UPD_CAP, HANDSHAKE_IN_THE_CLEAR_CAP>
      in
      [
        !CA(pkCA),
        !Cert(<M, pkM>, CA_SIG),
        !Ltk(M, ~ltkM),
        State_COMP_PROGRAMMED(firmware, config),
        !Cert(<C, pkC>, COMP_CA_SIG),
        !Ltk(C, ltkC),
        Fr(~KEY_EXCHANGE_STATE)
      ]
      --[
        TRUST_COMPONENT(M, C)
      ]->
      [
        REQ_STATE_9(~COMMON_STATE, ~KEY_EXCHANGE_STATE, cert, pkCA, mgmt_cert,
        ~ltkM, '0x11', FLAGS),
        COMP_STATE_4(~COMMON_STATE, ~KEY_EXCHANGE_STATE, '0x11', firmware, config,
    ltkC, cert,
        FLAGS),
        // Note: this is the state that the adversary could also have observed during
        // the component authentication, hence he knows it
        Out(~KEY_EXCHANGE_STATE)
      ]

```

Code 10: One sample of a rule that establishes the pre-required state in the requester. The `*_CAP` fields denote the capabilities that are established for this specific model

However, this state does also include messages from the SPDM component authentication protocol. As I decided to split these two protocols into separate theories in order to make proving over the theories feasible, this state has to be established before the SSE protocol can be executed. As also the capability negotiations happen during the authentication, multiple potential decisions affect the potential paths that can be taken later in the SSE protocol, e.g. support for a handshake in the clear or support for mutual authentication. In Code 10, a sample rule is given that establishes one of the several possible pre-required states for the requester.

This approach ensures that the model can explore all potential paths, without getting too large for the tool to handle. This allowed proofs over several aspects of the protocol. However, for some proofs, this was still too complicated, and at the time of writing, I did not find a workaround to proof all lemmata that I'd be interested in. For these lemmata, probably multiple theories are required, such that each theory only proofs the properties over a single path, which decreases the complexity of the theory and the proofs overall.

4.3.2 Evaluation

In this section, I am going to discuss how the Tamarin models performed with respect to the targets of a formal model and specification language and discuss alternative design decisions and which impact they might have.

As Tamarin, in general, is designed to provide us with proofs about protocols, all the models designed and implemented were able to prove some properties about them. It is however essential that one reads the proofs thoroughly or looks at the system state that proofs a lemma, as these often show paths that have been taken during an automated proof, that are not intended by the specification. One needs to be very careful during the implementation to not accidentally introduce other paths than specified, as facts that are generated can be consumed from any rule, hence following a naming convention helps a lot in defining a clear path. On the other hand, one needs to be careful not to reduce the modelling power of the implementation.

Concerning protocols run by hardware, Tamarin does a good job modelling the protocol and hence can be beneficial to be applied to protocol design intended to be run by hardware. Also, given the figures generated by Tamarin, it does an excellent job of describing the state of a system in a human-readable way. Given that one can specify a lemma that enforces the execution of the protocol in an orderly fashion, we could say that such models are also useful to describe a protocol for a specification purpose, while at the same time being able to prove properties over the same model. However, especially when working close to the hardware or when designing hardware, the engineer has to handle more complexity than just the bare protocol itself. Also does Tamarin not directly allow for a refinement workflow as described in [20]. As such, I can prove the protocol but cannot directly prove an implementation nor directly provide an executable simulator for the protocol, where one can interact with either role. For these types of proofs, we would require some modelling tools that are more specific, such as ASL or Sail, which on the other hand then make the proving of the protocol harder, as it requires to define an abundance of concepts that are readily available in Tamarin, such as the adversary, the transport or the protocol state update through messaging.

4.3.2.1 Tamarin Modelling Power

Tamarin provides us with an easy way to prove protocols and compared to other tools it is relatively fast (for performance measurements, see table 1 in [31]), it has its limits concerning the complexity of a protocol. As such, I had to reduce the theories quite a bit until they were simple enough to execute proofs effectively. If it were possible to encode the complete protocol in full detail in the Tamarin theory, arguing that the implementation does correspond to the specification would be straight forward, as one could read through the theory and directly compare it to the requirements from the specification. Now, with the reduced down theories, I have to argue why they are the same as the specifications and if the proofs would also hold true in practice.

The authors of Tamarin ran into a similar issue when proving TLS 1.3 [44] using Tamarin. In order to argue that their model is equivalent to the actual RFC, they provided the reader with an annotated version of the RFC, where they describe their design decisions and why certain decisions can be left out of the model. This is a structured approach, which prevents forgetting to mention design decisions, but does overall not guarantee equivalence between the model and the specification, which at last is what we are after.

5 Results

In this section, I will cover the results that were achieved by the different models and through the analysis of the Gen-Z specification.

5.1 Arm ASL and Sail Model Results

The ASL model clearly shows that this approach to modelling hardware is well suited for single hardware components and to model the behaviour of the hardware components, especially to model the hardware-software interface. With the support for transpiling to Isabelle/HOL these types of formal specifications can be directly proven to be correct with respect to the properties such a component should achieve, with the guarantee that the two outputs are equivalent modulo the transpiler. As such, we only need to prove the correctness of the transpiler in order to arrive at a wholly proven toolchain. As the transpiler to Isabelle/HOL and the actual interpreter share a large part of the codebase, proving the transpiler correct would also prove the interpreter, which then means that we have a provably correct executable model at our hand that can be used to test software, auto-generate test cases for real hardware or aid hardware design by fast and reliable prototyping.

Comparing the functionality of ASL to Sail, the common models and given the fact that the developers of ASL and Sail are currently working on a cross-compiler from ASL to Sail[58, 51], I can state that both languages are equivalent in terms of expressiveness with respect to classes of hardware interfaces that they can model. As such the results from ASL can be easily translated to Sail, which might be the better choice of language in the long-term as it is developed publicly and the sources to all the tools are publicly available.

Note that these findings are not specific to Gen-Z, but when applying this type of modelling to Gen-Z, I have been stretching the application area of these languages which were originally designed for modelling instruction set architecture to instructions that are non-fixed width, message-based and hence have some requirements to decoding and execution different to the requirements that an ISA imposes on a model language. With the proof-of-concept I have developped, I have shown that both concepts are equivalent in terms of expressiveness and hence tools and findings in one domain can be transferred to the other domain.

Additionally, using ASLi and my ASL implementation of the Gen-Z P2P64 protocol, I arrived at an executable simulator of a simple Gen-Z memory component. The main part of the work went into the simulator that takes over the tasks of a Gen-Z bridge and therefore can be reused to model arbitrary other components that should be compatible to Gen-Z.

5.2 Tamarin Model Results

With the help of the Tamarin models developed, I was able to prove multiple different properties of the Gen-Z protocol for component authentication and secure session establishment as well as find some specialised attacks.

As the proofs generated through the Tamarin prover tool are rather long and extensive, I do not include them in the thesis, but direct the interested reader to tamarin/results[4], where several proofs to the following results are given. Note that if a counterexample to a lemma has been found, the proof might have only be completed partially, as a single counterexample is enough to prove that a lemma does not hold for all existing traces.

5.2.1 Proofs over the Gen-Z Authentication Protocol Theory

Using the simple Gen-Z description of the authentication protocol, I was able to prove that an adversary can attack the authentication protocol if it obtains an arbitrary certificate from a trusted CA. As to the

[4] https://gitlab.inf.ethz.ch/OU-ROSCOE/Students/2020-msc-robrunne/2020-msc-robrunne-code/-/tree/master/tamarin/results

Gen-Z specification, we only verify if the certificate chain is valid. Hence we also accept a CA that has been obtained by the attacker itself. In fact, the initial implementation of the Gen-Z specification has shown that the attacker does not even need a component to attack the protocol, as he can completely take over the role of the component. However, please note that the attack on the Gen-Z described protocol might not be an attack on the intended protocol, as after clarification by the Gen-Z designers, they assumed the Gen-Z protocol and the SPDM protocol to be compatible to each other; hence the SPDM protocol should be applied for the authentication protocol.

5.2.2 Proofs over the SPDM Honest Component Authentication Protocol Theory

In this section, we go through the proofs proven using the SPDM Honest Component Authentication Protocol Theory. We show that the authentication protocol is secure with regards to achieving authentication under the specified assumptions of trusted hardware and non-extractability of the secret key. This model contains the additional assumption that the component is honest, hence follows the protocol strictly and does not cooperate with the adversary.

The first lemma proven using the honest component model states that in order for the protocol to reach the state where the initiator trusts the component, the component is required. So it is not possible for an attacker to completely take over the role of the component and fake this role to the initiator, but at least a component has to be started. Note that this statement is rather weak, as it does not rule out that the adversary tampers with the execution of the protocol and that in the end, both roles have the same view on the execution except for the fact that the initiator is convinced to trust the component.

For the second lemma, I was able to prove that an adversary that is in possession of a genuine component cannot use that component to mount an attack (similar to the cuckoo attack described in [18]) and establish trust into an adversarial component. This type of attack is known to work against some versions of the TPM attestation protocol, which usually relies on the fact that the verifier, which in our case is the initiator of the authentication, cannot check if the key that signed the measurements is controlled by the TPM that is built into the hardware. In order for this proof holding true, we have to either deploy our self-signed certificate to one of the components slot or trust that the CA issuing the certificate has not issued certificates to other entities, such as the attacker. Further, the certificate needs to be bound to the identity of the component, e.g. a serial number, which prevents an attacker from reusing the signed output of another component to fake knowledge of a private key.

The last lemma, I proved shows that both, requester and responder, share a common view on the state of the protocol under all possible execution traces. This equivalence in sharing the same view is a much stronger primitive than the two lemmata before, as it denotes that both the initiator as well as the component have the same understanding of the state of the component. This also implicates that an attacker cannot interfere with the authentication protocol actively, as in this case, it would need to be able to achieve differing views on the protocol and components state. Together with the second lemma, where I have shown that the adversary cannot use the honest component as an oracle to establish trust in an adversarial certificate and the fact that the adversary cannot change the views on the state by both parties, it hence fails to attack the authentication protocol for an honest component. As such, the SPDM protocol achieves to attest towards an initiator, that an untampered component is genuine and honest.

5.2.3 Proofs over the SPDM Dishonest Component Authentication Protocol Theory

In this section, we are going to look at the more interesting model, where the component actually is controlled by the adversary. This is the scenario against which the authentication protocol should actually defend the system, by not trusting the component, hence isolating the malicious component from the rest of the system.

Unfortunately, the additional rule that allows the adversary to update the component at any time lead to a state explosion. Even though the adversary has been restricted to a single compromise, it still

used up too much RAM and time to be useful for proving the lemmata. The issue most likely is that at every step, the prover needs to make at least one case distinction, which leads to an exponential growth in the number of achieveable states. This gets worse when considering that the adversary now has more information available to construct messages without requiring the involvement of the component. Hence we end up with an abundance of potential paths to check out during each step in the protocol run.

Even though I simplified the theory as far as possible such that it maintains equivalence to the specification, I was not able to effectively proof lemmata over the theory that are of interest. Some smaller lemmata were applied to sanity check the model and see if I can pinpoint down the exact moment where the state explosion happens, and some of these were provable. However, as of now, I cannot provide guarantees with respect to how the authentication protocol behaves under the assumption that the component might be malicious and if the attestation fails if the component has been tampered with.

The lemma I am most interested in is the one specifying that the protocol only achieves trust in the component by attestation of the components state if the adversary stays completely absent. The theory is designed that this case, the component behaves as intended. During my proof attempts, I was not able to find a counterexample to this lemma that would break the protocol, but I also was not able to complete the proof. As Tamarin allows for manual steering the prooving process, I tried to make better decisions than the prover itself, which produced smaller proofs in some cases, but could not prevent state explosions in most cases.

So, unfortunately, I cannot provide a definitive answer to the question if the authentication protocol actually achieves its goals when run under the given assumptions.

5.2.4 Proofs over the SPDM Secure Session Establishment Protocol Theory

In this section, we are going to look at the proofs over the SSE protocol theory and the guarantees I was able to prove. Note that for this theory, I assumed that the properties of the authentication protocol hold and hence we are communicating with an honest, unmodified component. If we would leave out this assumption, we could not achieve any secure session, as the attacker could learn the shared session secret immediately.

In this theory, I focussed on the establishment of the shared secret between the initiator and the component. I have been most interested in the case where no pre-shared knowledge is present. The protocol uses a DHE for the exchange of a session secret, which already had been proven secure. However, as DHE is vulnerable to a man-in-the-middle attack, I was wondering if the adversary can interfere with the protocol in a way such that both parties have different views on the shared secret state. For this, I introduce two helper variables, which denote the DHE private key as seen by the initiator or the component. The lemma states that given that both parties have been running up until the point where they trust that they share a secret with the other party, that after that point, the two secrets should be identical.

This lemma was successfully proven over all potential traces of the SSE protocol, including the ones that leave out mutual authentication. Even though leaving out mutual authentication does not allow the attacker to play a man-in-the-middle attack, as the adversary cannot impersonate the component, it still has a disadvantage in that the component does not provably know with whom it is communicating. It has to trust the initiator that the information with regards to the Gen-Z source subnet id (SSID) and source component id (SCID) is correct, and it is talking to the management for example. This leads to an imbalance for which I have not been able to prove an attack so far, but should be kept in mind, also for potential higher level attacks, that do not directly attack the SSE but might make use of this imbalance in knowledge between the two parties.

A secondary lemma that I proved only is concerned with the secrecy of the shared secret. The lemma states that given both parties established a shared secret that the attacker neither learned the secret as seen by the initiator nor by the component. This lemma is expected to be valid after the first lemma has been proven true, as this is a fundamental property of the DHE if no MitM attack

is launched. I used this lemma to sanity check the implementation and prove that no information is leaked to the adversary where this would harm the secrecy of the session.

The last lemma that I proved over this theory is concerned with the issue that an attacker could try to replace the component entirely, e.g. after the component is established through the authentication protocol, the attacker could try and replace the original component and completely take over the role of the component. This lemma states that if there has been established trust in the component that there exists no other identity that can establish a session with the initiator except than the component itself.

With the proofs outlined above, I can denote that the SSE protocol achieves a secure session independently of the negotiated configuration of both, initiator and component. While secrecy is given in any case by the secure session, the channel is only authenticated if the mutual authentication exchange takes place. If not, the channel is only authenticated in the direction from the component to the initiator. If we assume, however, that the authentication protocol works as intended, then only components that comply with the protocol should be allowed in the Gen-Z fabric, which the would render an authentication less critical, as any initiator that is already part of the fabric is trustworthy and hence sends his identity honestly (e.g. the SSID and SCID).

5.2.5 Discussion

In the previous sections, I made a multitude of assumptions in order to be able to prove or disprove certain aspects of the Gen-Z security protocols. Before we just close the books and assume that all is good, we first need to discuss if these assumptions are reasonable.

5.2.5.1 Trustworthy Hardware Assumption

The first assumption we make when receiving any hardware is that the hardware is genuine and untampered with. In legacy hardware systems, we often have no other option than just trust that no adversary interfered during the design phase, the construction or the shipping of the hardware. Gen-Z tries to elevate the guarantees with regards to trust into their components by requiring authentication of the hardware at any integration step. This sounds like a good thing to do, but as I outlined in subsection 2.2, detection of HTs is still an open research question. Hence authenticating a component may or may not make it harder for an attacker to hide malicious hardware within a component. As such the continuous authentication of hardware sub-components and components cannot provide additional security guarantees with respect to the hardware itself.

Additionally, this assumption also implies that the component has not been tampered with after the final production and authentication have taken place. This brings us to the issue of time-of-check-to-time-of-use attack (TOCTOU), where we check the hardware to be compliant and unmodified at one instance in time and sign the state at that time, but cannot necessarily trust that the component arrives in the same state as it has been signed for. If we want to ensure nothing has changed, all the hardware authentication would have to take place right before the hardware is inserted into the system.

The approach Gen-Z chooses for sure provides us with higher confidence in the hardware, but we cannot prove the absence of malicious hardware within the component. When compared to non-checked hardware, this is a step in the right direction. However, the claims by the Gen-Z specifications that the hardware must be correct due to the fact that at some point in time, a probably not complete authentication mechanism did not find any deviations, cannot be justified.

5.2.5.2 Hardware Support for Cryptographic Processing

A second assumption I made for the protocols above is that the secret key is stored in some specialised hardware that protects the secret from any software access. With the TPM there currently exists such a chip, which provides the guarantees required for the proofs above to hold. However, the Gen-Z specification does not call for such an implementation. A component could be entirely compatible

with the Gen-Z specification if it delegates this responsibility to the firmware, which then would load the secret key in order to authenticate itself during authentication and SSE. The issue with such an approach is that the component itself cannot necessarily detect an attack, perform the measurements of the firmware, the configuration and potentially other parts, and once it wants to sign the measurements, the adversary can extract the secret key from memory. After the adversary learned the secret key, it prevents the firmware from completing the original response and takes over the control over the component.

Once the adversary has learned the secret key, we cannot defend against the malicious component anymore, as the adversary can now sign whatever content it wants. The Gen-Z designers did clarify on request that some measurement needs to be taken but also stated that it would not need to be a TPM chip. I refrained from trying to abstract away from the TPM chip, as any other implementation would need to provide the same guarantees as to the TPM chip and protocol. However, if manufacturers start implementing their own version of a trusted hardware circuit to protect the private key and provide authentication service on behalf of the component, these protocols not necessarily have been checked to provide the guarantees they should.

Another option might be the usage of the software-based techniques for authentication and attestation, as presented in subsection 2.4. We need to be careful with this approach as it requires precise measurement of time and hence it requires precise knowledge of the speeds of the fabric up until to this component as well as that there might different approaches be required for different types of components.

5.2.5.3 Mitigating In Situ Insertion and Bump-in-the-Wire

The Gen-Z specification defines another protocol to detect if a component has been replaced in situ by another component. There are several attacks documented which require to remove memory from a system in order to read out its content, such as the cold-boot attack [13]. The other part of this protocol is to measure the timing of links in order to detect if a man-in-the-middle attack has been mounted in one of the links. Gen-Z does name this attack *bump-in-the-wire*.

For these protocols, I did not develop a model of the protocols, as these protocols do not provide us with additional guarantees than the other protocols would already do. First of all, if the component is authenticated and a secure channel is established then any other component that might have been used as a replacement in the in-situ attack could not correctly respond to the encrypted packets that arrive. In such a case, the session would need to be re-established, for which the initiator would expect that he could use the same certificate as before, e.g. the cached exchange of the certificates for the SSE protocol. However, the newly inserted component lacks knowledge of the private key of the removed component, which can be detected by the initiator. As such, no additional protocol is required in order to check if the component has been replaced. Note that aside from that the Gen-Z specification requires re-authentication of a component if the component has gone into a deep low-power state. This re-authentication can already detect that not the same component is present as before the deep low-power state, which renders the In Situ Insertion prevention protocol useless, as it only adds overhead without bringing additional value.

For the bump-in-the-wire, it behaves a bit differently, as SSE is not a strict requirement by the Gen-Z specification. An SSE may be established, but it is not enforced. As such, we could say that measuring the timing of a link in advance and then detect when something changes in the timing behaviour might indicate that a man-in-the-middle attack has been mounted on the link. However, given that Gen-Z supports not only fibre but also ethernet or PCIe buses as links on the PHY level, an attacker could easily passively eavesdrop on the link without changing the timing behaviour of the channel at all. So the attacker would still learn all the information that might not have been intended for the attacker. The only situation where an attacker might want to slice the link and put himself in between is in an active attack. However, as the attacker might has observed the full communication of that link and hence precisely knows how the attached component should behave, he could entirely replace the component without being noticed, as long as he does not need to re-authenticate. For

the regular service, the attacker would even be faster than the original component as the adversary has a shorter link in between himself and the network. To not be detected, he only needs to adapt his delay in answering, such that no time difference is noticed. If now a packet arrives that requires re-authentication, the attacker can route this to the original component, which happily authenticates itself. If a packet arrives that is used for timing measurements, the adversary can directly answer that in the expected time.

The adversary has multiple options to find the correct delay time he needs to apply when answering timing or in fact, any other request himself. One option is to observe a timing message exchange on the link and keeping track of the timing himself. The timing measured by Gen-Z is the round-trip time, for which a link-local message is used with a specialised sub-op code. The responding component is requested to respond as fast as possible to these messages with the minimal amount of processing required. The adversary does not need to know the full round-trip time; he only needs to know how much closer he is to the timing requester. To achieve this, he also measures the link-time between the path-time link-local (t_1) packet and the link acknowledge response (t_2) by the component. The adversary now knows that he has to respond to subsequent path-time link-local packets in $\Delta t = t_2 - t_1$. Another option for the adversary would be that he sends out a path-time link-local packet in both directions immediately after inserting himself into the wire. This would give the adversary more information but is also more detectable. However, any component can issue a path-timing packet on any occasion, so the adversary should still go unnoticed.

So far, we looked at the case with no secure session in place, let us now check how the bump-in-the-wire protocol could help us in the case that such a session is or has been established. As I have proven in subsubsection 5.2.4, the SSE is not vulnerable to bump-in-the-wire attacks, and once it is established, the packets that are sent over the link are encrypted and in the best case also authenticated in both directions. As such, the bump-in-the-wire protection described by Gen-Z does not provide us with additional guarantees that go beyond what the SSE already does.

To summarise, the two protocols above cannot provide additional guarantees independently of if a secure session has been established or not. They add additional overhead without additional benefits, and as such, can be removed from the specifications. Note that the timing information inferred denotes the round-trip time of a minimal packet, for which the receiving component should respond as fast as possible. This information might be valuable for other purposes such as resource allocation and scheduling, but do not provide additional security guarantees.

6 Future Work

In this section, we are going to look at several potential leads for follow up work to this thesis. While some of the proposed future work is merely cleanup, some other proposals could potentially foster follow-up projects itself.

6.1 Completing the ASL/Sail models

In the following, I am going to describe how the ASL models that I wrote could be developed further and brought to a state where we would be able to simulate a multitude of Gen-Z components. Note that this is not specific to ASL but could also be done with Sail. I am going to use ASL in the following paragraphs, but this does not mean that it could not be done with Sail.

The model I wrote in ASL provides a good entry point and can serve as a proof-of-work or as a point of reference for the implementation of additional components. Note that the central part of my implementation represents functionality that would reside in the Gen-Z bridge, and as such, could be applied to multiple different types of components. In my case, this has been a memory component, however in order to model and simulate a full Gen-Z system, at minimum a computing component would be required as well. For the computing component, one could take existing models, e.g. the Arm ISA model or the x86 model in Sail, add the bridge in front of these models and we got a complete Gen-Z computing component. Note that the current implementation of the bridge is minimal in the sense that it only covers the functionality that I needed in order for the memory component to work. Once this bridge is used more widespread though, it should probably be extended to support all kinds of configuration interfaces and spaces.

In order to prove properties over models implemented in ASL, a transpiler is required in any case, as ASL and the available tools do not offer proving functionality directly. There currently exist proof-of-concept implementations for Isabelle/HOL and Sail. Potential future projects could complete either of these transpilers, such that a complete translation of all ASL concepts to Isabelle/HOL or Sail is possible. Sail itself does also not allow proving properties, but it already provides a transpiler to Lem, from which Isabelle/HOL and Coq theories can be generated. An important part of such a project would be describing or even proving the correctness of the transpiler, as only this would guarantee that proofs over the transpiled output also hold for the actual model.

6.2 Sockeye and ASL/Sail

Since the ASL/Sail models only describe the behaviour of single components, these models are missing the fabric as an integral part of any Gen-Z system. Note that the fabric, in general, is mainly there to connect the components. Nevertheless, a description of the topology is required. Sockeye itself describes hardware using decoding nets. These decoding nets usually follow the specification of the hardware rather than defining the hardware topology itself. In the case of Gen-Z however, Sockeye could be a useful way to generate candidate topologies while ensuring that any potentially required properties hold for the candidates.

In order to simulate a Gen-Z system, multiple components need to work together. How they are connected could be specified using Sockeye. As the specification itself does not add much to the functionality yet, an interpreter would be required that can read Sockeye or one of the several transpilation outputs, such as Prolog, in order to actually connect the different components. This requires an interface between the interpreters of ASL/Sail and the interpreter of Sockeye, which would work like virtual connectors. It would also be possible to give such a Sockeye interpreter access to actual physical links, which could be used in conjunction with simulated components. This would allow greater flexibility in combining and building Gen-Z test systems for software development or research on disaggregated system designs and in particular memory-centric systems.

The second goal we have with ASL/Sail is to prove properties over the models themselves. As with the addition of Sockeye, this could be extended to prove properties not only over the components

but over the whole fabric. Sockeye already supports some approaches to proving properties with respect to address translation and interrupt routing. This functionality could be combined with the configuration and provable models of the components. Imagine a router component attached to other components via a Sockeye specified fabric. Through configuration, the router routes packets to and from any attached component or subnet. This routing table, however, has to be configured, for which the management could take the information from the Sockeye proved model. So the proofs would trickle down from Sockeye to the ASL/Sail components and inform the provable model generation with certain information, e.g. address ranges.

In the previous paragraph, I already hinted at another potential application of Sockeye in a Gen-Z system. Since Gen-Z systems allow for a very flexible combination of hardware, for which the management nodes are responsible for setting up the system in the desired topology. The Gen-Z specification denotes that the management steers the system to a state that is governed by a *grand plan*. This grand plan could be statically noted down or dynamically generated. However, in both cases, Sockeye could serve as the representation which directly allows proving properties of such a structure as well as generating code for the operating system to be executed on such a newly set up system. Comparing this to other previous systems, auto-generating code for the OS becomes more critical, as the system can change the topology several times, while rebooting in between reconfigurations or hardware changes is not even required. As such, the OS needs to become more flexible, also with regards to resource allocation and scheduling.

6.3 Sockeye Timing Model

As described in paragraph 5.2.5.3, Gen-Z has the built-in functionality for measuring the round-trip time of single links. As detailed, the bump-in-the-wire protocol that uses the timing in Gen-Z does not achieve additional security guarantees. However, the timing information might help the OS running more efficiently on top of such a Gen-Z system to make more informed decisions with regards to resource allocation and scheduling.

The Sockeye language could be extended to offer support for timing information during several phases of the model's application. In the first step, timing information could be added in advance during the implementation of a model, which then could be used to more accurately simulate message transport or serve as a sanity check during actual execution on the hardware. As a second application, timing information that has not been added in advance could be added dynamically to the compiled output of Sockeye. This information could be collected by the management from different components and be added to the model. This information then can be used by the OS to make better decisions with regards to ressource allocation and scheduling processes. If more timing measurements are required, e.g. over multiple links, this functionality could be added to the OS, which would send the smallest possible end-to-end message from one node to the node of interest and obtain the timing information that way.

6.3.1 Different Types of Timing Information

While the measurements will provide us with a single number per measurement and link, there are more options to timing information, especially during the design phase of a model. As such, the timing could be given as a single number as when doing the measurements, as a range of applicable timings or as a specific distribution of timing measurements. While a single timing measurement might be good enough for the simulation of an actual hardware link, for the runtime a range or distribution might be more useful, as these types of specifications would allow checking if the measured time is within the expected range. If timing is not within the expected range, it might be an indication for a change in hardware, a hardware defect or an attempted attack.

The range option might be the most useful for entering during the design phase, as minor deviations in timing might be possible depending on the condition of the link, the amount of messaging that is going on on the link and how much processing the component is currently doing. These ranges would

allow the OS to plan ahead of time, even before timing measurements are available. Additionally, ranges allow to sanity check measurements during execution and deviations from the provided ranges can be examined by the designer of the system.

Last but not least, timing should not be understood as only the information how long it takes for a request to reach an individual component, as this also depends a lot on the size of the packet and other influences, such as how computationally expensive a request is. The timing information should rather be seen as a proxy for closeness between different components.

6.3.2 Design Ideas

Sockeye does only explicitly express nodes and translations of addresses that take place at the given nodes. As such, the concept of a single link does not exist in Sockeye. Further complicating the matter is that Sockeye is modular, which might result in links going over the boundaries of a module.

I would argue that the timing information should be added separately by defining the time as a property between two nodes, independently of the address translation. The timing network should live below the address translation network, as the messages that are sent forth and back are sent over the physical links and only then are translated. Also, note that the timing information might have an influence on how packets are routed and hence potentially influence the address decoding.

For Sockeye modules, one could argue that the timing has to be added by the parent module, which instantiates a module and connects to the ports that the child module exposes. This assumes that a device usually exposes its ports without a long wire, but the wire is attached outside of the device. If this assumption does not hold true, we could transform ports, be they input or output, to separate nodes in both the child and the parent module, such that both can specify timing assumptions or behaviour on their side of the port.

The design described above would allow for a simple proof of concept for simple Gen-Z topologies, such as a star topology or any P2P topology. However, Gen-Z can get way more complicated and potentially allows a packet to be sent over multiple different paths. This results in a model where the OS cannot merely deduct the closeness of two components from the timing, as the packet might take a different path than expected. If in this scenario, the timing information still would be useful has to be tested out. One idea to cope with multiple paths would be that each path could represent a vote. E.g. each path from a computing component to a memory component contributes to the overall closeness of the core to the memory. We could then select to store data on the memory that is the closest under some measurement. Measurements could include the lowest average, shortest shortest path, shortest longest path and many more measurements could be found. Which of these strategies would prove to be best under differing conditions would be one of the main goals of such a project and most likely depend on the workload. Additionally, it would be interesting to see if the same strategy is optimal in both resource allocation and process scheduling.

Overall, I think that adding timing to Sockeye would provide multiple benefits to the OS, and we might find other applications where timing information could be useful. In order to be able to try this out, the Sockeye compiler has to be extended to support timing information. As long as Gen-Z hardware or similar systems are not available, it could be useful to also have a simulator, which lets us simulate different system topologies, where we can find out how the timing information influences the behaviour of the OS on top of such a system.

6.4 Isabelle/HOL to Sockeye

When we start proving properties over a model which combines Sockeye and ASL/Sail, we might find lemmata that we want to hold that can be proven incorrect by the given model. Suppose the prover finds a counterexample to the stated lemma. In that case, this counterexample could be useful to test either on the simulation or if the model represents an actual system, the counterexample could be tested on real hardware. Such an approach would have multiple benefits. When we start with running the counterexample on the simulation, such that we can step through the execution and identify where

the system takes a wrong turn. This could help designers in identifying issues faster in their model and also help build up a test suite to test new iterations of the model. If we have built a model of actual hardware and find a deficiency, running the counterexample on the actual hardware would be helpful to find out if the actual hardware also exposes this deficiency or if the model itself introduced this deficiency and therefore is not an accurate representation of the hardware. If the hardware itself also shows the deficiency, then the simulator could be used to step through the execution and support finding a workaround that prevents the deficiency, if it is possible at all.

In order for us to easily be able to come up with an instantiation of the counterexample that can be applied to the simulator or the actual hardware, we would like to have an automated generator that generates Sockeye configurations that expose the issue. This could also be done by hand, but as such would introduce additional points where errors can be introduced, and in some cases, it might be even difficult to find the exact right configuration to expose defficiency. In a first step, it would be helpful to see if it is possible to use the code generation feature of Isabelle/HOL to produce Sockeye according to the proof state. If this works, it might be possible to extend this also to write configurations outside of Sockeye that could serve us as test cases. As the proof state contains the counterexample to a stated lemma, it would therefore enable us to directly generate a Sockeye configuration that exhibits the faulty configuration.

Such an approach would make test case generation a whole lot easier and could automate certain parts of the test case writing. Further, it would reduce the number of overall required tests to the one that we know will fail according to the model's prediction. Such an approach would make testing a much more efficient process.

7 Conclusions

In this thesis, I have shown several approaches to formally specifying and modelling hardware-software boundaries as well as hardware security protocols. I have provided proof-of-concept implementations for multiple targets of formal specification languages, including simulators, transpilers to provable languages and proofs over the models themselves. I detailed which properties an optimal formal specification language would need to have and explained why we are not likely to achieve this with a single language. To solve this issue, we most likely require multiple languages that work together to describe a computer system completely. In section 6 I identified potential candidate languages that could well work together for a complete specification, which would be necessary to cover all possible Gen-Z systems.

Using the techniques and tools described in section 2 and section 3, I provided multiple models based on the Gen-Z specification as described in section 4, that have applications across the full specification of Gen-Z, starting at the physical layer and going up to the application layer of the OSI model. Using Tamarin, I proved certain aspects of the authentication and secure session establishment protocols of Gen-Z, while identifying certain additional assumptions that are required for these protocols to provide the promised guarantees. Using the proven properties, I was able to identify that the bump-in-the-wire and the in situ prevention protocols do not provide additional guarantees but introduce unnecessary overhead to the system.

The tools that have been developed during this thesis are designed in such a manner that they are easily extensible to other applications or additional concepts. The proposed future work hence can be easily integrated into the existing frameworks. While the tools in their current state are not able to cover the full extent of a Gen-Z system, they are able to cover certain aspects of the protocol. To summarize the experience with the Gen-Z specified protocols and interfaces, I would say that these types of systems are pretty exciting from a research point of view, as they allow an abundance of different system topologies, that can be easily adapted. This may foster additional developments in the area of operating systems and systems engineering in general. Even though Gen-Z claims that an OS can run on a Gen-Z system without modification and I would agree that there are cases where this might hold true, I would argue that in the general case the OS does benefit from adaptions to the underlying hardware structure. Additionally, some part of the software also has to take on the role of the manager. While this is not necessarily the OS, it would provide the OS with much more power to shape the OS according to higher level goasl to achieve. As Gen-Z systems can be reconfigured to have different topologies through software, automating the adaption process of an OS to the hardware will be one of the most pressing and challenging tasks there. In this thesis, I explored multiple ways of modelling Gen-Z systems, which allows further development of automated provers and code generation for modern operating systems for dynamic hardware systems.

A Arm ASL Grammar

The following grammar is generated directly from the ASLi project [55].

id
typeid
qualifier
intLit
bitsLit
maskLit
realLit
hexLit
stringLit
eol
i, *m*, *n*

l	::=			Source location
	|			
ident	::=			Identifier
	|	*id*		
typeident	::=			Type Identifier
	|	*typeid*		
leadingblank	::=			
	|	$\hookleftarrow$		
	|			
declarations	::=			
	|	*leadingblank declaration*$_1$ .. *declaration*$_n$		
declaration	::=			
	|	*type_declaration*		
	|	*variable_declaration*		
	|	*function_declaration*		
	|	*procedure_declaration*		
	|	*getter_declaration*		
	|	*setter_declaration*		
	|	*instruction_definition*		
	|	*internal_definition*		
type_declaration	::=			
	|	**__builtin type** *tidentdecl*; $\hookleftarrow$		
	|	**type** *tidentdecl*; $\hookleftarrow$		
	|	**record** *tidentdecl*{*field*$_1$ ... *field*$_n$}; $\hookleftarrow$		
	|	**type** *tidentdecl* **is** (*field_ns*$_1$, ... , *field_ns*$_n$) $\hookleftarrow$	S	DEPRECATED
	|	**type** *tidentdecl* = *ty*; $\hookleftarrow$		
	|	**enumeration** *tidentdecl*{*ident*$_1$, .. , *ident*$_n$}; $\hookleftarrow$		
field_ns	::=			
	|	*ty ident*		
field	::=			
	|	*ty ident*;		
variable_declaration	::=			
	|	*ty qualident*; $\hookleftarrow$		
	|	**constant** *ty qualident* = *expr*; $\hookleftarrow$		
	|	**array** *ty qualident*[*ixtype*]; $\hookleftarrow$	S	
ixtype	::=			
	|	*tident*		
	|	*expr*$_1$..*expr*$_2$		
function_declaration	::=			
	|	**__builtin** *ty qualident*(*formal*$_1$, .. , *formal*$_n$); $\hookleftarrow$		
	|	*ty qualident*(*formal*$_1$, .. , *formal*$_n$); $\hookleftarrow$		
	|	*ty qualident*(*formal*$_1$, .. , *formal*$_n$)*opt_indented_block*		
procedure_declaration	::=			
	|	*qualident*(*formal*$_1$, .. , *formal*$_n$); $\hookleftarrow$		
	|	*qualident*(*formal*$_1$, .. , *formal*$_n$)*opt_indented_block*		
formal	::=			
	|	*ty ident*		
getter_declaration	::=			
	|	**__function** *ty qualident*; $\hookleftarrow$		
	|	*ty qualident opt_indented_block*		
	|	*ty qualident*[*formal*$_1$, .. , *formal*$_n$]; $\hookleftarrow$		
	|	*ty qualident*[*formal*$_1$, .. , *formal*$_n$]*opt_indented_block*		
setter_declaration	::=			
	|	*qualident* = *ty ident*; $\hookleftarrow$		
	|	*qualident* = *ty ident opt_indented_block*		
	|	*qualident*[*sformal*$_1$, .. , *sformal*$_n$] = *ty ident*; $\hookleftarrow$		

| | *qualident*[*sformal*$_1$, .. , *sformal*$_n$] = *ty ident opt_indented_block*

sformal ::=
| | *ty ident*
| | *ty*&*ident*

instruction_definition ::=
| | __**instruction** *ident* ↩ {| *encoding*$_1$ ... *encoding*$_n$ *opt_postdecode* __**execute** *opt_conditional opt_indented_block* |}
| | __**decode** *ident* ↩ {| *decode_case* |}

encoding ::=
| | __**encoding** *ident*$_1$ ↩ {| __**instruction_set** *ident*$_2$ ↩ *instr_field*$_1$ .. *instr_field*$_m$ __**opcode** *opcode_value* ↩ __**guard** *expr* ↩ *instr_unpred*$_1$ .. *instr_unpred*$_n$ __**decode** *opt_indented_block* |}

opt_conditional ::=
| | __**conditional**
| |

opt_postdecode ::=
| | __**postdecode** *indented_block*
| |

instr_field ::=
| | __**field** *ident offset*$_1$+ : *offset*$_2$ ↩

offset ::=
| | *intLit*

opcode_value ::=
| | *bitsLit*
| | *maskLit*

instr_unpred ::=
| | __**unpredictable_unless** *intLit* == *bitsLit* ↩

decode_case ::=
| | **case** (*decode_slice*$_1$, .. , *decode_slice*$_m$) **of** ↩ {| *decode_alt*$_1$ ... *decode_alt*$_n$ |}

decode_slice ::=
| | *offset*$_1$+ : *offset*$_2$
| | *ident*
| | *ident*$_1$: .... : *ident*$_n$

decode_alt ::=
| | **when** (*decode_pattern*$_1$, .. , *decode_pattern*$_m$) => *decode_body*

decode_pattern ::=
| | *bitsLit*
| | *maskLit*
| | *ident*
| | !*decode_pattern*

decode_body ::=
| | __**UNPREDICTABLE** ↩
| | __**UNALLOCATED** ↩
| | __**NOP** ↩
| | __**encoding** *ident* ↩
| | ↩ {| *instr_field*$_1$ .. *instr_field*$_m$ *decode_case* |}

internal_definition ::=
| | __*operator1unop* = *ident*$_1$, ... , *ident*$_n$; ↩
| | __*operator2binop_or_concat* = *ident*$_1$, ... , *ident*$_n$; ↩
| | __**newevent** *qualident*(*formal*$_1$, .. , *formal*$_n$); ↩
| | __**event** *qualident possibly_empty_block*
| | __**newmap** *ty qualident*(*formal*$_1$, .. , *formal*$_n$)*opt_indented_block*
| | __**map** *qualident mapfield*$_1$, .. , *mapfield*$_n$ *optmapcond* **then** *possibly_empty_block*
| | __**config** *ty qualident* = *expr*; ↩

operator ::=
| | *unop*
| | *binop*
| | :

optmapcond ::=
| **when** *expr*
|

mapfield ::=
| *ident* = *pattern*

qualident ::=
| *ident*
| *qualifier*.*ident*

tidentdecl ::=
| *typeident*
| *ident*
| *qualifier*.*ident*

tident ::=
| *typeident*
| *qualifier*.*typeident*

ty ::=
| *tident*
| **bits** (*expr*)
| *tident*(*expr*$_1$, ... , *expr*$_n$)
| **typeof** (*expr*)
| **__register** *intLit*{*regfields*}
| **array** [*ixtype*] **of** *ty*
| (*ty*$_1$, .. , *ty*$_n$)

regfields ::=
|
| *regfield*
| *regfield*, *regfields*

regfield ::=
| *slice*$_1$, ... , *slice*$_n$ *ident*

stmt ::=
| *simple_stmt*
| *compound_stmt*

compound_stmt ::=
| *conditional_stmt*
| *repetitive_stmt*
| *catch_stmt*

simple_stmt_list ::=
| *simple_stmt*$_1$ ... *simple_stmt*$_n$

simple_if_stmt ::=
| **if** *expr* **then** *simple_stmt_list*$_1$ *simple_elsif*$_1$ .. *simple_elsif*$_n$ **else** *simple_stmt_list*$_2$ ↩
| **if** *expr* **then** *simple_stmt_list*$_1$ *simple_elsif*$_1$ .. *simple_elsif*$_n$ ↩

simple_elsif ::=
| **elsif** *expr* **then** *simple_stmt_list*

simple_stmts ::=
| *simple_stmt_list simple_if_stmt*
| *simple_stmt_list* ↩

stmts ::=
| *simple_stmts*
| *compound_stmt*

indented_block ::=
| ↩ {| *stmts*$_1$ ... *stmts*$_n$ |}

possibly_empty_block ::=
| *indented_block*
| *simple_stmts* statements on same line
| ↩

opt_indented_block	::=			
	|	*indented_block*		
	|	↩		
nonempty_block	::=			
	|	*indented_block*		
	|	*simple_stmts*		statements on same line
assignment_stmt	::=			
	|	*ty ident*$_1$, ... , *ident*$_n$;		
	|	*ty ident* = *expr*;		
	|	**constant** *ty ident* = *expr*;		
	|	*lexpr* = *expr*;		
lexpr	::=			
	|	–		
	|	*qualident*		
	|	*lexpr.ident*		
	|	*lexpr*.[*ident*$_1$, ... , *ident*$_n$]		
	|	*lexpr*[*slice*$_1$, .. , *slice*$_n$]		
	|	[*lexpr*$_1$, ... , *lexpr*$_n$]		
	|	(*lexpr*$_1$, ... , *lexpr*$_n$)		
lexpr_spice	::=			
	|	**__array** *lexpr*[*expr*]		spice for desugaring array assignment
	|	**__write** *ident*{{*expr*$'_1$, .. , *expr*$'_m$}}[*expr*$_1$, .. , *expr*$_n$]		spice for desugaring setter procedure call
	|	**__readwrite** *ident*$_1$ *ident*$_2${{*expr*$'_1$, .. , *expr*$'_m$}}[*expr*$_1$, .. , *expr*$_n$]		spice for desugaring read-modify-write function+procedure call
simple_stmt	::=			
	|	*assignment_stmt*		
	|	*qualident*(*expr*$_1$, .. , *expr*$_n$);	S	procedure call
	|	**return** *expr*;		function return
	|	**return** ;		procedure return
	|	**assert** *expr*;		assertion
	|	**UNPREDICTABLE** ();		underspecified behaviour
	|	**CONSTRAINED_UNPREDICTABLE** ;		
	|	**IMPLEMENTATION_DEFINED** (*ident*);		underspecified behaviour
	|	**UNDEFINED** ();		
	|	**__ExceptionTaken** ();		
	|	**UNPREDICTABLE** ;		DEPRECATED
	|	**IMPLEMENTATION_DEFINED** *stringLit*;		DEPRECATED
	|	**UNDEFINED** ;		DEPRECATED
	|	**SEE** (*expr*);		
	|	**SEE** *stringLit*;	S	DEPRECATED
	|	**SEE** *ident*;	S	DEPRECATED
	|	**throw** *ident*;		
stmt_spice	::=			
	|	*qualident*{{*expr*$'_1$, .. , *expr*$'_m$}}(*expr*$_1$, .. , *expr*$_n$);		spice for procedure call with explicit type parameters
conditional_stmt	::=			
	|	**if** *expr* **then** *opt_indented_block s_elsif*$_1$ .. *s_elsif*$_n$ *optional_else*		
	|	**if** *expr* **then** *simple_stmts s_elsif*$_1$ .. *s_elsif*$_n$ *optional_else*	S	
	|	**if** *expr* **then** *simple_stmt_list*$_1$ *simple_elsif*$_1$ .. *simple_elsif*$_n$ **else** *simple_stmt_list*$_2$ ↩		
	|	**case** *expr* **of** ↩ {| *alt*$_1$ ... *alt*$_n$ *opt_otherwise* |}		
s_elsif	::=			
	|	**elsif** *expr* **then** *opt_indented_block*		
	|	**elsif** *expr* **then** *simple_stmts*	S	
optional_else	::=			
	|	**else** *opt_indented_block*		
	|	**else** *simple_stmts*		
	|			
alt	::=			
	|	**when** *pattern*$_1$, ... , *pattern*$_n$ *opt_altcond possibly_empty_block*		
	|	**when** *pattern*$_1$, ... , *pattern*$_n$ *opt_altcond simple_if_stmt*	S	
opt_otherwise	::=			
	|	**otherwise** *possibly_empty_block*		

| |

opt_altcond ::=
| *&&expr =>*
| *=>*
| |

pattern ::=
| *intLit*
| *hexLit*
| *bitsLit*
| *maskLit*
| *qualident*
| *−*
| *(pattern*$_1$*, ... , pattern*$_n$*)*
| *{apattern*$_1$*, .. , apattern*$_n$*}*

apattern ::=
| *expr*$_1$*..expr*$_2$
| *expr*

repetitive_stmt ::=
| **for** *ident = expr*$_1$ *direction expr*$_2$ *indented_block*
| **while** *expr* **do** *indented_block*
| **repeat** *indented_block* **until** *expr*; ↩

direction ::=
| **to**
| **downto**

catch_stmt ::=
| **try** *indented_block* **catch** *ident* ↩ {| *catcher*$_1$ *.. catcher*$_n$ *opt_otherwise* |}

catcher ::=
| **when** *expr opt_indented_block*

expr ::=
| *conditional_expression*

conditional_expression ::=
| **if** *cexpr*$_1$ **then** *expr*$_1$ *e_elsif*$_1$ *.. e_elsif*$_n$ **else** *expr*$_2$
| *cexpr*

e_elsif ::=
| **elsif** *expr*$_1$ **then** *expr*$_2$

cexpr ::=
| *bexpr factor*$_1$ *.. factor*$_n$

zexpr ::=
| *expr*$_1$ *binop expr*$_2$

factor ::=
| *binop_or_concat bexpr*

binop_or_concat ::=
| *binop*
| :

binop ::=
| ==
| ! =
| >
| >=
| <
| <=
| +
| −
| *
| /
| ^

	|	**QUOT**	
	|	**REM**	
	|	**DIV**	
	|	**MOD**	
	|	<<	
	|	>>	
	|	&&	
	|	||	
	|	**IFF**	
	|	**IMPLIES**	
	|	**OR**	
	|	**EOR**	
	|	**AND**	
	|	++	
dummy_binop	::=		
	|		
bexpr	::=		
	|	*unop fexpr*	unary operator
	|	*fexpr*	
fexpr	::=		
	|	*fexpr.ident*	field selection
	|	*fexpr.*[*ident*$_1$, ..., *ident*$_n$]	multiple field selection
	|	*fexpr*[*slice*$_1$, .., *slice*$_n$]	bitslice
	|	*fexpr* **IN** *pattern*	pattern match
	|	*aexpr*	
aexpr	::=		
	|	*literal_expression*	
	|	*qualident*	
	|	*qualident*(*expr*$_1$, .., *expr*$_n$)	S
	|	(*expr*)	
	|	(*expr*$_1$, ..., *expr*$_n$)	tuple
	|	*ty* **UNKNOWN**	
	|	*ty* **IMPLEMENTATION_DEFINED** *opt_stringLit*	
expr_spice	::=		
	|	*qualident*{{*expr*$'_1$, .., *expr*$'_m$}}(*expr*$_1$, .., *expr*$_n$)	spice for desugaring function call with explicit type parameters
	|	__**array** *expr*$_1$[*expr*$_2$]	spice for desugaring array accesses
opt_stringLit	::=		
	|	*stringLit*	
	|		
unop	::=		
	|	−	
	|	!	
	|	**NOT**	
slice	::=		
	|	*sexpr*	
	|	*sexpr*$_1$: *sexpr*$_2$	
	|	*sexpr*$_1$+ : *sexpr*$_2$	
sexpr	::=		
	|	*scexpr*	
	|	**if** *cexpr*$_1$ **then** *expr*$_1$ *e_elsif*$_1$ .. *e_elsif*$_n$ **else** *scexpr*$_2$	
scexpr	::=		
	|	*bexpr sfactor*$_1$ .. *sfactor*$_n$	
sfactor	::=		
	|	*binop bexpr*	
literal_expression	::=		
	|	*intLit*	literal decimal integer
	|	*hexLit*	literal hexadecimal integer
	|	*realLit*	literal real
	|	*bitsLit*	literal bitvector

	| *maskLit*	literal bitmask
	| *stringLit*	literal string

expr_command ::=
- | ↩ *expr*

stmt_command ::=
- | ↩ *stmt*

impdef_command ::=
- | ↩ *stringLit* = *expr*

terminals ::=
- | {|
- | |}
- | ↩
- | ^

formula ::=
- | *judgement*

judgement ::=

user_syntax ::=
- | *id*
- | *typeid*
- | *qualifier*
- | *intLit*
- | *bitsLit*
- | *maskLit*
- | *realLit*
- | *hexLit*
- | *stringLit*
- | *eol*
- | *i*
- | *l*
- | *ident*
- | *typeident*
- | *leadingblank*
- | *declarations*
- | *declaration*
- | *type_declaration*
- | *field_ns*
- | *field*
- | *variable_declaration*
- | *ixtype*
- | *function_declaration*
- | *procedure_declaration*
- | *formal*
- | *getter_declaration*
- | *setter_declaration*
- | *sformal*
- | *instruction_definition*
- | *encoding*
- | *opt_conditional*
- | *opt_postdecode*
- | *instr_field*
- | *offset*
- | *opcode_value*
- | *instr_unpred*
- | *decode_case*
- | *decode_slice*
- | *decode_alt*
- | *decode_pattern*
- | *decode_body*
- | *internal_definition*
- | *operator*
- | *optmapcond*
- | *mapfield*
- | *qualident*
- | *tidentdecl*

| *tident*
| *ty*
| *regfields*
| *regfield*
| *stmt*
| *compound_stmt*
| *simple_stmt_list*
| *simple_if _stmt*
| *simple_elsif*
| *simple_stmts*
| *stmts*
| *indented_block*
| *possibly_empty_block*
| *opt_indented_block*
| *nonempty_block*
| *assignment_stmt*
| *lexpr*
| *lexpr_spice*
| *simple_stmt*
| *stmt_spice*
| *conditional_stmt*
| *s_elsif*
| *optional_else*
| *alt*
| *opt_otherwise*
| *opt_altcond*
| *pattern*
| *apattern*
| *repetitive_stmt*
| *direction*
| *catch_stmt*
| *catcher*
| *expr*
| *conditional_expression*
| *e_elsif*
| *cexpr*
| *zexpr*
| *factor*
| *binop_or_concat*
| *binop*
| *dummy_binop*
| *bexpr*
| *fexpr*
| *aexpr*
| *expr_spice*
| *opt_stringLit*
| *unop*
| *slice*
| *sexpr*
| *scexpr*
| *sfactor*
| *literal_expression*
| *expr_command*
| *stmt_command*
| *impdef_command*
| *terminals*

B Field Description for Explicit End-to-End Packets

Fieldname	Description
PCRC	Prelude Cyclic Redundancy Check that protects fields that determine type and length of a packet. Interpretation of the protected fields may vary but they are always in the same bit-location
OpCode	Operation Code: Field indicates the packet's request or response type
VC	Virtual Channel identifier
DCID	Destination Component ID: The component id of the intended receiver within the present or specified subnet
LEN	Packet length including protocol overhead plus payload. A value of 0x7F indicates a link-local packet, otherwise it is an end-to-end packet. The packet length in bytes is calculated as: Packet length = LEN * 4
SCID	Source Component ID: The component id of the component that initially injected this packet into the subnet
Tag	A Tag is an identifier to correlate request packets with response packets
OCL	The Operation Class Label identifies the explicit OpClass required to decode the packet
OS N	OpCode Specific Fields: Decoding of these fields varies by OpCode
PM	Performance Marker: Indicate if performance information for this packet should be collected or not
NH	Indicates if the Next Header field is present in the packet. The next header field is used for secure session related information, such as message authentication codes
GC	The GC bit indicates if the multi-subnet field is present in this explicit end-to-end packet. The multi-subnet fields contain the Source Subnet ID, identifying in which subnet a packet originated and the Destination Subnet ID, indicating the destination subnet of this packet
ECN	Explicit Congestion Notification: If at any point during transmission any congestion is detected, the ECN bit can be used to indicate congestion to the requester in the response packet
Deadline	Track a packets age in number of hops it has taken so far. The value in the Deadline field gets reduced by one for every hop the packet takes and is silently discarded if this value reaches zero. It works similar to the time-to-live (TTL) field in the IP header
Access Key	The Access Key allows the manager to restrict access to a certain component. Only packets with the correct access key are routed towards the destination component, others are discarded and an unsolicitated event is raised. Note that the Access Key field is not encrypted in a secure session and hence coul be sniffed by an attacker. As such it is not adding security guarantees against the Dolev-Yao attacker
Address	The address field identifies a location in the data space or control space of a component
ECRC	End-to-End Cyclic Redundancy Check: The ECRC field is present in all packets except for the link-local link idle packet. It is used to protect the complete packet against errors and has to be checked whenever a packet arrives
FPS	Forward Progress Screen: Indicates whether the packet has been transmitted. If the responder sent a not-ready packet, then the epoch (1 or 2) indicates if an error occured or not (1 or 0). If the packet has been transmitted successfully after receiving a not-ready packet, this is indicated by epoch 3

Table 3: Description of the single fields in a generic explicit end-to-end request. Adapted from [60]

C Indices

List of Figures

List of Tables

List of Codes

D Declaration of Originality

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Formal Verification and Modelling of the Gen-Z Specification

Authored by (in block letters):

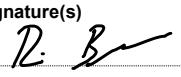
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Brunner	Roman

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, 07.10.2020	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.

References

[1] Whitfield Diffie and M. Hellman. "New directions in cryptography". In: *IEEE Transactions on Information Theory* 22.6 (Nov. 1976), pp. 644–654. ISSN: 0018-9448. DOI: 10.1109/TIT.1976.1055638. URL: http://ieeexplore.ieee.org/document/1055638/.

[2] Bruce J. MacLennan. "Simple metrics for programming languages". In: *Information Processing and Management* 20.1-2 (1984), pp. 209–221. ISSN: 03064573. DOI: 10.1016/0306-4573(84)90051-7.

[3] John L. Hennessy and David A. Patterson. *Computer Architecture: a Quantitative Approach*. Elsevier, 1990. ISBN: 978-0-12-383872-8.

[4] Sally A. McKee et al. "Experimental implementation of dynamic access ordering". In: *Proceedings of the Hawaii International Conference on System Sciences* 1 (1994), pp. 431–440. ISSN: 10603425. DOI: 10.1109/hicss.1994.323142.

[5] Sally A. McKee et al. "Increasing memory bandwidth for vector computations". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 782 LNCS (1994), pp. 87–104. ISSN: 16113349. DOI: 10.1007/3-540-57840-4_26.

[6] Wm. A. Wulf and Sally A. McKee. "Hitting the memory wall". In: *ACM SIGARCH Computer Architecture News* 23.1 (1995), pp. 20–24. ISSN: 0163-5964. DOI: 10.1145/216585.216588.

[7] Christoph Kern and Mark R. Greenstreet. "Formal verification in. hardware design: a survey". In: *ACM Transactions on Design Automation of Electronic Systems* 4.2 (1999), pp. 123–193. ISSN: 10844309. DOI: 10.1145/307988.307989.

[8] Bruno Blanchet. "An Efficient Cryptographic Protocol Verifier Based on Prolog Rules". In: *14th IEEE Computer Security Foundations Workshop (CSFW-14)*. Cape Breton, Nova Scotia, Canada: IEEE Compute rSociety, 2001, pp. 82–96. URL: https://prosecco.gforge.inria.fr/personal/bblanche/publications/BlanchetCSFW01.html.

[9] Arvind Seshadri et al. "SWATT: SoftWare-based ATTestation for embedded devices". In: *Proceedings - IEEE Symposium on Security and Privacy* 2004 (2004), pp. 272–282. DOI: 10.1109/SECPRI.2004.1301329.

[10] Arvind Seshadri et al. "Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems". In: *Operating Systems Review (ACM)* 39.5 (2005), pp. 1–16. ISSN: 01635980. DOI: 10.1145/1095809.1095812.

[11] Thomas Moscibroda and Onur Mutlu. "Memory performance attacks: Denial of memory service in multi-core systems". In: *16th USENIX Security Symposium* (2007), pp. 257–274.

[12] Arvind Seshadri et al. "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes". In: *Operating Systems Review (ACM)* (2007), pp. 335–350. ISSN: 01635980. DOI: 10.1145/1294261.1294294.

[13] J. Alex Halderman et al. "Lest we remember: Cold boot attacks on encryption keys". In: *Proceedings of the 17th USENIX Security Symposium* August (2008), pp. 45–58.

[14] Mainak Banga et al. "Guided test generation for isolation and detection of embedded trojans in ICs". In: *Proceedings of the ACM Great Lakes Symposium on VLSI, GLSVLSI* (2008), pp. 363–366. DOI: 10.1145/1366110.1366196.

[15] Cas J.F. Cremers. "Unbounded verification, falsification, and characterization of security protocols by pattern refinement". In: *Proceedings of the ACM Conference on Computer and Communications Security* (2008), pp. 119–128. ISSN: 15437221. DOI: 10.1145/1455770.1455787.

[16] Casimir Joseph Franciscus Cremers. "The Scyther Tool: Automatic Verification of Security Protocols". In: *Computer Aided Verification* 5423 (2008), pp. 414–418. ISSN: 0302-9743. DOI: 10.1007/978-3-540-70545-1_38.

[17] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008, p. 434. ISBN: 9780132350884.

[18] Bryan Parno. "Bootstrapping Trust in a "Trusted" Virtualized Platform". In: *HotSec* (2008), pp. 11–22. DOI: 10.1145/3338511.3357347.

[19] Andrew S. Tanenbaum. *Modern Operating Systems*. 3rd Editio. Pearson Education, 2008. ISBN: 978-3-8273-7342-7.

[20] Gerwin Klein et al. "seL4 : Formal Verification of an OS Kernel". In: *ACM SIGOPS 22nd Symposium on Operating systems principles* 97.1 (2009), pp. 207–220. ISSN: 00010782. DOI: 10.1145/1629575.1629596. URL: http://portal.acm.org/citation.cfm?id=1629596%7B%5C%%7D5Cnhttp://dl.acm.org/citation.cfm?id=1629596.

[21] Kevin Lim et al. "Disaggregated memory for expansion and sharing in blade servers". In: *Proceedings - International Symposium on Computer Architecture* (2009), pp. 267–278. ISSN: 10636897. DOI: 10.1145/1555754.1555789.

[22] Bryan Parno, Jonathan M. McCune, and Adrian Perrig. "Bootstrapping trust in commodity computers". In: *Proceedings - IEEE Symposium on Security and Privacy* (2010), pp. 414–429. ISSN: 10816011. DOI: 10.1109/SP.2010.32.

[23] Mohammad Tehranipoor and Farinaz Koushanfar. "A survey of hardware trojan taxonomy and detection". In: *IEEE Design and Test of Computers* 27.1 (2010), pp. 10–25. ISSN: 07407475. DOI: 10.1109/MDT.2010.7.

[24] Stéphanie Delaune et al. "Formal analysis of protocols based on TPM state registers". In: *Proceedings - IEEE Computer Security Foundations Symposium* (2011), pp. 66–80. ISSN: 19401434. DOI: 10.1109/CSF.2011.12.

[25] Russell A. Fink et al. "Catching the cuckoo: Verifying TPM proximity using a quote timing side-channel". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6740 LNCS (2011), pp. 294–301. ISSN: 16113349. DOI: 10.1007/978-3-642-21599-5_22.

[26] Kevin Lim et al. "System-level implications of disaggregated memory". In: (2012), pp. 1–12. DOI: 10.1109/hpca.2012.6168955.

[27] Benedikt Schmidt et al. "Automated analysis of diffie-hellman protocols and advanced security properties". In: *Proceedings of the Computer Security Foundations Workshop* (2012), pp. 78–94. ISSN: 10636900. DOI: 10.1109/CSF.2012.25.

[28] Ittai Anati et al. "Innovative technology for CPU based attestation and sealing". In: (2013), pp. 1–7. URL: https://software.intel.com/en-us/articles/innovative-technology-for-cpu-based-attestation-and-sealing.

[29] Sangjin Han et al. "Network support for resource disaggregation in next-generation datacenters". In: *Proceedings of the 12th ACM Workshop on Hot Topics in Networks, HotNets 2013* (2013). DOI: 10.1145/2535771.2535778.

[30] Frank McKeen et al. "Innovative instructions and software model for isolated execution". In: (2013), pp. 1–1. DOI: 10.1145/2487726.2488368.

[31] Simon Meier et al. "The TAMARIN prover for the symbolic analysis of security protocols". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 8044 LNCS (2013), pp. 696–701. ISSN: 03029743. DOI: 10.1007/978-3-642-39799-8_48.

[32] "TCG TPM Specification 2.0". 2013. URL: https://trustedcomputinggroup.org/resource/tpm-library-specification/.

[33] Benedikt Schmidt et al. "Automated verification of group key agreement protocols". In: *Proceedings - IEEE Symposium on Security and Privacy* (2014), pp. 179–194. ISSN: 10816011. DOI: 10.1109/SP.2014.19.

[34] Weijin Wang, Yu Qin, and Dengguo Feng. "Automated proof for authorization protocols of TPM 2.0 in computational model". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 8434 LNCS (2014), pp. 144–158. ISSN: 16113349. DOI: 10.1007/978-3-319-06320-1_12.

[35] Will Arthur, David Challener, and With Kenneth Goldman. "A Practical Guide to TPM 2.0". In: *Apress* (2015).

[36] Jianxiong Shao et al. "Formal analysis of enhanced authorization in the TPM 2.0". In: *ASIACCS 2015 - Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security* (2015), pp. 273–284. DOI: 10.1145/2714576.2714610.

[37] Jie Zhang et al. "VeriTrust: Verification for hardware trust". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.7 (2015), pp. 1148–1161. ISSN: 02780070. DOI: 10.1109/TCAD.2015.2422836.

[38] Bruno Blanchet. "Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif". In: *Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif* 1.1 (2016), pp. 1–135. DOI: 10.1561/9781680832075.

[39] He Li, Qiang Liu, and Jiliang Zhang. "A survey of hardware Trojan threat and defense". In: *Integration, the VLSI Journal* 55 (2016), pp. 426–437. ISSN: 01679260. DOI: 10.1016/j.vlsi.2016.01.004.

[40] Jean Pierre Lozi et al. "The Linux scheduler: A decade of wasted cores". In: *Proceedings of the 11th European Conference on Computer Systems, EuroSys 2016* (2016). DOI: 10.1145/2901318.2901326.

[41] Alastair Reid et al. "End-to-end verification of ARM® processors with ISA-formal". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 9780 (2016), pp. 42–58. ISSN: 16113349. DOI: 10.1007/978-3-319-41540-6_3.

[42] Reto Achermann et al. "Formalizing memory accesses and interrupts". In: *Electronic Proceedings in Theoretical Computer Science, EPTCS* 244.Mars (2017), pp. 66–116. ISSN: 20752180. DOI: 10.4204/EPTCS.244.4.

[43] Brad Benton. *CCIX, GEN-Z, OpenCAPI: OVERVIEW & COMPARISON*. 2017. URL: https://www.openfabrics.org/images/eventpresos/2017presentations/213%7B%5C_%7DCCIXGen-Z%7B%5C_%7DBBenton.pdf (visited on 07/23/2020).

[44] Cas Cremers et al. "A comprehensive symbolic analysis of TLS 1.3". In: *Proceedings of the ACM Conference on Computer and Communications Security* (2017), pp. 1773–1788. ISSN: 15437221. DOI: 10.1145/3133956.3134063.

[45] Kathryn E Gray et al. "The Sail instruction-set semantics specification language". In: (2017), pp. 1–25. URL: http://www.cl.cam.ac.uk/%7B~%7Dpes20/sail/manual.pdf.

[46] Lukas Humbel et al. "Towards correct-by-construction interrupt routing on real hardware". In: *Proceedings of the 9th Workshop on Programming Languages and Operating Systems, PLOS 2017* (2017), pp. 8–14. DOI: 10.1145/3144555.3144557.

[47] Alastair Reid. "Trustworthy specifications of ARM® v8-A and v8-M system level architecture". In: *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design, FMCAD 2016* (2017), pp. 161–168. DOI: 10.1109/FMCAD.2016.7886675.

[48] Hassan Salmani. "COTD: Reference-Free Hardware Trojan Detection and Recovery Based on Controllability and Observability in Gate-Level Netlist". In: *IEEE Transactions on Information Forensics and Security* 12.2 (2017), pp. 338–350. ISSN: 15566013. DOI: 10.1109/TIFS.2016.2613842.

[49] Daniel Schwyn. "Hardware Configuration With Dynamically-Queried Formal Models". Master. ETHZ, 2017. DOI: 10.3929/ethz-b-000203075. URL: http://hdl.handle.net/20.500.11850/203075.

[50] Reto Achermann et al. "Physical Addressing on Real Hardware in Isabelle/HOL". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10895 LNCS (2018), pp. 1–19. ISSN: 16113349. DOI: 10.1007/978-3-319-94821-8_1.

[51] Alasdair Armstrong et al. "Detailed Models of Instruction Set Architectures: From Pseudocode to Formal Semantics". In: (2018). URL: www.cl.cam.ac.uk/%7B~%7Dpes20/sail/..

[52] Jordan Robertson and Michael Riley. "The Big Hack: How China Used a Tiny Chip to Infiltrate U.S. Companies". In: *Bloomberg Businessweek* (Oct. 2018). URL: https://www.bloomberg.com/news/features/2018-10-04/the-big-hack-how-china-used-a-tiny-chip-to-infiltrate-america-s-top-companies.

[53] Syed Kamran Haider et al. "Advancing the state-of-the-art in hardware trojans detection". In: *IEEE Transactions on Dependable and Secure Computing* 16.1 (2019), pp. 18–32. ISSN: 19410018. DOI: 10.1109/TDSC.2017.2654352.

[54] Alexander Hedges. "Simulator Design Options for Hybrid CPU/FPGA Systems". Master's Thesis. ETH Zurich, 2019.

[55] Alastair Reid, Thomas Bauereiss, and Alasdair Armstrong. *GitHub ASL Interpreter*. 2019. URL: https://github.com/alastairreid/asl-interpreter (visited on 08/11/2020).

[56] "Sockeye in Barrelfish". 2019. URL: http://www.barrelfish.org.

[57] Shin Yeh Tsai and Yiying Zhang. "A double-edged sword: Security threats and opportunities in one-sided network communication". In: *11th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2019, co-located with USENIX ATC 2019* (2019). arXiv: 1903.09355.

[58] Reid Alastair and Thomas Bauereiss. *Tweaks for ASL-to-Sail translation*. 2020. URL: https://github.com/alastairreid/asl-interpreter/pull/2%7B%5C#%7Devent-3337372899 (visited on 09/23/2020).

[59] *Arm A64 Instruction Set Architecture*. 2020. URL: https://developer.arm.com/architectures/cpu-architecture/a-profile/exploration-tools%20https://developer.arm.com/-/media/developer/products/architecture/armv8-a-architecture/2020-06/A64%7B%5C_%7DISA%7B%5C_%7Dxml%7B%5C_%7Dv86A-2020-06.tar.gz?revision=f5431629-cc60-4e55-9dec-ed89f691d3 (visited on 09/17/2020).

[60] "Core Specification". 2020. URL: https://genzconsortium.org/specification/gen-z-core-specification-1-1/.

[61] Anakhi Hazarika, Soumyajit Poddar, and Hafizur Rahaman. "Survey on memory management techniques in heterogeneous computing systems". In: *IET Computers and Digital Techniques* 14.2 (2020), pp. 47–60. ISSN: 17518601. DOI: 10.1049/iet-cdt.2019.0092.

[62] Seokbin Hong, Won-Ok Kwon, and Myeong-Hoon Oh. "Hardware Implementation and Analysis of Gen-Z Protocol for Memory-Centric Architecture". In: *IEEE Access* 8 (2020), pp. 127244–127253. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.3008227. URL: https://ieeexplore.ieee.org/document/9137193/.

[63] Jakob Meier. "Tools for Cache Coherence Protocol Interoperability". Master's Thesis. ETH Zurich, 2020.

[64] "Security Protocol and Data Model (SPDM) Specification". 2020. URL: https://www.dmtf.org/dsp/DSP0274.

[65] *Arm TrustZone*. URL: https://developer.arm.com/ip-products/security-ip/trustzone (visited on 09/26/2020).

[66] *Intel Software Guard Extension*. URL: https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html (visited on 09/26/2020).