Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor's Thesis Nr. 248b

Systems Group, Department of Computer Science, ETH Zurich

Formally modelling hardware standards

by

Giuseppe Arcuti

Supervised by

Reto Achermann, Lukas Humbel, Prof. Dr. Timothy Roscoe

Wednesday 14th August, 2019

Informatik
Computer Science

Abstract

When systems developers want to port an operating system to a new platform, they need a description of that platform. Those descriptions are usually written in prose. The problem is that the English language is not precise enough, thus leaving room for interpretation.

In this thesis I analyze why current specifications are too vague and easily misinterpreted, with the examples being ARM's TrustZone, ARM's Server Base System Architecture and Cavium's ThunderX CN88XX. I report the problems I found in these specifications.

To remedy this situation I formalize TrustZone's memory subsystem in Sockeye, a DSL to describe address decoding nets, write a Prolog query to check whether an address decoding net is TrustZone-compliant and model the SBSA's memory subsystem in Isabelle as a predicate over address decoding nets.

As an example of a concrete system I express ThunderX' memory subsystem as an address decoding net in Isabelle.

I make suggestions on how to improve the address decoding net model and Sockeye based on the experience in this thesis.

Acknowledgements

I would like to thank my supervisors Prof. Dr. Timothy Roscoe, Lukas Humbel and Reto Achermann for giving me their valuable time, guidance and feedback throughout this thesis process.

I would also like to thank my family for always supporting me during my studies at ETH.

Contents

Chapter 1

Introduction

The job of operating system developers has become more and more difficult over the years with the increasing heterogeneity of hardware systems. Not only does the OS developer need to make special cases for the quirks of different architectures, they also have to infer from the reference manuals how exactly the hardware behaves.

One particular hard part nowadays is the memory subsystem, which is not as straightforward as it once was. It turns out that the reference specifications like the Server Base System Architecture (SBSA) [3], the TrustZone Reference [4] and the ThunderX CN88XX Reference [7] are too vague and ambiguous. Those references are written in plain English as is usual for hardware references but natural language is not precise enough and leaves room for interpretation.

The SBSA is a document describing some constraint on hardware systems that ARM hopes hardware manufacturer will adopt. It was created by ARM to try to stop the fragmentation of the ARM architecture. Otherwise, operating system cannot target ARM easily because every manufacturer's hardware behaves differently.

As an example of the SBSA not being precise enough: it states that there has to be a UART that can be configured to exist in the Secure memory address space. What does this mean that it can be configured to exist in the Secure memory address space? Can it be configured to be in Non-secure address space? If it can, doesn't that conflict with the statement that it's not allowed to be aliased in the Non-secure address space?

The x86 architecture [10] has been more or less able to standardize on the architecture front such that a single-built OS image can run unmodified on all x86 compatible architectures. One part of the solution is also the Advanced Configuration and Power Interface (ACPI) which is a standard on x86 that can be used by OSs to discover and configure hardware components [20]

But ARM can't learn much from it, given that this was more of an historical artifact which happened because at the time of x86 rise to power there were no Systems-on-a-chips (SoCs) yet and so hardware manufacturers tried to standardize the interfaces between devices to be compatible with each other. Obviously this cannot be replicated by ARM.

ARM has already tried remedying the situation by making Device Tree mandatory on new SoCs. Device Tree is a format describing some hardware components to the OS such that one OS image can support multiple hardware configurations. [8] But that isn't enough.

Using formal methods in this space is not unheard of, there's research in proving compliance of processors to the ARM ISA [14] and similarly for the new RISC-V ISA.[21] [18]

In chapter 3 I'm going to explain what TrustZone is and how it complicates the memory system. Then I'm going to show how to formalize TrustZone's memory system with Sockeye [17] and write a Prolog query to verify that a memory system is TrustZone-compliant.

In chapter 4 I'm going to explain the Server Base System Architecture and show its ambiguities. I'm then going to provide an Isabelle formalization of the SBSA's memory subsystem.

In chapter 5 I will show Cavium's ThunderX CN88XX, a modern system which claims to be SBSA-compliant. Its memory system is very complicated. I described it in the address decoding net model in Isabelle, which makes it possible for an OS developer to easily understand it, for the OS to automatically configure itself by querying facts about the memory system of the CN88XX and to prove something about the memory subsystem, like SBSA-compliance.

In chapter 6 I'm going to provide an evaluation of the address decoding net model and Sockeye and by providing some recommendations for improvements based on what I experienced while writing formalizations of specifications.

Chapter 2

Background

2.1 Memory system

A very important part of every computer system is the memory subsystem which is the way in which the CPU gets data to and from memory.

In a system with physical addressing (which is still used in embedded systems) each process directly shares address space with all the other processes executing on that core.

To have a more efficient use of main memory, simplifying memory management for programmers and isolating address spaces, virtual memory was introduced. Each process gets its own private memory space.

When that process emits an address that virtual address gets translated by the Memory Management Unit (MMU) with the mapping indicated by a process specific page table to a physical address or a page fault. The physical address corresponds to a memory cell or device register. That means each object can now have multiple virtual addresses but one physical address which uniquely describes that object. At least that's the mental model one often has. [15]

In figure 2.1: an example of a supposedly simple SoC, the Texas Instruments OMAP4460 Multimedia SoC, it has three different physical addresses for the GPTIMER5 timer depending on the core accessing it: an A9 uses 0x40138000, a DSP uses 0x01D38000 and a DMA-capable device on the L3 interconnect uses 0x49038000. This doesn't fit with the mental model explained before, where every object has a unique physical address after MMU-translation. Even without an MMU like e.g. in an embedded system the physical address space is not unique. Because what actually happens is that there are many lookup tables and other components in the system apart from the MMU that do translations or accept some addresses.

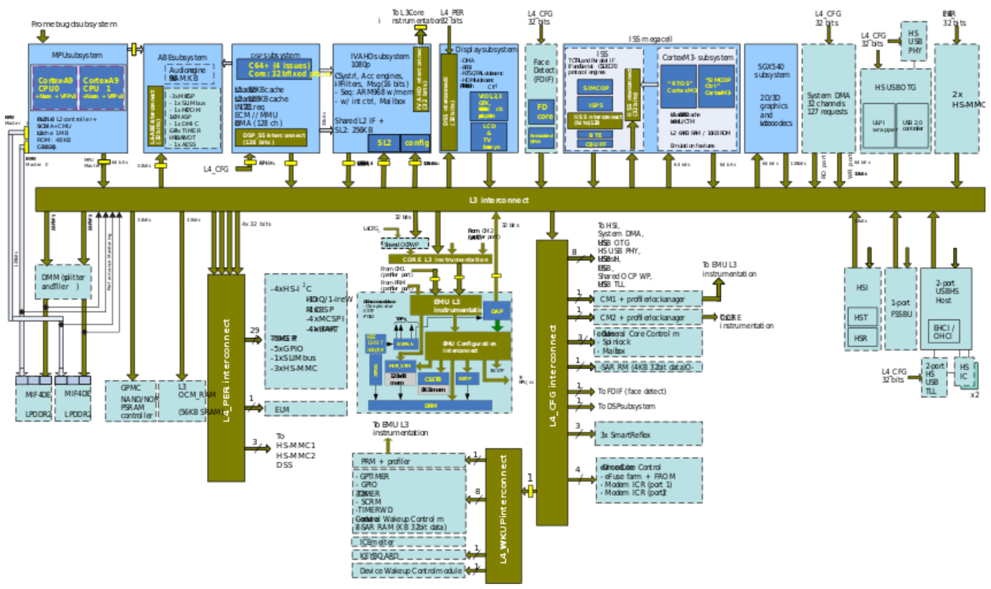

Figure 2.1: The OMAP4460 — A 'Simple' SoC (OMAP4460 TRM [19])

As can be seen a modern system doesn't only have multiple virtual address spaces but also has many physical address spaces i.e. even the physical address doesn't uniquely describe an object in a computer system. Therefor a location can have a different physical address depending on which core or device is trying to access it. A location can even be not reachable for some cores or devices.

So our simple mental model has become outdated.

"Modern systems are a complex network of cores, memory, devices, and translation units. Multiple caches in this network interpose on memory addresses. Virtualization support creates additional layers of address and interrupt translation." [1]

2.1.1 Address Decoding Net Model

To model this complexity the Address Decoding Net model was invented. [1]

It models memory addressing by a decoding net, which is a directed graph composed of nodes. Addresses are modeled as natural numbers.

A node is a unit which accepts a set of address (e.g. RAM or device registers), translates a set of addresses to other nodes (e.g. MMU or lookup tables), or both (e.g. caches), it can thus be described entirely by the set of addresses it accepts and the translation function which given an input address outputs a set of names it decodes to.

A name is defined as a tuple of an address and a node identifier (nodeid, which is also just a natural number) of the node at which the address is decoded.

An address' decoding process starts at a particular node. A resolution function can then be defined, which takes an input name, if the input node accepts this address it's added to the set of resolved names, and all the names it translates to are then recursively taken and the procedure then recursively applied to these. This returns all the names at which the input name could end up being accepted.

This formalization is useful for correctly configuring interrupts automatically [12] or automatic full PCI configuration, resource allocation, and interrupt assignment [16] and many other things.

2.1.2 Sockeye

Sockeye is a domain specific language to describe the address decoding net of a system with the goal of applying the model to hardware configuration.[17]

By modelling the address space translations in a declarative language, the mechanism by which the hardware gets configured can be separated from the calculation of how to configure the hardware.

This Sockeye description can, by having multiple backends, be compiled to different languages. One of these backends translates Sockeye into Prolog for use in Barrelfish's System Knowledge Base.

The Barrelfish's System Knowledge Base (SKB) is a central system service which stores knowledge it has found out about the system and makes it available to clients. The Prolog generated from Sockeye extends this knowledge to include information about the address decoding net of the system the OS is running on. Barrelfish then derives configuration parameters from this knowledge which are then used to configure the system with fast, low-level C code. This separation of policy and mechanism code helps to keep complexity out of low-level code and makes it easier to reason about complex policies. It also allows for complex algorithm's to be run off the system's fast path.

Sockeye isn't only an implementation of the address decoding net model but also made several improvements to it, which improve the ergonomics of using it in real life.

Instead of using node ids Sockeye uses strings to identify nodes because that is also how humans thinks about a system and makes it easier to recognize what is an MMU, DRAM etc.

```
memory (0 bits 4) LOOKUP
memory (0 bits 4) RAM
LOOKUP maps [
    (0x0 to 0xf) to RAM at (0x0 to 0xf)
]
```

The keyword ”memory” indicates the node’s domain. Other domains apart from the memory system are intr which indicates that the nodes are part of the interrupt system, power and clock domain. Domains are similar to types in that they prevent from accidentally mapping between two different systems and clarify the meaning of a description. The ”(0 bits 4)” indicates the address type of the node ”LOOKUP”. Meaning the address at ”LOOKUP” have to be element of the address set {0,1,....15}. The addresses 0x0 to 0xf then get mapped to the node called ”RAM” at the same base address.

Another improvement are modules, a module encapsulates an address decoding net to enable code reuse. By instantiating the module, the contained address decoding net can be integrated into a larger one. The interface to a module are the input ports and output ports. So if one wants to have that ”IN” node (or several nodes together) at multiple places in the system, or just wants to separate those nodes and give them a meaningful name, they can be put in a module, and designate the input and output ports of the module. It can then be instantiated as often as one wants.

```
module MMU {
    input memory (0 bits 4) IN
    output memory (0 bits 4) RAM
    IN maps [
        (0x0 to 0xf) to RAM at (0x0 to 0xf)
    ]
}
module SYSTEM{
    memory (0 bits 4) DRAM
    DRAM accepts [(0x0 to 0xf)]
    instance mmu of MMU
    mmu binds [
        RAM to DRAM
    ]
}
```

Modularity can be improved even further with the use of module parameters which allow leaving some values unbound in the module and then set those at instantiation-time, more or less like function parameters.

Other syntactic improvements are ”forall” which shortens the amount of code one has to write by allowing to write bounded loops. And the wildcard (*) which shortens the most common ”forall” loop, the one over all elements.

Sockeye also allow multi-dimensional addresses. The dimensions of the address are separated by semicolon. This isn't more powerful than having single-dimensional addresses (there exists a bijection with single-dimensional addresses by using diagonalization) but allows expressing semantics of some operations better.

2.2 ThunderX

The ThunderX CN88XX is a System-on-a-chip (SoC) developed by Cavium which contains up to 48 custom designed ARMv8.1 cores with a core frequency of up to 2.0 GHz. It claims to be SBSA-compliant. [7]

It was specifically designed for the needs of data centers with its high performance per watt, high performance per mm2 and various hardware accelerators designed for large data applications.

The cache-coherent interconnect allows connecting two CN88XX together and act like a single one.

Chapter 3

TrustZone

In section 3.1 I'm going to explain what TrustZone is and what it's trying to achieve. Then in section 3.2 I'm going to explain how it is implemented on the hardware-level. This information is based on "TrustZone Explained: Architectural Features and Use Cases" [13] and "ARM Security Technology Building a Secure System using TrustZone Technology". [4]

In 3.3 I'll show how it relates to address spaces and in 3.4 which parts of the specifications are ambiguous. As I will explain in section 3.5 there are different ways to model TrustZone in the address decoding net model, I'm going to show the Pros and Cons of each of them and which one I chose for the thesis. I will then in section 3.6 provide a Sockeye description of a simple TrustZone-aware system. I'll show which security properties TrustZone tries to uphold in 3.7 and how to check whether a certain address decoding net maintains them with the help of Prolog.

3.1 What's TrustZone?

TrustZone is an approach by AMD to stopping information leakage from a trusted process to one which isn't trusted running on the same machine.

The main idea is to have two worlds in which a resource can be, the secure world and the non-secure world, these worlds are orthogonal to privilege levels like EL1, EL2, EL3. The goal of TrustZone is guaranteeing that the non-secure world cannot access anything which is in the secure world. Everything can be accessed by the secure world.

TrustZone does not provide secure key storage nor a root of trust but just a system-wide isolation of two execution environments.

The secure world runs on the same CPU as the normal operating system. This means that the secure world is as fast as the non-secure world. Compared to a dedicated secure core ARM's solution runs much faster and has almost no additional costs.

3.2 Hardware implementation of TrustZone

TrustZone security properties are achieved by splitting all the hardware resources so that they exist in one of two worlds, the Secure world for the security subsystem, and the Normal world for everything else.

The CP15 coprocessor's secure configuration register determines the current processor state.

Through the monitor kernel mode, which acts like an ordinary context switch, the two worlds can communicate with each other.

The Advanced eXtensible Interface (AXI) main system bus, has an additional bit called the non-secure (NS) bit which indicates the world making the transaction. A Non-secure master cannot access a resource which is marked as secure. This is achieved by having the NS bit set to high in hardware for Non-secure masters. A transaction from a Non-secure master won't match any secure resource due to the NS bit being different.

The Advanced Peripheral Bus (APB) is the legacy bus which does not carry an equivalent of the NS bit due to backward-compatibility concerns. To compensate there is an AXI-to-APB Bridge which rejects transactions of inappropriate security setting and does not forward these requests to the APB. A TrustZone Protection Controller (TZPC) can be used to dynamically set the security of a peripheral connected to the APB.

The MMU acts as two virtual MMUs, one for the secure virtual processor and one for the non-secure virtual processor. This means that a virtual address emitted by a secure master can be translated to a different physical address than the same address emitted by a non-secure master.

The TrustZone Address Space Controller (TZASC) is a piece of hardware controlled by the secure world which allows dynamic classification of AXI slave memory-mapped devices as secure or non-secure. It allows an arbitrary number of partitions to be created.

The secure world can access secure or non-secure memory. This is achieved by having an additional field in the translation table descriptor, which tells the secure processor whether to set the NS bit or not when accessing that memory location. The non-secure world ignores that field, due to its NS bit being set to 1 in hardware.

Inside the cache the NS bit is treated like an additional address bit. This leads to a problem, if non-secure slaves are aliased in the secure world to allow accessing it through the secure world (instead of making the secure

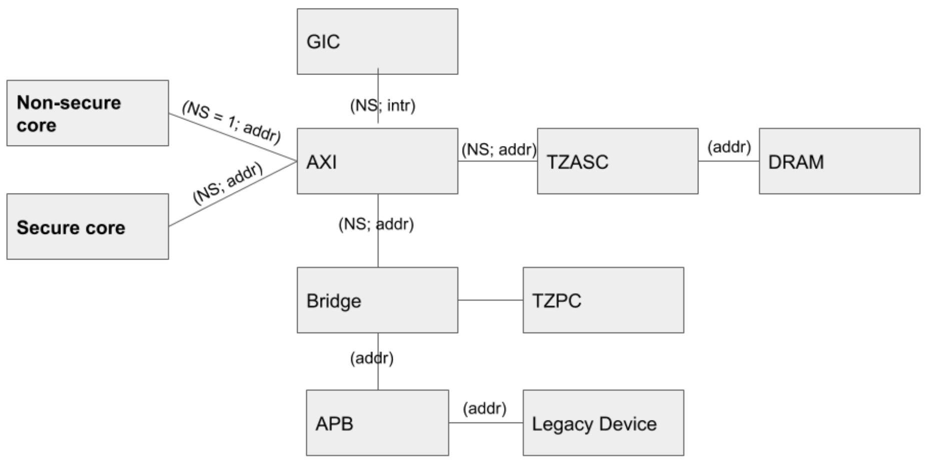

Figure 3.1: A minimal TrustZone-compliant system

world do a non-secure access) there can be two locations in the cache where the data is stored (a synonym). The system must pay attention to dirty cache lines to avoid causing problems.

The Translation Lookaside Buffers (TLBs) helps speed up switching from one world to another by allowing entries to be tagged with the world which walked the translation table leading to that entry, such that entries of both worlds are allowed to be in the buffer. Meaning there is no need to flush the TLB when switching worlds.

The Generic Interrupt Controller (GIC) allows classification of interrupts as secure or non-secure and then prevents non-secure interrupts from unauthorized access. After an interrupt has been marked as Secure, it's configuration can't be changed anymore by a Non-secure access. The GIC will ignore a signal if it's already in the right world (based on the secure configuration register), otherwise it switches into monitor mode. To prevent a denial-of-service attack through interrupts by the non-secure world, secure interrupts can always be configured to have a higher priority than non-secure ones.

In figure 3.1: an example of a minimal TrustZone-compliant system.

3.3 TrustZone and address spaces

TrustZone complicates the address space of the system because there can't be a true single address space anymore.

A core which is currently in the secure world has another view of memory than one which is in the non-secure world. The secure world PE doesn't get a fault when it tries to access a resource which is marked as secure. In some cases a register returns a different value based on the world making the request, these registers are called banked by security.

Another problem is the same core can have a different view of the system across time by switching between secure world and non-secure world. This can be easily solved by just modelling every core as two virtual cores, one secure and one non-secure.

3.4 Specification Ambiguities

Some examples of things which a hardware manufacturer wanting to implement a TrustZone-compliant system would have trouble knowing from the plain text specification of TrustZone

- Is a manufacturer allowed to implement the security properties differently than the way explained by ARM or is TrustZone the actual name of the implementation?

  I interpreted it to mean that there can be different implementations because this allows manufacturers to improve on the design.

- Is the translation path of a physical address allowed to be different based on the NS bit of the PE making the access, apart from resulting in a fault i.e. accessing a different register?

  I assumed that everything is allowed as long as the security properties are maintained because it was not excluded explicitly so it would be over-constraining the model.

Such ambiguities could not exist if TrustZone was formally specified.

3.5 Modelling TrustZone

As can be seen, TrustZone is quite complicated. How can a hardware manufacturer be sure that they correctly understood the specification. Once correctly understood, how can they be sure that they implemented it correctly. And how can an OS developer make sure that they don't accidentally break the TrustZone-compliance by e.g. aliasing a secure slave such that it is accessible by a non-secure master?

Sockeye is an ideal language to model the memory subsystem of TrustZone such that there are no ambiguities on what is meant by the specification.

Sockeye by default has no way of saying whether a name or node is secure or non-secure. Moreover, Sockeye doesn't have a way of saying which modules are masters and which are slaves. There are different ways to circumvent these problems such that I was able to represent TrustZone in Sockeye anyway.

3.5.1 Single-dimensional

I could add a bit to the system and the modules like e.g. the AXI would just map differently based on whether the address it receives is in the upper half or the lower half of the address space.

Pros:

- Sockeye parser doesn't have to be changed
- It resembles how it is implemented in hardware

Cons:

- When writing the Sockeye one has to always convert between the binary representation having a 0 or 1 as NS bit and the natural numbers that Sockeye needs (this could be solved by having better syntactic sugar for treating addresses as bit fields)
- Easy to make mistakes (this too could be solved by having better syntactic sugar for treating addresses as bit fields)
- It's not immediately obvious to someone reading the Sockeye specification why the mappings are the way they are
- I need a workaround to enforce that a non-secure node can only emit addresses which are in the upper half of the address space (i.e. have the NS bit set)

3.5.2 Multidimensional

I could make use of Sockeye's support for multidimensional addresses. Then I can put the NS bit in its own dimension and let nodes map differently based on that dimension.

Pros:

- It's easy to map based on the NS bit because it's in a separate dimension
- It matches the reality of the NS bit being special better than the single-dimensional approach

Cons:

- The backends for multi-dimensional address decoding net are not complete yet

- It's not immediately obvious to someone reading the Sockeye description why something is mapped the way it is
- I need a workaround to enforce that a non-secure node can only emit addresses where the NS dimension is set to 1.

3.5.3 Security domain

I could add support for TrustZone in Sockeye. That could be done for example by adding a security domain to names and modules and then have the ability to map differently based on that.

Pros:

- Very easy to input mappings
- Intuitive for someone reading to understand what is happening
- Support to automatically check that the Sockeye specification is TrustZone-compliant could be added

Cons:

- It's not implemented yet
- It would add custom stuff about TrustZone to Sockeye which would dilute the simplicity of the Address Decoding Net model

3.5.4 My choice

For the Sockeye implementation of TrustZone I chose to represent it with multi-dimensional addresses because the Prolog backend for multi-dimensional Sockeye works well enough. Also, it would have been a pain to write it in single-dimensional Sockeye because there is no support for bit fields in Sockeye and thus one has to convert back to natural numbers when setting the NS bit or clearing it. Moreover, implementing a security domain for Sockeye would have been outside the scope of this thesis.

The Isabelle formalization of SBSA doesn't really represent the NS bit because masters and names are just classified as secure or non-secure. An address decoding net which is SBSA compliant has the property that no non-secure master can access a secure name, but there's no restriction on how that address decoding net will implement this property.

For the ThunderX formalization in Isabelle I chose to represent it with single-dimensional addresses because Isabelle doesn't have the syntax problem that Sockeye has when working with single-dimensional addresses i.e. bit fields can be represented.

3.6 Sockeye Model

In minimal_trustzone.soc A.2 I tried implementing just the memory subsystem of a system that implements TrustZone. In trustzone.soc A.1 I extended it to include more hardware components.

I implemented the NS bit as a separate dimension, this allows mapping based on the NS bit.

To be TrustZone-compliant non-secure masters have to always have the NS bit set to 1. There are multiple ways to restrict the non-secure master not to output any addresses where the first dimension is set to 0.

With the security domain, this would be part of the language. In standard Sockeye this can be done in two ways.

One way is with what I call a mapping module, which is just a module (in this case called MAPPER) put after the output that one want to restrict, that takes the non-secure master's output and maps the first dimension always to 1.

Another way is to restrict it with the address type of the output, which is defined as an address set, that the address has to be an element of. The only address type which I have used so far in this thesis have been of this kind: (0 bits 8), (0 bits 48) etc. But it can be any address set e.g. the address type (1; 0 bits 48) would mean that the address' first dimension has to be element of the set {1} and the address' second dimension has to be element of $\{0,1,...,2^{48}\}$. That is exactly what I want to guarantee of the non-secure master.

This can also be applied to single-dimensional Sockeye by making the address type (2^{47} to 2^{48}) i.e. the upper half of the address space which implies the NS bit is set to 1. (Note: Sockeye doesn't support exponentiation, so these numbers would have to be written out)

The problem with the address type is that it's not part of the pure address decoding net model, also it doesn't get translated into Prolog or Isabelle. This is why I choose to describe it with an additional mapping module.

3.7 Maintaining TrustZone's security properties

From the description of TrustZone I inferred that the general security property a TrustZone-compliant system has to maintain is that no non-secure master should be able to access a secure resource.

Concretely, what needs to be guaranteed is that in a specific hardware decoding net, no master with the NS bit set to 1 can access a slave with the NS bit set to 0.

More precisely: The goal is to prove that for a given decoding net there is no path through translating nodes (nodes with incoming and outgoing edges) from any source node (node with only outgoing edges) of the form (1; *) to an accepting/target node (node with only incoming edges) of the form (0; *).

This is a graph reachability problem which can be solved in time linear to the number of nodes, thus in this case the complexity is linear in the number of addresses i.e. $O(2^{bitwidth})$.

Having the Sockeye description of a system which implements TrustZone, it would be useful to have a query which checks whether the described system maintains the security properties of TrustZone.

That would be especially useful if executed dynamically as a check on the OS. It could for example execute the query inside the SKB whenever the TZASC configuration change and be sure no mistake was made. If the query could not only check whether it's valid but also generate new valid assignments it could even be used to automatically configure the TZASC based on that.

```
add_SYSTEM([]).
secure([block{base:0, limit:0}]).
secure([block{base:1, limit:1}]).
secure([block{base:2, limit:2}]).
secure([block{base:3, limit:3}]).
non_secure([block{base:4, limit:4}]).
non_secure([block{base:5, limit:5}]).
non_secure([block{base:6, limit:6}]).
non_secure([block{base:7, limit:7}]).

can_access(Node, Temp, Addr) :- Temp = [I, Addr],
   node_overlay(Node, J), node_accept(J, [memory,
   Addr]).
can_access(Node, Temp, Addr) :- node_overlay(Node, A)
   , node_translate_dyn(A, Temp, C, D), can_access(C,
    D, Addr).
can_access(CPU, Addr) :- node_overlay(CPU, A), node_
   translate_dyn(A, B, C, D), can_access(C, D, Addr).
```

This Prolog code when added to the Prolog code generated by the Sockeye backend (e.g. minimal_trustzone.pl A.3 generated by minimal_trustzone.soc A.2) marks the addresses 0,1,2,3 as secure and addresses 4,5,6,7 as non-secure.

What the predicate can_access does is, given the variables CPU and Addr, it checks whether there are any assignment of an address such that the CPU maps that address to a name which can then recursively reach Addr.

Because of Prolog's way of working one can not only ask whether a specific assignment of CPU can reach a concrete Addr, one can also leave Addr as a variable and only assign CPU to a node and Prolog gives back all the names CPU can reach. Or the CPU variable can be left unassigned and a specific name given for Addr and Prolog returns all the nodes which can reach that name. One can even leave both unassigned and get back all the pairs of nodes and names where the node can in some way reach that name. Then it can be restricted to those where the CPU is marked as non-secure and Addr marked as secure.

The problem is this predicate becomes to slow at realistic address ranges because it uses the naive formulation of the reachability problem instead of exploiting the properties of intervals to speed it up.

Any instance of this problem on a graph of "address nodes" can be reduced into an instance of a problem on a graph of "interval nodes" (e.g. node (0; [0 to $2^{bitwidth}$])), then the graph reachability problem on this "interval nodes" graph can be solved and used to obtain the final solution.

In the worst case scenario where each address gets mapped differently, the number of nodes stays the same because each interval is of length 1. Given the properties of real hardware (locality, etc.) this case won't happen, the number of nodes is instead considerably reduced, thus getting a tractable problem.

What is left to prove is that if a path (doesn't) exist in the reduced problem then it also (doesn't) exist in the original problem.

3.8 Conclusion

As can be seen, the reference manual of TrustZone contains some ambiguities due to it being written in natural language. A hardware designer wanting to be TrustZone-compliant has no way of knowing whether they implemented the specification correctly.

By providing a Sockeye description of TrustZone I removed any ambiguities.

Having a Sockeye description of a system allows checking automatically whether the security properties of TrustZone are maintained by executing my query over it. This is quite useful for a manufacturer but also for an operating system having to configure for example the TZASC. That query is currently to slow for realistic systems due to it not exploiting the properties of intervals (see Chapter Conclusions and Future Work 7.1.5).

Chapter 4

Analysis of the Server Base System Architecture

In section 4.1 I will explain what the Server Base System Architecture is, and why it should be expressed formally in 4.2. I'm going to show how the SBSA relates to the problem of multiple address space in section 4.3. Following that I will show my Isabelle formalization of the SBSA in 4.4, while showing why each predicate I defined is needed based on quotes from the SBSA. Finally in section 4.5 I will show what a typical SBSA compliant system might look like and show the problems with the SBSA specification in 4.6.

4.1 What's the Server Base System Architecture?

The Server Base System Architecture (SBSA) is a system specification written by ARM which tries to standardize a minimum standard as a recommendation to manufacturers of Server Systems based on ARM.

Right now, there's a huge variability in the implementation of ARM based systems. Every manufacturer does things differently, this means the operating systems, hypervisors and firmware needs to be modified for each system. This cannot scale, it increases the cost of software system development and also the associated quality risks.

The primary goal of the SBSA is to enable a single OS image which targets the SBSA, to run on all systems compliant with the SBSA. Compliance to this specification is not mandatory but based on the idea that OEMs and software providers will demand compliance from the manufacturers to reduce their burden.

The Server Base System Architecture has multiple levels of functionality. Level N contains all the functionality of the N-1 previous levels. Level 3, 3 Firmware, 4 and 5 exist, the levels 0,1,2 have been folded into Level 3.

Systems that are compliant with a level of the Server Base System Architecture can include more features which aren't included in the definition of that level. Software written for a certain level, however, must run without modification on system that include additional functionality. [3]

4.2 Why formalize?

The problem with the Server Base System Architecture specification is that it's written in prose. Natural language is very ambiguous and hard to parse.

For example: the SBSA says that transactions coming from a PCI express device have to either address the memory system or be presented to a SMMU. Is it allowed to have some PCI express devices have an SMMU in front of them and some not? Or having some addresses of a PCI express device be sent to the memory system and some to the SMMU?

Moreover, being written in prose makes it impossible for the hardware manufacturer to check whether they are compliant to a certain level of the SBSA.

There is a project called SBSA Architecture Compliance Suite by ARM [6] which is a test suite one can run on a system to find out if it is SBSA-compliant. The suite relies on run-time behavior and asking the hardware for information so it isn't a replacement for a formal specification given that it doesn't describe the SBSA but only queries the hardware to check compliance and can only be used after the system is already manufactured which is too late to notice non-compliance. I was still able to make use of it to check whether my interpretation of the SBSA reference is correct.

Writing a formal specification of the Server Base System Architecture's memory subsystem using the Address Decoding Net Model in Isabelle allows to remove the ambiguity from the specification and allows one to prove formally that a system is SBSA-compliant.

The SBSA puts certain constraints on address decoding nets wanting to be compliant, an implementation can however include additional features. Meaning there is a set of address decoding nets which are compliant to SBSA Level 3, a subset of this set is compliant to SBSA Level 4 and so on. It follows that a predicate can be written which determines if an address decoding net is part of that set. Doing this allows a manufacturer which has a description of the address decoding net of their system to check whether they are compliant to a certain level of the SBSA.

There are various benefits to a formal specification:

- It makes imprecision in the specification more visible
- It forces one to think clearly about what one wants to express

- It's unambiguous
- It's easy to parse for a computer
- If the hardware manufacturer writes a formal specification of their hardware, compliance to the SBSA can be proven mathematically

The problem with trying to extract a formal specification from an informal specification is that assumptions have to be made on which way an ambiguous statement was meant. Thus, I will back every decision by an explanation on why I think the chosen interpretation is the more reasonable one.

As I explained in the Background chapter (2), memory systems are becoming ever more complicated e.g. with systems having more than one physical address space becoming more common. So is the SBSA trying to fight this trend of having multiple address spaces or is it encouraging it?

In the next section I will analyze whether a SBSA compliant system must have only a single address space, multiple address spaces or if both possibilities are allowed.

4.3 Address Spaces

Some definition of single address space from "Not your parents' physical address space" [9]:

- "Every memory location in a computer that could be addressed by the processor has a unique address"
- "All RAM, and all memory-mapped I/O registers appear in a single physical address space"
- "Any processor core (via its MMU) can address any part of this physical address space at any given time"
- "All processors use the same physical address for a given memory cell or hardware register."

Based on the address decoding net model I reformulated it to:

For every possible address every core resolves the address to the same name.

Some points in the SBSA that are an indication of a single address space:

Server Base System Architecture, Section 4.1.1 PE Architecture [3]
"All PEs are coherent and in the same Inner Shareable domain."

Server Base System Architecture, Section D.3 PCI Express device view of memory [3]

"In a system where the PCI express does not use an SMMU, the PCI express devices have the same view of physical memory as the PEs."

Server Base System Architecture, Section D.8 I/O Coherency [3]

"PCI Express transactions not marked as No_snoop accessing memory that the PE translation tables attribute as cacheable and shared are I/O Coherent with the PEs.

The PCI Express root complex is in the same Inner Shareable domain as the PEs."

Server Base System Architecture, Section E SMMUV3 INTEGRATION [3]

"All SMMU translation table walks and all SMMU accesses to SMMU memory structures and queues are I/O coherent"

But the naive view of a single address space quickly becomes impossible, because of statements like these:

Server Base System Architecture, Section 4.1.3 Memory Map [3]

"All Non-secure on-chip masters in a base server system that are expected to be under the control of the operating system or hypervisor must be capable of addressing all of the Non-secure address space."

Server Base System Architecture, Section 4.2.1 Memory Map [3]

"The system must provide some memory mapped in the Secure address space. The memory must not be aliased in the Non-secure address"

Further statements which mention secure address space can be found in the appendix B.1.

Being in secure address space means that non-secure PEs cannot access those address, which means that "Any processor core (via its MMU) can address any part of this physical address space at any given time"[9] does not hold anymore.

I weakened the definition of single address space to:

For every possible address every core either aborts or resolves to the same name.

But this doesn't hold either, for example in the GIC:

Generic Interrupt Controller Architecture Specification, Section 2.1 The GIC logical components [3]

"A Redistributor for each PE that is supported.

A CPU interface for each PE that is supported.

Generic Interrupt Controller Architecture Specification, Section 8.1.12 Register banking [2]

"If the GIC is implemented as part of a multiprocessor system:

-Some registers are Banked to provide a separate copy for each connected PE. These include the registers associated with PPIs and SGIs, and GICD_NSACR(n), where n=0, when implemented.

- The GIC implements the CPU interface registers independently for each CPU interface, and each connected PE accesses the registers for the interface to which it connects."

Therefore, there are some registers which are local to each core.

I tried weakening the definition of single address space even further to:

For every possible address except a small set of local only addresses, every core either aborts or resolves to the same name. The set of local only addresses are only accessible from one core.

This definition also fails, for example with register banking by security which happens in the GICv3 and the SMMU. Which means there are two registers with the same physical address, but a different one is accessed based on whether the master doing the access is secure or non-secure.

Server Base System Architecture, Section 4.1.4 Interrupt Controller [3]

"A level 3 base server system must implement a GICv3 interrupt controller. The GICv3 interrupt controller must support two Security states."

Generic Interrupt Controller Architecture Specification, Section 8.1.12 Register banking [2]

"If a GIC supports two Security states, some registers are Banked to provide separate Secure and Non-secure copies of the registers. The Secure and Non-secure register bit assignments can differ. A Secure access to the register address accesses the Secure copy of the register, and a Non-secure access accesses the Non-secure copy."

As can be seen, there are addresses that get resolved to different registers depending on which master is doing the access.

Thus, multiple address spaces exist.

Another point in SBSA, backing the idea that multiple address space are consistent with the specification is:

> **Server Base System Architecture, Section 4.1.3 Memory Map [3]**
>
> Compliant software must not make any assumptions about the memory map that might prejudice compliant hardware. For example, the full physical address space must be supported. There must be no dependence on memory or peripherals being located at certain physical locations.

Additional arguments are in the appendix B.2

Questions remaining are:

- What is meant with

  "In a system where the PCI express does not use an SMMU, the PCI express devices have the same view of physical memory as the PEs."[3]?

  The same view of physical memory of which PE? Of a non-secure PE? Does this imply that every non-secure PE has the same view of memory except the local only addresses?

- How is I/O Coherency implemented when multiple address spaces exist?

4.4 Isabelle Formalization

From the Server Base System Architecture Platform Design Document [3] I picked out every statement that relates to the memory subsystem and cited them in this section.

I will show my formalization in Isabelle and argue why I think what I formalized is what ARM meant in the SBSA.

The SBSA record contains everything that cannot be derived from the hardware decoding net alone.

```
  record SBSA =
all_pe :: "nodeid set"
non_secure_pe :: "nodeid set"
non_secure_on_chip_masters_os :: "nodeid set"
non_secure_on_chip_masters_firmware :: "nodeid set"
non_secure_off_chip_devices :: "nodeid set"
virtualized_devices :: "nodeid set"
pci_devices :: "nodeid set"
memory :: "nodeid"
secure_slave :: "name set"
```

```
non_secure_slave :: "name set"
uart :: "nodeid"
watchdog :: "nodeid"
generic_timer :: "nodeid"
wakeup_timer :: "nodeid"
gic_security_enabled :: "bool"
non_secure_distributor :: "nodeid"
secure_distributor :: "nodeid"
non_secure_redistributors :: "nodeid set"
secure_redistributors :: "nodeid set"
non_secure_smmus :: "nodeid set"
secure_smmus :: "nodeid set"
smmu_out_node :: "nodeid \<Rightarrow> nodeid"
pagesize :: "nat"
numpage :: "nat"
PCIe_address_space_and_IO_address_space :: "genaddr
    set"
```

Some notation: in the following list x := y means x is the set of node ids that represents y

- all_pe := PEs in the decoding net.
- non_secure_pe := PEs in the Non-secure state
- non_secure_on_chip_masters_os := masters which are in the Non-secure state, on-chip and which are under the control of an operating system or a hypervisor
- non_secure_on_chip_masters_firmware := masters, which are in the Non-secure state, on-chip and which are under the control of firmware
- non_secure_off_chip_devices := devices which are in the Non-secure state and off-chip
- virtualized_devices := virtualized devices
- pci_devices := PCIe devices
- memory is the node id representing DRAM
- secure_slave is the set of names (node ids + address) which should only be accessible from a master in the Secure state
- non_secure_slave is the set of names which should be accessible from Non-secure masters and from Secure masters
- uart is the node id representing the Generic UART
- watchdog is the node id representing the Generic Watchdog

- generic_timer is the node id representing the Generic Timer
- wakeup_timer is the node id representing the Secure Wakeup Timer
- gic_security_enabled represents whether the GIC security is enabled (true) or disabled (false)
- non_secure_distributor is the node id representing the distributor which is in the Non-secure state
- secure_distributor is the node id representing the distributor which is in the Secure state
- non_secure_redistributors := redistributors in the Non-secure state
- secure_redistributors := redistributors in the Secure state
- non_secure_smmus := SMMUs in the Non-secure state
- secure_smmus := SMMUs in the Secure state
- smmu_out_node is a relation which maps the node id of each SMMU in the system to the node id the corresponding SMMU outputs to
- pagesize is the pagesize of the SMMU
- numpage is the number of pages in the SMMU
- PCIe_address_space_and_IO_address_space is the set of addresses which are in PCIe address space or IO address space

The Server Base System Architecture states:

Server Base System Architecture, Section 4.1.1 PE Architecture [3]

"The number of PEs in the system must not exceed 2^{28} . This reflects the maximum number of PEs GICv3 can support."

In Isabelle this would be expressed as:

```
definition number_of_cores :: "SBSA ⇒ bool"
  where
    "number_of_cores sbsa =
    (card (all_pe sbsa) ≤ (2^28 :: nat))"
```

The complete Isabelle code can be found in the appendix C. For reasons of clarity from now on I'm going to use math notation instead of Isabelle. Mathematically expressed the same predicate is:

$$number\_of\_cores \ := \ |all\_pe| \leq 2^{28}$$

The cardinality of all_pe says how many PEs there are in the system, that number has to be less than 2^{28}

Server Base System Architecture, Section 4.1.3 Memory Map [3]

"Systems will not necessarily fully populate all of the addressable memory space.

All memory accesses, whether they access memory space that is populated or not, must respond within finite time, so as to avoid the possibility of system deadlock."

$$finite\_time\_resolution := \forall n.\ \exists f.\ wf\_rank(f, n, net)$$

What needs to be checked is that every name (node id together with an address) resolves in finite time. For that to hold there must exists a function f assigning a natural number to each step of the resolution, such that this number decreases at each step. This proves that the resolution terminates in finite time because at some point that number reaches 0.

Server Base System Architecture, Section 4.1.3 Memory Map [3]

"All Non-secure on-chip masters in a base server system that are expected to be under the control of the operating system or hypervisor must be capable of addressing all of the Non-secure address space."

I defined:

$$\begin{aligned} all\_non\_secure\_reachable\_masters\_os := {} & \forall m \in non\_secure\_on\_chip\_masters\_os. \\ & all\_non\_secure\_reachable(m) \end{aligned}$$

where all_non_secure_reachable is defined as

$$\begin{aligned} all\_non\_secure\_reachable(from) := {} & \forall ns\_name \in non\_secure\_slave. \\ & \exists addr.\ ns\_name \in resolve(from, addr) \end{aligned}$$

where resolve returns all the names that can be reached starting from the name given as an input.

I interpreted this to refer to Non-secure on-chip masters, which are not PEs (because PEs are treated differently as will be seen later) which are under the control of an OS or hypervisors (if the SBSA meant that Non-secure on-chip master are always expected to be under the control of an OS or hypervisor, there would be a comma before the "that" because it would be a nonrestrictive clause which offers additional information on something mentioned in the sentence).

These masters should be able to reach every address (name) which is marked as non-secure, concretely this means there is some address that the master can emit such that it resolves to a name which is in the set non_secure_slave. (capable of addressing could also be understood as meaning having a big enough output address size but in this case it doesn't make sense because it mentions Non-secure address space. The Non-secure address space could be anywhere address-wise so the output address size doesn't have anything to do with it)

Server Base System Architecture, Section 4.1.3 Memory Map [3]

"If the master goes through a SMMU then the master must be capable of addressing all of the Non-secure address space when the SMMU is turned off."

This can be written as:

$$\begin{aligned}&\mathit{all\_non\_secure\_reachable\_masters\_os\_smmu\_off} := \\ &\mathit{all\_non\_secure\_reachable\_smmu\_off}(\mathit{sbsa}, \mathit{net}, \mathit{non\_secure\_on\_chip\_masters\_os}(\mathit{sbsa}))\end{aligned}$$

where all_non_secure_reachable_smmu_off is defined as:

$$\begin{aligned}&\mathit{all\_non\_secure\_reachable\_smmu\_off} := \forall m \in \mathit{from}. \\ &(\exists \mathit{smmu} \in (\mathit{non\_secure\_smmus} \cup \mathit{secure\_smmus}).\ \mathit{behind\_smmu}(\mathit{net}, m, \mathit{smmu}) \implies \\ &\mathit{all\_non\_secure\_reachable}(\mathit{sbsa}, (\lambda n.\ \mathit{if}\ (n = \mathit{smmu})\ \mathit{then} \\ &(\mathit{accept} = , \mathit{translate} = (\lambda \mathit{addr}.\ \{((\mathit{smmu\_out\_node}), \mathit{smmu}, \mathit{addr})\}))\ \mathit{else}\ (\mathit{net}(n))), m))\end{aligned}$$

and behind_smmu is defined as:

$$\begin{aligned}&\mathit{behind\_smmu} := \forall \mathit{master\_addr}.\ \exists \mathit{smmu\_addr}. \\ &(\mathit{smmu}, \mathit{smmu\_addr}) \in \mathit{whole\_translation\_path}(\mathit{net}.(\mathit{node}, \mathit{master\_addr}))\end{aligned}$$

and whole_translation_path is:

$$\begin{aligned}&\mathit{whole\_translation\_path} := \mathit{fold}(\lambda \mathit{curr}, \mathit{acc}. \\ &\mathit{acc} \cup \mathit{whole\_translation\_path}(\mathit{net}, \mathit{curr})), \\ &\mathit{decode\_step}(\mathit{net}, \mathit{source}), \mathit{decode\_step}(\mathit{net}, \mathit{source}))\end{aligned}$$

The address decoding net model does not support modelling whether the SMMU is on or off so instead I modelled a turned off SMMU as an SMMU which has a one-to-one mapping, meaning the output address always corresponds to the input address. I then replace every SMMU with an SMMU which has a one-to-one mapping and check whether all the Non-secure names are reachable.

Server Base System Architecture, Section 4.1.3 Memory Map [3]

"Equally, all PEs must be able to access all of the Non-secure address space."

$$\begin{aligned} all\_non\_secure\_reachable\_pe \; &:= \forall c \in all\_pe. \\ &\quad all\_non\_secure\_reachable(sbsa, net, c) \end{aligned}$$

Analogously to all_non_reachable_masters_os but with PEs.

Server Base System Architecture, Section 4.1.3 Memory Map [3]

"Non-secure off-chip devices that cannot directly address all of the Non-secure address space must be placed behind a stage 1 System MMU compatible with the Arm SMMUv2 or SMMUv3 specification, that has an output address size large enough to address all of the Non-secure address space. See Section 4.1.6"

$$\begin{gathered} all\_non\_secure\_reachable\_off\_chip := \; \forall d \in non\_secure\_off\_chip\_devices. \\ (\neg all\_non\_secure\_reachable(sbsa, net, d)) \implies (\exists smmu.\ behind\_smmu(sbsa, net, smmu, d)) \end{gathered}$$

For every device first check whether it can reach every Non-secure slave. If it can't, it has to be behind an SMMU (I interpreted being behind an SMMU to mean that a node which is in the set smmu is in the path of translation of every address emitted by that device). Having a big enough output address size can't be represented in the address decoding net model so this part of the specification is omitted.

Server Base System Architecture, Section 4.1.3 Memory Map [3]

...", system software must not allocate any structures relating to a SMMU or GIC in PCIe address space or IO address space."

System Memory Management Unit Architecture Specification, Section 8.2 The global address space [11]

Server Base System Architecture, Section 4.1.4 Interrupt Controller [3]

Server Base System Architecture, Section 4.1.3 Memory Map [3]

..., “system software must not allocate any structures relating to a SMMU or GIC in PCIe address space or IO address space.”

Generic Interrupt Controller Architecture Specification, Section 8.1 About the programmers’ model [2]

“The Distributor, Redistributor, and ITS programming interfaces are always memory-mapped.”

“The CPU interfaces for physical and virtual interrupt handling, and the virtual machine control interface used by the hypervisor use: - System register interfaces for the operation of GICv3 and GICv4. - Memory-mapped interfaces for legacy operation.” “Note: Support for legacy operation is optional. ”

“Implementations are allowed to support legacy operation for virtual interrupts only, meaning that the GICV_* registers are the only memory-mapped CPU interface registers that are provided. In these implementations, GICC_* registers and GICH_* registers are not provided. ”

“GICC_* and GICH_* registers are only required to support legacy operation by physical interrupts.”

Generic Interrupt Controller Architecture Specification, Section 8.10 The GIC Redistributor register map [2]

Server Base System Architecture, Section 4.1.7 Clock and Timer Subsystem [3]

“The base server system must include the system counter of the Generic Timer as specified in the Arm ARM” [3]

ARM Architecture Reference Manual ARMv8, Section I2.2.3 Counter module control and status register summary [5]

Server Base System Architecture, Section A.3 Register summary [3]

Server Base System Architecture, Section B.2 Generic UART register frame [3]

$$uart\_frame := \exists base\_addr.$$
$$((uart, addr)|addr.\ addr \in range(base\_addr, 0x000, 0x047) \subseteq non\_secure\_slave)$$
$$\wedge\ (range(base\_addr, 0x000, 0x047) \subseteq accept(net, uart))$$

where range is defined as:

$$range := \{addr.\ \forall addr.\ (addr \geq basis + low) \land (basis \leq high)\}$$

The SBSA requires a compliant system to have an ARM GIC, Generic Timer, Generic Watchdog, Generic UART and depending on the system also an SMMU.

For each of these components there is a reference which states that there has to be some base address in the system such that its registers are memory-mapped at certain offsets from it. The memory-mapped registers can be represented as accepting addresses.

The problem is that there are some registers which can only be accessed from the Secure state. I represented this by saying there are two virtual component (e.g. SMMU, Distributor) for every real component, the Secure component and the Non-secure component. The Secure component only accepts the secure registers and these have to be part of the secure_slave set.

For the SMMU and GIC one also has to make sure that none of the addresses (base address + offsets) are in PCIe address space or IO address space.

In the GIC some registers are banked if the security is enabled, what this means is that the content of the register is different based on which of the secure states is making the access. In the address decoding net this can be represented by saying those are effectively two names, one is (secure_component, addr) and the other is (non_secure_component, addr)

The Redistributors and CPU interfaces are local to each PE which means that there needs to be a base address for each PE in the system.

Server Base System Architecture, Section 4.1.5 PPI assignments [3]

$$pe\_accept\_ppi := \forall c \in all\_pe.\ \{30, 29, 27, 26, 25, 24, 23, 22, 21\} \subseteq accept(net, c)$$

Server Base System Architecture, Section 4.1.5 PPI assignments [3]

$$redistributor\_translate\_ppi := \exists f.\ bij\_betw(f, redistributors, all\_pe) \land \forall r \in redistributors.$$
$$\forall intr \in \{30, 29, 27, 26, 25, 24, 23, 22, 21\}.\{(f(r), intr)\} = resolve(net, (r, intr))$$

Modelling the interrupt system of the SBSA is outside the scope of this thesis, but this would be a starting point. The PPI assignments are described by the SBSA and because the GIC reference requires that PPI have to pass through the redistributor I inferred that all redistributors have to translate these PPIs and all PEs have to accept them.

Server Base System Architecture, Section 4.1.6 I/O Virtualization [3]

"If a device is virtualized and passed through to an operating system under a hypervisor, then the memory transactions of the device must be subject to stage 2 translation, allocation of memory attributes, and application of permission checks, under the control of the hypervisor. This specification collectively refers to this translation attribution, and permission checking as policing. The act of policing is referred to as stage 2 System MMU functionality.

Stage 2 System MMU functionality must be provided by a System MMU compatible with the ARM SMMUv2 specification, where:

• Support for stage 1 policing is not required.

• Each context bank must present a unique physical interrupt to the GIC.

Or the Stage 2 System MMU functionality must be provided by a System MMU compatible with the ARM SMMUv3 spec where:

• Support for stage 1 policing is not required.

• The integration of the System MMUs is compliant with the specification in Section E.

All the System MMUs in the system must the compliant with the same architecture version."

$$\begin{aligned} virtualized\_device\_behind\_smmu \;&:= \forall d \in virtualized\_devices.\ \exists smmu \in smmus. \\ &\quad behind\_smmu(sbsa, net, smmu, d) \end{aligned}$$

For each virtualized device there has to be an SMMU such that each address has the SMMU in its translation path. (Could also be understood to mean that for each address there is any SMMU in its translation path)

Server Base System Architecture, Section D.3 PCI Express device view of memory [3]

"Transactions originating from a PCI express device will either directly address the memory system of the base server system or be presented to a SMMU for optional address translation and permission policing.

For accesses from a PCIe endpoint to the host memory system, in systems compatible with SBSA Level 3 or above, the following must be true:

• The addresses sent by PCI express devices must be presented to the memory system or SMMU unmodified.

• In a system where the PCI express does not use an SMMU, the PCI express devices have the same view of physical memory as the PEs. In a system with a SMMU for PCI express there are no transformations to addresses being sent by PCI express devices before they are presented as an input address to the SMMU."

$$\begin{aligned}
pcie\ :=\ & \forall pci \in pci\_devices.\ \forall addr.\ \exists smmu \in smmus. \\
& (smmu, addr) \in whole\_translation\_path(net, (pci, addr)) \lor \\
& \forall pci \in pci\_devices.\ \forall addr.\ (memory, addr) \in whole\_translation\_path(net, (pci, addr)) \land \\
& \neg(\exists smmu \in smmus.\ \exists any\_addr.\ (smmu, any_a ddr) \in whole\_translation\_path(net, (pci, addr)) \\
& \land (\forall pe \in all\_pe.\ resolve(net, (pe, addr)) = resolve(net, (pci, addr))))
\end{aligned}$$

For each address there has to either be an SMMU in the translation path and the address is unmodified when it arrives there or there is no SMMU in the translation path and the address arrives to memory unmodified and is resolved the same as a PE (two ambiguities here, can some addresses be behind an SMMU and other not, and does it mean that it has to resolve to the same as some PE or as every PE, in other words what if some PE resolve addresses differently e.g. difference between security states?)

$$\begin{aligned}
& sbsa3\_compliant\ :=\ number\_of\_cores \land finite\_time\_resolution \land \\
& all\_non\_secure\_reachable\_masters\_os \land all\_non\_secure\_reachable\_masters\_os\_smmu\_off \land \\
& all\_non\_secure\_reachable\_pe \land all\_non\_secure\_reachable\_off\_chip \land smmu\_frames \land \\
& distributor\_frame \land redistributor\_frames \land pe\_accept\_ppi \land redistributor\_translate\_ppi \land \\
& virtualized\_device\_behind\_smmu \land generic\_timer\_frames \land uart\_frame \land \\
& watchdog\_frames(non\_secure\_slave) \land pcie
\end{aligned}$$

This is the predicate which describes whether a decoding net is SBSA level 3 compliant. It's simply the conjunction of all the definitions. It's true when the combination of the sbsa set together with the decoding net of the system fulfills all the definitions, otherwise it's false.

Server Base System Architecture, Section 4.2.1 Memory Map [3]

"The system must provide some memory mapped in the Secure address space."

$$some\_secure := |secure\_slave| \geq 0$$

I interpreted "some" to just mean at least one name has to be in the secure_slave set.

Server Base System Architecture, Section 4.2.1 Memory Map [3]

"The memory must not be aliased in the Non-secure address space."

ARM Architecture Reference Manual ARMv8, Section D1.4 Security state [5]

"The ARMv8-A architecture provides two Security states, each with an associated physical memory address space, as follows:

- Secure state When in this state, the PE can access both the Secure physical address space and the Non-secure physical address space.

- Non-secure state When in this state, the PE:

• Can access only the Non-secure physical address space.

• Cannot access the Secure system control resources."

ARM Security Technology: Building a Secure System using TrustZone Technology [4]

- "Hardware logic present in the TrustZone-enabled AMBA3 AXI TM bus fabric ensures that no Secure world resources can be accessed by the Normal world components, enabling a strong security perimeter to be built between the two."

- "All Non-secure masters must have their NS bits set high in the hardware, which makes it impossible for them to access Secure slaves. The address decode for the access will not match any Secure slave and the transaction will fail"

- "If a Non-secure master attempts to access a Secure slave it is implementation defined whether the operation fails silently or generates an error. An error may be raised by the slave or the bus, depending on the hardware peripheral design and bus configuration, consequently a SLVERR (slave error) or a DECERR (decode error) may occur"

$$
\begin{aligned}
&\mathit{no\_aliasing} := ((\mathit{secure\_slave} \cap \mathit{non\_secure\_slave}) =)\\
&\wedge (\forall s \in \mathit{secure\_slave}.\\
&(\forall m \in (\mathit{non\_secure\_pe} \cup \mathit{non\_secure\_on\_chip\_masters\_os} \cup \mathit{non\_secure\_off\_chip\_devices}).\\
&(\neg(\exists \mathit{addr}.\ s \in \mathit{resolve}(\mathit{net}, (m, \mathit{addr}))))))
\end{aligned}
$$

Based on these statements by ARM I interpreted not aliased to mean that there is no way for a Non-secure master to access a secure slave.

no_aliasing checks that there is no name which is marked as secure slave and non-secure slave and that there is no virtual address such that a non-secure master can access a secure slave.

Server Base System Architecture, Section 4.2.1 Memory Map [3]

"All Non-secure on-chip masters in a base server system that are expected to be used by the platform firmware must be capable of addressing all of the Non-secure address space."

$$
\begin{aligned}
&\mathit{all\_non\_secure\_reachable\_masters\_firmware} :=\\
&\forall m \in \mathit{non\_secure\_on\_chip\_masters\_firmware}.\ \mathit{all\_non\_secure\_reachable}(\mathit{sbsa}, \mathit{net}, m)
\end{aligned}
$$

Similar to all_non_secure_reachable_masters_os but over the set of Non-secure on-chip masters that are used by the platform firmware.

Server Base System Architecture, Section 4.2.1 Memory Map [3]

"If the master goes through a SMMU then the master must be capable of addressing all of the Non-secure address space even when the SMMU is off."

$$
\begin{aligned}
&\mathit{all\_non\_secure\_reachable\_masters\_firmware\_smmu\_off} :=\\
&\mathit{all\_non\_secure\_reachable\_smmu\_off}(\mathit{sbsa}, \mathit{net}, (\mathit{non\_secure\_on\_chip\_masters\_firmware}(\mathit{sbsa})))
\end{aligned}
$$

Similar to all_non_secure_reachable_masters_os_smmu_off but over the set of Non-secure on-chip masters that are used by the platform firmware.

Server Base System Architecture, Section 4.2.4 Peripheral Subsystems [3]

"A system compatible with level 3-firmware must provide a second generic UART, referred to as the Secure Generic UART, that can be configured to exist in the Secure memory address space.

It must not be aliased in the Non-secure address space."

It's not clear what the SBSA means when it says that the Secure Generic

UART can be configured to exist in the Secure memory address space. Does that mean it can also be configured to exist in Non-secure memory address space?

$$\begin{aligned}&sbsa3\_firmware\_compliant := sbsa3\_compliant \land some\_secure \land no\_aliasing \land\\ &all\_non\_secure\_reachable\_masters\_firmware \land\\ &all\_non\_secure\_reachable\_masters\_firmware\_smmu\_off \land wakeup\_timer\_frames \land\\ &watchdog\_frames(secure\_slave)\end{aligned}$$

Being SBSA Level 3 Firmware compliant means being compliant to SBSA Level 3 and the additional predicates I listed must be true.

$$sbsa4\_compliant := ...$$

The SBSA doesn't state whether Level 4 requires all predicates from Level 3 or also of Level 3 firmware to be true to be compliant.

$$sbsa5\_compliant := sbsa4\_compliant \land ...$$

Being SBSA Level 5 compliant means being compliant to Level 4 and the additional predicates.

4.5 Typical SBSA-compliant system

One might wonder what a typical SBSA-compliant system looks like.

It necessarily will have an ARM GIC, an ARM Generic Timer, an ARM Generic Watchdog and an ARM Generic UART with the corresponding frames.

Some things about masters and PEs being able to access Non-secure slaves are known.

If it has virtualized devices then the system must have at least one SMMUv2 or SMMUv3.

If local PE timers aren't always on, it will contain a system wakeup timer.

It doesn't necessarily have PCI express but if it does, it is connected either directly to the AXI or will first pass through an SMMU. Its transactions will either reach the memory system or an SMMU unmodified.

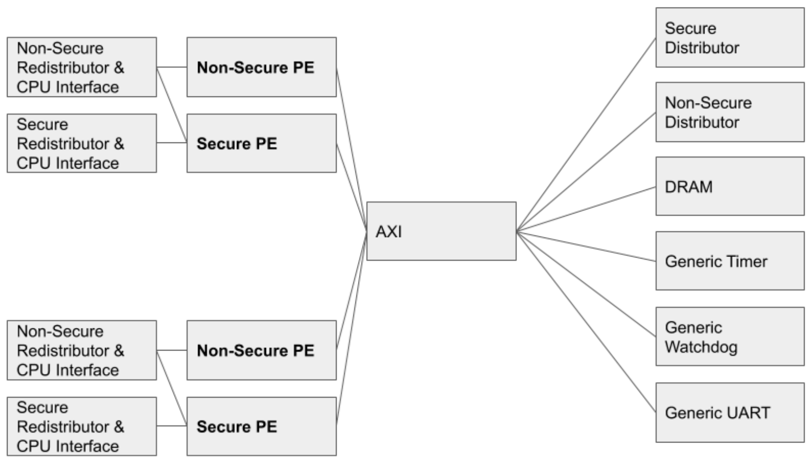
Non-Secure Redistributor & CPU Interface
Non-Secure PE
Secure Redistributor & CPU Interface
Secure PE
AXI
Non-Secure Redistributor & CPU Interface
Non-Secure PE
Secure Redistributor & CPU Interface
Secure PE
Secure Distributor
Non-Secure Distributor
DRAM
Generic Timer
Generic Watchdog
Generic UART

Figure 4.1: Minimal SBSA-compliant system

Not much can be said apart from that for a SBSA Level 3 compliant system. It might have any memory map and any additional devices and functions.

In figure 4.1 I show what a quite minimal SBSA system looks like.

4.6 Problems with the SBSA

- The SBSA doesn't say that two frames e.g. the UART's frame and the watchdog's frame can't be overlapping. This is very bad because if two frames are overlapping then writing to a location will cause multiple changes.
- In Level 3 - firmware it says that there has to be some memory mapped into the secure address space but doesn't say how much that is. Is one byte enough?

  Also, why does it only here say that the secure address space shouldn't be aliased into the non-secure address space? What about a Level 3 compliant system that has memory mapped into secure space?
- **Server Base System Architecture, Section 4.1.3 Memory Map [3]**

  "All Non-secure on-chip masters in a base server system that are expected to be under the control of the operating system or hypervisor must be capable of addressing all of the Non-secure address space."

  It's hard to parse whether this means that there is a subset of Non-secure on-chip masters which are under the control of the operating system or hypervisor or if it's just clarifying that Non-secure on-chip masters are under the control of the OS/hypervisor.

- Being able to address all the Non-secure address space could be understood to mean having a big enough output address size to output all the addresses which are Non-secure or they should be actually capable of accessing all the Non-secure slaves.

- **Server Base System Architecture, Section 4.1.3 Memory Map [3]**

  "If the master goes through a SMMU then the master must be capable of addressing all of the Non-secure address space when the SMMU is turned off."

  It's not clear what it means for a master to go through an SMMU, is it enough if there exists an address such that the translation path from that master goes through an SMMU or must all addresses have an SMMU in the translation path?

  What does it mean from the point of view of the address decoding net for an SMMU to be turned off? Is that implementation defined? Is it just equivalent to a one-to-one mapping in the SMMU?

- **Server Base System Architecture, Section 4.1.3 Memory Map [3]**

  "Non-secure off-chip devices that cannot directly address all of the Non-secure address space must be placed behind a stage 1 System MMU compatible with the Arm SMMUv2 or SMMUv3 specification, that has an output address size large enough to address all of the Non-secure address space. See Section 4.1.6"

  Do all addresses from that device have to go through the SMMU or only some. What if there are multiple SMMUs, can some addresses go through one and some through the other?

  Moreover, what part of the address space is Non-secure and the size of it, is a property that changes at run-time, so how can it have an output address size that is big enough if the hardware designer doesn't know how big the Non-secure address space will be? Either the hardware manufacturer assumes that the whole address space will be possibly Non-secure or the operating system has to limit the size of the Non-secure address space.

- It's not explained whether the table describing the frames of the timer refer to the Generic Timer or to the Wakeup Timer

- **Server Base System Architecture, Section D.3 PCI Express device view of memory [3]**

  "Transactions originating from a PCI express device will either directly address the memory system of the base server system or be presented to a SMMU for optional address translation and permission policing.

  For accesses from a PCIe endpoint to the host memory system, in systems compatible with SBSA Level 3 or above, the following must be true:

  • The addresses sent by PCI express devices must be presented to the memory system or SMMU unmodified.

  • In a system where the PCI express does not use an SMMU, the PCI express devices have the same view of physical memory as the PEs. In a system with a SMMU for PCI express there are no transformations to addresses being sent by PCI express devices before they are presented as an input address to the SMMU."

  Can some PCI express devices have an SMMU in front of them and some not? Can some addresses of a PCI express device be sent to the memory system and some to the SMMU? What does it mean for PCI express not to use the SMMU? What if only some devices use the SMMU? And most importantly what does it mean to have the same view of physical memory as the PEs if PEs might have different views of memory themselves, most prominent example being secure against non-secure PEs?

- **Server Base System Architecture, Section 4.2.4 Peripheral Subsystems [3]**

  "A system compatible with level 3-firmware must provide a second generic UART, referred to as the Secure Generic UART, that can be configured to exist in the Secure memory address space.

  It must not be aliased in the Non-secure address space."

  What does this mean that it can be configured to exist in the Secure memory address space? Can it be configured to be in Non-secure address space? Wouldn't that contradict it not being aliased in the Non-secure address space?

- Is Level 3 enough or is also Level 3 firmware a necessary condition to be Level 4 compliant?

4.7 Conclusion

I demonstrated that ARM's Server Base System Architecture can't fulfill its goals of unifying the architecture space because it is too ambiguous. Hardware manufacturer will parse the requirements differently leading to having incompatible systems.

Already answering whether the SBSA requires there to be a single physical address space or multiple ones is not straightforward to answer from the reference.

I found various other under-specifications and ambiguities. I then showed how to resolve those problems by formalizing the SBSA in Isabelle.

Chapter 5

ThunderX SBSA compliance

In section 5.1 I'll list what the ThunderX claims to support and I'm going to explain what the ThunderX' memory map looks like. Then in 5.2 I will list some ambiguities that I found in the ThunderX specification and how I interpreted them. After that in section 5.3 I'm going to show how I formalized it in Isabelle and which problems came up while doing it. Finally, in 5.4 I will show how a proof of SBSA-compliance would have looked like.

5.1 ThunderX in detail

5.1.1 What it claims to support

The ThunderX reference claims that the ThunderX is compatible with the ARM Server Base System Architecture but doesn't specify which level.

Although it claims that the ThunderX cores conform to the Level 2 of the SBSA and also implements features from higher levels, e.g. exception level EL3. [7]

5.1.2 Architecture

The ThunderX has a main bus called Near-Coprocessor Bus (NCB) which connects all the components together.

Connected to the main bus there are many accelerators like a true random number generator, a data compression/decompression unit and the Hyper Finite Automata Unit.

The Cavium Coherent Processor Interconnect (CCPI) allows seamlessly connecting two ThunderXs together and act as one big system.

5.1.3 Memory map

The memory map of the ThunderX is quite complex. The address space of the ThunderX is split into DRAM and I/O space based on the first bit. The second bit has to always be 0 for the address to be valid and in the case of DRAM also the 45th to 42nd bit. The field called "CCPI Node" signals to which CCPI node (0 - 3) this memory access has to be forwarded to. In the case of DRAM all the remaining offset bits are sent to DRAM. For NCB, SLI, AP and RSL there are some bits (NCB DID, SLI DID, AP DID and SLI DID) saying which device the offset is sent to. E.g. the access is to NCB if the 43rd to 36th bit are between 0x0 and 0x7d, that number then actually being the ID of the NCB device the offset gets forwarded to. [7]

What is missing from this memory map is the NS bit. In my Isabelle formalization I simply considered the NS bit to be 48th bit of the memory map.

5.2 ThunderX ambiguities

Some things which were not clear from reading the reference manual are:

Does the word "secure" being in parentheses in the description of a register mean that it is banked by security? E.g. the SMMU (Secure) Auxiliary Control Registers SMMU(0..3)_(S)ACR

If a register isn't called secure nor non-secure, does that mean that it is shared? Is it aliased, such that it can be accessed no matter whether the NS bit is set or not? E.g. the SMMU Identification Registers 0 SMMU(0..3)_IDR0

If a register is called non-secure does that mean that it is aliased such that it can be accessed with the NS bit set or not, or does it just mean that the secure register can also access it but has to do a non-secure access? E.g. GIC Redistributor Non-Secure Access Control Secure Registers GICR(0..47)_NSACR

5.3 Formalization of ThunderX

I'm now going to explain my formalization of the ThunderX. The full Isabelle theory can be found in the appendix D

Formalizing the ThunderX consists in writing for every node in the system which addresses it accepts and which addresses it maps to which name. Doing this for all the nodes in the ThunderX is outside the scope of this work as there are far too many.

First of all I map to a virtual non-secure or the virtual secure bus based on the most significant bit of the address, which corresponds to the NS bit. By using 0x0 as the base address only the offset gets sent to the corresponding bus.

The bus then checks the most significant bit to see if it's DRAM or I/O space and the second and third most significant bit to see which CCPI node this address goes to and maps differently based on that. I made a DRAM and I/O dispatcher node for each CCPI, and map to the correct one at address 0x0.

The I/O dispatcher for each CCPI node checks the most significant bits to see if it's a RSL, SLI or NCB address. If it's a NCB address it directly maps to the correct one for example the GIC or the SMMU, for RSL and SLI it maps it further to additional dispatchers.

The RSL dispatcher and SLI dispatcher then finally map the offset to the correct RSL or SLI device.

Inputting the memory-mapped registers of each node, e.g. the GIC and the SMMU, manually takes a lot of time, so I created a function which takes a list of addresses together with the security state of each register (shared, non-secure only, secure only, banked by security) and outputs three node specs one for the non-secure node, one for the secure node and one for the shared node.

One problem that arose is that some registers are secure-only depending on some other state e.g. if the interrupt is marked as secure, then the register is secure-only. This property cannot be represented in the address decoding net model.

Finally, in "sys" I assign to each node id its corresponding node.

5.4 Proof of ThunderX' SBSA compliance

Because I don't have a complete formalization, I can't prove formally that the ThunderX is SBSA-compliant. If I did, the next steps would be correctly initializing each field inside the "sbsa" record i.e. put the correct node ids of my ThunderX' formalization into secure_pe, non_secure_pe etc. Additionally, I would need to identify which names are secure or non-secure to put into the sets secure_slave and non_secure_slave.

The proof would then start by writing inside the SBSA Isabelle theory:

```
lemma thunderx_sbsa:
  "sbsa3_compliant sbsa (mk_net sys)"
  (* Insert proof here *)
  sorry
```

The proof would proceed by proving each term of the sbsa3_compliant conjunction separately. The most straightforward terms would be the ones about the existence of certain base addresses. Given that the ThunderX states concrete base addresses for each frame, a constructive existence proof is easy to do.

Proofs about reachability of non-secure names would proceed by splitting all the possible output addresses of PEs/on-chip masters and so on and showing that through the mappings it can reach every non-secure slave.

5.5 Conclusion

I exposed that not only specifications describing hardware standards like TrustZone and SBSA contain ambiguities and are hard to understand, but a concrete system like the ThunderX also is. The parts about security are especially bad, which could be an indication that the TrustZone specification is not precise enough.

I could not complete the formalization of the ThunderX' memory system because copying all the accepting offsets with their correct security behavior from the reference would have taken too long and been too tedious. Having a machine-readable reference manual would have sped up the formalization a lot, which goes shows how useful it would be to have a machine-readable format for the specification.

The proof of compliance would also be quite cumbersome to write for every machine. Automation could help quite a bit here, given that the actual steps of the proof are not hard but just take a long time to write. I'm going to expand on this in the Conclusions and Future work chapter 7.1.3.

Finally, problems with Sockeye became apparent, because Sockeye lacks good syntactic sugar to describe a memory map like the ThunderX' one. I will show my proposed fixes in the chapter Decoding net model and Sockeye evaluation 6.

Chapter 6

Decoding net model and Sockeye evaluation

I'm going to show which parts of the address decoding net model need improvement and also some things I noticed are cumbersome to write in Sockeye because of missing syntax. I'll also provide a proposal for each new syntactic element.

6.1 Keeping output address size

As can be seen in the SBSA formalization, often it's useful to say something about the number of output bits a node has. Right now this information is present in the Sockeye syntax but not in the Isabelle formalization. This could be fixed by adding a field to the node struct in Isabelle which contains the address type of the node i.e. the set of allowed addresses.

6.2 Sockeye Syntax Improvements

6.2.1 Bit fields

While writing the ThunderX decoding net in Isabelle, it became obvious that there had to be a better way in Sockeye to map an address based on certain bits of the address instead of only allowing ranges of natural numbers. This also helps when encoding the NS bit in single-dimensional Sockeye, where one often has to accept and map differently based on the NS bit. My proposed syntax is:

```
memory (0 bits 48) BUS
BUS[48] maps [
        if 0 then BUS[0 to 47] to DRAM at 0x0
        if 1 thenn BUS[0 to 47] to IO at 0x0
]
DRAM[40 to 41] maps [
        if 00 then DRAM[0 to 39] to CCPI_ZERO
        if 01 then DRAM[0 to 39] to CCPI_ONE
        if 10 then DRAM[0 to 39] to CCPI_TWO
        if 11 then DRAM[0 to 39] to CCPI_THREE
]
```

6.2.2 Tagging modules

Because the address decoding net model doesn't have any knowledge about which module represent secure cores, non-secure cores, SMMUs, GICs, timers and other devices, I had to pass in all this information manually through the "SBSA" record for the Isabelle formalization. It would be of great help in speeding up such formalizations in the future if Sockeye had support for tagging modules with such additional information. That information could not only be used when exporting to Isabelle for a proof but could also be useful to make Barrelfish's SKB more context-aware.

Proposed syntax:

```
#non_secure_core
module PE_1 {
}
#secure_core
module PE_2 {
}
```

6.2.3 Security domain

Whether one uses single-dimensional or multi-dimensional Sockeye when writing a description of a system which supports TrustZone, there's always the risk that one maps an address coming from a non-secure core to a secure slave. Moreover, the intent of the mapping isn't clear when reading the Sockeye description. Also, because the syntax doesn't allow expressing the semantics of it, no tool support can be provided to help in enforcing the security property that no non-secure master should be able to access a secure slave.

I propose adding syntax to Sockeye which allows expressing which address ranges are non-secure/secure and allow mapping based on the security of the master making the request. The added syntax allows to automatically check on compile whether the security property of TrustZone is maintained (see Chapter TrustZone 3.7).

Proposed syntax:

```
#non_secure_core
module PE{
    output memory (0 bits 4) RAMOUT
}

module MMU {
    input memory (0 bits 4) IN
    output memory (0 bits 4) OUT
    IN maps [
        (*) non_secure to OUT at 0
        (*) secure to OUT at 0
    ]
}

module DRAM{
    input memory (0 bits 4) IN
    IN accepts [
        (0 to 7) secure,
        (8 to 15) non_secure
    ]
}

module SYSTEM {
    pe instantiates PE
    mmu instantiates MMU
    dram instantiates DRAM
    pe binds [
            RAMOUT to mmu.IN
    ]
    mmu binds [
            OUT to dram.IN
    ]
}
```

This is an example where the security property of TrustZone doesn't hold (PE which is non_secure can access (0 to 7) at DRAM which is a secure address) and where it would have been helpful if some tool helped in recognizing it. Recognizing this violation of TrustZone isn't possible when using the unmodified Sockeye syntax.

6.3 Conclusion

I explained why the address type from a Sockeye description should be kept in Isabelle.

I also showed some additional syntax for Sockeye such that by using bit fields complicated memory maps become easier to specify.

I proposed syntax to tag module with additional information that can't be read out from the address decoding net alone but would be useful for external tools like a compliance checker.

Finally, I proposed a new domain for Sockeye, the security domain. Having such a domain allows expressing mapping based on security state more easily. It also allows an external tool to check that there are no security violations in the Sockeye description of a system.

Chapter 7

Conclusions and Future work

In this work I showed that writing specifications of hardware standards in prose as is the usual way to do it leads to them being ambiguous.

By writing TrustZone and the SBSA formally I not only removed ambiguities but also allowed tool support to automatically check compliance of concrete systems.

Not only hardware standards experience the problem of natural language but concrete systems like the ThunderX too. Writing these down in a formal specification language removes the ambiguities and allows checking properties like TrustZone or SBSA-compliance.

Finally, based on the knowledge gained I proposed specific improvements to the address decoding net model in Isabelle and to Sockeye.

In section Future work 7.1 I show some ideas on how to build on this work.

7.1 Future work

7.1.1 Isabelle backend for Sockeye

Writing the Isabelle formal model for each system to prove properties about it is a lot of work. It would be much nicer to write the description in Sockeye and then have the backend translate that Sockeye description to the Isabelle description. This is especially useful because the Sockeye description has to be written anyway for a system if one wants the SKB to automatically support it.

Adding an Isabelle backend to Sockeye is straightforward because the Isabelle model is also based on the address decoding net model, so it would just be a question of instantiating the modules, unrolling the loops and then translate all the nodes' accept and map operations into Isabelle.

7.1.2 Abstract Sockeye Language

In this work the Server Base System Architecture was described directly in Isabelle. Another possibility would have been to define a language for describing abstract specifications like the SBSA i.e. a language describing sets of address decoding nets.

Based on what I saw while writing the formal model for the Server Base System Architecture and what I think might be useful for other standards' description I'm proposing some primitives for an Abstract Sockeye Language to make describing such standards more straightforward. There is a trade-off to be made here between having a language that is more complex but allows describing more possible standards or a simpler language that allows writing fewer possible standards. But history and common sense show that the useful standards will behave nicely which means the language for describing them will be quite simple.

For reachability I would add a command to Sockeye called "reaches"

```
module PE {
    output (0 bits 48) OUT
    forall (ns_name in non_secure_slaves) {
        OUT reaches [ ns_name ]
    }
}
}
```

Having this syntax, the all_non_secure_reachable_masters_os, all_non_secure_reachable_pe and all_non_secure_reachable_masters_firmware (for no_aliasing additionally need logical not) definitions can already be expressed.

Adding the keyword "exists" would additionally allow expressing most of the remaining definitions:

```
module UART {
    output (0 bits 48} OUT
    input (0 bits 48) FRAME
    exists base in (0 bits 48) {
        FRAME accepts [
            (base+0x000 to base+0x047)
        ]
    }
}
}
```

For example in the redistributor $rd\_base + 0xFFFC \leq sgi\_base + 0x0080$ has to hold. This could be expressed by changing the (0 bits 48) to (rd_base+0xFF7C to 2^48) for the sgi_base set.

Having this syntax the smmu_frames, distributor_frame, redistributor_frames, generic_timer_frames, uart_frame, watchdog_frames and wakeup_timer_frames definitions of my SBSA formalization could already be expressed.

There needs to be a way to specify sets like non_secure_slaves, all_pe and so on. And a way to check the cardinality of those sets is inside some bounds. That way the definitions number_of_cores and some_secure could also be expressed.

7.1.3 Compliance oracle

By restricting what the Abstract Sockeye language allows, one can guarantee that finding a proof or a counterexample of compliance can be automated.

One way of guaranteeing it is by only allowing first-order logic in the Abstract Sockeye language. If more than first-order logic is allowed at some point over-approximation has to be done such that it either finds a proof or it's unknown whether its compliant or not.

As can be seen in the example of the SBSA, most predicates over the decoding net are of first-order (apart from proving finite time resolution all of them in this case).

A tool could be written, by making use of an SMT solver, that given an Abstract Sockeye file describing a standard and a Sockeye file describing a concrete address decoding net, outputs false when it finds a counterexample or can't prove it (in the case of first-order logic it only outputs false when it finds a counterexample) and true otherwise.

This would be useful when writing a new standard to check that it's not being over-constrained, by checking that a concrete address decoding net which should be allowed by the standard is compliant.

7.1.4 Generating compliant address decoding nets

Having such an oracle allows one to write a tool (e.g. naively by generating all possible address decoding nets of a certain size and check with compliance oracle) that automatically generates address decoding nets which are compliant to a certain standard.

This tool in turn would be useful when creating a new standard. A draft of the standard could be run through the tool to see if any of the compliant address decoding nets are weird and shouldn't be allowed by the standard i.e. it's currently under-constrained. This newfound information can then be used to iterate on the standard.

7.1.5 Speeding up TrustZone-compliance checking

My current Prolog query for checking whether a Sockeye description maintains the security properties of TrustZone becomes slow at realistic sizes because Prolog tries to find satisfying assignments of variables by going through each address.

One way of speeding up is to use a graph library which is optimized for finding reachability. A more complex solution involves keeping the information about ranges in the graph thus reducing the size of the graph considerably.

This TrustZone-compliance checker would not only be useful as a standalone tool for hardware designers but could also be integrated into the OS as an assertion to check whether the security properties still hold after changing the configuration of the TZASC. It also most probably would be useful as a subroutine in the compliance oracle.

Appendix A

TrustZone

A.1 trustzone.soc

```
module System {
    instance cpu of CPU
    cpu instantiates CPU
    instance dram of DRAM
    dram instantiates DRAM
    instance gic of GIC
    gic instantiates GIC
    gic binds [
        INTERRUPT_OUT to  cpu.INTERRUPT
    ]
    instance axi of AXI
    axi instantiates AXI
    instance apb of APB
    apb instantiates APB
    instance bridge of BRIDGE
    bridge instantiates BRIDGE
    instance device of DEVICE
    device instantiates DEVICE
    instance tzpc of TZPC
    tzpc instantiates TZPC
    cpu binds [
        ACCESS to axi.ACCESS;
        CONFIG_TZASC to axi.TZASC_CONFIG;
        CONFIG_TZPC to tzpc.CONFIG
    ]
    axi binds [
        BRIDGE to brdige.ACCESS_IN;
        DRAM to dram.MEMORY;
        DEVICE to device.ACCESS
    ]
    bride binds [
        ACCESS_OUT to apb.ACCESS
    ]
}

module AXI {
    input memory (0 to 1; 0 bits 48) TZASC_CONFIG
    TZASC_CONFIG accepts [
        (0; *)
    ]
    input memory (0 to 1; 0 bits 48) ACCESS
    output memory (0 bits 48) DRAM
    output memory (0 bits 48) BRIDGE
    output memory (0 bits 48) DEVICE
}

module DEVICE {
    input memory (0 bits 48) ACCESS
    ACCESS accepts [
        (*)
    ]
}

module APB {
    input memory (0 bits 48) ACCESS
    ACCESS accepts [
        (*)
    ]
}

module BRIDGE {
    input memory (0 bits 8) TZPCDECPROT
    input memory (0 to 1; 0 bits 48) ACCESS_IN
    output memory (0 bits 48) ACCESS_OUT
    ACCESS_IN maps [
        (0; *) to ACCESS_OUT at (*);
        (1; *) to ACCESS_OUT at (*)
    ]
}

module TZPC {
    input memory (0 to 1; 0 bits 8) CONFIG
    CONFIG accepts [
        (0; *)
    ]
}


module CPU {
    output memory (0 to 1; 0 bits 48) ACCESS
    output memory (0 to 1; 0 bits 8) CONFIG_TZPC
    output memory (0 to 1; 0 bits 48) CONFIG_TZASC
    input intr (0 bits 8) INTERRUPT
    INTERRUPT accepts [
        (*)
    ]
}


module DRAM {
    input memory (0 bits 48) MEMORY
    MEMORY accepts [
        (*)
    ]
}


module GIC {
    input intr (0 to 1; 0 bits 8) INTERRUPT_IN
    output intr (0 bits 8) INTERRUPT_OUT
    INTERRUPT_IN maps [
        (1; *) to INTERRUPT_OUT at (*); //IRQ
        (0; *) to INTERRUPT_OUT at (*) //FIQ
    ]
    input memory (0 to 1; 0 bits 8) INTERRUPT_
       SECURITY_REGISTER
    INTERRUPT_SECURITY_REGISTER accepts [
        (0; *)
    ]
}
```

A.2 minimal_trustzone.soc

```
module SYSTEM {
        instance non_secure_cpu of CPU
        non_secure_cpu instantiates CPU
        instance mapper of MAPPER
        mapper instantiates MAPPER
        non_secure_cpu binds [
                ACCESS to mapper.ACCESS_IN
        ]
        instance secure_cpu of CPU
        secure_cpu instantiates CPU
        instance tzasc of TZASC
        tzasc instantiates TZASC
        instance dram of DRAM
        dram instantiates DRAM
        mapper binds [
                ACCESS_OUT to tzasc.ACCESS_IN
        ]
        secure_cpu binds [
                ACCESS to tzasc.ACCESS_IN
        ]
        tzasc binds [
                ACCESS_OUT to dram.MEMORY
        ]
}

module CPU {
        output memory (0 to 1; 0 bits 10) ACCESS
}
module MAPPER {
        input memory (0 to 1; 0 bits 10) ACCESS_IN
        output memory (0 to 1; 0 bits 10) ACCESS_OUT
        ACCESS_IN maps [
                (0; *) to ACCESS_OUT at (1; *);
                (1; *) to ACCESS_OUT at (1; *)
        ]
}

module TZASC {
        input memory (0 to 1; 0 bits 10) ACCESS_IN
        output memory (0 bits 10) ACCESS_OUT
        forall addr in (0 to 512) {
                ACCESS_IN maps [
                        (0; addr) to ACCESS_OUT at (
                           addr)
                ]
        }
        forall addr in (513 to 1023) {
                ACCESS_IN maps [
                        (1; addr) to ACCESS_OUT at (
                           addr)
                ]
        }
}

module DRAM {
        input memory (0 bits 10) MEMORY
        forall addr in (0 to 1023) {
                MEMORY accepts [
                        (addr)
                ]
        }
}
```

A.3 minimal_trustzone.pl

```
add_SYSTEM(Id) :-
    is_list(Id),
    ID_non_secure_cpu = ["non_secure_cpu" | Id],
    ID_mapper = ["mapper" | Id],
    ID_secure_cpu = ["secure_cpu" | Id],
    ID_tzasc = ["tzasc" | Id],
    ID_dram = ["dram" | Id],
    add_CPU(ID_non_secure_cpu),
    add_MAPPER(ID_mapper),
    assert(node_overlay(["ACCESS" | ID_non_secure_cpu
       ],["ACCESS_IN" | ID_mapper])),
    add_CPU(ID_secure_cpu),
    add_TZASC(ID_tzasc),
    add_DRAM(ID_dram),
    assert(node_overlay(["ACCESS_OUT" | ID_mapper],["
       ACCESS_IN" | ID_tzasc])),
    assert(node_overlay(["ACCESS" | ID_secure_cpu],["
       ACCESS_IN" | ID_tzasc])),
    assert(node_overlay(["ACCESS_OUT" | ID_tzasc],["
       MEMORY" | ID_dram])).

add_CPU(Id) :-
    is_list(Id),
    (ID_ACCESS,INKIND_ACCESS,OUTKIND_ACCESS) = (["
       ACCESS" | Id],memory,memory).

add_MAPPER(Id) :-
    is_list(Id),
    (ID_ACCESS_IN,INKIND_ACCESS_IN,OUTKIND_ACCESS_IN)
        = (["ACCESS_IN" | Id],memory,memory),
    (ID_ACCESS_OUT,INKIND_ACCESS_OUT,OUTKIND_ACCESS_
       OUT) = (["ACCESS_OUT" | Id],memory,memory),
    assert(node_translate_dyn(ID_ACCESS_IN,[memory,[
       block{base:0,limit:0}],NYI],ID_ACCESS_OUT,[
       memory,[block{base:1,limit:1}],NYI])),
    assert(node_translate_dyn(ID_ACCESS_IN,[memory,[
       block{base:1,limit:1}],NYI],ID_ACCESS_OUT,[
       memory,[block{base:1,limit:1}],NYI])).

add_TZASC(Id) :-
    is_list(Id),
    (ID_ACCESS_IN,INKIND_ACCESS_IN,OUTKIND_ACCESS_IN)
        = (["ACCESS_IN" | Id],memory,memory),
    (ID_ACCESS_OUT,INKIND_ACCESS_OUT,OUTKIND_ACCESS_
       OUT) = (["ACCESS_OUT" | Id],memory,memory),
    assert(node_translate_dyn(ID_ACCESS_IN,[memory,[
       block{base:0,limit:0}],[block{base:0,limit
       :0}]],ID_ACCESS_OUT,[memory,[block{base:0,
       limit:0}]])),
    assert(node_translate_dyn(ID_ACCESS_IN,[memory,[
       block{base:0,limit:0}],[block{base:1,limit
       :1}]],ID_ACCESS_OUT,[memory,[block{base:1,
       limit:1}]])),
    assert(node_translate_dyn(ID_ACCESS_IN,[memory,[
       block{base:0,limit:0}],[block{base:2,limit
       :2}]],ID_ACCESS_OUT,[memory,[block{base:2,
       limit:2}]])),
    assert(node_translate_dyn(ID_ACCESS_IN,[memory,[
       block{base:0,limit:0}],[block{base:3,limit
       :3}]],ID_ACCESS_OUT,[memory,[block{base:3,
       limit:3}]])),
    assert(node_translate_dyn(ID_ACCESS_IN,[memory,[
       block{base:1,limit:1}],[block{base:4,limit
       :4}]],ID_ACCESS_OUT,[memory,[block{base:4,
       limit:4}]])),
    assert(node_translate_dyn(ID_ACCESS_IN,[memory,[
       block{base:1,limit:1}],[block{base:5,limit
```

```
        :5}]],ID_ACCESS_OUT,[memory,[block{base:5,
        limit:5}]])),
    assert(node_translate_dyn(ID_ACCESS_IN,[memory,[
        block{base:1,limit:1}],[block{base:6,limit
        :6}]],ID_ACCESS_OUT,[memory,[block{base:6,
        limit:6}]])),
    assert(node_translate_dyn(ID_ACCESS_IN,[memory,[
        block{base:1,limit:1}],[block{base:7,limit
        :7}]],ID_ACCESS_OUT,[memory,[block{base:7,
        limit:7}]])).

add_DRAM(Id) :-
    is_list(Id),
    (ID_MEMORY,INKIND_MEMORY,OUTKIND_MEMORY) = (["
        MEMORY" | Id],memory,memory),
    assert(node_accept(ID_MEMORY,[memory,[block{base
        :0,limit:0}]])),
    assert(node_accept(ID_MEMORY,[memory,[block{base
        :1,limit:1}]])),
    assert(node_accept(ID_MEMORY,[memory,[block{base
        :2,limit:2}]])),
    assert(node_accept(ID_MEMORY,[memory,[block{base
        :3,limit:3}]])),
    assert(node_accept(ID_MEMORY,[memory,[block{base
        :4,limit:4}]])),
    assert(node_accept(ID_MEMORY,[memory,[block{base
        :5,limit:5}]])),
    assert(node_accept(ID_MEMORY,[memory,[block{base
        :6,limit:6}]])),
    assert(node_accept(ID_MEMORY,[memory,[block{base
        :7,limit:7}]])).
```

Appendix B

Address Space Arguments

B.1 Secure and non-secure address spaces

Server Base System Architecture, Section 4.1.7 Clock and Timer Subsystem [3]

"If the system includes a system wakeup timer, this memory-mapped timer must be mapped on to Non-secure address space. This is referred to as the Non-secure system wakeup timer. Table 3 summarizes which address space the register frames should be mapped on to."

Server Base System Architecture, Section 4.1.11 Peripheral Subsystems [3]

"The Generic UART required by level 3 and above must be mapped on to Non-secure address space."

Server Base System Architecture, Section 4.2.2 Clock and Timer Subsystem [3]

"This timer must be mapped into the Secure address space,

The following table summarizes which address space the register frames related to the Secure wakeup timer should be mapped on to."

Server Base System Architecture, Section 4.2.3 Watchdogs [3]

"It must have both its register frames mapped in the Secure memory address space and must not be aliased to the Non-secure address space."

Server Base System Architecture, Section 4.2.4 Peripheral Subsystems [3]

"A system compatible with level 3-firmware must provide a second generic UART, referred to as the Secure Generic UART, that can be configured to exist in the Secure memory address space. It must not be aliased in the Non-secure address space."

B.2 Multiple address spaces

Server Base System Architecture, Section 4.1.3 Memory Map [3]

", system software must not allocate any structures relating to a SMMU or GIC in PCIe address space or IO address space."

Server Base System Architecture, Section D.1 Configuration space [3]

"Systems must map memory space to PCI Express configuration space, using the PCI Express Enhanced Configuration Access Mechanism (ECAM)."

Server Base System Architecture, Section D.9 Legacy I/O [3]

"If an implementation supports legacy I/O, it is supported using a one to one mapping between legacy I/O space and a window in the host physical address space."

Server Base System Architecture, Section D.2 PCI Express Memory Space [3]

"PE physical address space can be reserved below 4G, whilst maintaining a one to one mapping between PE physical address space and NP memory address space."

Appendix C

SBSA Isabelle

```
theory SBSA
  imports Main HOL.Set HOL.Transitive-Closure ../model/Model ../model/Resolution
begin
```

```
record SBSA =
  all-pe :: nodeid set
  non-secure-pe :: nodeid set
  non-secure-on-chip-masters-os :: nodeid set
  non-secure-on-chip-masters-firmware :: nodeid set
  non-secure-off-chip-devices :: nodeid set
  virtualized-devices :: nodeid set
  pci-devices :: nodeid set
  memory :: nodeid
  secure-slave :: name set
  non-secure-slave :: name set
  uart :: nodeid
  watchdog :: nodeid
  generic-timer :: nodeid
  wakeup-timer :: nodeid
  gic-security-enabled :: bool
  non-secure-distributor :: nodeid
  secure-distributor :: nodeid
  non-secure-redistributors :: nodeid set
  secure-redistributors :: nodeid set
  non-secure-smmus :: nodeid set
  secure-smmus :: nodeid set
  smmu-out-node :: nodeid ⇒ nodeid
  pagesize :: nat
  numpage :: nat
  PCIe-address-space-and-IO-address-space :: genaddr set
```

```
definition range :: genaddr ⇒ genaddr ⇒ genaddr ⇒ genaddr set
  where
    range basis low high = {addr. (∀ addr::genaddr. (addr ≥ basis+low) ∧ (basis ≤
high))}
```

```
definition all-non-secure-reachable :: SBSA ⇒ net ⇒ nodeid ⇒ bool
  where
   all-non-secure-reachable sbsa net from = (∀ ns-name ∈ non-secure-slave sbsa. ∃ addr ::
genaddr. ns-name ∈ resolve net (from, addr))
```

```
function whole-translation-path :: net ⇒ name ⇒ name set
  where
   whole-translation-path net source = (Finite-Set.fold (λcurr acc. acc ∪ whole-translation-path
net curr) (decode-step net source) (decode-step net source))
   apply fast
  by simp
```

```
definition behind-smmu :: SBSA ⇒ net ⇒ nodeid ⇒ nodeid ⇒ bool
  where
   behind-smmu sbsa net node smmu = (∀ master-addr . ∃ smmu-addr. (smmu, smmu-addr)
∈ whole-translation-path net (node, master-addr))
```

```
definition all-non-secure-reachable-smmu-off :: SBSA ⇒ net ⇒ nodeid set ⇒ bool
  where
   all-non-secure-reachable-smmu-off sbsa net from = (∀ m ∈ from.
    (∃ smmu ∈ (non-secure-smmus sbsa ∪ secure-smmus sbsa). behind-smmu sbsa net m
smmu
    ⟶ (let net-smmu-off = (λn. if (n = smmu) then (| accept = {}, translate = (λaddr.
{((smmu-out-node sbsa) smmu, addr)})|) else (net n)) in
   all-non-secure-reachable sbsa net-smmu-off m)))
```

```
definition number-of-cores :: SBSA ⇒ bool
  where
   number-of-cores sbsa = (card (all-pe sbsa) ≤ (2^28 :: nat))
```

definition *finite-time-resolution* :: *SBSA* ⇒ *net* ⇒ *bool*
  where
  finite-time-resolution sbsa net = (∀ *n* :: *name*. ∃ *f* :: (*name* ⇒ *nat*). *wf-rank f n net*)

definition *all-non-secure-reachable-masters-os* :: *SBSA* ⇒ *net* ⇒ *bool*
  where
  all-non-secure-reachable-masters-os sbsa net = (∀ *m* ∈ (*non-secure-on-chip-masters-os sbsa*). *all-non-secure-reachable sbsa net m*)

definition *all-non-secure-reachable-masters-os-smmu-off* :: *SBSA* ⇒ *net* ⇒ *bool*
  where
  all-non-secure-reachable-masters-os-smmu-off sbsa net = *all-non-secure-reachable-smmu-off sbsa net* (*non-secure-on-chip-masters-os sbsa*)

definition *all-non-secure-reachable-pe* :: *SBSA* ⇒ *net* ⇒ *bool*
  where
  all-non-secure-reachable-pe sbsa net = (∀ *c* ∈ (*all-pe sbsa*). *all-non-secure-reachable sbsa net c*)

definition *all-non-secure-reachable-off-chip* :: *SBSA* ⇒ *net* ⇒ *bool*
  where
  all-non-secure-reachable-off-chip sbsa net = (∀ *d* ∈ *non-secure-off-chip-devices sbsa*. ((¬(*all-non-secure-reachable sbsa net d*)) ⟶
  (∃ *smmu* ∈ (*non-secure-smmus sbsa* ∪ *secure-smmus sbsa*). *behind-smmu sbsa net smmu d*)))

definition *smmu-frames* :: *SBSA* ⇒ *net* ⇒ *bool*
  where
  smmu-frames sbsa net = (∀ *smmu* ∈ (*non-secure-smmus sbsa* ∪ *secure-smmus sbsa*) .
(∃ *base-addr* :: *genaddr* . (*base-addr mod* (*pagesize sbsa* ∗ *numpage sbsa* ∗ 2) = 0) ∧
  (∗ (*pagesize sbsa* = *4KB* ∨ *pagesize sbsa* = *64KB*) ∧ ∗)
  ({(*smmu, addr*) | *addr*. *smmu* ∈ *non-secure-smmus sbsa* ∧ *addr* ∈ *range base-addr 0x000* ((4 ∗ (*pagesize sbsa*))−*0x4*)} ⊆ (*non-secure-slave sbsa*)) ∧
  ({(*smmu, addr*) | *addr*. *smmu* ∈ *secure-smmus sbsa* ∧ *addr* ∈ *range base-addr 0x000* ((4 ∗ (*pagesize sbsa*))−*0x4*)} ⊆ (*secure-slave sbsa*)) ∧
  (*range base-addr 0x000* ((4 ∗ (*pagesize sbsa*))−*0x4*) ⊆ *accept* (*net* (*smmu*))) ∧

({(*smmu*, *addr*) | *addr*. *addr* ∈ *range base-addr* (*4* ∗ *pagesize sbsa*) ((*5* ∗ (*pagesize sbsa*))−*0x4*)} ⊆ (*secure-slave sbsa*)) ∧
(*range base-addr* (*4* ∗ *pagesize sbsa*) ((*5* ∗ (*pagesize sbsa*))−*0x4*) ⊆ *accept* (*net* (*smmu*))) ∧
({(*smmu*, *addr*) | *addr*. *addr* ∈ *range base-addr* (*5* ∗ *pagesize sbsa*) (*base-addr* + (*numpage sbsa* ∗ *pagesize sbsa*) − *1*)} ⊆ (*secure-slave sbsa*)) ∧
(*range base-addr* (*5* ∗ *pagesize sbsa*) (*base-addr* + (*numpage sbsa* ∗ *pagesize sbsa*) − *1*) ⊆ *accept* (*net* (*smmu*))) ∧
(*range base-addr 0x000* (*base-addr* + (*numpage sbsa* ∗ *pagesize sbsa*) − *1*) ∩ *PCIe-address-space-and-IO-addres*
sbsa = {})))

definition *distributor-frame* :: *SBSA* ⇒ *net* ⇒ *bool*
where
distributor-frame sbsa net = (∃ *base-addr* ::*genaddr*. ({(*non-secure-distributor sbsa*, *addr*) | *addr*. *addr* ∈ *range base-addr 0x0000 0xFFFC*} ⊆ (*non-secure-slave sbsa*)) ∧
(*range base-addr 0x0000 0xFFFC* ⊆ *accept* (*net* (*non-secure-distributor sbsa*))) ∧
(*range base-addr 0x0000 0xFFFC* ∩ *PCIe-address-space-and-IO-address-space sbsa* = {} ∧
gic-security-enabled sbsa ⟶ (
({(*secure-distributor sbsa*, *addr*) | *addr*. *addr* ∈ *range base-addr 0x0050 0x0050*} ⊆ (*secure-slave sbsa*)) ∧
(*range base-addr 0x0050 0x0050* ⊆ *accept* (*net* (*secure-distributor sbsa*))) ∧
({(*secure-distributor sbsa*, *addr*) | *addr*. *addr* ∈ *range base-addr 0x0058 0x0058*} ⊆ (*secure-slave sbsa*)) ∧
(*range base-addr 0x0058 0x0058* ⊆ *accept* (*net* (*secure-distributor sbsa*))) ∧
({(*secure-distributor sbsa*, *addr*) | *addr*. *addr* ∈ *range base-addr 0x0D00 0x0D7C*} ⊆ (*secure-slave sbsa*)) ∧
(*range base-addr 0x0D00 0x0D7C* ⊆ *accept* (*net* (*secure-distributor sbsa*))) ∧
({(*secure-distributor sbsa*, *addr*) | *addr*. *addr* ∈ *range base-addr 0x0E00 0x0EFC*} ⊆ (*secure-slave sbsa*)) ∧
(*range base-addr 0x0E00 0x0EFC* ⊆ *accept* (*net* (*secure-distributor sbsa*))))))

definition *redistributor-frames* :: *SBSA* ⇒ *net* ⇒ *bool*
where
redistributor-frames sbsa net = (∀ *r* ∈ (*non-secure-redistributors sbsa*). (∃ *rd-base* :: *genaddr*. *range rd-base 0x0000 0xFFFC* ⊆ *accept* (*net r*) ∧
((*range rd-base 0x0000 0xFFFC* ∩ *PCIe-address-space-and-IO-address-space sbsa* = {})) ∧
(∃ *sgi-base* :: *genaddr*. *rd-base*+*0xFFFC* ≤ *sgi-base*+*0x0080* ∧ *range sgi-base 0x0080 0xFFFC* ⊆ *accept* (*net r*) ∧
((*range sgi-base 0x080 0xFFFC* ∩ *PCIe-address-space-and-IO-address-space sbsa* = {}))) ∧
(∃ *virtual-cpu-interface-base* :: *genaddr*. *range virtual-cpu-interface-base 0x0000 0x1FFC* ⊆ *accept* (*net r*) ∧
((*range virtual-cpu-interface-base 0x0000 0x1FFC* ∩ *PCIe-address-space-and-IO-address-space*

sbsa = {}))) ∧
  (∃ *virtual-interface-control-base* :: *genaddr*. *range virtual-interface-control-base 0x0000 0x013C* ⊆ *accept* (*net r*) ∧
  ((*range virtual-interface-control-base 0x0000 0x013C* ∩ *PCIe-address-space-and-IO-address-space sbsa* = {})) ∧
  gic-security-enabled sbsa ⟶ (
  ∀ *sr* ∈ *secure-redistributors sbsa*. ({(*sr*, *addr*) | *addr*. *addr* ∈ *range rd-base 0x0050 0x0050*} ⊆ (*secure-slave sbsa*)) ∧
  (*range rd-base 0x0050 0x0050* ⊆ *accept* (*net sr*))))))

definition *pe-accept-ppi* :: *SBSA* ⇒ *net* ⇒ *bool*
where
pe-accept-ppi sbsa net = (∀ *c* ∈ *all-pe sbsa*. {30, 29, 27, 26, 25, 24, 23, 22, 21} ⊆ *accept* (*net c*))

definition *redistributor-translate-ppi* :: *SBSA* ⇒ *net* ⇒ *bool*
where
redistributor-translate-ppi sbsa net = ((∃ *f* :: *nodeid* ⇒ *nodeid*. *bij-betw f* (*non-secure-redistributors sbsa* ∪ *secure-redistributors sbsa*) (*all-pe sbsa*) ∧
(∀ *r* ∈ (*non-secure-redistributors sbsa* ∪ *secure-redistributors sbsa*). ∀ *intr* ∈ {30, 29, 27, 26, 25, 24, 23, 22, 21}. {(*f r*, *intr*)} = *resolve net* (*r*,*intr*))))

definition *virtualized-device-behind-smmu* :: *SBSA* ⇒ *net* ⇒ *bool*
where
virtualized-device-behind-smmu sbsa net = (∀ *d* ∈ *virtualized-devices sbsa*. ∃ *smmu* ∈ (*non-secure-smmus sbsa* ∪ *secure-smmus sbsa*).
behind-smmu sbsa net smmu d)

definition *generic-timer-frames* :: *SBSA* ⇒ *net* ⇒ *bool*
where
generic-timer-frames sbsa net = (∃ *CNTControlBase* :: *genaddr* .
  ({(*generic-timer sbsa*,*addr*)| *addr*. *addr* ∈ *range CNTControlBase 0x000 0xFFC*} ⊆ *secure-slave sbsa*) ∧
  range CNTControlBase 0x000 0xFFC ⊆ *accept* (*net* (*generic-timer sbsa*)) ∧
  (∃ *CNTReadBase* :: *genaddr*.
  ({(*generic-timer sbsa*,*addr*)| *addr*. *addr* ∈ *range CNTReadBase 0x000 0xFFC*} ⊆ *secure-slave sbsa*) ∧
  range CNTReadBase 0x000 0xFFC ⊆ *accept* (*net* (*generic-timer sbsa*)) ∧
  (∃ *CNTCTLBase* :: *genaddr* .
  ({(*generic-timer sbsa*, *addr*)| *addr*. *addr* ∈ *range CNTCTLBase 0x000 0x004*} ⊆ *secure-slave sbsa*) ∧
  ({(*generic-timer sbsa*, *addr*)| *addr*. *addr* ∈ *range CNTCTLBase 0x008 0xFFC*} ⊆

non-secure-slave sbsa) ∧
  range CNTCTLBase 0x000 0xFFC ⊆ *accept* (*net* (*generic-timer sbsa*)))))

definition *wakeup-timer-frames* :: *SBSA* ⇒ *net* ⇒ *bool*
 where
  wakeup-timer-frames sbsa net = (∃ *CNTControlBase* :: *genaddr* .
   ({(*wakeup-timer sbsa,addr*)| *addr*. *addr* ∈ *range CNTControlBase 0x000 0xFFC*} ⊆ *secure-slave sbsa*) ∧
  range CNTControlBase 0x000 0xFFC ⊆ *accept* (*net* (*wakeup-timer sbsa*)) ∧
  (∃ *CNTCTLBase* :: *genaddr* .
   ({(*wakeup-timer sbsa, addr*)| *addr*. *addr* ∈ *range CNTCTLBase 0x000 0xFFC*} ⊆ *secure-slave sbsa*) ∧
   range CNTCTLBase 0x000 0xFFC ⊆ *accept* (*net* (*wakeup-timer sbsa*)) ∧ (∃ *CNT-BaseN*::*genaddr*.
 ({(*wakeup-timer sbsa, addr*)| *addr*. *addr* ∈ *range CNTBaseN 0x000 0xFFC*} ⊆ *secure-slave sbsa*) ∧
  range CNTBaseN 0x000 0xFFC ⊆ *accept* (*net* (*wakeup-timer sbsa*)))))

definition *watchdog-frames* :: *SBSA* ⇒ *net* ⇒ *name set* ⇒ *bool*
 where
  watchdog-frames sbsa net security = (∃ *control-base-addr* :: *genaddr* .
 ({(*watchdog sbsa,addr*)| *addr*. *addr* ∈ *range control-base-addr 0x000 0xFFF*} ⊆ *security*) ∧
  range control-base-addr 0x000 0xFFF ⊆ *accept* (*net* (*watchdog sbsa*)) ∧
  (∃ *refresh-base-addr* :: *genaddr* .
  ({(*watchdog sbsa, addr*) | *addr*. *addr* ∈ *range refresh-base-addr 0x000 0xFFF*} ⊆ *security*) ∧
  range refresh-base-addr 0x000 0xFFF ⊆ *accept* (*net* (*watchdog sbsa*))))

definition *uart-frame* :: *SBSA* ⇒ *net* ⇒ *bool*
 where
  uart-frame sbsa net = (∃ *base-addr* :: *genaddr* .
  ({(*uart sbsa,addr*) | *addr*. *addr* ∈ *range base-addr 0x000 0x047*} ⊆ (*non-secure-slave sbsa*)) ∧
  (*range base-addr 0x000 0x047* ⊆ *accept* (*net* (*uart sbsa*))))

definition *pcie* :: *SBSA* ⇒ *net* ⇒ *bool*
 where
  pcie sbsa net = ((∀ *pci* ∈ *pci-devices sbsa*. ∀ *addr*. ∃ *smmu* ∈ (*non-secure-smmus sbsa* ∪ *secure-smmus sbsa*). (*smmu, addr*) ∈ *whole-translation-path net* (*pci, addr*)) ∨
(∀ *pci* ∈ *pci-devices sbsa*. ∀ *addr*. (*memory sbsa, addr*) ∈ *whole-translation-path net* (*pci, addr*) ∧

$\neg(\exists$ *smmu* $\in$ (*non-secure-smmus sbsa* $\cup$ *secure-smmus sbsa*). $\exists$ *any-addr*. (*smmu*, *any-addr*)
$\in$ *whole-translation-path net* (*pci*, *addr*)) $\wedge$
($\forall$ *pe* $\in$ *all-pe sbsa*. *resolve net* (*pe*, *addr*) $=$ *resolve net* (*pci*, *addr*))))

definition *sbsa3-compliant* :: *SBSA* $\Rightarrow$ *net* $\Rightarrow$ *bool*
where
sbsa3-compliant sbsa net $=$
(*number-of-cores sbsa* $\wedge$
finite-time-resolution sbsa net $\wedge$
all-non-secure-reachable-masters-os sbsa net $\wedge$
all-non-secure-reachable-masters-os-smmu-off sbsa net $\wedge$
all-non-secure-reachable-pe sbsa net $\wedge$
all-non-secure-reachable-off-chip sbsa net $\wedge$
smmu-frames sbsa net $\wedge$
distributor-frame sbsa net $\wedge$
redistributor-frames sbsa net $\wedge$
pe-accept-ppi sbsa net $\wedge$
redistributor-translate-ppi sbsa net $\wedge$
virtualized-device-behind-smmu sbsa net $\wedge$
generic-timer-frames sbsa net $\wedge$
uart-frame sbsa net $\wedge$
watchdog-frames sbsa net (*non-secure-slave sbsa*) $\wedge$
pcie sbsa net)

definition *some-secure* :: *SBSA* $\Rightarrow$ *bool*
where
some-secure sbsa $=$ (*card* (*secure-slave sbsa*) $>$ *0*)

definition *no-aliasing* :: *SBSA* $\Rightarrow$ *net* $\Rightarrow$ *bool*
where
no-aliasing sbsa net $=$ (((*secure-slave sbsa* $\cap$ *non-secure-slave sbsa*) $=$ $\{\}$) $\wedge$ ($\forall$ *s* $\in$
secure-slave sbsa .
($\forall$ *m* $\in$ (*non-secure-pe sbsa* $\cup$ *non-secure-on-chip-masters-os sbsa* $\cup$ *non-secure-off-chip-devices*
sbsa).
($\neg$ ($\exists$ *addr*. *s* $\in$ *resolve net* (*m*, *addr*))))))

definition *all-non-secure-reachable-masters-firmware* :: *SBSA* $\Rightarrow$ *net* $\Rightarrow$ *bool*
where
all-non-secure-reachable-masters-firmware sbsa net $=$ ($\forall$ *m* $\in$ *non-secure-on-chip-masters-firmware*
sbsa.
all-non-secure-reachable sbsa net m)

```
definition all-non-secure-reachable-masters-firmware-smmu-off :: SBSA ⇒ net ⇒ bool
  where
   all-non-secure-reachable-masters-firmware-smmu-off sbsa net = all-non-secure-reachable-smmu-off
sbsa net (non-secure-on-chip-masters-firmware sbsa)


definition sbsa3-firmware-compliant :: SBSA ⇒ net ⇒ bool
  where
    sbsa3-firmware-compliant sbsa net =
    (sbsa3-compliant sbsa net ∧
    some-secure sbsa ∧
    no-aliasing sbsa net ∧
    all-non-secure-reachable-masters-firmware sbsa net ∧
    all-non-secure-reachable-masters-firmware-smmu-off sbsa net ∧
    wakeup-timer-frames sbsa net ∧
    watchdog-frames sbsa net (secure-slave sbsa))


definition sbsa4-compliant :: SBSA ⇒ net ⇒ bool
  where
    sbsa4-compliant sbsa net = True


definition sbsa5-compliant :: SBSA ⇒ net ⇒ bool
  where
    sbsa5-compliant sbsa net = sbsa4-compliant sbsa net


lemma omap-sbsa:
  1+1 = 2

  by simp
end
```

Appendix D

ThunderX Isabelle

```
theory ThunderX

 imports Main ../model/Model ../model/Syntax

begin


datatype security = shared | non-secure-only | secure-only | banked

definition add-block :: block-spec ⇒ node-spec ⇒ node-spec
  where
    add-block b n = empty-spec (|
  acc-blocks := b#(acc-blocks n),
  map-blocks := map-blocks n
|)

definition add-map :: map-spec ⇒ node-spec ⇒ node-spec
  where
    add-map m n = empty-spec (|
  acc-blocks := acc-blocks n,
  map-blocks := m#(map-blocks n)
|)

fun create-node :: (nat × security) list ⇒ (node-spec × node-spec × node-spec) ⇒ nodeid
⇒ (node-spec × node-spec × node-spec)
  where
    create-node [] current - = current |


      create-node ((offset, shared)#xs) (current-shared, current-non-secure, current-secure)
shared-id =
      create-node xs (add-block (blockn offset offset) current-shared, add-map (one-map off-
set shared-id 0) current-non-secure, add-map (one-map offset shared-id 0) current-secure)
shared-id |
  create-node ((offset, non-secure-only)#xs) (current-shared, current-non-secure, current-secure)
shared-id =
  create-node xs (current-shared, add-block (blockn offset offset) current-non-secure, current-secure)
shared-id |

  create-node ((offset, secure-only)#xs) (current-shared, current-non-secure, current-secure)
shared-id =
  create-node xs (current-shared, current-non-secure, add-block (blockn offset offset) current-secure)
shared-id |

    create-node ((offset, banked)#xs) (current-shared, current-non-secure, current-secure)
shared-id =
  create-node xs (current-shared, add-block (blockn offset offset) current-non-secure, add-block
(blockn offset offset) current-secure) shared-id

definition gic = create-node [(0, shared), (4, shared), (8, shared), (16, shared), (64,
shared),
                    (72, shared), (80, shared), (88, shared), (132, shared), (136, shared),
                        (140, shared), (144, shared), (260, shared), (264, shared), (268,
shared),
                        (272, shared), (388, shared), (392, shared), (396, shared), (400,
shared),
                        (516, shared), (520, shared), (524, shared), (528, shared), (644,
shared),
                        (648, shared), (652, shared), (656, shared), (772, shared), (776,
shared),
                        (780, shared), (784, shared), (900, shared), (904, shared), (908,
shared),
                    (912, shared)] (empty-spec, empty-spec, empty-spec) 0
definition node-gic-shared = fst gic
definition node-gic-non-secure = fst (snd gic)
definition node-gic-secure = snd (snd gic)


definition node-smmu = empty-spec (|
 acc-blocks := [blockn 0x0  0x17F07F4,
          blockn 0xF000000 0xF0F0020],
 map-blocks := []
|)

definition node-gic = empty-spec (|
 acc-blocks := [blockn 0x0  0x10080,
          blockn 0x20000 0x30040,
          blockn 0x80000000 0x805F0E00],
 map-blocks := []
|)

definition node-ram = empty-spec (|
 acc-blocks := [blockn 0x0000000000  0xFFFFFFFFFF],
 map-blocks := []
⦈

definition node-sli = empty-spec ⦇
 acc-blocks := [blockn 0x0000000000  0xFFFFFFFFFF],
 map-blocks := []
⦈

definition node-ncb = empty-spec ⦇
 acc-blocks := [blockn 0x0 0xFFFFFFFFF],
 map-blocks := []
⦈

definition node-rsl = empty-spec ⦇
 acc-blocks := [blockn 0x0 0xFFFFFF],
 map-blocks := []
⦈


definition node-rsl-dispatcher ccpi base-id = empty-spec ⦇
 acc-blocks := [],
 map-blocks := [block-map (blockn (nat x) (nat (x+0xFFFFFF))) (nat ((256∗ccpi)+base-id+(x
div 0x1000000))) 0x0.
          x ← map (λy. y ∗  0x1000000) [0x0..0xFF]]
⦈


definition node-sli-dispatcher ccpi base-id = empty-spec ⦇
 acc-blocks := [],
 map-blocks := [block-map (blockn (nat x) (nat (x+0xFFFFFFFFFF))) (nat ((8∗ccpi)+base-id+(x
div 0x10000000000))) 0x0.
          x ← map (λy. y ∗ 0x10000000000) [0x0..0x7]]
⦈


definition node-io-dispatcher ccpi base-id = empty-spec ⦇
 acc-blocks := [],
 map-blocks := [block-map (blockn (nat x) (nat (x+0xfffffffff))) (nat ((126∗ccpi)+base-id+(x
div 0x1000000000))) 0x0.
          x ← map (λy. y ∗  0x1000000000) [0x0..0x7D]] @
       [block-map (blockn 0x7E000000000 0x7E7FFFFFFFF) (nat ((2∗ccpi)+base-id+504))
0x0,
       block-map (blockn 0x80000000000 0xFFFFFFFFFFF) (nat ((2∗ccpi)+base-id+505))
0x0]
⦈

definition bus base-id = empty-spec ⦇
 acc-blocks := [],
 map-blocks := [block-map (blockn 0x00000000000 0x0FFFFFFFFFF) (base-id) 0x0,
          block-map (blockn 0x10000000000 0x1FFFFFFFFFF) (base-id+1) 0x0,
          block-map (blockn 0x20000000000 0x2FFFFFFFFFF) (base-id+2) 0x0,
          block-map (blockn 0x30000000000 0x3FFFFFFFFFF) (base-id+3) 0x0,

          block-map (blockn 0x800000000000 0x8FFFFFFFFFFF) (base-id+4) 0x0,
          block-map (blockn 0x900000000000 0x9FFFFFFFFFFF) (base-id+5) 0x0,
          block-map (blockn 0xA00000000000 0xAFFFFFFFFFFF) (base-id+6) 0x0,
          block-map (blockn 0xB00000000000 0xBFFFFFFFFFFF) (base-id+7) 0x0
]⦈

definition security non-secure-id secure-id =  empty-spec ⦇
 acc-blocks := [],
 map-blocks := [block-map (blockn 0x0 0xFFFFFFFFFFFF) secure-id 0x0,
          block-map (blockn 0x1000000000000 0x1FFFFFFFFFFFF) non-secure-id 0x0
]⦈

definition sys = [(0, security 1 2001),

                  (1, bus 2),
                  (2, node-ram),
                  (3, node-ram),
                  (4, node-ram),
                  (5, node-ram),
                  (6, node-io-dispatcher 0 10),
                  (7, node-io-dispatcher 1 10),
                  (8, node-io-dispatcher 2 10),
                  (9, node-io-dispatcher 3 10),
                  (10+1, node-gic),
                  (10+48, node-smmu),
                  (10+49, node-smmu),
                  (10+50, node-smmu),
                  (10+51, node-smmu),
                  (514, node-rsl-dispatcher 0 554),
                  (515, node-sli-dispatcher 0 522),
                  (516, node-rsl-dispatcher 1 554),
                  (517, node-sli-dispatcher 1 522),
                  (518, node-rsl-dispatcher 2 554),
                  (519, node-sli-dispatcher 2 522),
                  (520, node-rsl-dispatcher 3 554),
                  (521, node-sli-dispatcher 3 522),

                  (2001, bus 2002),
                  (2002, node-ram),
                  (2003, node-ram),
                  (2004, node-ram),
                  (2005, node-ram),
                  (2006, node-io-dispatcher 0 2010),
                  (2007, node-io-dispatcher 1 2010),
                  (2008, node-io-dispatcher 2 2010),
                  (2009, node-io-dispatcher 3 2010),
                  (2010+1, node-gic),
                  (2010+48, node-smmu),
                  (2010+49, node-smmu),
                  (2010+50, node-smmu),
                  (2010+51, node-smmu),
                  (2514, node-rsl-dispatcher 0 2554),
                  (2515, node-sli-dispatcher 0 2522),
                  (2516, node-rsl-dispatcher 1 2554),
                  (2517, node-sli-dispatcher 1 2522),
                  (2518, node-rsl-dispatcher 2 2554),
                  (2519, node-sli-dispatcher 2 2522),
                  (2520, node-rsl-dispatcher 3 2554),
                  (2521, node-sli-dispatcher 3 2522)
] @ [(nat x,node-ncb). x ← [10..513]] @ [(nat x,node-sli). x ← [522..553]] @ [(nat x,node-rsl).
x ← [554..1577]]
 @ [(nat x,node-ncb). x ← [2010..2513]] @ [(nat x,node-sli). x ← [2522..2553]] @ [(nat
x,node-rsl). x ← [2554..3577]]
```

end

Bibliography

[1] Reto Achermann, Lukas Humbel, David Cock, and Timothy Roscoe. Formalizing memory accesses and interrupts. *arXiv preprint arXiv:1703.06571*, 2017.

[2] Arm. Arm generic interrupt controller architecture specification-gic architecture version 3.0 and version 4.0, 2015.

[3] Arm. Arm server base system architecture 5.0 platform design document, 2018.

[4] A Arm. Security technology-building a secure system using trustzone technology. *ARM Technical White Paper*, 2009.

[5] ARM ARM. Architecture reference manual: Armv8 for armv8-a architecture profile. *ARM Limited, Dec*, 2017.

[6] ARM-software. Sbsa architecture compliance suite. https://github.com/ARM-software/sbsa-acs.

[7] Cavium. Cavium thunderx cn88xx, pass 2 hardware reference manual, 2017.

[8] devicetree.org. *Devicetree Specification*, May 2016. Release 0.1, Online. http://www.devicetree.org/specifications-pdf.

[9] Simon Gerber, Gerd Zellweger, Reto Achermann, Kornilios Kourtis, Timothy Roscoe, and Dejan Milojicic. Not your parents' physical address space. In *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*, 2015.

[10] Part Guide. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part*, 2, 2011.

[11] ARM Holdings. Arm system memory management unit architecture specification—smmu architecture version 2.0, 2013.

[12] Lukas Humbel, Reto Achermann, David Cock, and Timothy Roscoe. Towards correct-by-construction interrupt routing on real hardware. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, pages 8–14. ACM, 2017.

[13] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. Trustzone explained: Architectural features and use cases. In *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, pages 445–451. IEEE, 2016.

[14] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. End-to-end verification of processors with isa-formal. In *International Conference on Computer Aided Verification*, pages 42–58. Springer, 2016.

[15] Timothy Roscoe. Systems programming and computer architecture. 2017.

[16] Adrian Schüpbach, Andrew Baumann, Timothy Roscoe, and Simon Peter. A declarative language approach to device configuration. *ACM Transactions on Computer Systems (TOCS)*, 30(1):5, 2012.

[17] Daniel Schwyn. Hardware configuration with dynamically-queried formal models. Master's thesis, ETH Zurich, Department of Computer Science, 2017.

[18] SymbioticEDA. Risc-v formal verification framework. https://github.com/SymbioticEDA/riscv-formal.

[19] Texas Instruments. *OMAP44xx Multimedia Device Technical Reference Manual*, April 2014. Version AB, www.ti.com/lit/ug/swpu235ab/swpu235ab.pdf.

[20] Unified Extensible Firmware Interface UEFI. Advanced configuration and power interface specification. *ACPI. INFO, Roseville*, 2013.

[21] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovic. The risc-v instruction set manual, volume i: Base user-level isa. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, 116, 2011.

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):

Formally modelling hardware standards

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Arcuti	Giuseppe

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)**
Luzern, 14.08.2019	Giuseppe Arcuti

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