The HASPOC High Assurance Platform
- An Overview -

Executive Summary

This document provides a high level overview of the HASPOC High Assurance Platform. This is a virtualization platform (a hypervisor supported by a secure boot component) for the ARMv8 CPU architecture with formally verifiable information isolation properties. The assurance in the platform is further supported by prepared Common Criteria documentation targeting EAL6.

Table of Content

1 Introduction

To save cost and increase usability there is an increased demand on reusing the same ICT hardware for multiple purposes, simultaneously handling applications, services and information with different requirements. A fundamental security requirement is to provide isolation between the different services and their associated information. This is a well-known problem area which first arose in the dawn of ICT usage within the military/defense sector and led to much research around so called multi-level security and access control: allowing many users with different security clearance to handle plural types of information with different security classification (restricted, secret) in the same computer system, while obtaining high assurance that information could not be accessed in non-authorized ways, see e.g. [BL]. Gradually, this problem has also become highly relevant in the private sector with issues around mixing personal and enterprise information in the same user device (e.g. a laptop), cloud computing (allowing tenants to share pooled resources), etc. At the same time as the problem has become relevant for all sectors (private, public, defense), the problem has also become relevant for most types of ICT devices, not only for mainframes in data centers, but also for smart phones and computationally very limited devices such as embedded systems and sensors.

For the purpose of allowing resource sharing, a commonly used component today is a so called *hypervisor*. Hypervisors are software components which, to a varying degree of transparency, virtualizes the underlying hardware (CPU, memory, etc.), allowing cross-architecture execution and allowing several *guests* to share the resources. The guests can range in complexity from simple bare-metal implementations to complete operating systems with applications running on top. Although hypervisors must obviously always ensure some form of *separation* between the guests in order to be useful (e.g. ensuring execution correctness), the amount of *isolation* (in the sense of information flow control) between the guests can vary, both in terms of which security functions that are in place to control information flow, as well as the *assurance* in those security functions. The assurance in the hypervisor itself is, needless to say, irrelevant unless we can also provide assurance that the hypervisor has been properly instantiated (booted), e.g. that the integrity of the hypervisor code is intact and that no other "Trojan" software component acts as "man in the middle" between the hypervisor and the hardware.

The highest available assurance level is that offered by including a formal verification. Here, formal methods including manual "pen-and-paper" mathematical proofs and/or application of interactive theorem provers, provide verifiable evidence that the security objectives are met. Further, Common Criteria (CC) is a standardized framework for documentation, evaluation and certification of an ICT product's security assurance. Indeed, the highest assurance levels of CC require some form of formal verification.

This document provides a high level overview of a *high assurance platform* developed by the HASPOC project, offering such formally verified isolation properties. In addition, a so called Common Criteria Security Target has been specified which can be used as a basis for CC evaluation of products based on the HASPOC platform. Specifically, "high assurance" here translates into CC assurance level EAL6.

2 The HASPOC Platform

2.1 Overview

We first provide a very high level overview of the HASPOC platform on a more structural level, and then later dig further into the details.

The HASPOC platform is intended as basis for securely executing multiple guests on the same hardware with high isolation (confidentiality and integrity) assurance for the guests. Note that in some literature, the term *compartments* is used to refer to *isolated guests*. For the purpose of this document, all guests considered are isolated (through a hypervisor) and thus the term compartment is superfluous.

For the targeted formal verification of the isolation (verification at machine code level), it becomes necessary to rely on specifics of the underlying hardware, e.g. to verify or at least make assumptions on the correct execution of instructions by the CPU. Therefore, it is important to understand that the formal verification is connected to the use of a specific hardware architecture, here based on the ARMv8-A v8.0 specification. No formal claims are made about security properties when porting the HASPOC platform to some other hardware architecture. On the other

hand, for Common Criteria assurance, the CPU itself is considered outside the boundary of the verified component (the so called TOE) and large parts of an ST might be re-usable also for another CPU architecture.

Additionally, for the HASPOC platform, the hypervisor, guests, and their parameters are statically configured. There are no provisions made for dynamically spawning more guests, only those guests that are initially defined at boot time will be executed. Neither does the platform provide support for patching/upgrading the software. Of course, when an updated version of the platform becomes available, that version can be booted instead.

Specifically, the assumed hardware on which the HASPOC platform is running is some ICT device of the following simplified form (Fig 1).

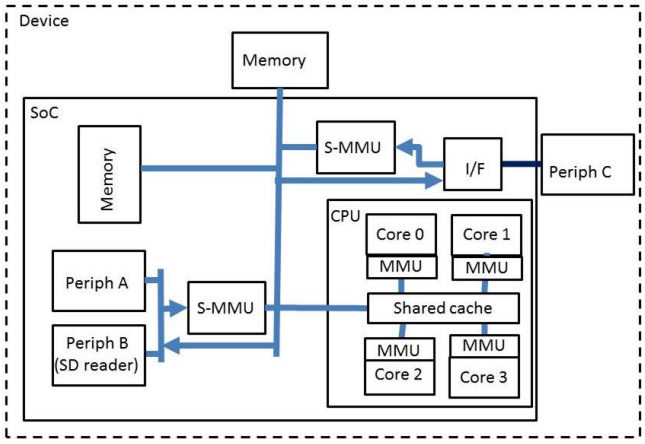


Figure 1: HASPOC Compliant System

In the sequel any reference to a *HASPOC compliant system* (or simply *compliant system*) should be understood as an instantiation of a hardware architecture as above. The exact compliance requirements will be elaborated later but one of the key requirements is that the CPU cores correctly implement the ARMv8-A specification [ARMv8]. In addition, there are also further external dependencies besides those on the hardware, e.g. on software signature creation processes.

The *HASPOC virtualization platform* (or, in simpler terms, the *HASPOC platform*) consists of the so called *HASPOC hypervisor* executing on a compliant system after having been initiated (booted) through the *HASPOC secure boot* service. The logical structure of the HASPOC platform during operation thus has the following form.

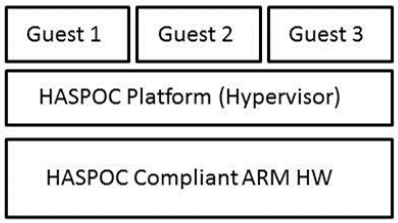


Figure 2: HASPOC platform during operation.

The HASPOC secure boot ensures that the hypervisor, the guests, and configuration data (e.g.

security policies) are loaded into memory with assurance on authenticity and integrity. The assurance is provided through cryptographic protection (signatures) on image and associated data files and formal verification of the secure boot code. (Secure boot is provided also for guests, see below.)

The HASPOC platform provides two different categories of services to guests: *built-in services and platform security services.* It may also provide *trusted guest services*.

The built-in services are those which the guest cannot (or need not) explicitly invoke at runtime, specifically the secure boot and the virtual machine services (CPU/memory/peripherals) provided. The secure boot service for guests is provided by the HASPOC secure boot service while the other two services are provided by the HASPOC hypervisor.

Platform security services are those which the guest may invoke on its own initiative. There is one platform security service: *secure communication between guests* which will be elaborated further below.

Keeping the number of platform security services limited is highly desired as a large code foot print would also imply a large attack surface and would make verification very complex. Observe that while the above small set of services may seem limiting, it is possible to rely on the HASPOC platform in order to provide other, high assurance service to guests. This can be done by implementing such services in one (or more) special *trusted guests* as discussed later, providing the aforementioned trusted guest services. Securely implementing (and possibly formally verifying) such additional trusted guest services is however outside the scope of the HASPOC platform.

As mentioned, the assurance in the isolation is anchored in formal verification. Informally, the verification establishes that different guests executing on the HASPOC platform enjoy the same isolation as if they had been executing in an *ideal model*, comprising two separate, air-gap separated systems (allowing for dedicated inter guest communication channels across the air-gap). The attacker is modelled as an arbitrary, adversarial guest on the HASPOC platform executing code with the goal to infer information about other guests' resources such as memory/register content, information exchanged with peripherals, etc, or, to affect/modify that information, by using its own available resources and accessible memory content. No claims about assurance against side channels (e.g. timing or power consumption analysis) are made.

2.2 Use-case Examples

To further, and more concretely, illuminate some possible use cases for the HASPOC platform, three exemplary use cases are given below.

2.2.1 Secure Smartphone

A Smartphone is basically a battery powered computer with great communication capabilities in a form factor that can be carried in a pocket. In the discrete world, a secure mobile communication system would contain the user device to be protected (red), an encryption device and a communication device (black).

In this example the HASPOC platform is used to create virtual instances of all devices in the secure communication system. Each device is assigned to a guest and the HASPOC platform is used to

regulate access to peripherals and to apply an information flow policy.

The User guest is given access to all peripherals (touch screen, keyboard, …) except the communication peripherals (4G and Wifi). The communication guest Com is only given access to the communication peripherals. The encryption guest Enc is forbidden access to any peripherals and has only communication channels to User and Com. This setup will enforce the red/black separation information flow between User and Com by mandating the flow to go through Enc. The picture below shows the structure.

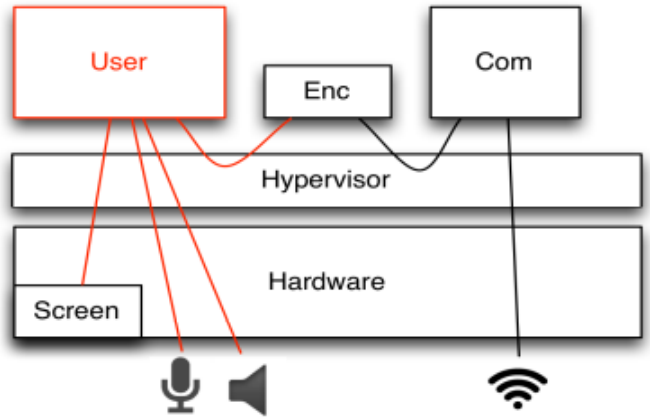


Figure 3: Secure Smartphone

2.2.2 Network Security Appliance

In this case, a form of security gateway is provided to ensure red-black separation of a trusted and an untrusted network domain. The main security policy to enforce is that data is not allowed to transfer from red to black domain unless it is first encrypted. There are thus two guests R and B for the red resp. black domain (performing some service in each domain, R may for instance comprise an untrusted Linux/Android system). R and B each exclusively own a peripheral network interface card in order to communicate with the respective domains. Exclusive ownership of the peripherals is necessary to prevent B from accessing traffic in the red domain. Furthermore, there is a guest E performing policy enforcement (i.e. encryption) on all traffic going from R to B. Finally, a specific (tamper resistant) peripheral is responsible for management of encryption keys etc. This is shown in the picture below.

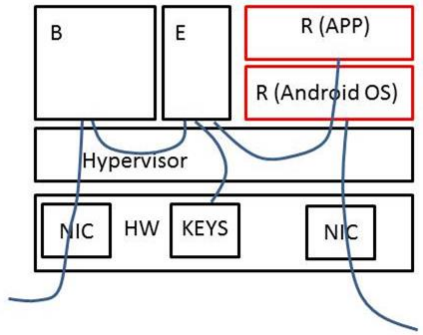


Figure 4: Network Security Appliance

2.2.3 Car OBU

In this example, the HASPOC platform is used as basis for a car on-board unit (OBU). Five guests are running on the platform, one for the entertainment system (music streaming etc), one for the car's propulsion control system, a safety control system (e.g. anti-collision) and finally a management domain. There is also one dedicated guest implementing two “Software SIM-cards” as trusted guest services, responsible for security set-up for WAN (4G) connectivity. The reason for having two “SIMs” is that (in this example) the entertainment system is associated with a mobile subscription of the car owner, whereas the other is associated with a subscription of the car manufacturer.

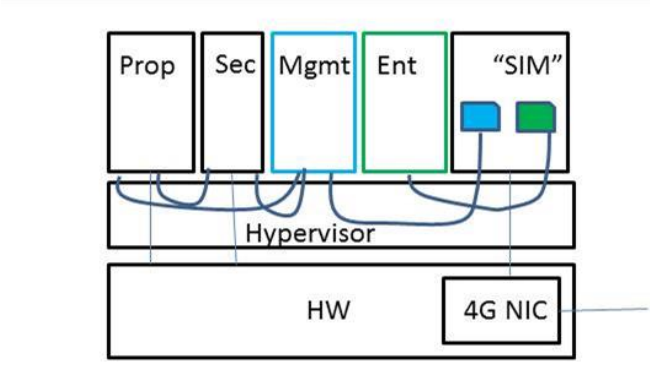


Figure 5: Car OBU

3 Terminology

The following terminology is used throughout the document (further notation is defined as needed in the sequel). As the reader may have noticed, *italic* type face is used when normative terminology is first introduced and/or defined.

architecturally mapped peripheral,	non memory-mapped peripheral, e.g. accessible via special CPU instruction
BAP,	Binary Analysis Platform, an infrastructure for static analysis and formal verification of binary code

Bisimulation,	Formal proof methodology, used to show equivalence between systems
BIL,	BAP Intermediate Language, code representation language used by BAP
BL,	Boot loader/level, a stage of the ARM boot process
Built-in service,	Mandatory (non-negotiable) services provided by the HASPOC platform
Cold boot,	Full boot process at initial power-on (BL1 to BL3)
Communication object,	Virtual resource provided by the HASPOC hypervisor for supervised communication between guests
data sharing communication object	Communication object offering transfer of arbitrary data between guests
DMA,	Direct memory access
event signalling communication object	mechanism for inter-guest signalling implemented through interrupts between cores
EL,	Exception level, a privilege level of executing ARM code
Execution trace,	Sequence of consecutive machine states, derived by the chained application of machine steps, so that the post-state of one step is the pre-state of the next step
Exclusively owned peripheral,	Peripheral which is allocated to exactly one guest
Guest,	A software component with a certain allocated subset of platform resources hosted by the hypervisor
handler,	A collective term for all functions of the hypervisor that can be invoked after the initiation phase is finished, such as: interrupt handlers, hypercall handlers, data/prefetch abort handlers. For further details see Section 7
HASPOC compliant system,	See Section 4.1
HASPOC hypervisor,	The hypervisor/virtualization software component of the HASPOC platform
HASPOC secure boot,	The secure boot software component of the HASPOC platform
HASPOC secure boot firmware,	The software implementing HASPOC secure boot
HASPOC (virtualization) platform,	The HASPOC hypervisor and HASPOC secure boot
Ideal model,	A fictional set-up consisting of guests running directly on individual, physically separated HASPOC compliant systems,

	used to model a setting with “optimal” isolation to which the HASPOC hypervisor is compared in the formal verification, see Section 7
HSBF,	HASPOC Secure Boot Format, file storage format for code to be booted (boot firmware, hypervisor, guests)
Hoare logic,	A formal system for reasoning about program correctness
HOL,	Higher Order Logic, a family of interactive theorem proving systems
Hyper property,	Property that constrains a set of execution traces and thus in general cannot be expressed as a property of a specific (single) trace alone
Isolation,	Property that that platform resources assigned to a guest (or of the HASPOC hypervisor) are not directly accessible by another guest. For guests it means that all access to resources is equivalent to those in the ideal model. In particular, results of resource usage (e.g. memory read) are equivalent and direct access to foreign memory is blocked
Memory mapped peripheral,	Peripheral accessible as part of physical memory space
Peripheral,	Separate hardware module of the SoC which may be made available to guests, e.g. graphic or network card, non-volatile storage such as SD card reader, etc
Platform resources,	CPU cores, memory, and peripherals
Platform security service,	Services offered as option to guests by the HASPOC platform (secure communication)
sequential consistency,	The intuitive view of a concurrent system where memory accesses are observed according to a global order that is consistent with the order of memory accesses in each participant's sequential programming
SMC,	Secure monitor call, invoking a TrustZone service in EL3
S-MMU,	System MMU, the ARM term for an I/O MMU, i.e., an entity on the bus that can be configured to prevent peripherals from accessing protected memory or other peripherals
SoC,	System-on-a-chip
Shared peripheral,	Peripheral which is shared between two or more guests
Side channel,	Non-intentional information flow between systems caused by physical realization/implementation of the systems
Supervised communication,	Authorized information flow, passing through a dedicated

	communication object allocated by the HASPOC hypervisor
Trusted guest,	A guest which, due to some criteria, is trusted by another guest
Trusted guest service,	Any service provided by a trusted guest to another guest
TrustZone,	An ARM extension providing an additional security mechanism by introducing a disjoint “secure memory” address space and a new exception level EL3 (a.k.a. secure monitor mode)
TSF/TSFI	Trusted Security Function/Interface (Common Criteria terms)
Unsupervised communication,	any information flow, direct or indirect, between guests that is not supervised, e.g. due to side channel
Weakly consistent memory,	the concurrent memory model provided by most modern processors, where different participants observe memory accesses in different orders due to pipelining, out-of-order execution, store buffers, cache effects, etc. In general, a sequential consistent view can only be established if certain software conditions are met, e.g. absence of self-modifying code, appropriate cache management and memory barrier usage

4 Assumptions and Requirements

As mentioned, the HASPOC platform assurance holds assuming operation on a compliant system, and, when some other external prerequisites are in place.

4.1 HASPOC Compliant System

A HASPOC compliant system follows the structure depicted in Illustration 1, with the following more specific requirements. The requirements are divided into requirement areas. For each requirement, we briefly motivate how/why this requirement is essential and give reference to other sections of the paper for further information about the requirement and its usage.

Requirement Area	**Requirement**	**Req Type***	**Rationale**	**References**
CPU Core(s)	ARMv8-Av8.0 compliance	C, S	Needed for - privilege level support - formal verification at assembly level, dependent on specific model for	Sections 5.3, 7

			CPU cores	
Memory Management	MMU supporting virtual memory based on two stage translation	C, S	Memory isolation between guests	5.3.2
	System MMU (S-MMU) support through MMU-400/401 (v1).	C, S	Required to to prevent the (DMA) accesses that peripherals might attempt to foreign guest memory or to other peripherals	5.3.3
Trust anchors	E-fuses	S	Required for trust-anchoring of secure boot	5.1.2
Peripherals	SD card reader	C	Required for boot	5.1
Other	Evaluated configuration	A	For the Common Criteria assurance, this assurance will hold for a specific evaluated configuration.	6.4.3

** Requirement type: S = Security, C = Compatibility, A = Assurance*

4.2 Other environmental assumptions

The IT environment must provide a secure function for the support of the generation of evidence that the soft- and firmware of the HASPOC platform has not been tampered with at boot time, this for the purpose of providing the secure boot as described in Section 5. A tamper free hardware for the HASPOC platform is left strictly as an assumption as will not be discussed further.

Specifically, the *HASPOC secure boot firmware* is assumed available in flash or on an external media, signed by a secure private key, and the public *boot image signature verification key* (corresponding to the secret key) is assumed available in the same location with preserved authenticity and integrity. In case of external media, it must provide tampering protection for the verification key with sufficient assurance for the operational IT environment of the HASPOC platform.

The IT environment must also provide a trusted *signature generation tool* that is able to generate signed *boot images* of the HASPOC hypervisor and the guests. Note that the boot image includes configuration parameters and security policies which must also be protected. The HASPOC hypervisor and the guest boot images may have been signed with respect to the same key, or, may be using different keys.

The boot image signature verification key(s) must have a suitable cryptographic strength.

5 Technical Description

After the high level overview, we now turn to a more in-depth description of the HASPOC platform.

As mentioned, we can distinguish between two main components, the secure boot and the hypervisor.

5.1 HASPOC Secure Boot

5.1.1 Overview and Motivation

The objective of the isolation of the HASPOC platform is to ensure certain security properties, e.g. that the (memory) states of guests are unaffected by other guests. This cannot be ensured unless some initial invariants can be established: that the authentic HASPOC hypervisor has been booted up, that the right configuration data (security enforcement policies and other initial parameters) are in place etc. The key security objective of the *HASPOC secure boot* is to provide assurance on the integrity of these initial invariants, i.e. that the hypervisor is started in a state that is considered a secure initial state in the formal verification of the hypervisor. Therefore, the secure boot code is also formally verified and included in the TOE of the CC Security Target.

The integrity is ensured in a bootstrapping fashion forming a chain-of-trust anchored in formal verification and assumptions on the IT environment.

I. Initially, the *secure boot firmware* as well as the parts of the boot image (parts of the HASPOC hypervisor) performing step V below have been formally verified.

II. We assume that the (correct) boot firmware is available to the IT environment where the HASPOC platform is to be started.

III. We also assume that correct and sufficient security processes and mechanisms have been used to cryptographically sign the (correct) boot images of the HASPOC hypervisor, guests, policies, etc, and that these signatures are available.

IV. As the HASPOC compliant system is started, the CPU will start executing the boot firmware in a number of bootstrapped stages. Each stage loads and verifies the next stage before passing on control to the next stage.

V. In the final stage, completing the secure boot, the hypervisor has been loaded, the execution level is lowered to EL2 and control is passed to the hypervisor.

VI. The hypervisor will perform additional configuration of the system and guests, e.g. (S-)MMU configurations, guest memory mappings, etc.

VII. The hypervisor will spawn the guests.

Step I (the formal verification) is described in Section 7.2. Steps II and III have many different (secure) implementations and are largely outside the scope of this paper. However, we will below discuss some of the concrete choices done by our own proof of concept implementation, see Section 6. We will also be providing some general high level guidance on these aspects in Section 6. The focus of this section is therefore on step IV, which in the sequel is simply referred to as “the (secure) boot (process)” and is treated in depth in Section 5.1.3. It consists of three steps, initial hardware configuration, dynamic memory configuration and the final boot.

Since formal verification does not yet cover the entire boot code-base, traditional testing is needed for full coverage of security functions. The modules are therefore divided into generic and specific. Generic modules could be tested extensively with unit tests.

The secure boot follows ARM Trusted Firmware (ATF) model, with relevant aspects thereof

described in Appendix A for the interested reader. The specific module which initializes hardware is named bios_low and the implementation overlaps with ATF. Memory configuration and final boot are implemented in the bios_high module.

5.1.2 Boot Environment Assumptions

In a multi-phase boot process, such as used by the HASPOC platform, the integrity of the code executing the initial phase is of course critical. As the CPU is started, it will start to execute code and load various configurations from ROM. As such, ROM provides good protection against tampering.

The initial ROM boot code needs to have a quite limited footprint to allow formal verification and needs to have as restricted capabilities as possible. In addition to formal verification, the design must also comply to Common Criteria which requires that all external security interfaces must be described in relation to all dependent modules. The design also reflects overall CC principles such as non-bypassability, domain isolation and self-protection. As a result of the design two external attack surfaces has been identified. The first attack surface (AS1) relates to reset/power off, the other attack surface (AS2) relates to images written by an attacker on the boot media. External interfaces for security functions related to the target of evaluation are listed as TSFI: according to the Common Critera.

Complex hardware requires proprietary binary images to be retrieved and executed as part of the secure boot low procedure. Details of the actual images and it functional purpose could not be exposed in this paper, since a non-disclosure agreement had to be signed. Such details are therefore only identified and listed, but verification of these components was not deemed meaningful since it would be of limited use outside our project.

As the boot process continues, the ROM boot is therefore responsible for loading the code for the next boot phase and, most importantly, to verify this code before handing over control. To do this in a flexible way, the secure boot high code is assumed digitally signed in relation to a public root key and that public root key is securely made available to the platform. The common way to protect public keys is to provide a tamper resistant fingerprint, and that is also the approach used here. In order to arrive at a reasonably secure yet flexible way to store this fingerprint, it is assumed that e-Fuses are used. Storage in the chip would again be possible but less flexible. Fingerprint provisioning is assumed done at initial deployment but due to e.g. need for revocation, this should be accompanied by a root key life-cycle-management process which is largely out of scope here. However, we briefly discuss some caveats related to revocation and in-field reprogramming in Section 6.

In any case, the root key is assumed to be used to sign the rest of the boot code. Optionally, it may also be used to sign further public keys, e.g. keys usable to sign and verify guests. In our proof of concept, we have (again for simplicity) chosen to store the fingerprint hard coded in the “ROM boot” code. We use SHA-256 to generate the fingerprint and the root public key is an RSA 2048 bit key. In the proof of concept we also use a “raw” public key, not a full X.509 certificate. This key is denoted ROOT_PK.

5.1.3 Secure Boot Process

When the target is released from reset, all core becomes active and executes the early part of Secure Boot in parallel. The primary core is selected for exclusive control, while the remaining cores being suspended. From this point hardware configuration is executed, followed by MMU/Cache configuration. The early hardware specific module exists by passing control to the high level generic portion of Secure Boot.

The Secure Boot High module retrieves and processes all object in the boot image. The processing is divided into a initial truncated read (pass one) and a final complete read of the entire boot image (pass 2). The initial pass configures dynamic memory, required to store the boot image for the subsequent final pass.

When the boot is completed, parts of the secure boot code are still involved due to power management of secondary cores.

5.1.3.1 Secure boot low

Sequence diagrams are used to illustrate interaction within and between modules in this subsystem. Instances with capital letter represent modules, lower case letters denote entities within an actual module.

The entry point for secure boot low is shown here. Before any access to memory takes place, alignment and other parameters must be assigned correct values. After interrupt have been configured, secondary cores are detected and detached from further execution of secure boot low. The C-runtime sequence then copy from the ROM resident region into static ram. After some early configuration, the MMU/Cache is configured. Static configuration of MMU combined with a non bypass implementation of the low level module reduces the attack surface to just power on.

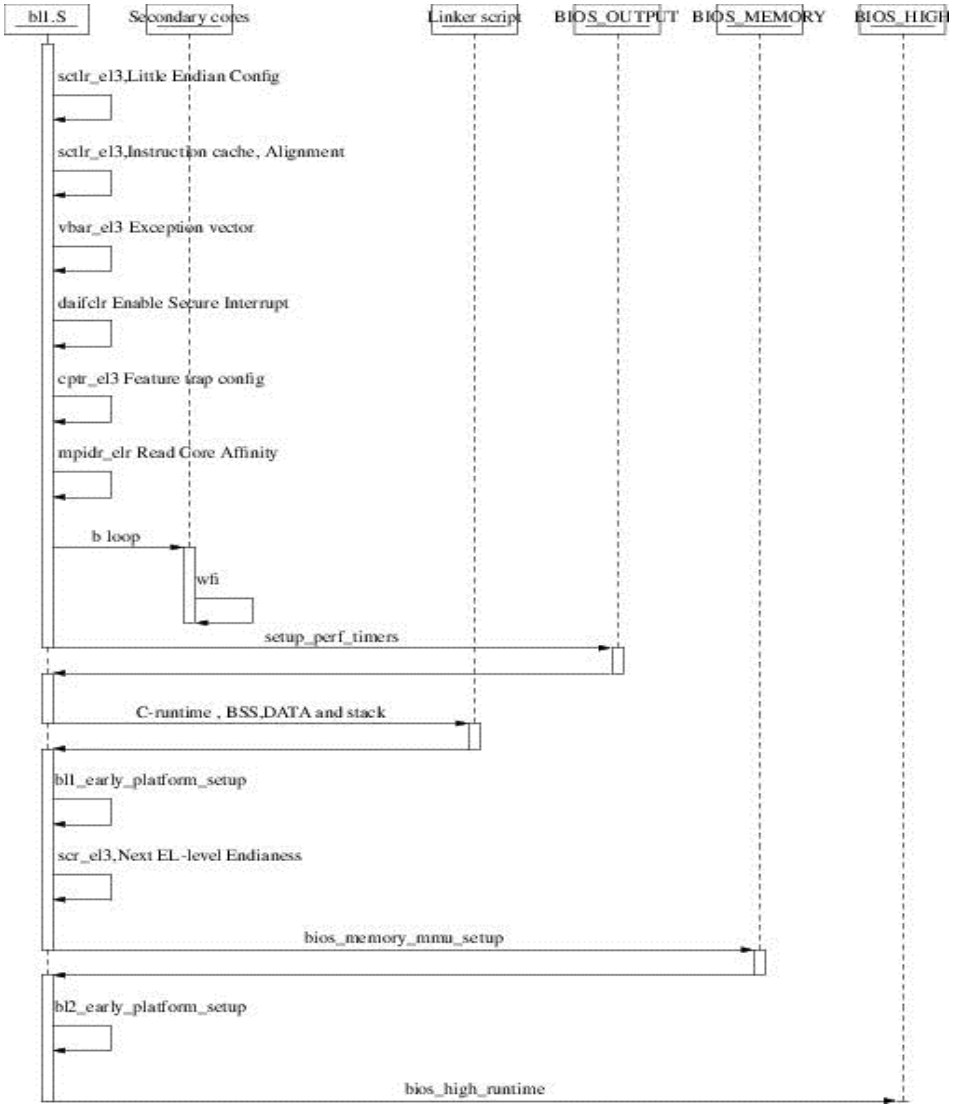


Figure 6: Secure Boot Low

5.1.3.2 HASPOC Secure Boot Format (HSBF)

The boot image consists of concatenated objects. The order is important since Secure Boot lacks functionality to process objects if they do not follow the boot image format. The image is retrieved in two phases, initial read and final read. The initial read is required to configure DRAM memory. The subsequent final read process all objects found in the entire image. This approach enables dynamic allocation of memory assigned for heap. During the initial read, heap is assigned to the limited static RAM (SRAM). During final read, heap is assigned to dynamic memory. The image could be retrieved from a file system on a target, or as a continuous sequence of sectors. For the latter, size of the total image must be contained inside the HSBF image.

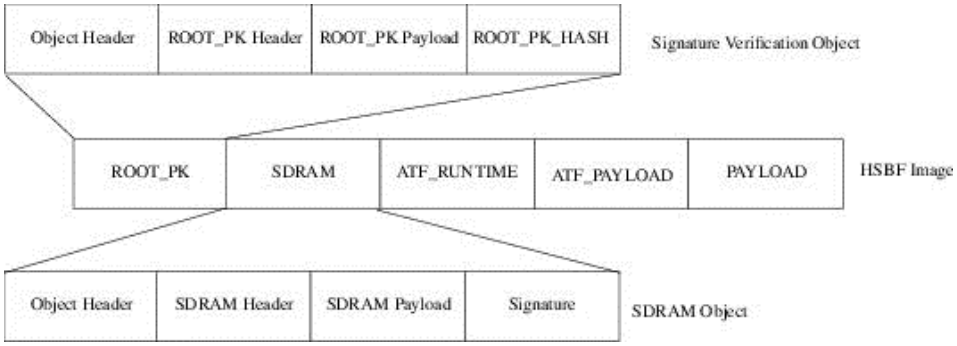


Figure 7:HSBF File format

Each block contains two headers, the outermost header defines the chained format of HSBF, the innermost header represents the contained object.

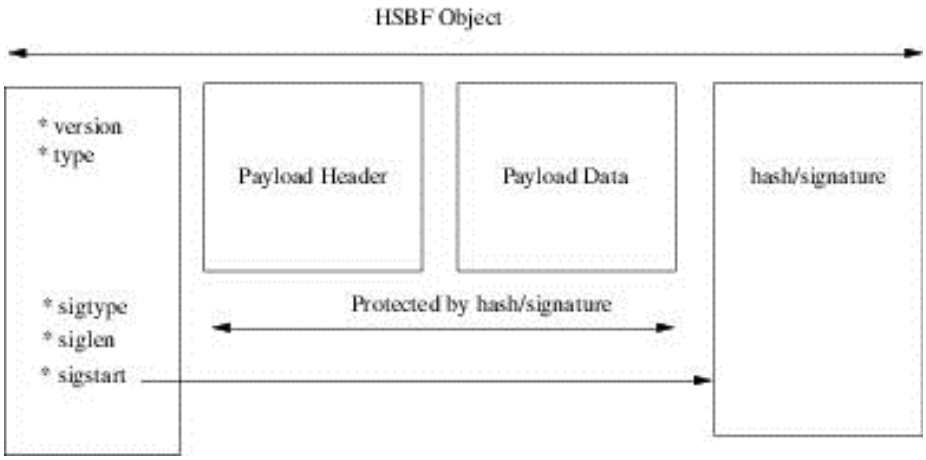


Figure 8: HSBF Object Format

The hypervisor objects are represented by a structure where start and size members maps to the HSBF format. The bl33_load_address parameter defines the destination address on the target platform.

```
struct  atf_payload {
 uint64_t start;
 uint64_t bl33_load_address;
 uint32_t size;
};
```

The corresponding manifest used by the signer tool to create the image maps directly to the above structure. All security critical parameters such as exception levels, register content, memory address for storing verified images are specified in plain text for easy inspection. The parameters are then covered by a signature.

5.1.3.2 Secure boot high

Upon power on, the low level module will either transfer control to the secure boot high module or abort with an error code indicating detection hardware problem.

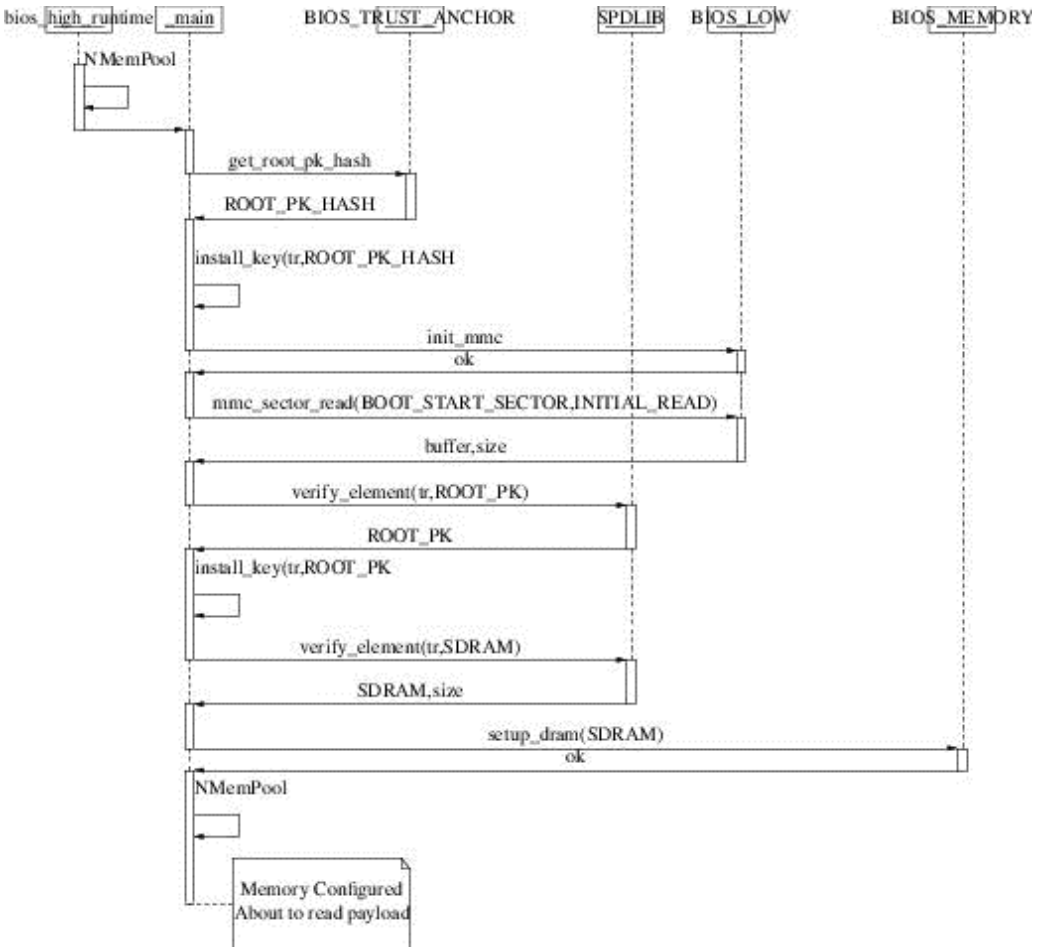


Figure 9: Secure Boot high

The entry point into bios_high is the runtime module, that assigns a region of the limited static ram to be used as heap.

The implementation of main is highly portable so that all exit codes could be tested thoroughly, either by unit test on the build system, or integration tests on the actual target. The second attack surface (AS2), as discussed above, relates to storage. It is addressed as follows.

Trust anchoring in Secure Boot conforms to ARM Trusted Firmware. The modular design allows both use of OTP/E-fuses as well as embedding the fingerprint ROOT_PK_HASH into ROM.

Since static memory is limited, only the two first objects are retrieved from the HSBF image in the initial read. The first object is the ROOT_PK key which is verified against the hash value (ROOT_PK_HASH). Upon success the SDRAM object is verified and processed. The heap is from now on assigned to the newly configured DRAM memory.

The entire HSBF image is now retrieved from storage into DRAM. All objects are processed,

including verification and copying to final predefined destination in the memory region.

The ATF_BOOT object is responsible to create the parameters needed to be compatible with ARM Trusted Firmware boot.

Passing control from Secure Boot to hypervisor requires downshifting exception level, changing from secure mode to normal mode, and finally the runtime module serving power management requests must be initialized. Since Secure Boot is fully compliant with ARM Trusted Firmware, a precompiled binary could be be injected into the HSBF image and invoked. The design promotes the HASPOC principle of static configuration, without sacrificing compatibility.

Until this point, the entire execution has taken place in EL3, with a static one-to-one translation table.

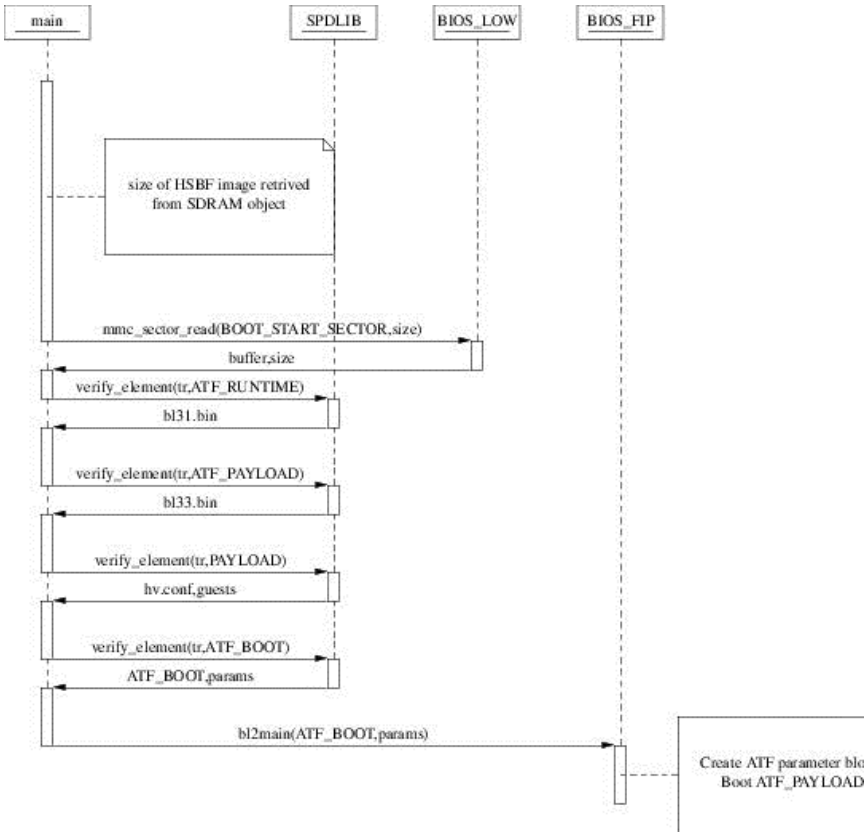


Figure 10: Processing HSBF Image and preparing for boot

Compatibility with RM Trusted Firmware also enables use of runtime modules required for power management (bl31.bin). This module contain two functional modules. The boot module uses the boot configuration to assign values in the ARMv8 parameter register, and also change from secure into normal domain. Further, the runtime module lowers the privilege level (from EL3 to EL2). Upon success the ATF_PAYLOAD is invoked, which for HASPOC this is the hypervisor, but Uboot have also been used to verify the ATF boot interface during the project. The hypervisor initially execute on the primary core, but sends requests to Secure boot too power on new cores. Currently power management is very simplistic, due to formal verification. Each guest gets its primary core from the hypervisor, and could then require new cores with SMC/PSCI calls. The hypervisor traps all SMC calls and then generates new power-on request on the behalf of the guest.

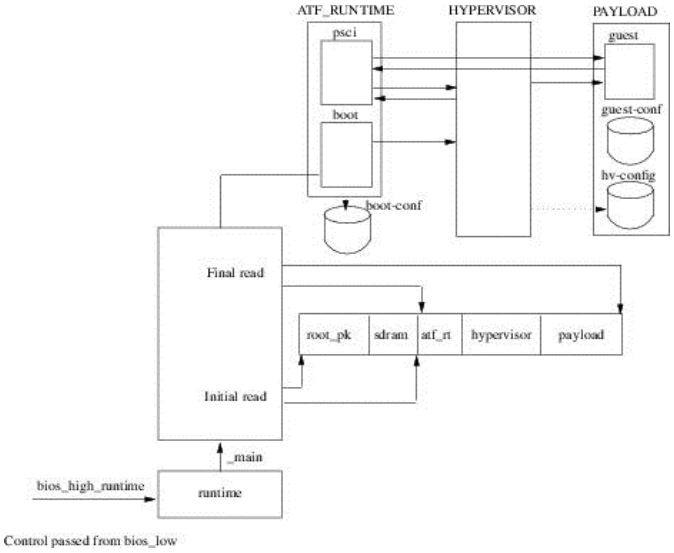


Figure 11: Secure boot interface to ATF runtime

5.1.3.3 Remarks

As mentioned, there are some points where the HASPOC boot differs from what is specified by the ARM Trusted Firmware.

1. The finger print of the root public key (ROOTPK) is stored in e-Fuses, rather than in hardware key registers. This is chosen as it offers higher flexibility.
2. We do not use full X.509 certificates for the public key, but allow raw public keys for simplicity. (Formally verifying a full X.509 implementation would be quite cumbersome.)

Note also that at the end of the secure boot process, as control is handed over to the HASPOC hypervisor, it is crucial for the formal verification that this is done in a system state which agrees with a verified starting state of the hypervisor (one for which the formal verification is valid). In principle the system could at this point, as an extra precaution, make a self-test that such a state indeed is obtained. This would provide extra protection against accidental misconfiguration by the user since the fact that the signatures of the policy files and other configuration data verify correctly, does in itself not imply that these policies and configurations are free of errors and mistakes. While no such verification is currently made, a handle is provided where the user can add such checks.

5.2 The HASPOC Hypervisor

The HASPOC hypervisor is a so called Type-1 (or “bare metal”) hypervisor allowing, in principle, full virtualization. This means that the hypervisor does not require an underlying operating system and that the hypervisor is transparent to guests. An exception lies in inter-guest communication which is performed by special hyper-calls. However, for certain types of guests, e.g. rich OSs such as Linux, even these calls may be handled transparently through OS drivers.

5.2.1 Services to Guests

5.2.1.1 Built-in Services

5.2.1.1.1 Secure Boot

It would be undesirable from verification point of view for the hypervisor to require functionality to read and load from external storage, to have cryptographic algorithm support etc since the footprint of verified code gets smaller if functionality that is not generally needed can be factored out. Therefore, much of the responsibility for secure boot of guests is handled by the HASPOC secure boot component, comprising functionality to verify the integrity of the loaded guest image before guest execution is allowed. To be precise, the contents of a guest image on file may be (temporarily) loaded into memory for signature verification purposes, but if signature verification fails, the HASPOC platform will not further process the guest data file (e.g. parse guest parameters or allocate resources to the guest), in particular the guest will not be started and the platform will instead perform system reset. The guest images may be signed with a user-provided public key, where said public key is in turn signed relative to the root public key and guest images can be verified by the HASPOC secure boot.

In terms of secure boot service for guests, the security dependency on the hypervisor is that the hypervisor must correctly handle the verified guest data, e.g. properly allocate memory (second stage virtual memory mapping, see below) and configure input parameters to the guest etc before starting the guest. The secure boot of the guests thus also comprises the configurations done by the hypervisor in steps VI and VII, during which the following parts are active.

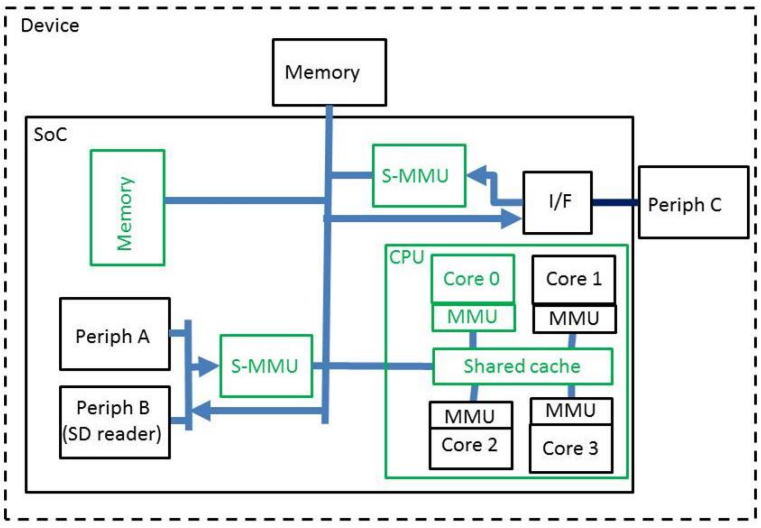


Figure 12: HASPOC Platform during hypervisor initializations

5.2.1.1.2 Virtual Machine Services

This is manifested in the provisioning of isolated CPU core, memory and peripheral instances, as discussed in Section 5.3.1 and 5.3.2.

5.2.1.2 Platform Security Services

5.2.1.2.1 Inter-guest Communication

Clearly, the fact that guests are allowed to communicate implies that information *can* flow between them. Since communication implies information exchange, to still be able to ensure a meaningful concept of information isolation, two main types of (potential) communication/information flow between guests (between a *source* and a *target* guest) need to be considered.

- *Supervised communication*: information flow, explicitly authorized by, and passing through a dedicated resource allocated by the hypervisor.
- *Unsupervised communication*: any other information flow, direct or indirect, between guests. An example of an indirect information flow could be a so called side-channel, for example information deducible from timing measurements.

The first relation between isolation and inter-guest communication in the HASPOC context is that the hypervisor shall ensure that only supervised communication will occur. This is however not sufficient as the concept of supervised communication would not be meaningful to isolation properties unless both source and target identities of the communication are properly authorized. Therefore, the HASPOC hypervisor provides a secure communication service to guests, meaning a supervised communication with access control, integrity and authenticity properties as follows:

- *Read access control***:** Only the guest for which the communication is intended shall be able to access it. This relies on a verified access control mechanism, to support the isolation properties. There is thus no need for explicit confidentiality protection of communicated data.
- *Write access control (source origin authenticity and integrity):* Again, by relying on verified isolation properties, no cryptographic protection is needed as only the claimed source guest shall be able to send data and no other 3rd party guest can modify the information content.

Implementation of inter-guest communication is done through *communication objects*. There are two types of communication objects provided by the hypervisor: *data sharing* and *event signaling*.

Data sharing communication objects are implemented by shared memory, and shall be unique to a given (source, target) pair. The source guest may write to, and read from, the associated memory while the target guest may read but shall not be permitted to write. Note that the hypervisor does not enforce write-only policy for the source. Doing so could provide additional assurance against side-channels. For example, it could be possible, through timing analysis related to memory caching, to determine if/when the target has read specific parts of the shared memory, which in turn could leak information about the target. However, no such enforcement is currently implemented. Thus, two distinct communication objects are necessary for bi-directional communication. The hypervisor enforces the read/write access control to the resource associated with the communication object.

The basis for this is a policy file, which is securely made available to the hypervisor by the secure boot.

Note that the hypervisor does not guarantee delivery of every individual message written to the communication object. For example, a source guest could submit a first message and then modify/overwrite it with a second message before the first has been read by the target guest. Therefore, guests are responsible for implementing synchronization, acknowledgments, etc, if so needed.

Event signalling communication objects are implemented through interrupts between cores for further study and is used to allow guests to notify one another about incoming messages.

More details on the implementation of inter-guest communication can be found in the section on interrupt handling.

5.2.1.2.2 Secure Peripheral Access

Guests may also communicate with peripherals. The physical peripherals may be *shared* or *exclusively owned* by guests. The hypervisor enforces that a peripheral is exclusively owned by a specific guest according to an integrity protected *peripheral ownership policy* provided through the secure boot, except for hardware timers, where the underlying hardware timer may be shared between guests through virtualized, guest-exclusive timers provided by the hypervisor.

Exclusive ownership and isolation is enforced through various mechanisms. The type of mechanism depends on the type of peripheral. For example, for *memory mapped peripherals*, access control policy is enforced on the associated memory. This will be described in more detail in Section 5.3.3.

As a note, Graphics Processing Units (GPUs) are particularly difficult as they are very complex and their internal specifications are not always available at the level of detail necessary to formally verify isolation properties. Consequently, it would become much harder to provide assurance on inter-guest isolation if GPUs were to be shared.

5.2.1.3 Trusted Guest Services

The HASPOC platform can enable means to enable “outsourcing” of specific services to other guests. Since the HASPOC platform provides secure communication between guests (in the sense defined above), any guest A can share information with another guest B (e.g. for processing such as encryption or key management) and can also rely on that results of such processing is only made available to the guest itself. However, for this to make sense, guest A must clearly trust guest B, and the properties on which such trust is to be based is outside the scope of HASPOC. The HASPOC platform does not specify the implementation of such “trusted” guests and their services, nor which security functional and assurance requirement to put on them: a “trusted” guest is just another guest from the HASPOC platform perspective. On the other hand, as a future extension of HASPOC, one could envision another type of trusted guests, more directly anchored into the HASPOC platform, e.g. by assigning them special authorization to access e.g. certain peripherals, not allowed be other guests. The trust could/should here be supported by (formal) verification also of the trusted guest.

5.3 Isolation Enforcement

The basic policy enforced by the HASPOC hypervisor is that each guest exclusively owns a subset of the available *system resources* (memory, core(s) and peripheral(s)) for its own use. The basic objectives of the platform are to ensure:

(a) A guest cannot directly access memory areas of the hypervisor.
(b) A guest cannot directly access memory areas of other guests (unless shared in communication).
(c) A guest cannot directly access other guests' cores.
(d) A guest cannot directly access other guests' peripherals (expect through one guest providing proxy access to the peripheral).

The word “directly” above signifies that, as mentioned, the HASPOC hypervisor does not provide assurance against existence of side-channels, e.g. timing-analysis and we also need to allow “controlled” leakage, e.g. through the use of supervised inter-guest communication. The ideal model with “air-gap” isolated guests, to which we formally prove equivalence, also provides inter-guest communication and thus allows information to flow between guests. In other words, the memory content of a first guest can only affect the memory content of another guest if the first guest so allows through communication.

This equivalence between the isolation offered by the HASPOC hypervisor and the ideal model is formally proven by verifying, at machine code level, that all execution flows on the HASPOC platform always have an equivalent execution flow when implemented on two physically separate systems. Before looking into the formal proofs, we next discuss how isolation of the resources are implemented by the hypervisor.

5.3.1 Isolation of CPU core resources

Different guests execute on exclusively owned cores in the current version. A guest is therefore allocated to one (1-to-1) or more (1-to-N) cores. The option to share core among guests (N-to-1) is currently not supported. This frees us from the necessity of context switches and thus helps to keep the hypervisor light-weight.

The guests however share the CPU cores with the hypervisor. At core level, isolation between a guest and the hypervisor is enforced using the *privilege (or “exception”) levels* provided and page tables/access permissions. For the assumed 64-bit ARMv8-A architecture, four different exception levels are provided:

- EL3 for TrustZone (this is the highest level)
- EL2 for the hypervisor
- EL1 for the operating system
- EL0 for applications (this is the lowest level)

Thus, the HASPOC hypervisor operates at exception level 2 enabling control of guest behavior

(running at exception level 1 and 0) executing on the same core while still maintaining the isolation. The hypervisor is not involved in switches between EL1 and EL0.

Guests must be prevented from executing code at EL3 as such execution is outside the control of the hypervisor. Execution at EL3 cannot be prevented entirely since the system will always start at that level. EL3 execution takes place during the HASPOC platform boot and initialization, after which privilege is lowered to EL2 as the hypervisor starts. The ability for guests to issue Secure Monitor Calls (SMC) to enter EL3 (once they are launched) is also disabled at boot. If a guest still attempts to invoke SMC it will cause a trap to EL2. During execution, SMC calls can be used for power management, but these are also intercepted by the hypervisor, checked/translated and sent again from EL2. Another potential way to reach EL3 is through exceptions, e.g. interrupts. These are handled as described below.

5.3.2 Memory isolation

Physical memory (of hypervisor and guest(s)) is exclusively owned and statically configured via an interface between the secure boot and the hypervisor. The hypervisor must enforce that a guest cannot directly (outside the scope of supervised inter-guest communication) access or modify information in the memory exclusively owned by another guest.

The memory management unit (MMU) provides a mechanism to translate virtual addresses to physical addresses. When executing natively, the OS requires one level of address translation (virtual to physical). But when the MMU supports virtualization, it will provide an additional translation level that is controlled by the hypervisor as shown in Fig 13.

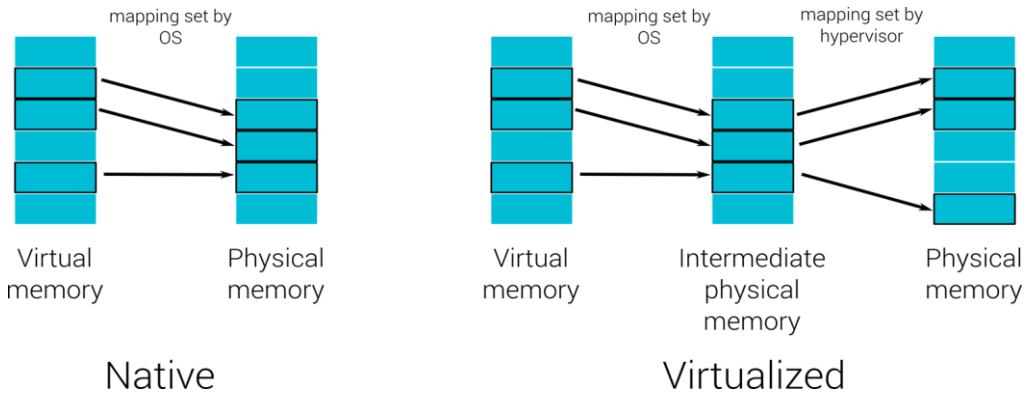


Figure 13: One and two level memory virtualization

This mechanism is used to allow the hypervisor to enforce access control policy pertaining to a guest's access to the physical memory. The entries of the second stage page tables that map the guest's working memory shall be configured by the hypervisor in conjunction with initialization and subsequently be kept unmodified.

5.3.2.1 Cache Isolation

The cores share caches. It has been identified [P,RTSS,IGIES,ZJBR,GNBD] that this is a potential

source of information leakage (side-channels). When a guest resumes execution, timing analysis allows the guest to figure out if its entries in the cache were evicted while another guest was executing. Since memory addresses are mapped to cache lines in a known way, this enables a guest to deduce information about which memory addresses another guest has accessed. Suppose for example that bits in an encryption key determines which addresses the other guest should access (which is common in encryption algorithm source implementation), it effectively means that a guest can figure out bits of the encryption key. Such attacks have been demonstrated in practice, [IGIES,ZJBR,GNBD]..

Various techniques have been proposed to mitigae these issues, e.g. [SM]. In our project, no specific countermeasures are provided however. As a practical way to limit impact on e.g. guest performing cryptographic operations, one may opt to choose cryptographic algorithms that to less extent use key-dependent memory accesses and/or add dummy memory accesses to the code.

5.3.3 Peripheral isolation

5.3.3.1 Memory Mapped Peripherals

In a modern SoC, most components (including peripherals and parts of the CPUs) are *memory mapped*, meaning access is mapped to various locations of the address space. Whenever a guest makes store or load accesses to “memory” (i.e., an address within the accessible address space), it is either directed to the actual system memory or to a port of a peripheral that is mapped to the addressed location. An example of this is shown below.

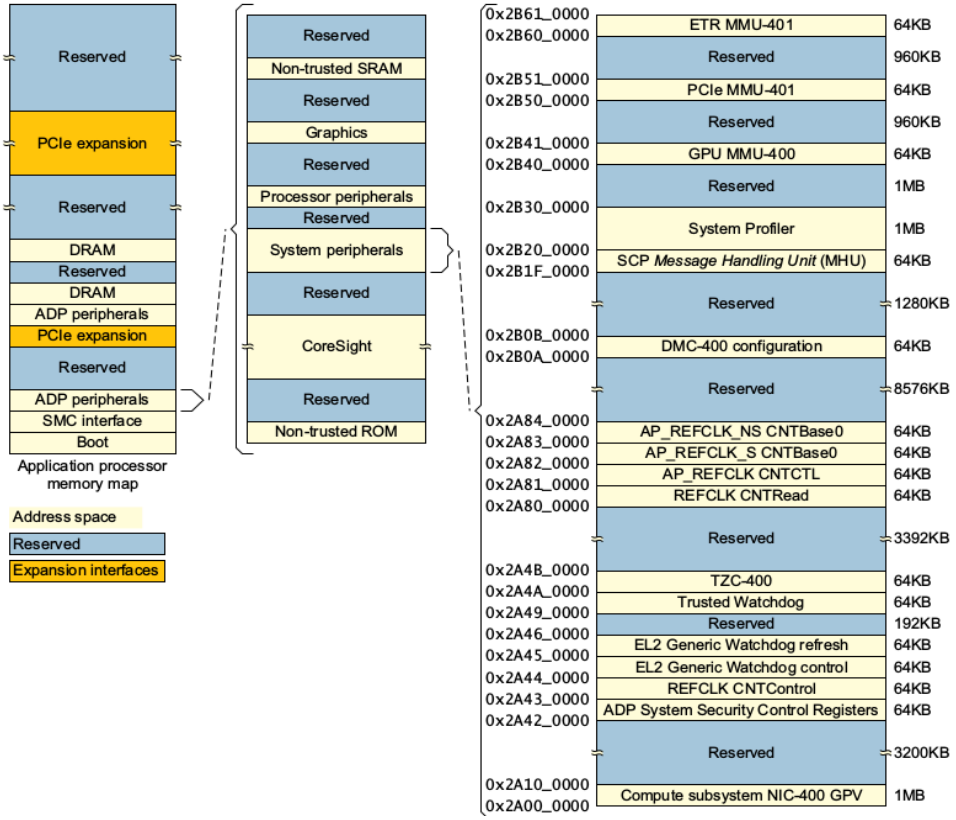


Figure 14: Example of memory mapped devices from the Juno board (source: Juno ARM Development Platform SoC Technical Reference Manual)

Guest access to a peripheral can therefore be controlled by enforcing the access control policy on associated memory addresses.

5.3.3.2 Architecturally Mapped Peripherals

Certain peripherals and CPU internal components are not fully memory mapped. For example, in ARMv8-A the floating point co-processor and the physical timers are parts of the CPU and accessed via special CPU instructions. Two approaches for isolation is used. When applicable, guest access is simply disabled altogether from higher privilege levels. For other cases, the hypervisor must maintain separate versions of data associated with the peripheral to avoid information leakage. For example, in the case of the FPU co-processor this is done by saving and restoring FPU state and registers during a context switch between different guests.

5.3.3.3 Interrupt Controller

Slightly simplified, the interrupt controller receives messages from various peripherals and forwards them to the processor after appropriate filtering and priority check. In a multi-core system the interrupts need to be forward to the correct core(s), as shown in Fig 15.

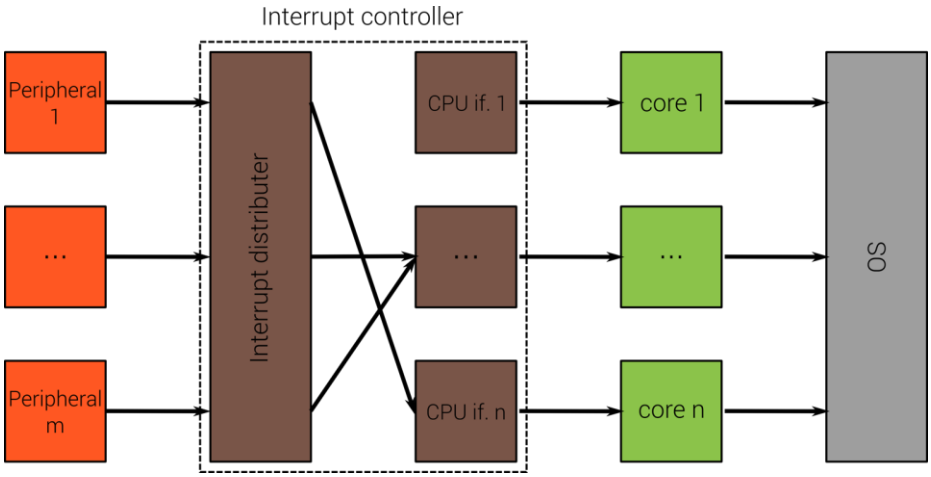


Figure 15: Interrupt forwarding to cores

In the HASPOC platform, the following principle applies. The access control policy for a given peripheral shall define access control policy also for interrupts from that peripheral. Thus, a certain guest is permitted access to interrupts from a peripheral if, and only if, said guest is allowed access to said peripheral, which as noted is equivalent to the guest exclusively owning the peripheral.

To this end, the interrupt controller must be configured to route incoming interrupts to precisely that core where the owning guest is located. The hypervisor must forward physical interrupts to the guest that owns the source device (using virtual interrupts).

In a HASPOC compliant system, the target platform provides virtual CPU interfaces that can be (and are) controlled by the hypervisor. From a guest point of view these interfaces act just like the physical CPU interfaces. This setup is shown in Fig 16.

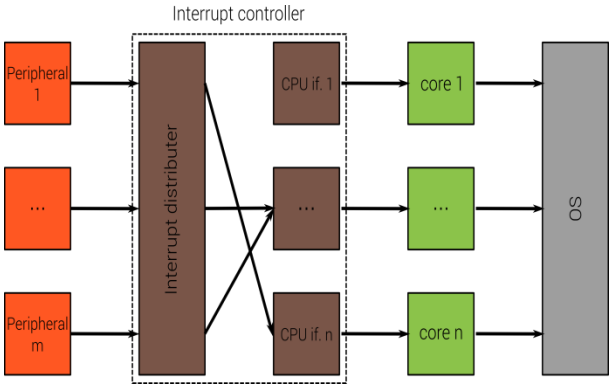


Figure 16: Interrupt forwarding to cores

Note on inter-guest communication: For the purpose of implementing inter-guest communication, interrupts are used as follows. A source guest informs another guest about a new message associated with the communication object, by requesting the local part of the hypervisor to issue a Software Generated Interrupt (SGI) directed to the target guest. On the associated receiving core, the hypervisor then informs the target guest via a virtual interrupt.

5.3.3.4 System-MMU

Some peripherals require access to the internal buses and the memory allocated to the peripheral via

the DMA mechanism. This poses a threat to isolation since even if the MMU can restrict processor access to memory, it cannot restrict a peripheral's access:

- A peripheral owned by guest A could write into memory space of guest B.
- In a red-black separated system as in Section 2.1.2, data could potentially flow directly between interfaces without first passing an encryption module.

ARM-based HASPOC compliant systems however provide a so called *System-MMU* (S-MMU) which is a secondary MMU placed between the peripheral and the system bus, see below. The HASPOC platform uses this mechanism to enforce a policy that a peripheral cannot make e.g. memory accesses outside its allocated space. Each DMA-capable peripheral's access to internal resources must go through an S-MMU.

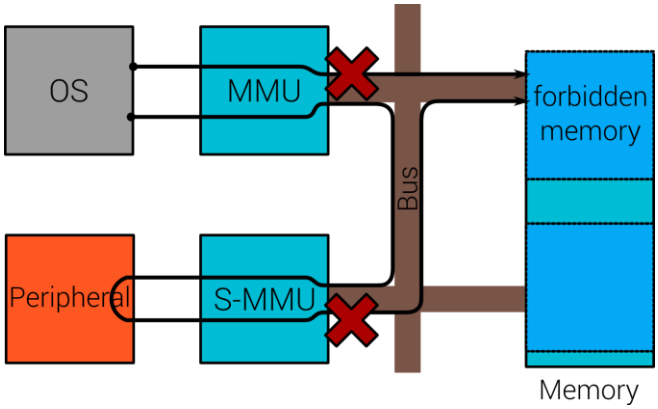


Figure 17: A System-MMU is used to limit peripheral access to the system bus as complement to processor access policy enforcement by the MMU.

5.3.3.5 Peripheral Access Without an S-MMU

Sometimes an S-MMU is not available or does not fulfill our security requirements. A common example of the latter is an S-MMU which is shared between multiple peripherals allotted to different guests. In such cases there are two options to consider: (1) not giving the guests access to these peripherals and (2) allowing access only through hypervisor drivers.

In short, a hypervisor driver emulates the target peripheral while at the same time enforcing the isolation rules. For example, the DMA peripheral mentioned earlier can be replaced with a hypervisor driver that denies DMA transfers that would read from or write to memory not belonging to the initiating guest. Note however that hypervisor drivers have two major drawbacks, performance and added complexity.

6. Life-cycle Guidance

In this section, we provide some guidance to the life-cycle management of a system using the HASPOC platform. The assumed starting point is an organization who imports an open source distribution of the HASPOC platform code base to deploy it, possibly after modification, together with one or more guests, on HASPOC compliant platforms.

6.1 General

6.1.1 Import of HASPOC code base

Before deployment and/or updates of the code base, the integrity of the originally imported code base must be verified, else any assurance of systems built on-top is void. The HASPOC project is investigating the possibility to provide signed images of source and how to distribute the associated verification key. Meanwhile, developers interested in obtaining an ensured authentic copy of the source are kindly requested to get in touch by email through the contacts found at https://haspoc.sics.se/.

6.1.2 Upgrade of the HASPOC Platform

First of all, it is important to be aware that the formal verification and accompanying CC Security Target may be invalidated by any change or modification to the HASPOC code base or if deploying it on a non-compliant platform. Depending on the assurance level desired, one may apply different approaches. For highest possible assurance it is of course necessary to redo relevant parts of the formal verification and produce an updated ST, taking into account the modifications done. This document provides some directions as to which parts that must be re-done. Even if this level of assurance is not required, one should at least carefully analyze the modifications to determine if (and how) they affect the behavior of the secure boot or isolation mechanisms described here (and preferably also in the more detailed accompanying HASPOC technical documents, [HASPOC]).

During development work, standard security procedures (access control etc) for the development environment needs to be applied. In case a CC evaluation of the final system is desired, CC provides concrete requirements that must be fulfilled.

A (rough) idea of impacts when deploying HASPOC code base on a non-compliant platform can be obtained by analyzing the platform requirements and their rationale as listed in Section 4.1.

6.1.3 Cryptography and Key Management

A foundation for the security offered by systems built on the HASPOC platform is the integrity of the boot images. Since ARM-based systems are common in embedded systems and constrained computation platforms, it could be desirable to replace the cryptographic algorithms, e.g. using elliptic curves instead of RSA signatures in the secure boot. Indeed, the part of the HASPOC code base which is most modular and therefore easiest to replace while maintaining assurance is probably the cryptographic algorithms. Replacing the default RSA, SHA-256, and HMAC algorithms by (secure/verified) implementations of any other algorithm(s) with required strength should be reasonably straight-forward.

Update and revocation of the root signing key can be supported if e-Fuse technology is used, see next section for a discussion.

6.2 Software Updates and Revocation

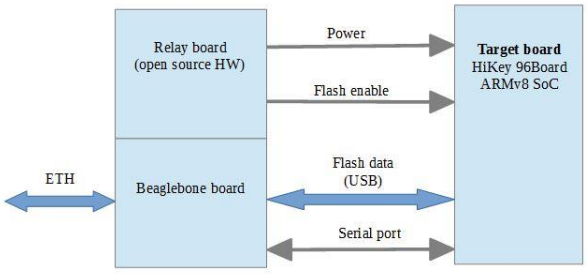
In a setting where a system based on the HASPOC platform undergoes updates, one should

normally revoke previous versions to avoid security issues related to roll-back. Since we propose to use e-Fuses for storage of the root signing key, this also provides means for flexible version management if field-programmable e-Fuse technology is used. By assigning a version number to the code base (which is supported by the HSBF format) one would “burn” e-Fuses corresponding to the current version and during boot reject any image with version below that indicated by the e-Fuses. (This is not supported by the current code base and thus needs to be added). Clearly, one needs to be aware of caveats of using field-programmable e-Fuses since it could e.g. allow an attacker to disable a system's operation by burning all fuses, thereby revoking all version. Even in the absence of attackers, random fault conditions could prevent the current version from booting and some high-availability deployments may therefore choose to maintain a “golden version” which is always allowed to be booted, simply to make the system available. This version of course still needs to be signed/verified. In the HASPOC platform this could be implemented by keeping the golden version on external media and perform boot from that if internal ROM/flash boot fails.

Concerning root key revocation, one should clearly not permit any fallback to previously revoked keys.

6.4 Environment and Tools Used in the Project

The HASPOC software is divided into secure boot and hypervisor build systems. The secure boot build-and-test system uses exported binaries from hypervisor build-and-test and vice versa. The secure boot build-and-test uses the T2data proprietary MAIA system and the hypervisor build-and-test uses git-repos and make-scripts for build and custom python scripts for testing. Both systems interact with the test object via off-the-shelf electronics with custom wiring for the serial channels. It contains a Beaglebone black with a relay cape connected to a HiKey ARMv8 board.



The above block diagram shows how jumpers on the Hikey ARMv8 board are connected to remote controlled relays that control boot mode and power.

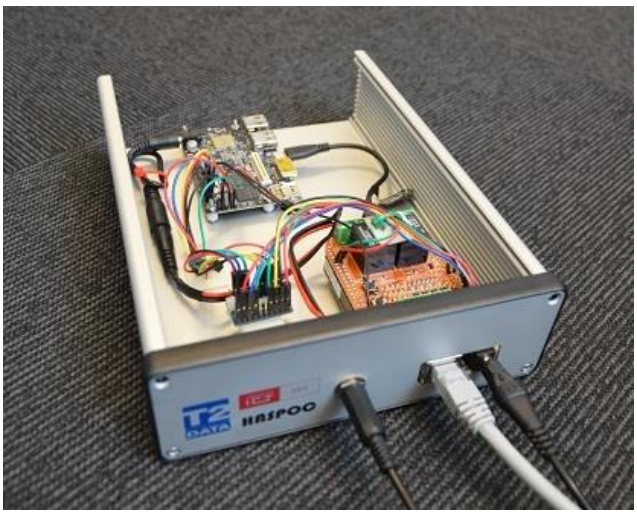


Figure 18: The two boards enclosed in a box

6.4.1 Secure Boot build and test

6.4.1.1 Secure Boot build

Secure boot is built on MAIA, a T2data proprietary Continous Delivery system. Every night all source code is built and the results are stored in an .iso file ready for delivery. As much as possible of the source code are platform generic which enables basic tests already in the build machine, typically a x86 platform. MAIA receives the hypervisor binaries every night before the build.

6.4.1.2 Secure Boot test

Every night the built code is tested in various ways. For example, the cryptographic functions are first tested on the build machine and later false binaries are loaded to the ARM platform to exercise error handling.

6.4.1.3 Sign tool

The process to have a secure execution starts with a signed and verified binary. The sign tool packages boot data, hypervisor, guests and configuration into a monolith. How to package, what images and where they should be loaded, is defined in a manifest.

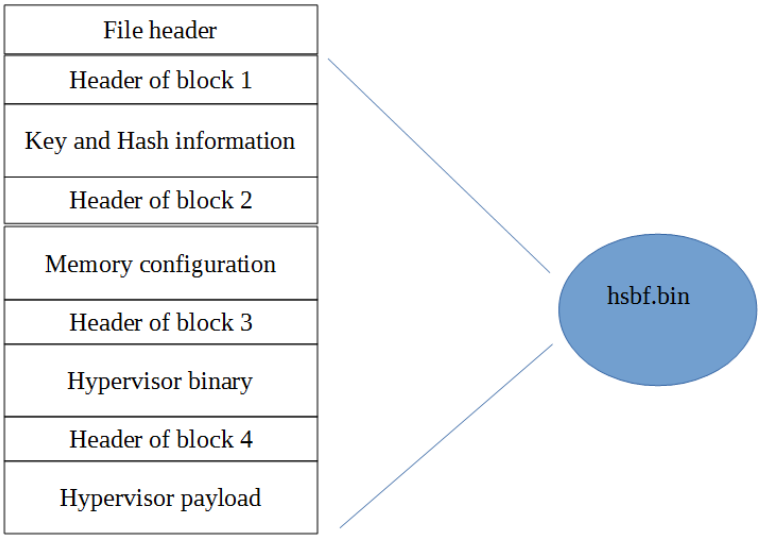


Figure 19: Principal design of HSBF file, for actual design, see svn:/HASPOC Documents/Secure boot design/

This hsbf.bin is later verified and loaded by secure boot. The foundation of this verification is a SHA2 function that generates a “checksum” or hash on the content of the file.

6.4.1.4 Secure boot runtime, HiKey platform

The chosen platform for HASPOC is the HiKey 96-board. Since this project was early to adopt ARMv8, there was a limited number of available platforms. This platform has worked pretty well but the project has been suffering from the lack of SoC specification.

The source code is based on Arm-Trusted-Firmware 1.1 (ATF). In this project this source code has been simplified and reduced where possible. The main reason for reduction is to be able to execute formal verification with reasonable complexity.

The major part of the HW setup is unchanged since the SoC designer has not shared important specifications.

A cold boot starts with a short ROM-sequence and then jumps to a open source sequence that does some critical HW setup and changes execution state to AArch64. After that, the execution of Boot Loader 1(BL1) starts. BL1 is redesigned for HASPOC to handle the rest of boot. BL1 copies the HSBF image to from Flash to DRAM, checks the integrity and copies the different blocks to the addresses specified in the block header. BL1 also loads the secure monitor(BL31).

At the end of boot, the specified EL-level is set and X0 is loaded with the hypervisor payload address, finally a jump to the hypervisor terminates the boot.

The secure monitor resides in memory the entire runtime and is called when for example Linux tries exercising power management. Secure monitor runs in EL3 and calls functions in the closed source BL30 which runs on a small helper processor on the SoC.

6.4.2 Hypervisor build and test

Secure boot is built with a cloned git repository. The address is:
https://bitbucket.org/vahidi2/sth_armv8/

Every night source code is built and the results are stored in a git repository: https://bitbucket.org/vahidi2/HASPOC-binaries

This repository also contains prebuilt boot binaries that is used in tests. The source code also includes guests and scripts for nightly builds and tests.

6.4.2.1 Hypervisor build

After the command to clone the repository, the current procedure involves a build setup, build guest and then build hypervisor. Details can be found in the directory doc/. Today we have support for a number of platforms including simulator, HiKey and Nvidia TX1. We have support for a number of guests including Linux, SICS-mini-OS and Android. The necessary boot sequence can be selected, ATF or secure boot.

The guests and configuration is packaged in a payload format. Details can be found in svn: HASPOC Documents/Hypervisor design/

There is also a tool to join the hypervisor and guests into one binary for compatibility to ATF.

All significant configuration both for hypervisor and guests are specified in device-tree format, a format that is for example used to configure Linux kernels.

6.4.2.2 Hypervisor test

Any time during development, there is support for test on simulator and HiKey-HW.

However the test for HiKey is more extensive as it also includes automated checks and performance tests both for hypervisor and the guests. This test mechanism is used every night when a autobuild/autotest tool triggers build and test of the latest on the development branch. If the tests pass, the results and binaries are posted to MAIA for the boot tests.

6.4.2.3 Hypervisor runtime, HiKey platform

When the boot terminates, execution starts at the defined start address of the hypervisor. The hypervisor prints some build information and dumps several important registers to the console. The device tree files specifies how many cores and where to put guests. For HiKey we typically assign all 8 cores; 3 cores for guest1, 3 cores for guest2 and 2 cores for guest3.

6.4.3 Common Criteria and Module Dependencies

Common criteria introduces requirement of documents, which is challenging for agile projects with research focus. Module interfaces are documented with doxygen, which provides strong relation between implementation and documentation.

External security related interfaces, must be documented from a dependency perspective, also the subsystem should be modular, which results in several layer. Combined with agile development, where the team learns during the project require doxygen like tools for generating module dependency for the so called TSFI:s.

A future evaluation will also need to specify an “evaluated configuration” under which the evaluation holds, e.g. a platform configuration that ensures that debug ports are disabled to prevent bypassing TSF:s, etc. Details of this are left to the evaluation.

Below is a requirement which was early introduced in the HASPOC project:

Provide a semiformal description of each module in terms of its purpose, interaction, interfaces, return values from those interfaces, and called interfaces to other modules, supported by informal, explanatory text where appropriate.

The CC requirement has great impact of the modularity, but also on the build system that have been tailored to generate data needed to generate how modules depend on other modules.

7 Formal Verification Methodology

The formal verification of the HASPOC virtualization platform aims at establishing mathematically proven and machine checked guarantees of the isolation of guests in the system. The top-level property we aim to show in the project is non-interference: that no information can flow between guests except through explicitly defined channels. This means in particular that guests cannot change the private data of other guests or the hypervisor (integrity or no-exfiltration) and that they cannot observe the contents of these private resources (confidentiality or no-infiltration), see [HALM].

Formally, integrity is stated as a safety property[1] on execution traces of a system. Assume that *Wi* is the set of resources that guest *i* is allowed to modify according to the access policy governed by the hypervisor. Resources in this setting cover not only memory cells but also registers and output signals. If a system fulfills the integrity property for guest *i*, then for all possible execution steps of this guest, indeed only resources in *Wi* are modified (thus all other resources are unchanged).On the other hand confidentiality is a trace hyperproperty (relating different executions of the same system) concerning the resources *Ri* that guest *i* is allowed to read or write[2]. Roughly speaking it says that for two starting states *S1* and *S2* where the contents of *Ri* are identical (but everything else might differ) if guest *i* executes the same code in both states, then after any number of steps the contents of *Ri* are mutually identical in the trace starting from S1 and the trace starting from S2. That means

[1]We refer to a safety property in the computer scientific sense here as opposed to a liveness property. Safety properties demand that an undesirable behavior will never occur, i.e., in the case of integrity, that a guest will never modify data for which it has no rights according to the hypervisor's access policy. In contrast, liveness properties guarantee that a desirable behaviour will eventually occur, e.g., termination.

[2]Mentioning writable resources here might seem redundant since normally something that is writable is also readable. However there might be corner cases, like write-only output signals and device ports.

that updates to the resources in *Ri* do not depend on anything that is not accessible to guest *i*, therefore no private data of others is leaked to guest *i*.
Naturally one has to show these properties for all guests in the system. In the formal proof that the HASPOC hypervisor exhibits them, several refinement steps are taken. Broadly, the first step employs a so called bisimulation-based approach to ensure that the platform design resembles a system where non-interference holds by construction. The second part of the proof focuses on the handler code and establishes the correctness of its binary implementation by means of (semi)-automated code verification methods. The overall structure of the verification methodology is depicted in Fig 20.

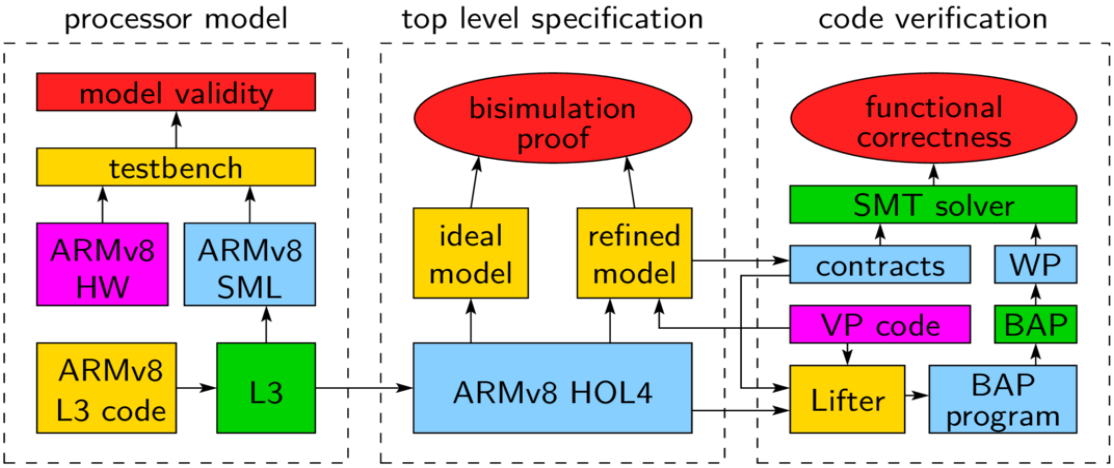


Figure 20: Overall formal verification methodology. Red containers denote goals, light blue boxes are derived automatically, green boxes are tools, yellow boxes are hand-written, arrows denote dependencies.

7.1 Real-Ideal Bisimulation

In general a simulation proof between two systems A and B is conducted to show that B is refining (implementing) A, or – equivalently – that A abstracts from (simulates) B. The two systems are usually coupled with a simulation relation *sim* between states of A and B. It is then proven by induction that for any computation of B starting in state *b* that is related to a state *a* of A by *sim*, there exists a simulating execution of A such that the resulting states *b'* and *a'* are again contained in the simulation relation *sim*. Note that this means that there might be execution traces in A that are not exhibited by B, however all possible behavior of B is modeled by the abstract model A, which is called *soundness* of the simulation.

A bisimulation is a special case of simulation between two systems where both systems are simulating each other. That is, also for every computation of A one must show that there exists a simulating computation of B. Thus all possible behaviors of A are also implemented in B, which is called *completeness* of the simulation. In case A and B are deterministic systems whose computations only depend on the initial state, it is enough to show the simulation once, because for every pair of initial states *a* and *b* there is only one pair of corresponding traces in A and B.

A typical example for a bisimulation relation is compiler correctness, e.g., between a high-level language model A and its assembly implementation B. The relation in this case would link high-level concepts with low level hardware components, e.g., high-level variable values with the content of registers and memory cells, the high-level code with the generated instructions in the

code segment, as well as the current program rest or continuation of the high level programming language semantics with the program counter of the executing processor. For soundness one needs to show that all the generated code in B implements some behavior of A, i.e., that a compiler only generates code that is covered by the original program semantics in A. On the other hand one would like the compiler to generate complete code, e.g., if a program exhibits several entry points, code should be generated for all possible execution traces and not just a subset of those. If the bisimulation is proven to hold for all execution traces of A and B, any safety property shown for A also holds on all traces in the implementation B (in a translated from). In addition, trace hyperproperties proven about A can be transferred to B as well. A bisimulation proof in this example could be performed for a particular pair of programs from A and compiled programs from B, by structural induction over the call graph of both, showing that the compiler correctness relation holds in corresponding, entry, branch, input/output, and exit nodes. Alternatively, one could target the compiler itself and proof the bisimulation for all programs and their compiled versions, by arguing over the code generation and memory allocation functions of the compiler which define the bisimulation relation in a general way. Doing this for an optimizing compiler is complicated and usually requires several refinement steps. Therefore for the verification of machine code in HASPOC we stick to the first method.

7.1.1 Ideal vs. Real Model

The idea of the real-ideal approach to proving information flow properties now is to formulate an idealized abstract model of the system which *by construction* has the desired properties (e.g., non-interference). The main idea of virtualization (and isolation) is to provide an illusion to the guest processes that they are running on the machine alone. Each guest owns a virtual machine which in principle completely isolated from the others'.

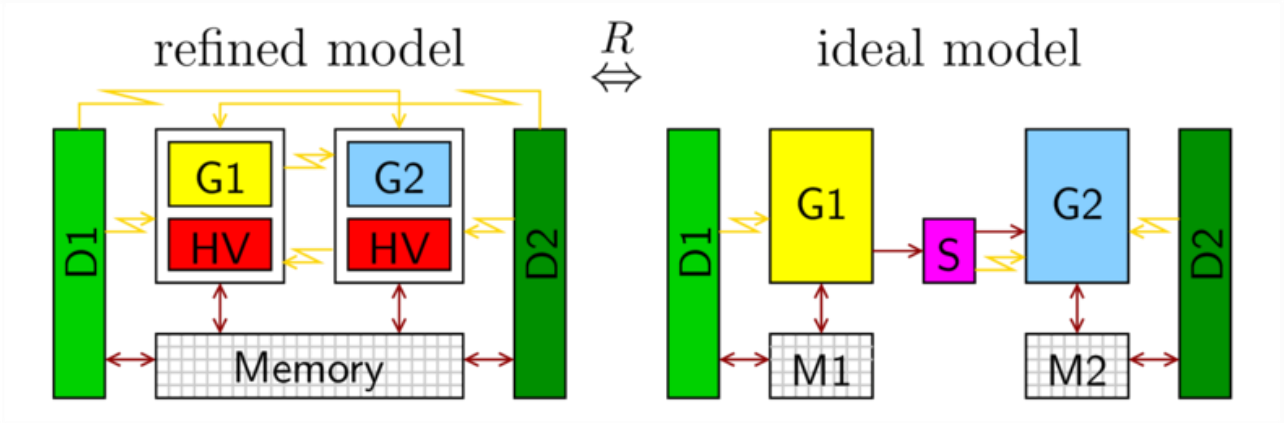


Figure 20: Example of real and ideal model two guests on two cores running on top of the hypervisor under simulation relation R

We pick up this idea and model the ideal system as a number of separate virtual machines, which are reduced versions of the original ARMv8 processor cores. Specifically, these machines can only run in EL1 and EL0 and there is only one stage of address translation. There is only a subset of the platform's interrupts available to each guest and the memory allocated to each guest has a smaller size than the complete physical memory. Similarly, peripherals are only connected to machines that are allowed to use them. The communication mechanism between the guests provided by the hypervisor can be modeled abstractly by a message passing protocol using device-like uni-

directional message channels between the virtual guest machines. For other services provided by the hypervisor or the ARM Trusted Firmware code running in EL3, abstract data structures may be added to the ideal model which can be affected by the guest using hypercalls/SMCs. These are represented as atomic transitions of the model.[3] If two guests were running on the same core, they would be executed in an alternating fashion, controlled by a dedicated scheduler/timer, however in the HASPOC setting cores are owned exclusively. Nevertheless, there is a global scheduler which interleaves the parallel execution of the cores in a sequential way. Note that apart from the shared memory used to implement the communication between guests, the guest operating systems are free to configure their private memory as they wish. This means that their virtual memories exhibit all properties of the weakly consistent memory model of ARM (see 7.3) and sequential consistency cannot be assumed for the execution of the guests. Especially if one guest is running on several cores its virtual machine is a multicore processor with a weakly consistent memory.

One the other hand there is the model of the “real” system which represents the HASPOC platform on the design level. The system resembles the actual physical machine with all execution levels and the second stage address translation for the guests. Similarly the interrupt controller and system MMU become visible and the interrupts to the guests are routed through the hypervisor. Hypercalls and SMCs are not executed atomically but according to a transition system that resembles their C and assembly implementation. The interleaving of hypervisor and guest execution on each core becomes explicit including the saving and restoring of the virtual machine contexts. Moreover there is now only one weakly consistent memory shared between all guests. However the sequentially consistent hypervisor memory is separate from this memory and shared between the hypervisor threads. The implementation counterparts of the abstract data structures from the ideal model are located in this part of memory using virtual addresses. Figure 21 shows an example of the real and ideal model.

Once the models are fixed, one performs a bisimulation proof between the ideal model and the real model in order to show that they behave equivalently (see Fig 21). Proving simply a refinement from the ideal to the real model (i.e., a simulation between ideal and real model) is not enough here, because confidentiality is a hyperproperty. In order to transfer confidentiality from the ideal model (where it holds trivially by construction) to the real model we need completeness of the simulation, i.e., that both traces in the ideal model have a counterpart in the real model. Then the simulation relation which holds between the corresponding resulting states can be used to establish that the accessible state of the executing guest is still identical. The bisimulation relation for this proof is quite straight-forward. All the components of the ideal model need to be mapped to counter-parts in the real model. This are first and formost the (intermediate physical) memory regions and the register state(s) of each guest. They are mapped to the actual physical memory regions and register states to either the actual register on a given core, or the saved guest context for this core, depending on whether hypervisor or guest are currently running. The contents of abstract message channels are mapped to the shared IGC memory regions. Similarly abstract hypervisor data structures that may be visible in the ideal model are mapped down to their implementation in hypervisor memory. Devices are mapped one-to-one between the ideal and real models. If there is a part of the interrupt controller visible in the ideal model and configurable by the guests, it has to be coupled accordingly

[3]In case the handlers involve concurrent to shared hypervisor resources, synchronization might be required. If more than one such synchronization occurs in the handler code, we need to model them as several atomic transitions.

with the actual IGC hardware as represented in the real model.

It should be noted that parts of the real model, e.g., the second stage MMU, the SMMU, (parts of) the interrupt controller, and parts of the hypervisor memory are not covered by the bisimulation relation. This makes sense since these resources are either fixed and crucial in establishing the idealized picture of the system or they are internal hypervisor structures that should not influence the behavior of guests and can thus be hidden from the outside view. However an invariant guaranteeing the isolation property must be proven on the real model along with the bisimulation. Note also, that the real model is still quite abstract in that it separates the hypervisor memory from the guests. In order to justify it formally, another bisimulation theorem is established with the hypervisor code running in the actual weak memory model that is shared between all cores. This theorem needs to argue about the restrictions we place on the hypervisor code that allow establishing sequential consistency, e.g., using only cacheable memory, not modifying the hypervisor page tables, using a locking discipline on shared hypervisor data structures to avoid data races, etc.

However, proving this theorem is currently outside of the scope of HASPOC and is being studied in a related project “PROSPER” [PROSPER]. In what follows we will focus on the real-ideal bisimulation proof which splits into four parts.

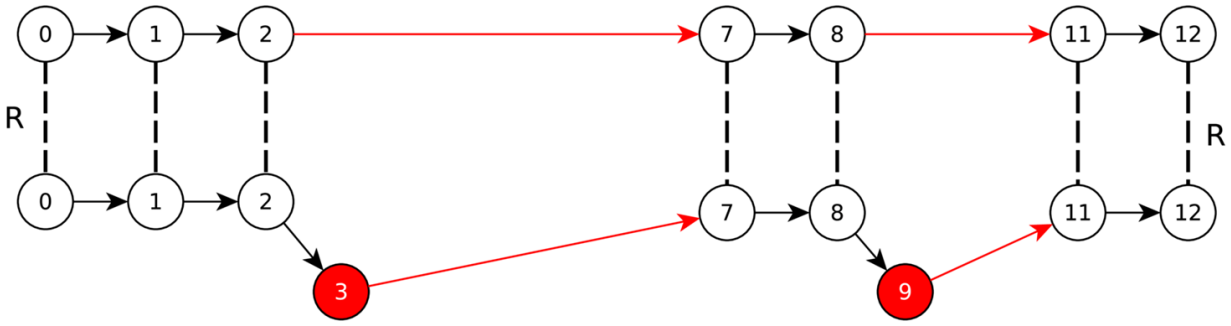


Figure 21: Bisimulation between guest and handler execution (red) between the ideal and real model

7.1.2 Initialization/Boot

The bisimulation concerning the hypervisor execution starts in an initial ideal and a real state where the bisimulation already holds. In order to obtain this initial property we need to argue about the boot code and the initialization code of the hypervisor. After the execution of this code, execution must arrive in a real state that is bisimilar to the initial state of the ideal model. This means that the guests are set up and ready to boot. Moreover the second stage of the address translation for EL1 and EL0 must be configured to guarantee the isolation of the guests and the interception of accesses to the inter guest communication control structure. Similarly the SMMU and interrupt controller are set up to conform to the configuration of peripherals in the hypervisor configuration file. Finally all data structures supporting hypercalls and SMCs are initialized correctly. We refer to these conditions as the *implementation invariant*.

This setting is implemented mainly[4] by the hypervisor initialization code. In order to run the hypervisor it first has to be loaded into DRAM and called at its entry point. This is the task of the secure bot loader of the HASPOC platform. Before that the boot loader also needs to configure low-level peripherals of the SoC that are necessary to execute according to the ISA specification. This includes activating timers and clocks, DRAM, caches, as well as configuring power management and the TrustZone controller that determines which regions of memory belong to the Secure World. Furthermore, the boot loader needs to install Secure World services and interrupt handlers as well as copy the guest images and hypervisor configuration file. The authenticity of all files loaded by the boot code need to be validated cryptographically.

To verify these requirements on the initialization of the HASPOC platform we mainly employ sequential code verification. The details will be discussed below. Proving the soundness of the approach is easier than for the runtime phase, because all the code is executed on the primary processor core while all secondary cores are suspended. Therefore we can reduce the weakly consistent concurrent model to a sequential setting where no other actors in the system interfere. Again, proving this theorem is out of the scope of the HASPOC project.

7.1.3 Guest/Switch Lemma

The verification of the bisimulation for the runtime phase of the platform covers three cases: execution of a guest process in its virtualized execution environment, switching from the guest into the hypervisor, running the hypervisor handlers and SMCs. The first two are handled by the so-called “Guest/Switch Lemma” which mainly depends on the ARMv8 architecture and the configuration of the platform obtained from the initialization phase.

For every possible step of the guest process we need to show that the step preserves the bisimulation relation between the real and ideal model, i.e., it behaves as if it was running on its own isolated machine. In particular this includes to show that it can only access its own memory and other parts of the system stay unchanged (integrity), and that it cannot obtain leaked information from other guest or the hypervisor (confidentiality). If any of this were possible, the bisimulation relation could not be established since in the ideal model guest steps cannot affect or read from foreign resources directly. To prove the properties we require a detailed model of the ARMv8 architecture which covers all features of the architecture that a guest running in EL1 or EL0 could abuse to break these properties. Then a big case-split over all possible transitions of this model is conducted. Guests in this context are also peripherals which can access the memory using DMA.

The main arguments that integrity and confidentiality are preserved by guest execution steps is the configuration of the second stage address translation and the SMMU, guaranteeing memory isolation between hypervisor and guests. Moreover, the correct setup and virtualization of the interrupt controller ensures that guests cannot send interrupts arbitrarily to other guests or intercept interrupts not meant for them. As noted above these properties are guaranteed by the implementation invariant that has to be maintained by all steps of the system. Additionally the hardware virtualization support plays an important role to ensure that guests can only execute EL1 and EL0 operations.

[4]Everything located in the Secure World must be set up by the boot code.

As the model of the complete platform is highly complex, we follow a decompositional approach to prove the non-interference properties on the architecture. We decompose the model into smaller automata that model the execution of the instruction core, MMU, SMMU, peripherals, the interrupt distributer, and the weakly consistent shared memory. These automata are communicating with each other via message passing and their execution is interleaved arbitrarily according to an external oracle. Also single components (e.g., the shared memory) can be non-deterministic automata and be decomposed further.

This decomposition allows us to easily extend the hardware model without needing to re-verify the entire non-interference theorem for the architecture. Also the proof itself becomes more manageable, since for instance the memory isolation properties are independent of the instruction definition in the processor core component. Furthermore, the decomposition opens the door for automating parts of the proof. We are developing a tool which automatically processes the definition of the instruction semantics for the core component, analyses the possible information flow between core registers, and proves integrity and confidentiality properties in a mechanized fashion. For example, there are a lot of system registers in the core that cannot be accessed in EL1 and EL0. The tool then detects that these registers belong to the confidential part of the state and returns a soundness-checked theorem that indeed a guest can never read or write them.

As mentioned before, the memory isolation is mainly guaranteed by the configuration of the (S)MMU. A separate proof on the (S)MMU component can in deed show, that – if second stage page tables are set up accordingly – it will never generate a memory request that will access a protected region of memory, no matter what requests are sent to it from guests/peripherals.

Special care has to be taken for proving the bisimulation between the virtual memory components and the real shared memory. For instance, all cores store data in the same second level cache. While the set up of the page tables guarantees memory isolation also for the caches, i.e., programs cannot directly read data corresponding to addresses for which they do not have permissions, it can be observed by an attacker (via timing analysis or uncacheable aliasing) whether its own data is stored in the cache or not.

Since the cache is much smaller than the memory, only a subset of each core's data can be stored in it. If space to store new data is running out, old entries in the cache are removed from the cache according to some eviction policy (e.g., least recently used). By filling the cache and measuring the evictions due to accesses from other cores, an attacker can probe if certain address ranges are accessed by the target of the attack and thus identify memory access patterns. Depending on the code executed in the target such patterns may reveal sensitive information like encryption keys, when branches are taken based on this sensitive data, as is the case in contemporary encryption standards like AES. There are several countermeasures against these cache-based channels, e.g., partitioning the cache, however the solution within the HASPOC project is not determined yet. From a verification perspective, cache partitioning is perferable since it allows to disregard the caches in the ideal model after one proves the counter-measure to be effective.

The second case of the Guest/Switch Lemma treats the switch from EL1 or EL0 to hypervisor mode or the Secure World. These switches occur voluntarily, e.g., by a guest executing the hypercall instruction (HVC), by external interrupts that are routed to a higher execution level, by (potentially illicit) actions of the guest that are trapped to a higher level according to the virtualization and

TrustZone hardware support. It has to be shown on the architecture model that these interrupts and exceptions bring the system into a trusted state, i.e., relevant parts of the execution context of the guest are stored in banked registers, the context of the trusted software is restored in the right exception level, and computation continues at the right entry point to the hypervisor code. Thus the right virtualization code is executed, implementing the ideal hypercall/interrupt/exception semantics. The proof of the context switch of the execution level is relatively easy since it only considers the processor core component of the detailed ARMv8 model.

7.1.4 Handler Execution

The final part of proving the real-ideal bisimulation is concerning the functional correctness of the hypervisor handlers. On this level of abstraction we do not perform code verification yet, but only verify that the design of the hypervisor – represented as a transition system resembling the actual implementation of the handlers – behaves exactly like the ideal specification of the handlers as an atomic action.

Standard Hoare logic-style reasoning about pre- and post-conditions can be employed to show the functional correctness. The bisimulation relation couples the abstract hypervisor data structures with their counter-parts in memory. Possibly several refinement steps, showing bisimulation with intermediate handler models of finer step granularity, might be necessary to couple the real with ideal model. Since we still assume a sequential memory for the execution of handler code in the real model, we do not need to care about the weak memory model here.

However, when refining this model to a more realistic one where all memory is in principle weakly consistent we also need to make sure that no sensitive information processed by the hypervisor is leaked to the guests through the caches. This can be achieved for instance by flushing the caches before returning to the guest. Nevertheless also guests executing in parallel on other cores could observe the modifications of shared caches by the hypervisor. Thus there is still an open issue here that is outside the scope of the HASPOC project.

7.2 Code Verification

So far we have only considered the correctness of the hypervisor design on an abstract level. The effect of the handlers is modeled as a transition system closely following their C and assembly implementation. However, strictly speaking, the proofs on this model give no guarantees at all for the actual hypervisor machine code running on the ARMv8 processor.

To close this gap we apply binary verification techniques on the compiled code of the hypervisor. Doing so without having an overlying formal specification of the hypervisor design is tedious and probably impossible to get right. Luckily we have a model that is close to the hypervisor implementation to guide the code verification effort and provide function contracts (pre- and postconditions) which can be discharged using automated tools.

7.2.1 Methodology

The formal specification of the hypervisor design is basically a finite state automaton where each transition represents the sequential execution of a portion of the handler code. Case splits and function calls are natural candidates for start or end points of these transitions, i.e., the states of the automaton. In this way it resembles the control flow graph of the handler implementation.

Now it is relatively easy to map these transitions to the portions of the compiled code that implement them.[5] In particular we define another bisimulation *R'* which describes this refinement (see Fig 23).

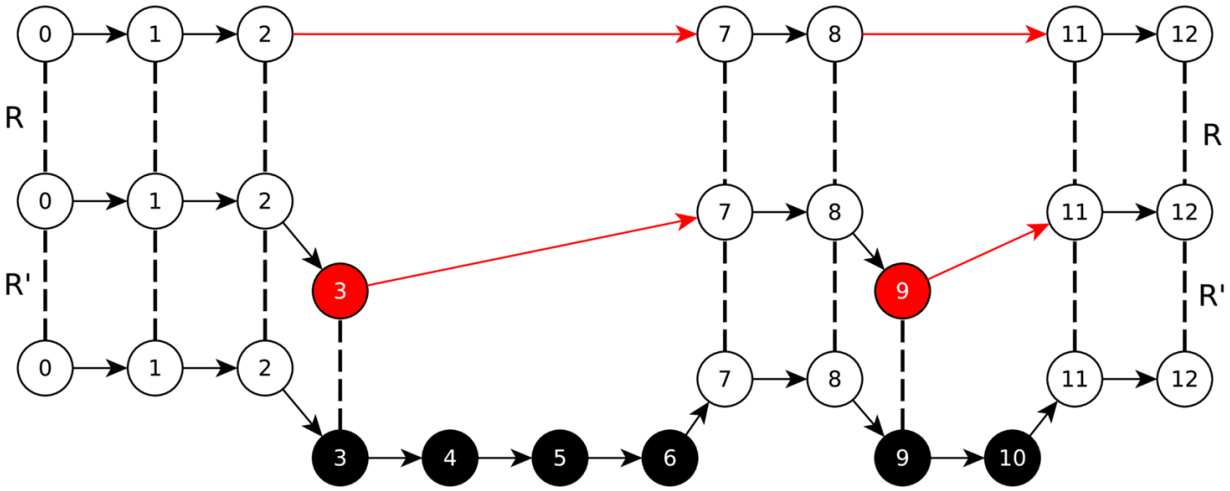


Figure 23: Refinement R' down to the machine code level

Since the states in the real model and the binary verification have the same type (ARMv8 machine configurations), defining *R'* is rather trivial From the semantics of the real model one can derive pre- and post-conditions for the code that allow to show that *R'* is preserved. Formally we define Hoare triples *{P} C {Q}*, denoting that, if in a state where property *P* holds we execute code *C* and it terminates, then property *Q* holds in the resulting state. Here *P* and *Q* are derived from the hypervisor specification and the definition of *R'*. For instance, part of *R'* for the hypervisor and SMC code is that the exception level is the same in the real model and the machine code implementation. If in the real model during the execution of a handler fragment *C* the exception level would change from EL2 to EL3, then *P* would assume that it is EL2 (derived from *R'*) and one had to prove about *C* that it changes the exception level to EL3 in order to satisfy *Q* and reestablish *R'*. In order to verify that the code *C* of a handler actually satisfies the post-condition *Q* for pre-condition *P* a technique called *weakest pre-condition propagation* is used.

[5]Since the compiler employs several optimizations, e.g., optimizing code across function calls and reordering memory accesses, drawing an exact borderline in the compiled code for two subsequent transitions can become hard. However, since we focus mainly on the effect of the handlers on global variables, system registers, and the page tables, rather than on local variables and GPRs (which are the usual targets for compiler optimization), this should still be possible. In case there are problems one can verify less optimized code instead of the one running on the processor. Then however the correctness of the compiler becomes part of the trusted computing base.

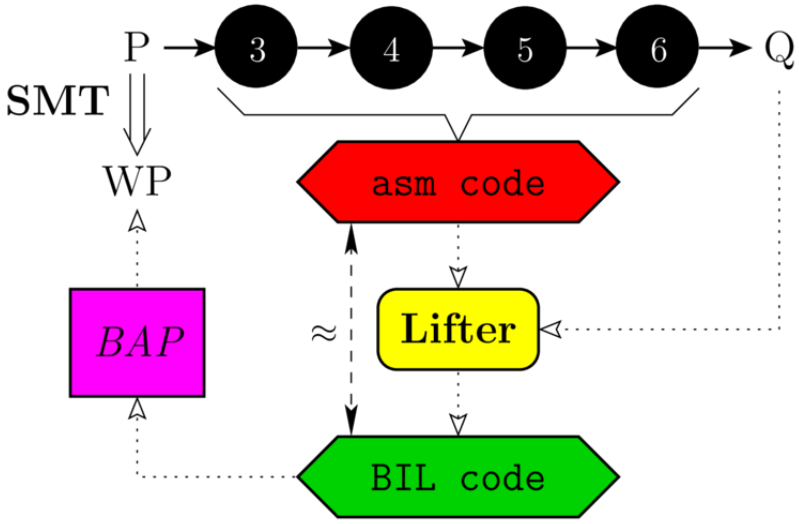


Figure 22: Code verification approach using Lifter, BAP, and SMT solver.

Intuitively, the weakest precondition for code *C* and post-condition *Q* represents the minimal assumptions on the initial state needed, so that *C* terminates and the resulting state satisfies *Q*. Then if *P* implies the weakest precondition, it is a sufficient pre-condition and *{P} C {Q}* holds. Weakest precondition propagation now aims to find the weakest precondition by starting at condition *Q* and going backwards instruction by instruction in *C*, accumulating the weakest preconditions for each step. In our exception level example, the weakest precondition depends on how *C* sets the new level. If it just assigned EL3 to the level (e.g., using an SMC call) the weakest precondition would be empty and trivially implied by P. If it added one to the current exception level (e.g., using an exception that automatically traps into the next higher exception level) the weakest precondition would be that the current exception level is one less than EL3, which is guaranteed by *P*. A tool that implements this method for binary programs is BAP (Binary Analysis Platform).

BAP defines an intermediate language (BIL) abstracting from any specific architecture, and provides tools for the formal analysis of such programs. The given code *C* thus has to be translated from ARMv8 machine code into a BIL program. To this end we are developing a so-called *lifter* based on the ARMv8 hardware model in HOL4. It takes the desired Hoare triple *{P} C {Q}* and translates it into an equivalent BIL program annotated with pre- and post-conditions. Additionally one can add intermediate assertions representing additional verification conditions on *C*. To ensure that the result of the translation is correct, the lifter also generates a corresponding theorem, stating that the program behaves equivalently in BIL and in the ARMv8 model.

After translating *C* into BIL we can run the analysis to generate the weakest precondition for *Q*. Then a SAT solver is used to show automatically that *P* implies the weakest precondition and we obtain *{P} C {Q}* as a proven theorem in HOL4. Manual effort is usually needed to massage the formulation of *P* and *Q* in HOL4, so that the automated prover can handle them. Specifically, one must transform them into quantifier-free, first order logic statements. The overall approach is illustrated in Fig 24.

In this way one proves the correctness of all transitions of a handler and combines the corresponding Hoare triples into one that describes the complete execution of the handler. The pre- and post-conditions imply that the refinement relation *R′* is preserved by the handler code and it is

plain to see that the compiled machine code of the hypervisor correctly implements its specification.

Moreover, the verification conditions are shown to hold for the execution of the hypervisor code. Such verification conditions are usually necessary to prove the soundness of the overall approach, e.g., we need to show that the hypervisor does not modify its own page table while running on it, breaking the sequential consistency of memory assumed in the real model. They are the conditions under which one can prove that the sequential ARMv8 model is a sound and complete abstraction of the detailed weakly consistent model.

We apply our code verification methodology on the hypervisor as well as on the secure boot code. The main effort lies in showing that the boot code and hypervisor initialization routine correctly establish the implementation invariant. For the hypervisor handlers the main goal is to show that they do not break this invariant, which can be as easy as showing memory safety and control flow integrity, where the code does not touch page tables or interrupt controllers (which is usually the case). Functional properties only need to be proven to be correctly implemented if they result in behavior that is visible in the ideal model, i.e., the delivery of inter-guest interrupts for the IGC mechanism.

7.2.2 Hypervisor Handlers

In the verification of the hypervisor code, however, several additional challenges arise in applying the verification methodology described above. First of all the hypervisor code is executed concurrently on several cores. If they would never interact we could simply verify the handlers like a sequential program. However, there are shared resources, like the interrupt controller, which are accessed concurrently. Therefore the hypervisor needs to implement a locking discipline for shared hypervisor resources and its correctness must be proven.

There are several established methods to handle concurrency in code verification. Most notably there is rely-guarantee-based reasoning and concurrent separation logic. It is yet undecided what verification methodology we will employ in the HASPOC project, however it is clear that any existing method needs to be adapted to support reasoning about low-level ARMv8 code.

In general, if concurrent access is limited to a few explicit shared resources all code that does not touch them can be handled by sequential code verification, assuming that all handlers follow this programming discipline, i.e., they do not modify the local state of hypervisor handlers running on other cores. Shared resources are protected by locks which implement exclusive access to them, i.e., the lock guards the entrance to a so called *critical section* so that only one handler at a time can manipulate the protected resource.

Acquiring and releasing a lock requires low-level synchronization primitives which are inherently racy. Thus their correctness needs to be proven in the detailed concurrent machine model with weakly consistent memory. If the locking primitives (acquire, release) in deed guarantee exclusive access, then one can integrate them and the critical sections into the sequential code verification methodology, if one shows that all handlers obey the following locking policy: *Shared resources may only be accessed if they have been acquired before*.

In this case the handler code can be split up into chunks which contain at most one critical section (and the accompanying locking primitives). These blocks of code are then arbitrarily interleaved in

executions of the ideal model, which means in turn that hypervisor handlers containing more than one critical section cannot be modeled as atomic transitions. Instead they are executed in several abstract atomic steps, similar to the situation in the real model.

Showing that this abstraction is sound requires a reordering proof between the real model and the detailed, weakly consistent ARMv8 model. We do not go into the details of such a proof here.

Another issue in the handler verification are interrupts. If handlers are interruptible, i.e., the hypervisor is preemptive, they can be interrupted at any time, especially in the middle of a block of code that is supposed to be atomic. One can reorder the interrupt handling to the beginning or the end of the block assuming that they do not touch shared resources or mess with the local state of the interrupted handler. However the proof of this theorem is non-trivial and even harder if handlers actually can interfere with the shared resources. To avoid this complexity the HASPOC hypervisor is not preemptive at the moment and there is another verification condition that the hypervisor code does not cause any exceptions itself, e.g., data or prefetch aborts.

In addition to the functional correctness of the handlers one must also consider information flow properties. Execution of a handler for one guest should not be visible to others and the caller should not be able to leak information about other guests through the hypercall. In a sequential model, proving this is rather trivial and already proven by the real-ideal bisimulation, arguing that the handler implementation does not read from or write to other guests' private memory. However, in the weakly consistent memory model, where the shared caches become visible proving the non-interference for the handlers becomes a real challenge. There are several solutions to this problem, including cache partitioning, but we omit a deeper discussion here.

7.2.3 Secure Boot and Initialization

The code verification of the boot and initialization code is conducted as described in the methodology section. Since only one core is executing with interrupts disabled we do not have to care about concurrency or preemption. However, a large portion of the code is configuring peripherals like the DRAM or TrustZone controllers. Since we do not wish to provide models for such potentially complex devices, we do not verify functional correctness for these portions of the code, but only that they do not break confidentiality of the information, e.g., the root key that is used to check the signatures of the modules that are loaded by the boot process. Similarly we do not verify the correctness of the implementation of the cryptographic functions.

Thus the code verification focuses mainly on the setup of the page tables and the copying of the right data to the specified locations. Still there is an issue here which cannot be handled by our automated code verification approach. As boot and hypervisor initialization codes set their own page tables and activate the MMU, it breaks some of the verification conditions guaranteeing sequential consistency. Basically the code changes the execution context that is assumed by the automated verification with BAP. In order to handle the part of the code that switches the context, we need to perform manual code verification in HOL4 based on the detailed ARMv8 model. After the new execution context is established, automated code verification can continue in the new context.

Obviously these context switches are costly. Therefore we verify a simplification of the boot process based on the ARM Trusted Firmware, skipping the switch from EL3 to Secure EL1 and

back, to run phase BL2 of the boot process, before the hypervisor is called in EL2. Instead the complete boot code is running in EL3 and reuses the page tables set up in phase BL1 also for BL2. Additionally there is third-party code borrowed from the ARM trusted firmware reference implementation used in order to configure the hardware platform and set up the SMC handlers for power supply control. Some of this functionality is implemented by executing binary images on memory controller co-processors. Since these are basically black boxes we need to add them to the trusted computing base. For all other code we can actually prove that it does not break the setup or leak information.

In the hypervisor initialization a similar context switch as described above occurs when it installs its own page tables and turns on the MMU (again); we handle it in the same way. Another challenge here is the configuration of the interrupt controller and the SMMU. We need to provide HOL4 models for both components to ensure that the hypervisor code configures them correctly according to the isolation requirements. While both interrupt controller and SMMU are external components that need to be configured through memory, the configuration is implemented as a one-way communication, therefore we do not need to model it as a concurrent process but we can reorder the responses of both peripherals to the configuration commands to the end of the initialization phase.

Apart from these issues the code verification of boot and initialization is straight-forward.

7.3 ARMv8 Hardware Model

We need a formal model of the ARMv8 architecture for several purposes. First and foremost, our code verification approach is based on a formalization of the ARMv8 architecture in HOL4. Besides that, the top-level specification and the real model are specified as transition systems over (virtualized) ARMv8 processor cores and a shared (partially weak) memory. Finally, for proving the soundness of our approach we need a detailed model of the hardware including weakly-consistent memory that can be reduced to the “real” model under certain verification conditions on the hypervisor code. We can only be sure to have identified a sufficient set of conditions if we prove a bisimulation between the two models. Below we describe the modeling approach, the versions of the ARMv8 architecture we obtain, as well as the evaluation of the models.

7.3.1 Modeling approach

For producing the models we use a domain specific language called L3 which was developed by Anthony Fox in Cambridge. It is an imperative language with native support for modeling low-level hardware functionality such as bit manipulation and instruction decoding. L3 programs can be translated into a HOL4 theory as well as an SML program. Using the MLton compiler we can even generate executable binaries from SML.

Several architectures have already been specified in L3, e.g., MIPS, ARMv3-v7, as well as a subset of x64. There also already exists an L3 model of ARMv8, covering user level instructions. We have enriched this model with system level instructions as well as a model of the two-stage memory management unit.

For evaluation of the original model in HOL4, Anthony Fox also provided a collection of so called *step theorems* which give a simplified Hoare-style semantics for each instruction of the architecture. These step theorems are very useful for proving properties of ARMv8 code in HOL4, as well as for

constructing the Lifter to BAP explained above. For the extended model we need also to adapt these step theorems in order to be able to obtain a verified BAP lifter for ARMv8.

The translation to SML is used to construct a simulator of the model, allowing us to test the model itself and explore the possible execution traces for a selection of test programs. See the evaluation section for more details and Fig 25 for an illustration.

7.3.2 SHARMA8 model

The first model we obtain is the Sequential HASPOC ARMv8 Architecture (SHARMA8) model. It is a straight-forward extension of the original Cambridge model with system instructions. However the memory is still sequential and uses virtual addresses. The main addition is a large number of system registers that can now be set and retrieved using dedicated system level instructions. Additionally the model supports cache/TLB invalidation instructions and memory barriers. However these instructions have no effect, since we assume a sequential memory with transparent caches and TLBs/address translation at this level of abstraction.

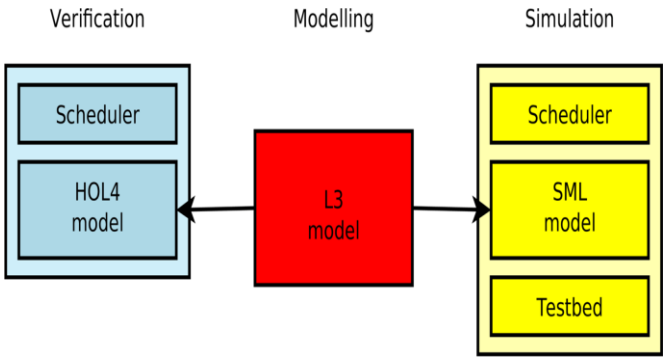


Figure 23: Hardware modelling for verification and simulation using L3.

The SHARMA8 model is used as the base for hypervisor code verification and the construction of the verified BAP lifter. To be useful we also need to provide step theorems for the new instructions in HOL4. The original step theorems are still usable for all other instructions as we have not changed their semantics. In the analysis with BAP we need to add verification conditions that ensure that the verified code does not change the execution context or break the sequential consistency, e.g., by reconfiguring the MMU or bypassing the caches. Only then the code verification approach is sound.

One particularity in the SHARMA8 model is the treatment of AT instructions. These are used to obtain an address translation from virtual to intermediate or physical addresses for a given exception level that is equal to or lower than the current one. This functionality is useful for the hypervisor which can use it to simulate the address translation of a guest and deduce the targeted intermediate or physical address corresponding to data faults or other exceptions that occurred in the guest. The instruction is forwarded to the MMU which performs the translation for the given address as if the processor would run at a different exception level, then the result is returned to the core and it updates the PAR_EL1 register accordingly.

Since we do not want to add a sequential model of the MMU to the SHARMA8 model, we underspecify the effect of AT instructions. If it is used for a lower exception level than EL2, we

simply set the PAR_EL1 register to an unknown value. This is sound since the hypervisor never uses the result of the translation for memory accesses directly. Instead it performs a case split on the address, determining which part of memory the guest was trying to access. As we are not trusting the guest we assume that it will try to access every possible address in the system, therefore we need to verify the correctness of the handler for all possible values of PAR_EL1 anyway.

We do not model AT instructions for EL2 or EL3 since they are not used in the hypervisor code.

7.3.3 DHARMA8 model

The Decomposed HASPOC ARMv8 Architecture (DHARMA8) is used as the base for our simulation stack. It is developed to model all the possible low-level behaviours of the ARMv8 architecture and peripherals on the SoC. Specifically it should capture the effects of the weakly consistent memory that is shared between all observers in the system. A weakly consistent memory is different from a sequentially consistent memory in that different observers may “see” shared memory accesses of others in different orders. In particular memory coherence, i.e., the guarantee that the latest written value to a memory address is returned upon a read, is usually only guaranteed for a subset of addresses or even just every single address in such systems.

In order to handle the complexity of such a model, we decompose it into several components which are communicating through message passing. Components are for instance processor cores, MMUs, and the memory subsystem. The latter can be further decomposed into, e.g., caches, physical memory, peripherals, and SMMUs.

The benefit of such a decomposed model was already explained in the section about the Guest/Switch Lemma. Instead of having a monolithic model where each modification prompts a re-verification of the whole non-interference theorem of the architecture, we can modularize the verification effort and split the information flow property of the whole system into sub-properties that can be verified separately for each new instantiation of a component, without having to touch the overall theorem.

Another benefit of such a decomposed model is the possibility to interleave steps of the components non-deterministically. This is necessary to represent multi-core systems and the weakly consistent memory model.

At the moment the DHARMA8 model still contains a sequential memory, the MMU has no TLB yet, and the core does not support out-of-order instruction execution. These features will be added incrementally in the near future, but outside of the HASPOC project.

7.3.4 Evaluation

The main target of evaluation is the DHARMA8 model. Since it is very complex, we need to make sure by extensive testing that our model actually resembles a real ARMv8 system. To this end we have defined a simulator for the model in SML which allows to load test programs (ELF files) into the model's memory, set up page tables, and execute the model in different granularities of steps.

Our aim here is two-fold. On the one hand we want to be able to execute test programs on the model to spot possible inconsistencies with the Architecture. As a reference we can use other ARMv8 emulators or actual ARMv8 hardware. In the future we may construct a stand-alone simulator that is able to run larger programs and provide outputs through a UART driver.

On the other hand, to exercise the weakly ordered memory model by exploring all possible interleavings of the model for the execution of small concurrent programs, e.g., the ARM barrier litmus tests. For these programs we know all the possible allowed outcomes so that we can validate our model.

8 Demonstrators

To demonstrate the capabilities of the HASPOC platform in solving realistic security related issues, two demonstrators were developed on the 96boards Hikey. The Hikey board is intended for Android development and features 8 ARMv8 Cortex A53 cores, 2GB RAM and 8GB eMMC storage. One of the factors for choosing this hardware as the platform for the demonstrators was the feature of an S-MMU which protects against attacks using DMA access of a peripheral. Without an S-MMU a guest can use the peripheral DMA to write to any part of memory, thereby circumventing the hypervisor's control and opening up to any type of attack. As it turned out the S-MMU on the Hikey was limited to the media block only and did not work as intended when choosing the platform. The consequence of this limitation is that each guest with access to a peripheral must know the physical address space it is able to use, otherwise the guest cannot setup a DMA area with correct addresses. In other words, there are two limitations to this platform: 1) guests with access to a peripheral are given it's location in the actual physical address space instead of a virtualized address space starting from 0x0, 2) a guest with access to a peripheral could potentially use the peripheral's DMA to overwrite another guest's memory. Note that the Hikey board therefore is not a piece of HASPOC compliant hardware, with a HASPOC compliant S-MMU these weaknesses does not exist.

8.1 Tutus demonstrator – Secure smartphone

The Tutus demonstrator was designed to show more specifically how the HASPOC platform can be used to isolate security critical services on a smartphone, and to some extent investigate which performance penalties such a design might entail.

8.1.1 Background

As mentioned in 2.2.1 Secure Smartphone it is possible to achieve a red black separation on a smartphone where the main OS (Android) is considered a red domain and all information leaving the phone is considered sensitive so that it needs to be encrypted (black). By establishing an encrypted tunnel (Virtual Private Network, VPN) an internet connected smartphone can have secure access to sensitive resources on an internal network. The tunnel is established by running a VPN service on the cell phone, which would run in Android or possibly in the underlying Linux kernel. This means however that the safety of the VPN service is dependent on the safety of the operating system (OS) it is running in. If there is a security hole in the OS the VPN can be circumvented or even worse, cryptographic credentials along with sensitive information can leak. By isolating the VPN service from any OS, the safety of the VPN is dependent on only the service itself and the isolating properties of the hypervisor.

8.1.2 Solution design

The design of the demonstrator for an isolated VPN service is similar to the already described red black separation, an illustration is shown in Fig 26. The main OS is Android (Android Open Source Project, AOSP, specifically) which has access to all peripherals except for communication devices.

The second middle guest is a VPN service running bare bone and in memory only. The third and last guest is a minimal Linux kernel also running in memory only operating the communication hardware. The middle guest communicates with a VPN server and negotiates parameters for the symmetric encryption. By using e.g. Diffie-Hellman key exchange the third Linux guest while seeing all communication cannot derive the key used for symmetric encryption. Since the VPN service is running as a bare bone guest and given that the hypervisor isolation properties hold, the VPN parameters cannot leak to either Android or the Linux kernel. Furthermore the Android guest cannot circumvent the VPN as all communication from Android passes through the middle guest and is encrypted, before arriving at the Linux kernel and the communication devices – meaning the VPN is enforced under all circumstances.

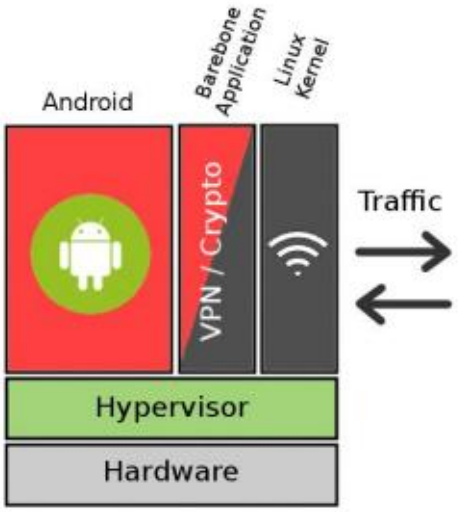


Figure 24: Tutus design for a secure VPN service on a smartphone.

8.1.3 Implemented solution

The Android guest was given 4 of the cores and 1664 MB of the memory with access to all peripherals except WiFi and Bluetooth (integrated on the same chip/device). The Linux guest was given 2 cores, 256 MB of memory and access to the Wifi/Bluetooth device. Both the guests need shared access to power management, a few clocks, etc. which is handled by virtual drivers as part of the hypervisor. The VPN guest was given 2 cores, although used only 1 core, and 56 MB of memory.

The VPN guest was limited in certain ways to simplify development. A real VPN was never established but instead simulated, incoming traffic was indeed encrypted however the plain text data was forwarded unmodified and vice versa for decryption – thereby achieving the same delay caused by encryption as when using a real VPN. The simulated encryption was performed using the OpenSSL implementation of AES128, including the possibility to use the ARM ISA cryptography extension for accelerated performance.

Another limitation to the demonstrator was the non-existence of a communication channel between the Android and Linux guest. Such a channel, using a small well defined protocol, must exist to be able to display and log in to available networks, toggling WiFi on and off, getting antenna signal strength, etc.

Communication between the guests are performed as described in 5.2.1.2.1 Inter-guest Communication, using a hypervisor controlled shared memory area. A virtual Ethernet driver was

developed for Linux so that the Android and Linux guest could use this shared memory as a network device. The shared memory area was implemented as a ring buffer containing IP packets. The guest on the other side is alerted by getting a virtual interrupt when there is a packet to read, efforts were made to minimize the number of interrupts and optimize performance by allowing multiple packets in queue before sending interrupts and using pre-allocation of resources when possible. The VPN guest could read and write directly to the shared memory although using the same data structure as for the Linux driver. The Linux guest used **ip-tables** to relay traffic between the virtual Ethernet and Wifi device.

8.1.4 Performance

With the virtual Ethernet driver the bandwidth between the Android and Linux guest was measured using **iperf** to around 700-750 Mbit/s using TCP with the middle guest simply forwarding traffic. When using the ARM ISA extension for encryption the bandwidth was around 650 Mbit/s while performing cryptography using software only gave a bandwidth of around 140 Mbit/s. The latter is similar to what has been measured when running a VPN service without a hypervisor as a service in Android and using software encryption.

The demonstrator was connected to a 23-inch monitor with a touchscreen interface to allow for an experience more similar to a phone or tablet. The perceived experience is identical to running Android directly on the Hikey board, including playing Android games and displaying videos in HD-quality. With the WiFi chip on the Hikey board (TI WL1835MOD) having a listed throughput of 80 Mbit/s for TCP and 100 Mbit/s for UDP, there was as expected no measured difference between using the demonstrator and running Android directly on the hardware.

Due to the nature of the Hikey board and to limitations of available time an analysis of any added power consumption using the demonstrator compared with bare metal was not performed.

8.1.5 Extending the demonstrator

As the only necessary modification done to Android to be able to run it on top of the hypervisor is making changes to the device-tree used by Linux, this isolation of a service is not limited to using Android as main OS. As a proof of concept and showing the versatility of the HASPOC platform two similar demonstrators were developed using the exact same configuration however using either Debian with LXDE desktop or Ubuntu Server as main OS instead of Android.

8.2 Sectra demonstrator

The Sectra demonstrator focuses on how to incorporate the HASPOC platform in a networked cryptographic system, given requirements from standards such as FIPS 140-2.

8.2.1 Background

As described in 2.2.2 Network Security Appliance a red-black separation can be used to achieve communication between two red domains (i.e. domains handling sensitive data) over a public network. The separation can be used to ensure that only legitimate traffic enters or leaves a device and that no sensitive data leaks to the public network. This can be achieved through a VPN as described in 8.1.1 Background and the same reasoning holds for this scenario as well. However, the solution does not need to be a VPN, since the same reasoning holds for network communication between two devices in general. However, for this demonstrator a VPN solution was implemented.

8.2.2 Solution design

The demonstrator hardware consists of the ARMv8 development board Hi-Key and a separately developed hardware, called a daughterboard, which is used to address certain security requirements. The daughterboard consists of two microcontrollers that mainly handle the system monitoring and software upgrade functionalities.

On the HASPOC platform the guests consists of a red guest, a crypto guest, and a black guests, as illustrated in Fig 26. In addition to interfacing with a trusted network/device the red guest also interacts with the daughterboard. The interaction consists of heartbeat messages from the red guest to the daughter board and notifications from the daughterboard to the red guest, e.g. tamper tamper events. The system was designed to minimize the number of peripherals used and ensure that there is no overlap in the usage of peripherals. An example of this is that only one guest can use the USB ports. Even though there are two ports they are controlled by the same hardware peripheral.

The demonstrator communicates with the VPN server through an encrypted tunnel. The black guest performs all of the networking activities and the encryption guest is responsible for all the cryptographic operations both during and after tunnel establishment as well as transmitting data between the red and black guests. The red guest controls the overall behavior of the guests, whereas the daughterboard ensures that the system enters a secure state if an error event occurs.

8.2.3 Implemented solution

The hardware was setup with a serial device communicating with the Red guest using an UART interface. The black guest was in turn connected to a network using a 10/100Mbps USB-to-Ethernet converter.

Each guest was implemented with a Linux OS running on one core each. The communication is performed as described in 5.2.1.2.1 Inter-guest Communication, using a hypervisor controlled shared memory area. A Linux driver was developed to handle the communication for each guest. The shared memory was implemented as a ring buffer containing TLV-encoded messages and a guest on the other side of a communication channel is alerted by getting a virtual interrupt when there is a packet to read. The communication was setup to be synchronous, which is not optimal from a performance perspective.

A fully functioning VPN was implemented using mbedTLS with AES128 and a Diffie-Hellman handshake.

8.2.4 Performance

No actions were taken to optimize the Sectra demonstrator. However, the bandwidth of the data flow from the red guest to the VPN server was measured to 50Mbps. Specific areas to optimize would be the IGC drivers, which were quite simplistic, and the implementation of the crypto guest.

8.2.4 Extending the demonstrator

The HiKey development board is lacking in terms of hardware. A hardware solution with two Ethernet ports that are not part of the same hardware block would be necessary to achieve a higher bandwidth. In general, a system where several guests want to use several peripherals it might

become necessary to develop custom hardware.

The encryption guest could also be optimized further by hardware acceleration or using a baremetal implementation.

9 Benchmarking

Even though there has been little to no effort to optimize the performance of the HASPOC hypervisor some benchmarking has been performed anyway – there is e.g. most likely room for improvement regarding context switching between guest and hypervisor. The following section includes benchmarks like boot time comparisons, cost of context switches between guest and hypervisor as well as results from a more extensive test suite ranging from I/O to CPU and memory performance where the HASPOC platform is evaluated against running bare bone.

9.1 Secure boot

The most important boot benchmarking parameter is elapsed time, where the total boot is the sum of the cold start phase and the storage phase. The cold start phase starts when reset is released and ends before external storage is initiated. The storage phase starts when media controller is initiated and ends when control is passed to the hypervisor. There is a performance register (pmcr_el0) described in the ARM Architecture Reference Manual from which you can read the number of elapsed clock cycles and which was used to acquire the following results.

Use case	**Total time**	**Cold start phase**	**Storage phase**	**Comment**
15 Mb signed HSBF	4.866 s	1.929 s	2.937 s	Sha256 / RSA2048
15 Mb signed HSBF	4,126 s	2.359 s	1.767 s	Verification disabled
15 Mb FIP image	4.941 s	-	-	Arm Trusted Firmware

For large HSBF images the time depends on two factors: IO Bandwidth for storage device (SD / MMC) and the Sha256 calculation.

9.2 Context switching

The same register, pmcr_el0, was used in conjunction with a custom guest developed with the soul purpose of making performance measurements regarding hypervisor driver latency and context switching when doing a hypervisor call (HVC) and secure monitor call (SMC). For HVC and SMC the measurements were made using a loop making 4 null calls each round for 1000 rounds (i.e. 4000 calls). The driver latency measurements used a dummy driver which was triggered 1000 times in a loop. The cycle counts reported below is the average for one call. The time reported was calculated using the number of CPU cycles and with the assumption that the CPU was running at a speed of 1.2 GHz as according to specification.

Use case	**CPU cycle count**	**Time**
HVC	13955	11.6 µs
SMC	13968	11.6 µs
Driver latency	12917	10.8 µs

9.3 Caliper test suite

The Caliper test suite consists of several well known and commonly used functional tests and benchmarking tools. It is part of Open-Estuary, a “complete open source solution for ARM based systems” which provides a “total solution for general-purpose computer based on aarch64 architecture” [OPENEST]. Open-Estuary currently supports Huawei D01 and D02 boards as well as 96boards Hikey. Caliper was used to compare performance on the HASPOC platform compared to bare bone performance both in terms of functionality and benchmark performance. For more detailed information please visit http://open-estuary.org/caliper-benchmarking/.

The tools used in the Caliper test suite are cross-compiled and administered by a host which is connected by network to the Hikey board. For these tests a Lenovo Thinkpad X230 running Ubuntu 14.04 was used as the host with it's Wifi set up as an Access Point to which the Hikey board connected to. The Hikey itself ran Ubuntu Server 15.04 (file system and kernel downloaded from http://download.open-estuary.org/).

Presented below is the results from a selection of benchmarking tools, mostly concerning low-level performance for Memory and CPU. Bare metal performance is used as reference compared to running on the HASPOC platform with access to all peripherals. A comparative percentage is given as well as the raw results. The tool **cachebench** measures the memory bandwidth performing vector operations of different lengths. The final result is the peak performance of memory bandwidth with optimal cache reuse. **CoreMark** is a tool for measuring the performance of the CPU by doing list processing, matrix manipulation, state machine operations and CRC. Another tool for CPU performance is **Dhrystone** which consists of a set of programs which simulates CPU usage of typical applications. **NBench** is a suite of ten different benchmarking tasks as given in the table below. Finally **Iperf** is a tool for measuring network bandwidth over TCP/UDP. Higher value in the following table always mean better.

Test performed	**Bare metal**	**HASPOC platform**	**% Overhead**
cachebench: read bandwidth	2124.2 MB/s	2105.1 MB/s	-0.90%
cachebench: write bandwidth	3011.0 MB/s	3001.8 MB/s	-0.31%
cachebench: read/modify/write bandwidth	2164.5 MB/s	2156.6 MB/s	-0.36%
CoreMark	3622.1 iter/s	3610.8 iter/s	-0.31%
Dhrystone: VAX MIPS rating	7348.4 MFLOPS	7328.3 MFLOPS	-0.27%
Dhrystone: MWIPS	1052.1 MFLOPS	1049.1 MFLOPS	-0.28%
NBench: Fourier coefficients	8882.5 iter/s	8852.7 iter/s	-0.34%
NBench: LU decomposition	456.68 iter/s	452.55 iter/s	-0.90%
NBench: Neural net	14.401 iter/s	14.229 iter/s	-1.20%
NBench: Assignment algorithm	10.419 iter/s	10.380 iter/s	-0.37%
NBench: Bitfield manipulations	1.7552 iter/s	1.7493 iter/s	-0.34%
NBench: FP Emulation	148.26 iter/s	147.77 iter/s	-0.33%

NBench: Huffman decompression	969.49 iter/s	965.90 iter/s	-0.37%
NBench: IDEA encryption	2585.5 iter/s	2577.5 iter/s	-0.31%
NBench: Numeric sort	548.54 iter/s	538.63 iter/s	-1.81%
NBench: String sort	133.29 iter/s	132.83 iter/s	-0.35%
Iperf: mean value of 9 measurements	1.381 Mbits/s	1.193 Mbit/s	-13.60%

As expected the results regarding memory bandwidth and CPU performance are close to running bare metal, while network bandwidth is not. A possible explanation to this lower performance is presented later in this section.

The following table shows results from running **lmbench** covering different areas of benchmarking divided into categories shown on the left in bold font. The suite consists of simple, portable, ANSI/C microbenchmarks for UNIX/POSIX. It mainly measures latency and bandwidth including context switches, the latter is especially of interest when running with a hypervisor.

	Test	**Bare metal**	**HASPOC Platform**	**Overhead**
CPU	(Lower value is better)			
multicore_double:	double_add	3.33 ns	3.34 ns	+0.30%
	double_bogomflops	27.57 ns	27.64 ns	+0.25%
	double_div	18.32 ns	18.36 ns	+0.22%
	double_mul	3.33 ns	3.34 ns	+0.30%
multicore_float:	float_add	3.33 ns	3.35 ns	+0.60%
	float_bogomflops	20.08 ns	20.12 ns	+0.20%
	float_div	10.82 ns	10.85 ns	+0.28%
	float_mul	3.33 ns	3.34 ns	+0.30%
multicore_int:	integer_add	0.83 ns	0.84 ns	+1.20%
	integer_bit	0.83 ns	0.83 ns	0%
	integer_div	5.00 ns	5.01 ns	+0.20%
	integer_mod	5.00 ns	5.01 ns	+0.20%
	integer_mul	-0.00 ns	0.00 ns	0%
Latency	(Lower value is better)			
ctx	16p/16K context switch	9.23 µs	40.18 µs	+335.32%
	16p/64K context switch	22.38 µs	52.07 µs	+132.66%
	2p/0K context switch	6.64 µs	35.78 µs	+438.86%
	2p/16K context switch	7.10 µs	36.45 µs	+413.38%
	2p/64K context switch	6.52 µs	36.17 µs	+454.75%
	8p/16K context switch	7.51 µs	37.35 µs	+397.34%
	8p/64K context switch	11.41 µs	41.80 µs	+266.35%
file/vm	0k file create	35.79 µs	36.09 µs	+0.84%

	0k file delete	30.52 µs	30.40 µs	-0.39%
	100fd select	6.54 µs	6.57 µs	+0.46%
	10k file create	77.11 µs	77.15 µs	+0.05%
	10k file delete	46.00 µs	46.53 µs	+1.15%
	Mmap (KB)	21.35 µs	22.26 µs	+4.26%
	Page fault	1.69 µs	1.73 µs	+2.37%
	Prot fault	0.28 µs	0.36 µs	+28.57%
mem	L1	2.51 ns	2.51 ns	0%
	L2	6.54 ns	6.56 ns	+0.31%
	Main Memory	25.32 ns	25.93 ns	+2.41%
process	exec proc	1564.75 µs	1660.25 µs	+6.10%
	fork proc	535.60 µs	592.22 µs	+10.57%
	null IO	0.98 µs	0.95 µs	-3.06%
	null call	0.19 µs	0.17 µs	-10.53%
	open close	5.17 µs	5.19 µs	+0.39%
	sh proc	3403.00 µs	3730.00 µs	+9.61%
	sig hndl	2.76 µs	2.79 µs	+1.08%
	sig inst	0.41 µs	0.42 µs	+2.44%
	slct TCP	19.53 µs	19.57 µs	+0.20%
	stat	1.95 µs	1.96 µs	+0.51%
Network	(Lower value is better)			
local lat	AF Unix	20.88 µs	77.33 µs	+270.35%
	Pipe	18.03 µs	76.36 µs	+323.52%
	TCP	54.55 µs	116.23 µs	+113.07%
	TCP con	111.74 µs	128.17 µs	+14.70%
	UDP	43.54 µs	103.51 µs	+137.74%
Memory	(Higher value is better)			
local speed	AF Unix	2153.91 MB/s	2077.63 MB/s	-3.54%
	Bcopy (hand_par)	859.82 MB/s	929.93 MB/s	+8.15%
	Bcopy (hand)	1466.76 MB/s	1446.16 MB/s	-1.40%
	Bcopy (libc)	1658.34 MB/s	1648.58 MB/s	-0.59%
	Bcopy (libc_a)	1650.53 MB/s	1671.79 MB/s	+1.29%
	Bzero	5723.45 MB/s	5704.47 MB/s	-0.33%
	File reread	1090.71 MB/s	1093.64 MB/s	+0.27%
	Mem RW par	1742.49 MB/s	1647.56 MB/s	-5.45%
	Mem read	1783.87 MB/s	1751.64 MB/s	-1.81%
	Mem read par	1742.49 MB/s	1647.56 MB/s	-5.45%
	Mem write	4716.01 MB/s	4691.53 MB/s	-0.52%
	Mem write par	1958.77 MB/s	1743.01 MB/s	-11.02%

	Mmap O2C	1460.99 MB/s	1459.41 MB/s	-0.11%
	Mmap reread	2669.96 MB/s	2646.20 MB/s	-0.89%
	Pipe	854.15 MB/s	667.99 MB/s	-21.79%
	Reread O2C	1091.68 MB/s	1091.86 MB/s	+0.02%
	TCP	749.91 MB/s	643.29 MB/s	-14.22%

The results that stand out are network performance and latency for context switches. Note that for the Latency:ctx results there is a constant performance penalty of close to 30 µs. We believe that these results relate to the Linux rescheduling interrupts (SGI-0) which trap to the hypervisor before it is forwarded to the guest. This is specific to the GIC used on Hikey, with a GIC that could forward this kind of interrupts directly to the guest we would expect to see similar performance in these areas as for the other areas. Of course faster context switching would be desirable as well, however as already mentioned not much effort have been spent at this stage of development towards optimizing the hypervisor performance.

Part of the Caliper test suite is also the Linux Test Project (LTP) which consist of regression and conformance tests designed to confirm the behavior of the kernel and glibc. For these tests running bare bone actually performed worse than the hypervisor. Two tests for CPU concerning scheduling failed when running bare bone but passed when running on the HASPOC platform. Similarly when running bare bone two tests related to Process ID namespaces while on the HASPOC platform they passed. Also all tests (8 of them) concerning Posix Message Queue namespaces were skipped when running bare bone, but succeeded when running on the HASPOC platform. The reason for these results are unclear. The following table presents a summary of the LTP test, 'bm' means 'bare metal' and 'hv' means HASPOC platform with the hypervisor.

LTP test	**bm PASSED**	**bm FAILED**	**bm SKIPPED**	**hv PASSED**	**hv FAILED**	**hv SKIPPED**
cpu	8	4	2	10	2	5
dio	37	8	2	37	8	2
memory	39	25	26	39	25	26
filesystem	80	2	60	80	2	60
kernel	25	14	8	35	12	0
proc	40	4	2	40	4	2
Total	229	57	103	241	53	95
in precent	58.87%	14.65%	26.48%	61.95%	13.62%	24.42%

10 Summary and Conclusions

We have presented a solution for isolating critical from commodity software on ARMv8, while still providing controlled information flow between the guests. The virtualization based approach allows secure resource sharing between different parties and the minimization of the trusted computing base that services need to rely on. For example, we can enforce encryption between network domains, even in cases where the sending commodity system and the network drivers are compromised. Particularly strong assurance is achieved by a holistic trust framework. The

hypervisor is small in size, currently 8 kLOC and 38 KB and thus suitable for formal verification on binary level. Its uncompromised execution is rooted in a secure boot scheme following ARM Trusted Firmware. The overall solution is suitable for Common Criteria certification and we provide the required starting points towards an EAL6 certification of a concrete product which is based on this high assurance virtualization platform. More information on the ongoing research project and open source releases will be made available via [HASPOC].

Acknowledgment The HASPOC project presented herein is in part funded under contract 2014-00702 of the Swedish Governmental Agency for Innovation Systems' Challenge-driven Innovation Programme.

10 References

[ARMTF] ARM Trusted Firmware, http://git.linaro.org/arm/arm-trusted-firmware.tar.gz

[ARMv8] http://infocenter.arm.com/help/topic/com.arm.doc.ddi0487a.h/index.html

[BL] D. E. Bell and L. J. LaPadula. "Secure Computer Systems: Mathematical Foundations", 1973.

[GNBD] R. Guanciale, H. Nemati, C. Baumann, and M. Dam, "Cache storage channels: Alias-driven attacks and verified countermeasures". In IEEE Symposium on Security and Privacy, SP, 2016.

[HALM] Heitmeyer, C.L. and Archer, M.M. and Leonard, E.I. and McLean, J.D. (2008). "IEEE Transactions on Software Engineering Vol. 34".

[HASPOC] HASPOC project homepage, http://HASPOC.sics.se

[IGIES] M. S. İnci, B. Gülmezoğlu, G. Irazoqui, T. Eisenbarth, and B. Sunar, "Seriously, get off my cloud! Cross-VM RSA key recovery in a public cloud". Technical report, IACR Cryptology ePrint Archive, 2015.

[OPENEST] Open-Estuary homepage, http://open-estuary.org/estuary/

[P] C. Percival, "Cache missing for fun and profit". In Proc. of BSDCan, 2005.

[PROSPER] PROSPER project homepage, http://prosper.sics.se/

[RTSS] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds". In Proc. CCS '09, pages 199–212, URL http://doi.acm.org/10.1145/ 1653662.1653687.

[SM] T. Kim, M. Peinado and G. Mainar-Ruiz, "STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud", USENIX Security Symposium, 2012.

[ZJRR] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. "Cross-VM side channels and their use to extract private keys". In Proc. CCS'12, pages 305–316. ACM, 2012.

Appendix

A ARM Trusted Firmware

ARM provides their own reference boot loader (ATF) , that implements a subset of the requirements listed in Trusted Board Boot Requirements (TBBR).

The boot process in ATF involves a number of modules that interacts.

Module	Location	Format	Purpose
Bl1 (required)	ROM	ARM Assembler	Initial module, selects primary core, loads BL2
BL2 (required)	FLASH	FIP	Loads BL30, BL31, BL33, calls BL31
BL31 (required)	FLASH	FIP	Runtime services and boot of BL33
BL32 (optional)	FLASH	FIP	Trusted Application
BL33 (required)	FLASH	FIP	Payload , Hypervisor , uboot, UEFI, etc

A.1 Trust Anchoring

Integrity control requires that algorithms and public are resilient against tampering. The algorithm for signature verification is not customer specific and could therefore be implemented in ROM. The public key however should be stored outside ROM if possible, since it must be unique for a lot of reasons. Isolation between customers, recovery, updates etc. One time programming (OTP) memory provides registers that could be written only once. Since the size of the public key exceeds the size of the OTP register, the hashed value of this key is stored instead. The public key used by the algorithm in BL1 to verify BL2 is called ROOT_PK. The hashed value of the key is notated ROOT_PK_HASH, for sha256 the size is 32 bits. Verification of the remaining modules could involve other keys than ROOT_PK, therefore ATF support X509v3 certificates with some unique attributes representing the vendor of each module.

A.2 Trust Anchoring

ARMv8 supports four privilege levels, from the lowest EL0 to the highest EL3. Unlike other architectures, there is no fixed memory regions assigned for each level, instead the memory is controlled by registers, and the registers are connected to exception level. Application could only shift from a higher level to a lower, not the opposite. Secure Monitor Call enables applications in low level to exchange messages with application in a higher exception level. The implementation of ATF downshifts from EL3 to EL1, when the first module (BL1) invokes the second (BL2). The first module implements a service listening for SMC calls, required when BL2 pass control to the final module BL33. For native system with pure linux, the kernel should execute in EL1 , and applications in EL0, however the exception level for BL33 is obtained by reading a register that inform ATF if the hardware supports hypervisor. Therefore UEFI or Uboot is required to assure that native linux executes in EL1.

A.3 Trustzone

Cellular phones utilizes SIM-card to protect sensitive data. The interface to the SIM-card doest not allow high speed communication. Trustzone provides the same isolation on the normal hardware. Each processor has a state indicating if the core executes in the secure or the normal world. Each peripherals have access to the security state, via the Trustzone extended databus and could either allow or block access from a specific core, based on the security state of the actual core. For trusted boot, the static memory is exclusive reserved for secure execution. Dynamic memory could be divided into regions with different access criteria. Some implementation of ATF utilize this feature, but the main driver for this is decoding of video streams. The security state is independent of exception level, but could only be altered from EL3.

A.4 Firmware Package

All modules, except BL1 are embedded into a FIP image. The format could be used with or without signatures. The format specification is defined in a C header file, which is shared by external tools, that generates the FIP image. The integration of FIP in the boot code (ATF) involves other components, such as IO-layer, that also converts sequential block media to appear as memory mapped. Since BL2 configures memory, this module must reside in a fixed position , defined during compile time. The other modules (BL31 and beyond) could be reallocated.

A.5 Memory Management

ATF lacks dynamic (free/malloc). Since each module could be developed by independent vendors, the processing of FIP-image must keep track of memory usage, and pass information upon invocation of the called module.

A.6 Alternative Implementation

In the firmware-design.md document found in the ATF source code, the interface to BL31 is described.

Some platforms have existing implementations of Trusted Boot Firmware that

would like to use ARM Trusted Firmware BL3-1 for the EL3 Runtime Firmware. To

enable this firmware architecture it is important to provide a fully documented

and stable interface between the Trusted Boot Firmware and BL3-1.

Since the module BL2 is responsible for memory management, everything in the FIP image is retrieved successfully.

A.7 ATF and Secure Boot

The FIP format has been replaced with HSBF, which only support signed objects. The support of many keys for different kind of modules is removed. The ROOT_PK is used to sign everything, however the design allows additional public keys be embedded in a HSBF object, even if not implemented. Some business model is vertical and have no use of this additional complexity. The X509v3 format has been replaced with signatures only.

Dynamic memory management with free and malloc enable an alternative to the iolayer, further the widespread malloc/free paradigm also enables integration of light weight cryptographic modules.

With heap management very large HSBF could be managed, the only limiting factor is the amount of memory.

Secure Boot substitutes BL1 and BL2, and consequently eliminates the management of shifting EL-levels. The interface to BL31 is implemented, which allows that a binary version of this module could be embedded in the HSBF object. The footprint of BL1 and BL2 without cryptographic support is 37 kb + 57 kb. Secureboot with signature verification is 57 kb.