A High Assurance Virtualization Platform for ARMv8

Christoph Baumann
Royal Institute of Technology
SE-100 44 Stockholm, Sweden
Email: cbaumann@kth.se

Christian Gehrmann, Oliver Schwarz
SICS Swedish ICT
SE-164 29 Kista, Sweden
Email: {chrisg, oliver}@sics.se

Mats Näslund
Ericsson Research
SE-164 80 Stockholm, Sweden
Email: mats.naslund@ericsson.com

Hans Thorsen
T2 Data AB
SE-16440 Kista, Sweden
Email: hans.thorsen@t2data.com

***Abstract*—This paper presents the first results from the ongoing research project HASPOC, developing a high assurance virtualization platform for the ARMv8 CPU architecture. Formal verification at machine code level guarantees information isolation between different guest systems (e.g. OSs) running on the platform. To use the platform in networking scenarios, we allow guest systems to securely communicate with each other via platform-provided communication channels and to take exclusive control of peripherals for communication with the outside world. The isolation is shown to be formally equivalent to that of guests executing on physically separate platforms with dedicated communication channels crossing the air-gap. Common Criteria (CC) assurance methodology is applied by preparing the CC documentation required for an EAL6 evaluation of products using the platform. Besides the hypervisor, a secure boot component is included and verified to ensure system integrity.**

Keywords—hypervisor, isolation, assurance, formal verification, Common Criteria, ARMv8.

I. Introduction

Our daily life and our entire society is increasingly dependent on software and its security and reliability. Software allows us to make dumb devices smarter and to replace expensive, inflexible hardware with low-cost, programmable components and systems. While we cannot remove hardware completely, we can investigate savings possible from sharing the same hardware between different software based services. A fundamental security requirement is to provide *isolation* between the different services. In this work the focus lies on isolation of *information*, a well-known problem area with roots in the military/defense sector and research around multi-level security and access control: allowing plural types of information with different security classification in the same computer system, while obtaining high assurance that information cannot be accessed in non-authorized ways. Isolation also helps to minimize the trusted computing base that services need to rely on. For example, ensuring encrypted network communication even in cases where the sending commodity system and the network drivers are compromised. Gradually, this problem has also become highly relevant in the private sector with issues around the mix of personal and enterprise information in the same user device (e.g. a laptop), cloud computing (allowing users to share pooled resources), network sharing between mobile operators, etc. Simultaneously, 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. In this work we focus on the highly popular and successful ARMv8 architecture [1], which is already common in embedded systems, smartphones, etc, and also starts to see usage in network infrastructure such as 4G radio base stations.

Platform virtualization is commonly based on a *hypervisor*, a software that, to a varying degree of transparency, virtualizes the underlying resources (CPU, memory, etc.) to a set of *guests* sharing these resources. Guests range from simple bare-metal implementations to complete operating systems with applications running on top. Although hypervisors must always ensure some form of separation between the guests in order to be useful (e.g. ensuring execution correctness), the strength of isolation between guests can vary, both in terms of the security methods to control information flow, as well as the assurance in those methods. Our project includes the development of a hypervisor with strong isolation in both respects.

The probably highest attainable, yet practically feasible, assurance level is given by formal verification at machine code level. This is also the aim of this work. Based on a formalization of the ARMv8 architecture, interactive theorem provers and other tools are applied to show that our hypervisor provides isolation which is essentially equivalent to the isolation provided by identical, but physically separate systems, each running one guest. The term “essentially equivalent” is used as a disclaimer regarding certain side-channels, e.g. from power consumption analysis. Also, we allow inter-guest communication, controlled by the hypervisor. Through this, guests *may* share information, but *only* via dedicated channels that were defined at configuration time.

Assurance in the hypervisor is in itself irrelevant unless we can also provide assurance that the hypervisor has been properly booted. Therefore, our platform contains a verified secure boot component which starts the hypervisor in an initial trusted state where the assumptions of the formal proofs hold.

In the critical infrastructure segment, compliance to the assurance standard Common Criteria (CC, ISO 15408, [2]) is typically required, and is also being requested for devices such as USB drives, SIM-cards etc. CC defines assurance levels (EAL 1–7), which affects the depth, rigor, and formalism of the assurance documentation. As part of our work, CC documentation for EAL6 is produced. Still, our formal verification in many regards goes beyond what is required for EAL6.

Related verification projects [3], [4], [5] targeted different architectures, where only seL4 took formal verification down to machine code level [6] and none of them treated boot loader verification. The HASPOC project strives for a holistic high-assurance platform, combining a custom hypervisor, a custom secure boot, formal verification, and CC. We plan to release hypervisor and boot solution as open source. The project is still in progress. Its current status is described in Section VI.

II. The Virtualization Platform

A. Hardware Architecture

For formal verification at machine code level, it is necessary to rely on specifics of the underlying hardware, e.g., to assume the correct execution of instructions by the CPU. In addition, to enforce isolation properties where peripherals are involved, certain support from the hardware is required. Specifically, we assume an underlying system-on-a-chip (SoC) architecture compliant with the ARMv8-A specification [1] and hardware support in terms of memory management units (MMUs) providing virtual memory based on two-stage translation and System MMUs (S-MMU, MMU-400/401 v1) handling peripherals (§IV-B). The CPU must implement all four exception levels (the protection rings of ARMv8), where the most privileged one (EL3) is used to execute the secure boot code and EL2 is used to host our hypervisor. Figure 1 shows a simplified high level form of the underlying hardware platform. The hardware must also provide trust anchors (storage of verification keys) for the secure boot (§IV-A).

B. Use-case examples

Consider a network gateway appliance separating ”red” and ”black” domains, executing on the virtualization platform as in Fig. 2. Each domain is only available to a dedicated guest, and any communication between the domains passes through a third guest, enforcing encryption. Even if both the red and the black guest (including drivers) are compromised, the trust in the encryption only depends on the encryption service, the hypervisor, hardware, and the secure boot (not shown).

In mobile networks, network slicing [7] is an emerging trend, allowing slices to provide more optimal service delivery for specific types of services. Indeed, slicing may extend down to the base station [8]. In such settings, purely physical isolation is no longer cost-efficient. Instead, our solution enables logical isolation, protecting resource multiplexers from resource users, as well as resource users from each other.

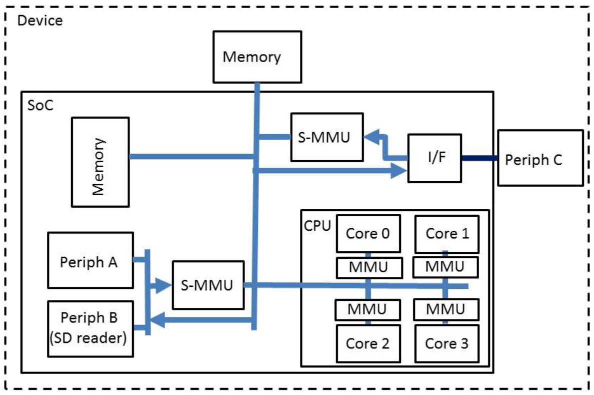

Fig. 1. Platform Overview

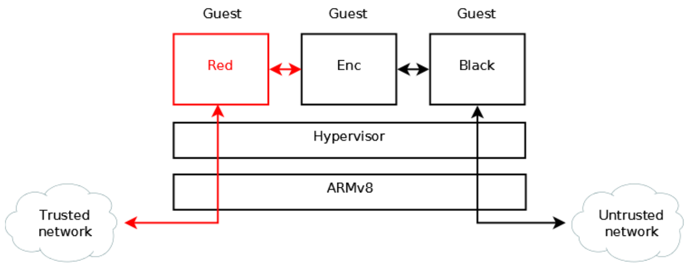

Fig. 2. Example Use Case: Encrypting network gateway

III. Platform Software Overview

A. Secure Boot

The isolation security objective of the virtualization platform hinges on a trustworthy initialization routine that boots up the authentic hypervisor with the signed configuration data (security enforcement policies and other initial parameters) and guest images in place. In other words, the hypervisor execution starts in a state were a number of trusted (verified) invariants hold. This is achieved by the secure boot loader.

1) Background: ARM Trusted Firmware: ARM defines a Trusted Firmware [9] concept that is built upon ARM’s TrustZone technology [10] and provided as a reference implementation by ARM. It comprises a three stage secure boot design where the first boot block (BL1) executes in EL3. BL1 is responsible for verifying integrity of the second boot block, the ”trusted boot firmware” (BL2), and running it in the ARM TrustZone secure execution container (Secure EL1). Finally, BL2 executes a number of boot blocks (BL3x), which set up the platform configurations etc. The trusted boot firmware also loads and verifies the application system blocks, including the hypervisor code if such is present.

2) Boot Overview: The secure boot in our project is based on the ARM Trusted Firmware reference implementation, but omits/simplifies certain steps/features. We omit BL2 and a new simplified but flexible boot block format has been specified, minimizing functionality and code footprint, making the formal verification easier and reducing attack surface. The secure boot loader runs in EL3 and proceeds roughly as follows:

1) Establish trust anchors (verification keys etc) by reading from secure storage.
2) Read and verify (integrity and authenticity) boot configuration information using the trust anchors from step 1.
3) Read and verify boot image (hypervisor).
4) Lower execution privilege level (to ARM EL2) and hand over execution to the hypervisor.

A failure in any of the verifications will abort the process.

B. Hypervisor

To enable formal verification, a key element is to make the hypervisor design very simple. As control is handed over from boot to hypervisor, the hypervisor will take care of initializing virtual memory and peripheral allocation to guests by configuring mappings in MMU and S-MMU, according to configuration data in the loaded boot image. The general principle is that peripherals are exclusively owned by a specific guest. Clearly, memory must also be exclusively owned, except for special memory areas that enable inter-guest communications (§IV-B2). Exclusive ownership also applies to the CPU cores: no two guests share the same core. However, one guest may be assigned multiple cores, and each guest will also be multiplexed with the hypervisor itself on each of the cores.

IV. Design Details - Assurance Elements

A. Secure Boot

The secure boot design empoloys the typical multi-stage approach and is based on ARM Trusted Firmware but with considerable simplifications to allow formal verification of the boot code: a static MMU configuration, a simplified control flow, and two (instead of three) boot stages. The BL2 boot stage from the ARM Trusted Firmware has been eliminated and only the most necessary platform set-up blocks from the last boot stage have been kept, such as power management drivers and the block implementing the switch from EL3 to EL2. The whole boot code is executed in EL3 (omitting switch to/from Secure EL1 to install the payload). In the formal verification of the boot loader one has to show that EL3-specific features (e.g. configuring security-related registers, lowering the exception level) are used as expected.

Images in the boot chain are signed with a private root key. The trust anchoring requires that a hash of the corresponding public root key and the first stage boot code are resilient against tampering. To this end, the first stage loader resides in ROM and the root key hash can be programmed into a special register (e.g. eFuse). At power on/reset, the public root key is retrieved and verified against the trusted hash value. Upon success, the public key is added as the first item to the list of trusted keys used to verify subsequent objects in the second stage boot loader. In this phase, only trusted (SoC-internal) RAM is used. Depending on the target, the next object to verify may contain the firmware required to enable use of dynamic (SoC-external) RAM to store subsequent objects, i.e., the guest images, their configuration, as well as the hypervisor security enforcement policies. The final object contains the hypervisor. After its signature has been verified against the trusted public root key, the hypervisor is invoked on EL2. The design allows to add trusted keys into the verification chain (3rd party keys signed by the private root key), so that guest and configuration data does not need to be signed with the root key.

B. Hypervisor

The main objective of our hypervisor is to ensure the isolation of the resources allocated to each guest. Since every guest owns all its cores exclusively, the hypervisor implements a security policy focusing on shared hardware components. A guest must not directly access:

1) hypervisor memory,
2) memory areas of other guests (unless shared in communication, see below),
3) other guests' peripherals (but proxy access through communication channels is permitted).

Of course, side effects on foreign resources can occur when stimulated by inter-guest communication or system calls to the hypervisor. Below, key enablers of isolation are discussed.

1) Exception levels: The existence of multiple exception levels allows the hypervisor to be more privileged than guests when setting up isolation mechanisms. While the boot executes at EL3, the hypervisor always execute at EL2 and guests are confined to EL1 or EL0 (e.g. a guest OS can execute at EL1 and its user applications at EL0). The ability for guests to issue Secure Monitor Calls (SMC) to enter EL3 is disabled at boot, and any later attempt to invoke SMC will cause a trap to EL2/hypervisor. During execution, SMC can be used for power management, but these are intercepted by the hypervisor, checked/translated and resent from EL2. Another way to reach EL3 is through exceptions, e.g. interrupts, discussed below.

2) Inter-guest communication (IGC): Since guests are allowed to communicate, information *can* flow between them. The hypervisor ensures that only *supervised communication* can occur, meaning an information flow explicitly authorized by and passing through a dedicated resource allocated by the hypervisor, a so-called *communication channel*. Authorization in this context means that these channels are created according to a parametrized communication policy. Each communication channel is associated with a unique (sender, receiver)-pair and is thus in nature uni-directional. No other guests are able to read or write from the channel. Concretely, a shared memory area with restricted access permissions is used and interrupts signal the presence of a message.

3) Memory isolation: Physical memory (of hypervisor and guests) is exclusively owned. Boundaries are defined in the runtime configuration. The hypervisor must enforce that a guest cannot directly (outside the side-effects of IGC) access or modify information in the memory exclusively owned by another guest. The MMU provides a two-level mechanism to first translate virtual addresses into intermediate physical memory and then to actual physical memory addresses. This mechanism allows the hypervisor to enforce access control policy pertaining to guests' access to the physical memory. The entries of the second stage page tables that map a guest's working memory are statically configured by the hypervisor.

4) Peripheral isolation: Since most peripherals are memory mapped, guest access to them can be controlled by enforcing the access control policy on associated memory addresses. However, 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 instructions. Since only one guest is executing on a given core and the hypervisor is not using these per-core components, their isolation is given by construction. In any other scenario, if applicable, guest access could simply be disabled altogether from higher privilege levels, or the hypervisor could save and restore separate guest contexts for the registers associated with the peripheral. Some peripherals have direct access to the buses, e.g., for direct memory access (DMA). This poses a threat to isolation since even if the MMU can restrict processor access to memory and peripherals, it cannot restrict a peripheral's access to the bus. For example, a peripheral owned by guest A could write into memory space of guest B and in a red-black separated system such as that in Fig. 2, data could potentially flow directly between interfaces without first passing encryption. Here, the S-MMU functionality steps in. The S-MMU is a secondary MMU placed between the peripheral and the system bus, as shown in Fig. 1. The hypervisor programs the S-MMU to control which internal resources a peripheral can access.

5) Interrupt handling: In a multi-core system, the interrupt controller receives messages from various peripherals and forwards them to the correct cores after appropriate filtering and priority check. In our case, a guest is permitted access to interrupts from a peripheral only if that guest owns the latter and if the policies in the configuration do not block those interrupts. Thus, the interrupt controller is configured by the hypervisor to route incoming interrupts to precisely those cores where the owning guests are located.

V. Formal Verification Methodology

In order to obtain high assurance of the platform's isolation we combine formal verification of the hypervisor design in the theorem prover HOL4 [11] with a mechanized binary code verification approach. Both parts depend on a formalization of the ARMv8 architecture. The overall goal of the formal verification is to establish a bisimulation between an ideal model of the system that is secure by construction and the binary hypervisor implementation. Details of our verification methodology are depicted in Fig. 3 and explained below.

A. ISA Model

A formal notion of the ARMv8 instruction set architecture (ISA) is needed in three places: (1) to model the guest systems that are virtualized, (2) to prove that the hardware support for virtualization, e.g., the stage-2 MMU as well as the S-MMU, are configured in a way that ensures the desired isolation properties, and (3) as a base for the code verification approach. Thus a suitable ISA model needs to include all system level functionality available to guests, including the stage-1 MMU and virtualized system registers, as well as the stage-2 MMU, the S-MMU, and all instructions that the hypervisor and secure boot code may execute to implement the virtualization platform.

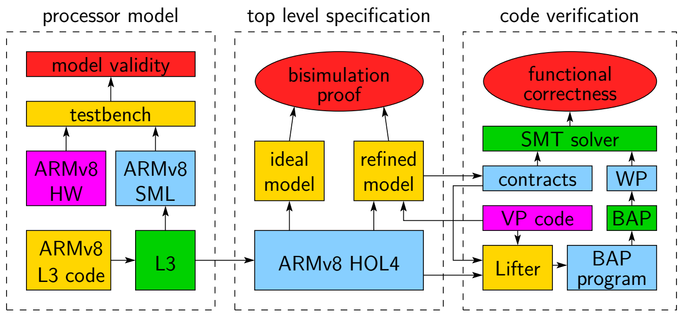

Fig. 3. Overview of the formal verification methodology. Violet boxes show the actual virtualization platform, i.e., its code and the underlying hardware. Yellow boxes are handwritten artifacts. Light blue boxes represent derived artifacts (in HOL4 if not stated otherwise) while green boxes are third-party tools. Red containers denote results, where ellipses are formalized in HOL4. Arrows illustrate dependencies.

We extend the pre-existing ARMv8 user mode model by Fox [12] with the aforementioned system level features to obtain the desired ISA model, defined in the domain specific language L3 [13]. The L3 framework allows to generate corresponding theories in HOL4 and executable SML code. The HOL4 translation serves as a base for formal proofs of architectural properties and as an input to our binary code verification framework. Using the SML representation we have developed a simulator that runs test programs given in ELF format. It is part of a testbench where the same tests are run on ARMv8 hardware in order to validate our ISA model.

B. Top Level Specification & Bisimulation Proof

As mentioned before, the virtualization platform should resemble a system where all guests are running on their own isolated virtual machines with private memories, only communicate through dedicated channels, and have exclusive access to the peripherals associated with them. We formalize this idea in HOL4 and define an *ideal model* of the system that has these desired properties by construction.

Additionally, we capture the design of the virtualization platform in a *refined model* that closely follows its C and assembly implementation, describing internal data structures and control flow of the hypervisor and secure boot loader. Thus we obtain a functional representation of the C and assembly code in HOL4. We use this specification later to derive function contracts for the binary code verification.

The aim is to show that the ideal model simulates the refined one, i.e., that all computations of the refined model can be represented in the ideal model where guests are isolated by construction. Since we also want to cover information flow properties like confidentiality, a simple refinement proof is insufficient. We instead prove a bisimulation.

The bisimulation relation maps refined guest states to ideal guest states in the obvious way, taking into account the second-stage translation and stored CPU contexts that are only visible in the refined model. Internal hypervisor steps and states only become visible in the ideal model if they are observable by the guests, e.g., through the effect of hypercalls. The secure boot loader is invisible in the ideal model; we prove that the bisimulation relation holds for the initial state after execution of the boot and hypervisor initialization code.

Inherently, the bisimulation proof relies on the components guaranteeing memory, peripheral, and interrupt separation. For memory separation we show that the second-stage MMU is configured initially in a way that isolates private guest memories properly and that this configuration is never changed. For peripheral separation we prove a similar property for the S-MMU. Interrupt separation relies mainly on the fact that the hypervisor injects interrupts into the right guests, as all interrupts are routed through the hypervisor by the current generation of ARM interrupt controllers. Further isolation properties guaranteed by the architecture are, e.g., that guests cannot access hypervisor and TrustZone registers or execute certain instructions. We are in the process of developing semi-automated proof tools that allow to deduce these kind of properties for different architecture models.

C. Binary Code Verification

To show that the refined model correctly captures the binary implementation of the virtualization platform, we use the refined model as a functional specification for the binary code and prove functional correctness of the latter. We follow an approach that has successfully been tried and tested on a previous ARMv7 version of the hypervisor [14], [15].

Technically we employ the Binary Analysis Platform (BAP) [16] to perform the formal analysis based on a representation of the ARMv8 assembly semantics that we derive from our HOL4 model through a so-called *Lifter*. The Lifter is a HOL4 procedure that evaluates a given ARMv8 program according to the HOL4 model and generates a corresponding program in BAP's internal language, that is annotated based on the function contracts derived from the refined model. It also gives a proof that both programs behave equivalently.

For each translated code fragment we use BAP to deduce the weakest pre-condition for the given post-condition. Finally an SMT solver is employed to show that this weakest pre-condition is implied by the actual pre-conditions from the function contracts. Thus we obtain that the binary code of the virtualization platform correctly implements the refined specification and—through the bisimulation proof—also resembles the ideal model.

VI. Current Status

Prototypes of secure boot and hypervisor have been implemented on several different ARMv8 platforms, such as the ARM Cortex A53 HiKey and ARM Juno boards. We have been able to securely boot multiple Linux guests on these boards, proving the feasibility of the suggested approach. Hypervisor, guests, and guest configurations are statically defined. There are no provisions made for dynamically spawning more guests, only those guests that are defined at boot time are executed.

So far the verification work has focused on developing the ARMv8 model and the top level specification. The current ISA model covers all relevant registers and instructions of the four exception levels (excluding interrupt semantics) as well as the 2-stage MMU (excluding TLBs). Memory is currently modeled in sequentially consistent fashion excluding caches, but we plan to cover weakly consistent behaviour of guests in the future. The certifying Lifter interfacing BAP has been implemented for Fox's original ARMv8 model.

Formal verification cannot cover information flow through "analog" side channels such as power consumption or timing behaviour. For instance, sensitive information might leak between guests through access time discrepancies on the shared L2 cache. One way to close this channel is to partition the L2 cache lines between guests (cache coloring). Another option is to adapt guest software such that its cache line access patterns do not leak sensitive information.

For the Common Criteria work, a so called Security Target (ST) with the required supporting design documentation is being written. While the formal verification aspects of assurance requires us to rely on details of the ARMv8-A specification [1], the situation is somewhat different for the CC assurance since the CPU itself is considered outside the boundary of the verified component (the so called TOE). Thus, large parts of the ST *might* be re-usable also for another CPU architecture.

VII. Summary and Conclusions

We have presented a solution for isolating critical from commodity software on ARMv8, while still providing controlled communication. 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. Our 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. 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 [17].

Acknowledgment The HASPOC project presented herein is in part funded under contract 2014-00702 of Vinnova's Challenge-driven Innovation Programme. Parts of the verification work are supported by a framework grant from the Swedish Foundation for Strategic Research.

References

[1] "ARMv8-A Architecture Reference Manual," http://infocenter.arm.com/help/topic/com.arm.doc.ddi0487a.h/index.html.

[2] "Common criteria for information technology security evaluation," https://www.commoncriteriaportal.org/cc/.

[3] Green Hills, "INTEGRITY real-time operating system," http://www.ghs.com/products/rtos/integrity.html.

[4] D. Leinenbach and T. Santen, "Verifying the Microsoft Hyper-V hypervisor with VCC," in *FM 2009. Proceedings*, 2009, pp. 806–809.

[5] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an OS microkernel," *ACM TOCS*, vol. 32, no. 1, pp. 2:1–2:70, feb 2014.

[6] T. A. L. Sewell, M. O. Myreen, and G. Klein, "Translation validation for a verified OS kernel," in *Programming Language Design and Implementation*, ser. PLDI '13, 2013, pp. 471–482.

[7] NGMN alliance, "Description of network slicing concept," editor: Peter Hedman, White Paper version 1.0, January 2016.

[8] G. Tseliou, F. Adelantado, and C. Verikoukis, "Resources negotiation for network virtualization in LTE-A networks," in *International Conference on Communications (ICC)*, June 2014, pp. 3142–3147.

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

[10] "ARM TrustZone technology webpage," http://arm.com/products/processors/technologies/trustzone.

[11] "HOL4 project," http://hol.sourceforge.net.

[12] A. Fox, "ARMv8 L3 model," http://www.cl.cam.ac.uk/%7Eacjf3/l3/isa-models.tar.bz2.

[13] A. C. J. Fox, "Improved tool support for machine-code decompilation in HOL4," in *Interactive Theorem Proving (ITP)*, 2015, pp. 187–202.

[14] M. Dam, R. Guanciale, and H. Nemati, "Machine code verification of a tiny ARM hypervisor," in *TrustED@CCS'13*. ACM, 2013, pp. 3–12.

[15] M. Dam, R. Guanciale, N. Khakpour, H. Nemati, and O. Schwarz, "Formal verification of information flow security for a simple ARM-based separation kernel," in *Computer & Communications Security (CCS)*, 2013, pp. 223–234.

[16] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A binary analysis platform," in *CAV*, 2011, pp. 463–469.

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