Specifying the *de facto* OS of a production SoC

Ben Fiedler
ETH Zürich
Zurich, Switzerland

Roman Meier
ETH Zürich
Zurich, Switzerland

Jasmin Schult
ETH Zürich
Zurich, Switzerland

Daniel Schwyn
ETH Zürich
Zurich, Switzerland

Timothy Roscoe
ETH Zürich
Zurich, Switzerland

Abstract

Verification of any operating system is inevitably relative to a model of the underlying hardware. Within the context of kernel verification, the underlying hardware model usually comprises of architectural correctness of the executing cores, but pays little attention to devices *underneath* barring the assumption that they are "trusted".

Recent work has pointed out that the *de facto* operating system of a machine includes not only the kernel and processes running on top, but the multitude of other devices driving the actual hardware: security (co-)processors, DMA engines, network firmware, and more. The concept of the *de facto* OS shines light on a critical boundary between a kernel and the rest of hardware which is crucial to reasoning about both kernel isolation, and the security properties of the whole operating system. In this paper we report on our experience to date in specifying the *de facto* OS environment of a production System-on-Chip, and the implications of this effort so far for assured kernel isolation.

ACM Reference Format:
Ben Fiedler, Roman Meier, Jasmin Schult, Daniel Schwyn, and Timothy Roscoe. 2023. Specifying the *de facto* OS of a production SoC. In *Workshop on Kernel Isolation, Safety and Verification (KISV '23), October 23, 2023, Koblenz, Germany.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3625275.3625400

1 Introduction

We have recently introduced term *de facto operating system* [7] to describe the body of software in a modern computer that performs the functions traditionally ascribed to an OS: securely multiplexing the hardware resources among principles, abstracting and virtualizing these resources to applications, etc. The term is useful because it captures the fact that a traditional OS kernel like Linux or seL4 [9] no longer performs these functions, except as part of a larger whole which is typically unspecified and lacks an intentional design.

Such traditional kernels aim to provide isolation between mutually distrusting processes, and consequently to also isolate themselves from such processes to avoid being compromised.

The existence of *de facto* OSes implies a the third isolation boundary a kernel must pay attention to: the one *underneath* it that protects it from other parts of the *de facto* OS, system software running on other cores and DMA-capable devices like NICs, WiFi chips, video co-processors and everything else that can issue reads and writes to address space. A steady stream of security problems sometimes called *cross-SoC vulnerabilities* in Linux come down to paying insufficient attention to this boundary. While verified kernels like seL4 [9] or Hyperkernel [12] acknowledge its existence, they defer the question of how to deal with it [12, 17], and rely on vague statements such as "hardware correctness" instead.

In this paper we present our experience so far trying to formally represent the full hardware environment of a widely deployed System-on-Chip (SoC), the NXP i.MX 8X. We follow our approach proposed in [7] and try to specify the complete set of software principles in the system, and what trust relationships obtain between them, based on source code and documentation for the hardware.

Our results so far have been instructive. For example,

- Modern, mature memory management unit (MMU) designs can display an extraordinary level of complexity when attempting to specify their semantic behavior.
- The information provided by hardware vendors to facilitate secure OS implementations is often not the information needed to formally specify the security properties of the hardware.
- System-wide hardware features designed to provide flexible isolation often introduce arbitrary limitations on security policies, and complexity that can be challenging to capture in a formal specification.
- The security behavior of some hardware modules can configured externally, making modular composition of specifications difficult

KISV '23, October 23, 2023, Koblenz, Germany
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0411-6/23/10…$15.00
https://doi.org/10.1145/3625275.3625400

Perhaps most significantly, we find that a focus purely on *kernel isolation* misses the point: the Linux kernel, say, plays a relatively minor role in the dynamic security properties of an SoC, and it may be better to shift attention to the entire *de facto* OS.

We present these results in the rest of this paper, and reflect on the implications for the concept of kernel isolation.

2 Approach

We use the approach proposed in our earlier work [7], leveraging a Rust-based implementation of decoding nets [1] to describe the *de facto* OS of the NXP i.MX 8X SoC [19]. Our goal is to identify a set of behavior assumptions for the components of the i.MX 8X that the components of the *de facto* OS make about each other and the other soft- and hardware agents on the SoC to guarantee each component's integrity. In general, we expect the same techniques to be applicable to other SoCs.

Within a decoding net, the address translation state of an SoC is modeled as a directed graph, where nodes are address spaces and edges represent translation between address spaces. Addresses in a decoding net are always qualified with an address space they are located in. Besides the primitives of adding address spaces and mappings, we add additional annotations, as we have proposed previously [7].

We annotate some address spaces as *contexts*, meaning they generate memory operations, which are reads and writes to addresses within the local address space. Examples of contexts include CPUs, GPUs, DMA engines and other devices.

Address regions can be defined as *accepting*, which means that they terminate the address decoding process. The main examples of these are memory addresses backed by DRAM or SRAM, and memory-mapped device configuration registers.

Translation and *protection* regions are address space regions holding the metadata to configure some translation or protection operation. They specify in which address region the configurations are stored, as well as the source and destination address spaces of the translation or protection process this data configures. Access to a translation region allows the accessing context to arbitrarily change the mapping from source to destination addresses, while access to a protection region only allows changing which parts of the source and/or destination address space are accessible, but cannot alter the underlying mapping(s).

Note our assumptions about the semantics of translation and protection regions are oversimplified: few components provide byte-granularity modification of address translation or protection behaviour. We take care to always *over-approximate* the authority conferred in our specification to ensure that we never incorrectly identify an insecure specification as secure.

To facilitate re-use of common hardware features and their specifications, we additionally provide concrete *components*, which are Rust programs that invoke decoding net primitives to specify the behaviour of complex hardware blocks. This way components are maximally flexible in their implementation, which may be quite complex, e.g. in the case of an i.MX 8X's Extended Resource Domain Controller (XRDC). We describe a number of components that make up the i.MX 8X platform in section 3.

Once we have constructed a decoding net for our SoC, we proceed to analyze the integrity and isolation guarantees of individual contexts. Using our SoC model and a set of assumption about the behaviors of contexts in the system, we compute the set of possible overlaps between a prospective victim context and the other contexts in the system. When considering a set of contexts whose guarantees we are interested in, we re-run our analysis for each individual context.

Our analysis then computes all (potential) address space overlaps between our victim's address space and all other contexts in the system, under the assumption that every context adheres to the behavioural assumptions we make of it.

Any overlaps found during our analysis constitute a possible integrity violation, and thus has to be addressed by the user. There are multiple ways that these overlaps can be addressed via behavioral assumptions: (1) assuming that the overlap between the victim and accessing contexts is validated by the victim, or (2) by restricting modifications to the intermediate translation and protection structures. By default, a context is unrestricted in its behaviour.

Whenever a computed overlap utilizes some context's ability to modify translation or protection configuration, then we also derive a possible sequence of modifications that facilitates this access. Note that it is possible that there are multiple possible ways to modify the translation structures, and in order to guarantee isolation/integrity *all* possible exploit paths must be constrained.

Once the user has assembled a list of assumptions that cause our tool to longer flag vulnerabilities, these assumptions constitute a set of proof obligations that guarantee the victim's isolation within our memory addressing model.

At first, it seems that we haven't gained much: the derived assumptions must still be trusted for correct isolation. However, from this point we can make quantitative statements about the *amount* of trust that we have to place in the platform, as well as how *trustworthy* it is. For instance, due to its verified functional correctness, we know that trust we place in seL4 [9] restricting access to certain regions of its address space is justified.

In the following section, we detail our experience of writing down the *de facto* OS of an i.MX 8X platform. Even with the conservative simplifying assumptions above, the result so far is remarkably complex, itself a reflection on the current state of hardware semantics.

3 Experience

In this section we discuss our experience so far specifying the hardware execution environment provided by the NXP i.MX 8X SoC, as a foundation for reasoning about the *de facto* OS running on this chip in a typical deployment.

For the most part, we have found that the relationships between the various hardware execution contexts and devices on the i.MX 8X can be captured cleanly using a model based on decoding nets. However, in the following sections we identify some features of the SoC where we feel our experience so far has more general implications for formal reasoning about isolation in *de facto* OSes.

3.1 The NXP i.MX 8X

We chose the i.MX 8X for several reasons: it is widely available (including in the form of low-cost System-on-Modules (SoMs) and hobbyist boards), it has substantial documentation and a support community, and much of the default firmware is readily available as open source code. While it is not a new part (it was announced in 2013), it remains representative of the complexity of modern SoCs.

The i.MX 8X is a SoC designed for the automotive market. The main application processors are a cluster of 4 ARM Cortex A35 cores, which typically run Linux. However, the i.MX 8X features many more cores than this: a Cortex M4 for application use, another Cortex M4 as a System Controller, a Cortex M0+ core as a “Security Co-processor”, a HiFi4 DSP also capable of running an operating system, multiple hardware media encoders and decoders, at least two further Cortex M0+ cores associated with the Vision Processing Unit, a sophisticated GPU, and more.

All these subsystems are able either to directly access memory or initiate Direct Memory Access (DMA)-based copies to or from main DRAM, which sits on a 36-bit wide main interconnect called the DRAM Block or DB. This interconnect can route transactions not only to main DRAM but also to every other subsystem on the i.MX 8X, and therefore its corresponding address space has a rich memory map.

In addition to the main banks of system DRAM, some of these components also feature their own memory areas, some of which are shared between some subsystems, and others are globally reachable via the DB. As is common in modern SoCs, the “physical” address at which a given resource is accessed sometimes depends on the core originating the access.

3.2 The Armv8-A Memory Management Unit

The Armv8-A MMU is ubiquitous in 64-bit ARM-based SoCs, but is noteworthy due to the sheer complexity of its semantics. The MMU provides virtual address spaces for processes running on top of the Cortex-A compute cores on the i.MX 8X SoC, but also virtual machine support, TrustZone, pointer authentication, 32-bit compatibility, and a host of other features, all of which concern a specification of its isolation properties. It is programmable via device registers that configure address translation properties such as granule (page) size, input and output address space sizes, number of translation levels and stages, and more. These registers aside, most of the rest of the behavior of the MMU is determined by data structures in memory, access to which from other contexts might be problematic.

Nothing about this MMU is fundamentally difficult to model: it is well documented as part of the Armv8-A Architecture Reference Manual [10]. However, a full description of the semantics of this MMU is likely to be very large. Our current specification of a rather restricted subset of its capabilities runs to about 480 lines of specification. Achermann *et al.* [3] report similar complexity with the x86 architecture, although (complementary to our work) they focus on the syntax of data structures rather than the semantics of translation.

An excerpt from our (simplified) spec is shown in Listing 1. Every level of page table entries has a set of possible states: an `Invalid` entry faults the current translation and does not induce further mappings, a `TableEntry` points to a next-level table filled with the respective entries, and a `BlockEntry` translates the current offset in the virtual address space to the specified physical address. The function `add_new_vas` correctly sets up a new virtual address space and inserts mappings as dictated by the Armv8-A MMU's semantics. Furthermore, it also annotates the page table regions as holding *translation configuration*.

```
enum L0PTE { Invalid, TableEntry(usize, L1Table) }
type L0Table = [L0PTE; 512];
enum L1PTE { Invalid, TableEntry(usize, L2Table),
     BlockEntry(usize) }
type L1Table = [L1PTE; 512];
// ...

enum MMUConfiguration {
  Disabled, OneStage(usize, L0Table), // ...
}

struct MMU {
  output_as: ASID, translation_walk_as: ASID,
}

impl MMU {
  fn add_new_vas(DecodingNet, MMUConfiguration) -> ASID
}
```

Listing 1. Excerpt from our Armv8-A MMU specification

We assume that a hardware MMU correctly multiplexes multiple virtual address spaces based on their individual configurations: the MMU only supports one active context issuing memory operations (per exception level), but supports multiple contexts running in parallel on top of the MMU.

3.3 Security and System Controllers

The System Controller Unit (SCU) and Security Controller (SECO) are two Cortex-M cores that drive significant parts of the i.MX 8X, responsible for booting the big Cortex-A cores, the other Cortex-M4 core, and initializing and configuring the XRDC. NXP provides firmware with the i.MX 8X

SDK that runs on the SCU and the SECO, though it is possible to modify and/or replace the SCU firmware as needed. Both components share 256KiB of Tightly Coupled Memory (TCM) which are used during the boot process to bring up the main DRAM.

The SCU is the primary intended communication partner of other system components for interacting with the SECO. Communication with these components is done via messaging units (MUs), which are two-way memory-mapped buffers reachable from (at least) the Cortex-A core cluster and the user-controlled Cortex-M.

Given their privileged access to the XRDC (see subsection 3.4), any context that has access to the SCU or SECO will need to be almost completely trusted by other contexts. Most SoCs include some form of security processors, so this situation is not unusual. However, it is not at all clear from the i.MX 8X documentation exactly what protection applies to these components.

This shows that the information vital to formally reason about security of a *de facto* OS is sometimes not regarded as useful by the hardware vendors, who assume that trusting the firmware is sufficient for OS developers.

3.4 The Extended Resource Domain Controller

The XRDC [14] is a reconfigurable, distributed, tree-based partitioning access protection unit intended to enforce isolation between SoC components. The XRDC is significant not only because it is the central protection and isolation unit on the i.MX 8X, but also it represents the solution proposed by the hardware vendors to the isolation problem yet remains outside the scope of the traditional OS kernel.

While not as large as the MMU, the XRDC is still complex. It can isolate any two resources it considers separate on any bus it controls on the level of bus transactions in hardware. Bus masters (in decoding net terminology, contexts), peripherals, and memory regions are assigned to partitions, which are organized hierarchically based on a 5-bit mask. Bus transactions from by contexts are tagged with an ID identifying their partition, and disallowed accesses generate bus faults.

Contexts can by default only access resources in the same partition, but peripherals can have different access permissions from each partition. However, Peripherals can only be configured from within the same partition.

Partitions are also the power management granule, and rebooting a partition deletes all child partitions and returns their resources. Communication across partitions is possible through a limited number of messaging units which operate like interrupt-enabled postboxes.

Memory Regions are defined by physical start and end addresses and are always owned by exactly one partition. The owning partition has broad configuration powers over the memory region, but can grant other partitions access rights to the memory region. The number of memory regions that a particular memory can be split into is restricted and particular to that memory.

Finally, cores have an associated "process ID" register, changed on a context switch, which can effectively move the core from one partition to another.

The XRDC illustrates the expressive power of decoding nets: we model the XRDC protection regime with a different decoding net per partition, and assign unique address spaces to any shared memory.

However, at the same time it shows the difficulty of interpreting ambiguous vendor documentation written informally in English, and translating this into a specification. Many of the details of how the XRDC hardware operates are unclear, and since it operates below the Linux kernel it is hard to link to OS behavior.

An implication for faithfully specifying the *de facto* OS is that it may be necessary to generate "litmus tests" in the form of code and deployment scripts from the specification, that can execute on the hardware and validate the spec against the real *de facto* OS. This was always going to be a requirement for verification, but documentation ambiguities may mean it is also a vital *a priori* part of the specification process.

3.5 ENET DMA engine

The ENET is an Ethernet MAC module in the i.MX 8X "Connectivity Subsystem". It includes descriptor-based DMA engines to transfer data bidirectionally between the MAC and the system interconnect.

This is straightforward to model, but ambiguity in the documentation shows how relevant details configuration and implementation of the core can become important. While the documentation does not specify the exact aperture the ENET DMA engine has into the system, it accepts 32-bit pointers as addresses when configuring its descriptors, and we assume conservatively that the DMA engine can access the full lower 32-bit address range of the main system bus.

This module illustrates the need to make explicit trust choices when reasoning about the *de facto* OS. In this case, the ENET hardware can potentially access the low 32-bits of system address space, which might include a range of critical data structures including page tables. If the ENET firmware is not to be trusted (for example, if we believe it could be remotely compromised), neither can any OS component whose critical data structures can be accessed by it.

If we choose instead to trust the ENET to only ever access memory specified by the transmit and receive descriptors, there is now an obligation both on the device driver to only supply "safe" descriptors to memory, but *crucially* we must *also* trust any other context which could modify the descriptors in memory and use the ENET DMA engine itself as a vector for attack.

3.6 WiFi Modem

A similar situation with a known vulnerability is seen on our i.MX 8X SoM which includes an AzureWave AW-CM276NF WiFi modem [23] based on the NXP 88W8997 chipset [20], connected to the SoC via PCIe and USB 2.0 and is DMA-capable. A remote code execution vulnerability has been found for the firmware on this modem [18, 21]. Exploits for the application processor kernel using vulnerabilities in WiFi module firmware as an attack vector have been shown [8].

However, a different issue for specifying the *de facto* OS arises because the interface used by the WiFi module and the similar Bluetooth interface is configured by pins brought out on the edge connector of the SoM.

This illustrates a further challenge in modeling and specifying the behavior of the *de facto* OS: what happens within a module (including DMA accesses) can be dependent on external hardware outside that module.

3.7 Inter-component DMA

The i.MX 8X features a sophisticated audio subsystem. One component of this is a Tensilica HiFi4 Digital Signal Processor (DSP) [22, 24], which is used for processing audio signals. This is a Very Long Instruction Word (VLIW) Single Input Multiple Data (SIMD) core with 64KB of TCM that is capable of running FreeRTOS [25]. On the i.MX 8X the DSP has an additional 448KB of local SRAM, and can access RAM and other subsystems over the main interconnect.

One of the audio inputs on the i.MX 8X is an Enhanced Synchronous Audio Interface (ESAI), which itself is a sophisticated peripheral incorporating an Enhanced Direct Memory Access (eDMA) controller for transferring audio data. This eDMA controller can access two different interconnects, one of which is the system interconnect, and crucially can transfer data directly between the DSP and the ESAI, without involving main memory or the "traditional" OS kernel in any way. Neither the source nor destination address spaces of the eDMA engine are accessible to the kernel.

This does not seem to be unique to this SoC or even the i.MX 8X audio subsystem (camera data can be transferred to the display subsystem).

The implication is that viewing the traditional OS kernel as the arbiter of isolation decisions, and even the primary object to be isolated in the system, will fail to capture accesses and potential security vulnerabilities in other parts of the SoC.

3.8 Inline encryption

A final feature we have encountered so far in our efforts is the Inline Encryption Engine (IEE) on the i.MX 8X, which interoperates with the XRDC. Memory regions can be tagged as IEE-encrypted, causing bus traffic to be re-routed through the IEE where it is either decrypted or encrypted before continuing. This operation is transparent to the requesting context.

The IEE loads keys over a private bus to the SECO subsystem from a secure component called the Cryptographic Acceleration and Assurance Module (CAAM), which can either generate keys or safely store them for persistent use.

While specific to the i.MX 8X and its variants, it seems plausible that such privacy features would have value on other SoC designs, and the functionality also has much in common with enclave technologies like ARM's Confidential Compute Architecture (CCA) [11] and Intel SGX [5].

Formally modeling IEE functionality can be done using decoding nets by introducing two views on encrypted memory: one able to read the plain text, and the other only seeing the encrypted version. If keys are securely stored, we might assume that encrypted memory provides isolation from untrusted components that cannot decrypt it. However, the untrusted component might still overwrite encrypted memory with garbage data.

We conjecture that decoding nets may need to be augmented to efficiently express hardware functionality such as this, but it remains an open question.

4 Discussion

The process we have been going through can be described as modelling the resource access relationships of a modern SoC. The approach we have adopted is to use (and extend) decoding networks and focus on access to memory addresses.

The clearest result from our work so far is that modern computers are, indeed, enormously complex when it comes to their use of addresses, address translation, and protection. This is, of course, no surprise, but what is significant about a formal specification approach is that, for the first time, it gives us some kind of *measure* of this complexity, and a means of *comparing* the complexity of hardware designs which, on the face of it, might provide equivalent functionality.

4.1 Implications for Kernel Isolation

It is also clear that as OS researchers we have to adapt our reasoning about kernel isolation properties to modern SoCs.

Traditionally, kernel isolation analyses are processor-centric: they consider the application processor and DRAM and examine if *lower-level privilege* contexts can interfere with the kernel. In this setting, higher-privilege contexts are limited to the hypervisor and the firmware of the application processor, which are typically assumed *trusted* and sufficiently isolated themselves and can thus be ignored.

Once we extend the scope of such an analysis to a modern SoC, these assumptions are no longer tenable: The remainder of the *de facto* OS is not sufficiently *manageable* to simply extend blanket trust to it as previously done to higher-privilege contexts. Indeed, not even the SoC vendors fully trust their own platforms any more due to the inclusion of third-party IP blocks that the limited time to market and growing hardware-complexity necessitate [16].

It is therefore vital to model all possible resource access relationships of an SoC, as we are doing initialling the i.MX 8X. If the platform allows the kernel to configure system-level access control mechanisms, this model can be used to reduce the required trust in the remainder of the *de facto* OS to a minimum: not all the contexts of the OS need the ability to modify the application kernel's memory and its isolation configurations as part of their functionality.

Even if no such mechanisms are available, the model allows us to make trust statements that are much more qualified. Such precise trust statements can be useful, for example to generate proof obligations for formally verified firmware. They can also direct developers to the areas of code that are especially vital to increase its assurance, or alternatively, direct security researchers to the areas most likely to enable system-wide exploits via complex cross-SoC vectors. The definition of *trust* and its incorporation into our model constitutes future work.

4.2 Viability of the approach

A different question raised by our analysis so far is whether decoding networks (or extensions thereof) are really sufficient to capture the important features of isolation in a modern SoC.

So far, we have found decoding nets to be a good fundamental abstraction, although considerable additional complexity is required above them to capture units like the Armv8-A MMU. The component that is likely to stress the abstraction the most is the inline encryption engine.

A different issue is the potential lack of composability of isolation boundaries, e.g. in the case of the external SoM wireless chip. It remains to be seen if a better fundamental abstraction for describing isolation in a modern SoC exists.

5 Related work

The role of MMUs in inter-process isolation has been extensively studied since their emergence in the 1960s and is considered well-understood. However, the complexity of correctly configuring a heterogeneous set of MMUs and memory protection units (MPUs) across a complete SoC has only been recently studied and addressed by Achermann *et al.* [2].

The configurations of other platform-level isolation mechanisms, such as IOMMUs, the XRDC on the i.MX 8X or the MPU proposed in [15] are similarly complex. Prior work [6] has addressed this problem by automatically generating configurations from desired information flow properties and specifications of the isolation mechanisms of a platform.

This is approach is similar to our own tool, but it presupposes an intimate knowledge of the platform's interactions to formulate appropriate information flow properties. In contrast to this, our tool is intended to help reason about these interactions and derive the necessary trust assumptions to ensure isolation given a particular configuration.

Other communities have considered the security implications of modern SoCs. The trusted computation community is concerned with defining and limiting the Trusted Computing Base (TCB) of security-critical software. Most research restricts the TCBs to the application cores [26]. Thus, all contexts and resources beyond the application processors are considered untrusted—mostly to guard against physical attackers, but justifying trust in the unknown firmware blobs executing on additional (general purpose) cores on the system would certainly be problematic, too. The resulting TCB security analyses and isolation mechanisms therefore do not consider the SoC interactions in-depth which is vital for kernel-SoC isolation, because the kernel *must* interact with the rest of the platform, unlike the critical code in a Trusted Execution Environment (TEE).

SoC vendors struggle with the complex SoC supply chain and the limited time-to-market that forces them to introduce foreign IP blocks into their design. These IPs may include Hardware Trojans or security vulnerabilities and must therefore be considered untrusted components [16]. Solutions such as [4] propose adding additional hardware to perform online security policy enforcement. The inclusion of the platform-level isolation mechanisms mentioned above has also been promoted by this mixed-trust setting [15]—the authors imagine that a blanket platform-trust might have been expected of application-core kernels otherwise.

In [13], an information-flow analysis is performed to ensure non-interference between mixed-trust IPs. This is conceptually similar to our own tool, but performed on the much lower-level of abstraction of hardware gates. Furthermore, it does not target dynamically allocated resources such as memory, but only considers access to special-purpose cores that perform security-critical tasks such as encryption.

6 Conclusion and future work

Our work to specify the i.MX 8X, and at the same time evolve our specification language and tool set to capture the hardware protection features we encounter on the way, is ongoing. We are also starting to specify other SoCs, as a way of extending the convex hull of what we can capture in the tool.

Beyond this, we note that static analysis is a good start, but is limited with respect to expressibility. At some point we would like to make statements based on things that can only be determined at runtime, for example the statement "we trust a DMA engine to only access the descriptors it gets handed".

We do our best to diligently read and understand the hardware manuals and transcribe them to our formal language. However, errors in understanding can occur, and the informal manuals themselves are well-known for acquiring numerous errata over time. Developing a method of tying our model to the actual hardware implementation would greatly increase the confidence that we are faithful to the real hardware. A possible implementation of this could be in the form of litmus tests, where we generate some translation configurations, pick two contexts and then see whether their actual views match the ones we compute.

Sometimes it may be impossible to run tests on all involved contexts. WiFi modems are an example of legal barriers to modifying firmware. In those cases, we might be able to rely on SoC models or simulation to gain some confidence.

Currently, context behavior restrictions have to be identified and written down by us. Automated methods to suggest restrictions would present a significant usability improvement. For example, we already compute reconfiguration paths for some overlaps, and these can give us a good indication where restrictions could be introduced.

Furthermore, we consider our behavior constraints to be absolute: i.e. when we assume a context never accesses a certain part of memory, then we will *always* assume it does so, even if it were compromised by another, malicious context. A more sophisticated (not to mention well-founded) approach to specifying (1) the behavior assumptions and (2) the trust implications of those assumptions would help us in judging how *realistic* the assumptions we are making are.

Nevertheless, we argue that without clear specifications of the protection and translation semantics of the entire hardware platform, reasoning about the isolation properties of the *de facto* OS in any modern computer system will not survive contact with reality.

References

[1] Reto Achermann, Lukas Humbel, David Cock, and Timothy Roscoe. 2018. Physical Addressing on Real Hardware in Isabelle/HOL. In *Proceedings of the 9th International Conference on Interactive Theorem Proving, 2018, Held as Part of the Federated Logic Conference, FloC 2018* (Oxford, UK, 2018) *(ITP'18)*. 1–19. https://doi.org/10.1007/978-3-319-94821-8_1

[2] Reto Achermann, Lukas Humbel, David A. Cock, and Timothy Roscoe. 2017. Formalizing Memory Accesses and Interrupts. In *Proceedings 2nd Workshop on Models for Formal Analysis of Real Systems, MARS@ETAPS 2017, Uppsala, Sweden, 29th April 2017 (EPTCS)*, Vol. 244. 66–116. https://doi.org/10.4204/EPTCS.244.4

[3] Reto Achermann, Ilias Karimalis, and Margo Seltzer. 2023. Why Write Address Translation OS Code Yourself When You Can Synthesize It?. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems* (Providence, RI, USA) *(HOTOS '23)*. Association for Computing Machinery, New York, NY, USA, 174–180. https://doi.org/10.1145/3593856.3595895

[4] Abhishek Basak, Swarup Bhunia, Thomas Tkacik, and Sandip Ray. 2017. Security Assurance for System-on-Chip Designs With Untrusted IPs. *IEEE Transactions on Information Forensics and Security* 12, 7 (July 2017), 1515–1528. https://doi.org/10.1109/TIFS.2017.2658544

[5] Intel Corporation. 2018. *Intel® Software Guard Extensions Developer Guide*. Technical Report. Intel Corporation.

[6] Tobias Dörr, Timo Sandmann, and Jürgen Becker. 2021. Model-Based Configuration of Access Protection Units for Multicore Processors in Embedded Systems. *Microprocessors and Microsystems* 87 (Nov. 2021), 104377. https://doi.org/10.1016/j.micpro.2021.104377

[7] Ben Fiedler, Daniel Schwyn, Constantin Gierczak-Galle, David Cock, and Timothy Roscoe. 2023. Putting out the Hardware Dumpster Fire. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems* (Providence, RI, USA) *(HOTOS '23)*. Association for Computing Machinery, New York, NY, USA, 46–52. https://doi.org/10.1145/3593856.3595903

[8] Xiling Gong, Peter Pi, and Tencent Blade Team. 2019. Exploiting Qualcomm WLAN and Modem Over the Air. , 58 pages. https://www.blackhat.com/us-19/briefings/schedule/index.html#exploiting-qualcomm-wlan-and-modem-over-the-air-15481

[9] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

[10] Arm Ltd. 2021. *Arm Architecture Reference Manual: Armv8, for A-profile architecture*. Arm Ltd.

[11] Arm Ltd. 2021. *Arm CCA Security Model 1.0*. Technical Report. Arm Ltd. ARM-DEN-0096.

[12] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-Button Verification of an OS Kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 252–269. https://doi.org/10.1145/3132747.3132748

[13] Jason Oberg, Timothy Sherwood, and Ryan Kastner. 2013. Eliminating Timing Information Flows in a Mix-Trusted System-on-Chip. *IEEE Design & Test* 30, 2 (April 2013), 55–62. https://doi.org/10.1109/MDT.2013.2247457

[14] Scott O'Brien. 2018. *Extended Resource Domain Controller xRDC*. NXP Semiconductors. NXP AMF-AUT-T3382.

[15] Joël Porquet, Alain Greiner, and Christian Schwarz. 2011. NoC-MPU: A Secure Architecture for Flexible Co-Hosting on Shared Memory MPSoCs. In *2011 Design, Automation & Test in Europe*. 1–4. https://doi.org/10.1109/DATE.2011.5763291

[16] Sandip Ray, Eric Peeters, Mark M. Tehranipoor, and Swarup Bhunia. 2018. System-on-Chip Platform Security Assurance: Architecture and Validation. *Proc. IEEE* 106, 1 (Jan. 2018), 21–37. https://doi.org/10.1109/JPROC.2017.2714641

[17] seL4 Foundation. 2023. Frequently Asked Questions on seL4. online. https://docs.sel4.systems/projects/sel4/frequently-asked-questions.html accessed on 2023-08-17.

[18] Denis Selyanin. 2018. Researching Marvell Avastar Wi-Fi: from zero knowledge to over-the-air zero-touch RCE. https://2018.zeronights.ru/en/reports/researching-marvell-avastar-wi-fi-from-zero-knowledge-to-over-the-air-zero-touch-rce/ Accessed 2022-10-20.

[19] NXP Semiconductors. 2019. *i.MX 8DualX/8DualXPlus/8QuadXPlus Applications Processor Reference Manual*. NXP Semiconductors. NXP IMX8QXPSRM.

[20] NXP Semiconductors. 2019. *NXP® 88W8997 802.11ac wave 2 2 x 2 Wi-Fi® Dual Band with Bluetooth® 5 SoC*. NXP Semiconductors.

[21] NXP Semiconductors. 2022. *NXP-Wireless-Chipset-Release-Notes*. NXP Semiconductors. L5.10.72_2.2.0_WIFI-Doc.

[22] NXP Semiconductors. 2023. *i.MX DSP User's Guide*. NXP Semiconductors. NXP IMXDSPUG.

[23] AzureWave Technologies. 2019. *AW-CM276NF IEEE 802.11 2X2 MU-MIMO ac/a/b/g/n Wireless LAN + Bluetooth 5.0 NGFF Module Datasheet*. AzureWave Technologies.

[24] Bryan Thomas. 2018. i.MX 8 Audio and Tensilica HiFi 4 Overview. NXP Semiconductors presentation AMF-AUT-T3362, https://community.nxp.com/t5/Technology-Days-Training/i-MX-8-Audio-and-Tensilica-HiFi-4-Overview/ta-p/1098861.

https://community.nxp.com/t5/Technology-Days-Training/i-MX-8-Audio-and-Tensilica-HiFi-4-Overview/ta-p/1098861

[25] Yuzuki Tsuru. 2023. FreeRTOS for HIFI4 DSP. https://github.com/YuzukiHD/FreeRTOS-HIFI4-DSP. https://github.com/YuzukiHD/FreeRTOS-HIFI4-DSP

[26] Zhenyu Xu, Thomas Mauldin, Zheyi Yao, Shuyi Pei, Tao Wei, and Qing Yang. 2020. A Bus Authentication and Anti-Probing Architecture Extending Hardware Trusted Computing Base Off CPU Chips and Beyond. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 749–761. https://doi.org/10.1109/ISCA45697.2020.00067