Putting out the hardware dumpster fire

Ben Fiedler
ETH Zurich
Zurich, Switzerland

Daniel Schwyn
ETH Zurich
Zurich, Switzerland

Constantin Gierczak–Galle
ENS Paris
Paris, France

David Cock
ETH Zurich
Zurich, Switzerland

Timothy Roscoe
ETH Zurich
Zurich, Switzerland

ABSTRACT

The immense hardware complexity of modern computers, both mobile phones and datacenter servers, is a seemingly endless source of bugs and vulnerabilities in system software.

Classical OSes cannot address this, since they only run on a small subset of the machine. The issue is interactions within the entire ensemble of firmware blobs, co-processors, and CPUs that we term the *de facto* OS. The current "whac-a-mole" approach will not solve this problem, nor will clean-slate redesign: it is simply not possible to replace some firmware components and the engineering effort is too great.

Our response, instead, is to build a high-level model of exactly what a given real hardware and software platform consists of, and captures for the first time the necessary and assumed trust relationships between the software contexts executing on different components (CPUs, devices, etc.).

This principled but pragmatic approach allows us to make rigorous statements about the hodgepodge of soft- and firmware at the heart of modern computers. We expect these statements to be, at first, depressingly weak, but it may be the only way to identify changes that provably increase the trustworthiness of a real system, and quantify the benefits of these changes.

CCS CONCEPTS

• **Software and its engineering → Operating systems**; *Formal methods.*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
HotOS '23, June 22–24, 2023, Providence, RI, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0195-5/23/06…$15.00
https://doi.org/10.1145/3593856.3595903

KEYWORDS

operating systems, address spaces, hardware specification

ACM Reference Format:
Ben Fiedler, Daniel Schwyn, Constantin Gierczak–Galle, David Cock, and Timothy Roscoe. 2023. Putting out the hardware dumpster fire. In *Workshop on Hot Topics in Operating Systems (HotOS '23), June 22–24, 2023, Providence, RI, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3593856.3595903

1 INTRODUCTION

From a security, reliability, and manageability perspective, computer hardware is a dumpster fire.

Modern computer systems, whether phones or datacenter servers, are composed of hundreds of different processing cores of various designs, most of which are invisible to Linux or whatever environment is presented to applications. These cores execute complex firmware for devices such as security coprocessors, radios, NICs, or storage devices, perform power sequencing, or provide remote system management.

This is causing increasing alarm in the security and system software communities and beyond [7, 10, 19], due to the slew of vulnerabilities and other bugs (see section 2) appearing based on this hardware complexity, often hidden from, or ignored by, OSes like Linux. Indeed, these cores and their firmware usually explicitly sandbox Linux on a corner of the chip (the "application cores"), preventing it taking any meaningful role in managing and securing the platform. On an Android phone, Linux is effectively an application runtime.

Worse, the constant game of "whac-a-mole" fixing these bugs in the face of rapidly changing hardware is a distraction from the deeper problem: we simply lack a coherent framework for talking about the sum total of privileged software running on a modern computer. If, following [19], we take the operating system to mean the privileged software responsible for managing and securing the hardware, it is clear that the "*de facto* OS" of a modern System-on-Chip (SoC) includes not just Linux (or another "classical" OS), but also all firmware and "hidden" cores in the system. Critically, many components of this *de facto* OS cannot practically be

replaced (e.g. deeply-embedded firmware). We must, therefore, construct a holistic model of an operating system for real hardware that reaches some overall security/functionality goal by composing mutually-untrusting (or *partially*-trusting) components.

This paper advocates a radical, but realistic, approach to this crisis. Instead of trying to fix point bugs, or to redesign the software ecosystem from scratch, we start from with a **formal model of the semantics of the hardware platform** which can capture the full panoply of SoC and server designs. From here, we derive a **network of strong trust statements** between execution contexts: (cores, devices, etc.). This is then extended to **virtual contexts** (processes, privilege levels, virtual machines, enclaves, etc.).

What pops out for the first time is, for a given platform and collection of system software, a clear statement about exactly what *must* be blindly trusted by an application program. We expect this to be, initially, "everything", but it provides a solid basis to replace parts of the system software stack on a machine with components that can begin to narrow this trust obligation down. In this way, we start to rule out *a priori* whole classes of cross-SoC hardware-enabled software bugs rather than waiting for exploits to appear.

2 THE PROBLEM

We first discuss the state of hardware-related software bugs and vulnerabilities, the failures that led us here, and why existing approaches are doomed to fail.

2.1 The growing threat of cross-SoC bugs

Protection in a modern computer system is about much more than programming the MMU correctly to ensure isolation between different user processes and the kernel: the OS must interact with hundreds of devices that can access arbitrary memory locations via DMA and increasingly have their own processors, running their own system software. Mutual trust between such devices and their drivers can lead to serious problems, and despite the existence of IOMMUs or System MMUs, new "cross-SoC" attacks which rely on compromising an intelligent device are regularly demonstrated.

Some exploit incorrect or incomplete IOMMU configuration [14, 17], while others exploit subtle features of how data structures are shared with peripheral devices [16]. Preventing such bugs by verification seems hard: many years after its introduction, seL4's correctness proofs either rely on the absence of DMA devices, or assume they are trusted [20].

Often, device firmware is much less rigorously engineered than, say, the Linux kernel, and less likely to be updated. Remote code execution vulnerabilities have been demonstrated for many of the Wi-Fi chips in mobile devices [4, 11, 21]; all exploit buffer overflows using specially crafted packets.

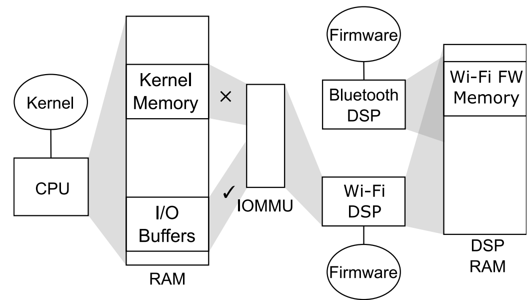

Figure 1: A cross-SoC attack vulnerability

OS kernel's mitigations like Kernel Address Space Layout Randomization (KASLR) and Executable Space Protection (ESP) [13] are often missing from peripherals [4].

Figure 1 shows this: a Wi-Fi Digital Signal Processor (DSP) is compromised over the air, and a further bug in the device driver allows arbitrary RAM to be mapped to the Wi-Fi DSP via the IOMMU, allowing the DSP firmware to access the CPU kernel's private memory and compromise it.

The authors of these attacks all suggest that this is likely the tip of the iceberg for these kinds of problems. Classen et al. show exploits spreading between peripherals without involving the OS kernel on the CPU [8], using buffers shared between Bluetooth and Wi-Fi chips to attack one from the other. In Figure 1, compromised Bluetooth firmware can in turn compromise the Wi-Fi firmware since it can access on-chip RAM containing the firmware.

The current software response is to fix each particular bug in the device driver that allowed the exploit to spread to the main CPU and move on – a game of "whac-a-mole" that results in every new bug opening a window of vulnerability in a large number of deployed devices. With new hardware appearing all the time, this approach is doing nothing to make the problem go away.

2.2 What's really going on?

Stepping back from the endless sequence of vulnerabilities and bugs, there is a fundamental problem here. In principle, it is the function of the OS to provide, using the hardware facilities, protection and isolation between application programs. In these cases, however, the kernel has been completely bypassed by other, highly privileged software running on the machine. Focussing on kernel design is missing the point.

For this reason, we adopt the definition of "operating system" used in [19] as, roughly, *that body of software that manages the machine and securely multiplexes it between applications.* This definition finesses the issue of firmware sidestepping protection in the Linux kernel: this still constitutes a bug in the OS as we define it, even if it is not a bug in Linux per se. It also emphasizes that the OS for

modern platforms is a complex mixture of kernels, firmware blobs, monitors, management controllers and other software entities collectively sharing the hardware.

This is why the "whac-a-mole" approach to bugs will not improve the situation with modern OS deployment, and may make it worse: without a concept of the totality of the "OS" running on a machine, one cannot even start to define what correct or secure behavior might be, let alone provide criteria for improving the system's security or correctness.

Furthermore, this "*de facto* OS" has two important characteristics: First, *it has no design* as such, let alone specification. It has accreted out of parts dictated by hardware designers. Second, parts of this OS *simply cannot be changed* (e.g. proprietary or hard-coded firmware).

2.3 How did we get here?

The current state of affairs, where almost every production computer runs a *de facto* OS which nobody designs, and whose behavior and functionality cannot be specified, has come about due to a set of inter-dependent factors and trends.

One is the need to reuse hardware components, macrocells, chiplets, etc. to cut development time and costs. This leads to "cut'n'paste" hardware design, with additional cores and firmware being added in any way that gets the job done. However, most of these extra processors are also really necessary for energy efficiency, performance, or simply to implement complex I/O functionality. Since it's not clear at design time where in a given platform these units (such as a Bluetooth interface) will sit, the hardware vendors ship them as self-contained units with their own firmware. In some cases, sealing the firmware in the device is a legal requirement, for example for the DSPs implementing wireless basebands, or simply desired by the vendor to protect intellectual property or competitive advantage.

At the same time, traditional OS kernels simply avoid engaging with these parts of the hardware. This trend started long ago, with boot firmware and system management code separated off from the traditional OS, but has accelerated dramatically with the rise of complex heterogeneous SoCs. A kernel like Linux only runs on a small fraction of the processors in the whole system.

This is, in part, because a Unix-like OS is simply incapable of running on more than a homogeneous set of cores sharing a uniform physical address space, and OS developers have preferred to retain their old architecture rather than engage with hardware reality. The result is that, under the definition above, Linux running on a phone is not functioning as an OS (merely a component of one). This abdication further incentivizes hardware vendors to move functionality out of the kernel and into firmware.

2.4 Why a new perspective is needed

We point out that many existing approaches will not, by themselves, solve this problem and result in a deployable OS that is not subject to a stream of hardware-enabled bugs.

Verified kernels [12, 15] are based on a simplistic hardware model, rendering their proofs invalid on realistic hardware. They also, like Linux, are simply not designed to execute on heterogeneous cores which might not fully share memory. We discuss the role that systems like seL4 *might* play in a comprehensive solution in section 3.

IOMMUs have a role in OS security, but do not solve the problem. Modern SoCs include interconnects and devices beyond the scope of the platform IOMMU, and IOMMU drivers are themselves a constant source of OS bugs, in part (we argue) because there is no clear model of the hardware.

Clean-slate OSes can provide insight into good design choices, but ultimately cannot include in their purview all the proprietary firmware in a given platform, nor feasibly achieve compatibility with existing software. The hardware dictates that the *de facto* OS on a modern hardware platform is already a multikernel [6], but a new multikernel by itself does not address firmware blobs or resource management across heterogeneous parts of the hardware, and will not provide application compatibility without a huge engineering effort. Moreover, with no formal description of how hardware components interact such an effort would be futile.

HW/SW co-designs like M^3 [5] similarly limit both software that can run on the OS, and hardware that can be used with it. Future SoC designs can benefit from such insights, although this is unlikely to fully solve the problem of hardware evolution, and the frequent economic imperative to assemble a hardware platform from existing components from diverse vendors. That said, at least one new server company is trying to address the problem in this manner [18].

Secure boot and attestation provide no assurance about the security of firmware othe software, except that it was the software supplied by the vendor. This is a useful feature, but orthogonal to the problem this paper addresses.

Our approach, in contrast, tries to be both *pragmatic* and *rigorous*. Rather than ignoring or bracketing the features that characterize a *de facto* OS (absence of a priori design, and components that cannot be changed), we treat them as a research challenge - arguably the primary OS design challenge today: how can we make strong security and correctness statements about, and then improve, a real-world *de facto* OS under these constraints?

3 PUTTING OUT THE FIRE

What can be done to improve *de facto* OSes, given that the problems they suffer arise from the full system rather than

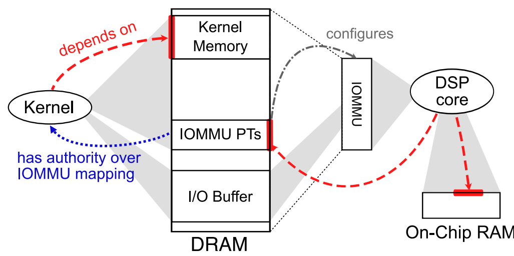

Figure 2: Simplified view of the address spaces, contexts, and authorities involved in QualPwn [11]

individual components, and some of those components themselves may be impossible to change? The new (albeit ambitious) approach we propose is to formally model the structure of entire, real-world *de facto* OSes. We will then use these models to determine what components must be trusted to make the system as a whole trustworthy, and subsequently guide replacement of software and hardware features to provably increase the trustworthiness of the system as a whole.

3.1 Modelling a *de facto* OS

To reason about the behavior of a *de facto* OS, we need to model it in a way amenable to formal and/or informal analysis. As it (by definition) manages the whole machine, the model must encompass the complete platform, but it need not be exhaustive: it's neither necessary nor realistic to cover *every* hardware detail in the presence of closed firmware blobs and components. The problem is therefore not hopeless.

Trust relations between components in the system, both hardware and software, are primarily mediated through memory mapping and address spaces. Isolation between components (*this* process cannot read *that* process' data, the NIC cannot modify kernel data, etc.) can be expressed in terms of which agents (processes, DMA engines, etc.) can read/write which RAM locations and control registers.

Decoding nets [2, 3] already provide a well-studied formal model of the interaction of memory-mapped address spaces on modern hardware, which enables this kind of access-control reasoning *within* both Linux [1] and other, less traditional OSes. Decoding nets describe a system's address mapping hardware as a directed graph, where nodes are distinct address spaces and edges are address translations between them. Figure 2 shows a simplified net: the kernel executes in one address space and there are translations from this to the DRAM address space. Similarly, the DSP core has a distinct address space, with partial mapping from this into on-chip RAM and also the IOMMU address space.

We extend decoding nets to include configurable translation hardware whose behavior is, itself, determined by data in memory. This way we can reason about the fact that, while

```
1  fn qualpwn_example(pt: PageTable) -> DecodingNet {
2    let dn = DecodingNet::new();
3    let linux = dn.add_address_space(0, 4096);
4    let dram = dn.add_address_space(0, 4096);
5    let dsp = dn.add_address_space(0, 4096);
6    // ...
7    let iommu = IOMMU::new(pt);
8    let iommu_as = mmu.insert_into_decoding_net(&mut dn,
        dram);
9    dn.add_mapping(dsp, iommu_as, 0, 0, 4096);
10   // ...
11   dn.mark_context(linux);
12   dn.mark_context(dsp);
13   dn.mark_accepting(dram);
14   dn.mark_dependent(linux, ram, 0, 1024);
15   // ...
16   return dn;
17 }
18
19 enum PTE { Invalid, PageMapping(phys_addr) }
20 struct PageTable { addr: u128, entries: [512]PTE }
21
22 struct IOMMU { pt: PageTable }
23 impl IOMMU {
24   fn new(pt: PageTable) -> MMU { ... }
25   fn insert_into_decoding_net(
26     &self, dn: DecodingNet, pasid: ASID
27   ) -> ASID {
28     let vasid = dn.add_address_space(offset, length);
29     dn.mark_map(vasid, pasid, pt.addr, page_size);
30     dn.mark_grant(pasid, pasid, pt.addr, page_size);
31     for (i, entry) in pt.entries.enumerate() {
32       let va = i * page_size;
33       match entry {
34         PTE::PageMapping(pa) =>
35          dn.add_mapping(vasid, pasid, va, pa, page_size),
36         PTE::Invalid => (),
37       }
38     }
39     return vasid;
40   }
41 }
```

Figure 3: Part of a Sockeye3 description of Figure 2

a core might be unable to access an area of memory, it might access or compromise another core to modify in-memory tables to grant itself access to that memory.

For this we need a specification language expressive enough to encode, for example, the semantics of the ARMv8-A MMU. Our language, Sockeye3, is defined as an abstract syntax and we concretely use Rust to create abstract syntax trees (ASTs) for it. Figure 3 shows part of the Sockeye3 description of the system in Figure 2; address spaces and mappings are added to the decoding net with the corresponding Rust calls. *Accepting* address spaces are final destinations for reads and writes, like DRAM or device registers, while other address spaces translate addresses between them.

A read or write to an address is initiated by a *context*. CPU cores, DMA-capable devices, or any other hardware entity that accesses memory is viewed as a context. However, we also extend decoding nets to *virtual contexts* created by hardware and software running in another context: processes,

enclaves, realms, privilege levels, virtual machines, etc. are all represented as virtual contexts.

The behavior of a context often depends on regions in other address spaces, like the DRAM addresses which hold a kernel's text segment or device registers that control a NIC's operation. These dependencies are depicted using dashed red arrows in Figure 3 and are expressed in Sockeye3 by the `mark_dependent` primitive.

So far we have talked about the decoding net as a static snapshot of the platform, but to model real systems with configurable address translation we also need facilities to create mappings and address spaces based on e.g. in-memory data structures like page tables.

We illustrate this with a simple IOMMU design and a single level page table. In Figure 3, we define the structure of the page table as an array of Page Table Entries (PTEs). A PTE is an enumeration that in this case either represents an invalid entry or a page mapping to a physical address. The translation behavior of the IOMMU is translated to the decoding net representation using `insert_into_decoding_net`.

We must also specify where the page table resides. We call the set of address space regions determining a translation function's behavior its *configuration space*, shown with a dash-dotted gray arrow in Figure 2. Real MMUs with multi-level page tables or nested paging are expressed by modelling the effects of each translation step and composing the input and output address spaces via decoding net mappings.

We have confidence this works: we created an extended decoding net specification of the ARM Confidential Compute Architecture (CCA) hardware component in Isabelle/HOL, including execution levels, worlds, realms, and concurrent access to the protection tables. While not fully complete, it convinced us of the practicality of our approach, particularly in the implications of access to translation tables.

3.2 Deriving trust relationships

From the decoding net constructed by Sockeye3 we now derive a graph of trust relationships between contexts in the system. For each context we extract the set of *critical regions*: all address space regions which either directly influence the behavior of a context, or are configuration regions for translation hardware that translates reads or writes for the given context. If context A can access a critical region of context B, context A can influence context B's behavior.

For example, a Linux kernel's critical regions include not only the kernel text and internal data structures in memory, but also the page tables of *any* MMU or IOMMU which might grant a user process or a device access to that memory.

It follows that to trust a context A to behave "correctly" necessarily entails trusting, transitively, any contexts that can have access to the critical regions of A. This might include other cores or devices in the system.

Conversely, it also makes it explicit precisely which (nontrival) invariants the IOMMU driver must maintain to preserve kernel integrity. The QualPwn bug [11] shows the consequences of not precisely defining these invariants.

3.3 Challenges

This project comes with serious challenges. Some we can address by extending prior work, others are more open-ended.

Firstly, hardware protection mechanisms are recursive: the configuration of an MMU, the page table, is itself stored in MMU-secured memory. This is a source of security bugs, e.g. by fooling an OS into mapping its page tables as DMA-able memory, and introduces formal complexity but is surmountable. Building a secure system entails carefully layering trust relationships such that no such insecure loops can occur.

Decoding nets so far are insufficient to capture this as their current formulation describes only a *current snapshot* of the configuration of translation units and not how the configuration evolves over time. Figuring out exactly which additional semantics are sufficient to write down a useable and useful model of modern hardware is ongoing work, as we require it to be both expressive enough (to capture non-trivial behavior), but still allow us to derive useable trust statements. Our current work exploiting algebraic data types in Rust (building on an earlier version using F*) provides a clean solution to this problem.

Secondly, we need to express different degrees of trust in different memory regions. For example, both page tables and DMA buffers (e.g. filesystem blocks) are held in Linux' kernel memory, and both are potentially vulnerable to compromised components. However, overwriting the page table compromises the kernel itself (and by extension any applications relying on it), while overwriting the buffer might only compromise the application depending on it. If that application itself is untrusted, then from a system-level security perspective, establishing the trust relations for the application buffer is unnecessary. A too-conservative model (e.g. treating all kernel memory as fully trusted) would falsely declare most systems insecure. The right balance between tractability (favoring a conservative model) and precision (favoring a more fine-grained one) is a key challenge.

Finally, establishing the trust requirements for immutable black-box components (such as a cellular baseband stack) will be delicate, but enormously valuable. Some will be simply isolated by rigorous analysis of IOMMU configuration, while others (e.g. power-management cores) will require some degree of trust. A key result of this work as applied to existing hardware will be whether it is possible to assign less-than-complete trust to any of the black-box components in

an SoC, or whether current HW design practices are incompatible with building secure, correct systems. Either we will manage to answer this with: "Yes, you *can* build secure systems on real hardware" or, less optimistically, "This hardware is *fundamentally* insecure, and something needs to change". In either case the answer will be an important signpost for work on securing systems at the HW/SW boundary.

3.4 What do we do with it?

Given a Sockeye3 specification of the hardware and virtual contexts, together with a set of explicit trust assumptions, we can derive the guarantees the *de facto* OS actually provides. We are particularly interested in the familiar concepts of *integrity* (no other components can alter my memory) and *confidentiality* (no other components can read my memory) on a per-component basis. These are, for example, formally verified for processes running under seL4.

Deriving these guarantees involves combining our hard- and software specifications with our trust assumptions, and establishing whether read/write access by an untrusted context is possible. It can be done with modern theorem provers: we are augmenting the existing models for describing memory addressing using decoding nets (as in Figure 3) to reason about physical and virtual contexts, and record our trust assumptions in the same framework, ensuring that we can make precise and formal statements about authority and its derivation. Ideally, this process will be mostly mechanized.

Out of the box, we expect to derive no useful guarantees for an unmodified system without further trust assumptions which we must introduce explicitly: they are not part of our platform specifications, but rather supplement them.

Consider an Android phone which stores biometric data about the user on its fingerprint reader. We want to ensure that biometric data is not accessible by another application, or by a compromised Wi-Fi chip. Assuming we find a sufficiently powerful set of trust assumptions (e.g. the fingerprint reader *itself* does not compromise the data) about the hard- and software deployed on the system, then our model should allow us to derive confidentiality of the biometric data, given a correctly-configured IOMMU.

There are many possible trust assumptions we could choose, ranging from very general ones, such as "the Wi-Fi chip is trusted to never access the fingerprint reader's memory", to very specific statements about the behavior of individual components, such as Linux' ability to correctly program the IOMMU to isolate the Wi-Fi chip from the fingerprint reader. No trust assumption is better or worse than any other; it is simply a question of whether it is at all possible to derive the guarantees we need.

The essence of our approach is a holistic model of the *de facto* OS and hardware that, given a set of trust assumptions which abstract the assumed behavior of unverified components, allows us to establish (or refute) particular security properties, such as integrity.

3.5 Backsolving for trust

Given such a model, we could use a range of techniques (backsolving, parameter search, etc.) to answer "what if?" questions. This is crucial, as without large trust assumptions, the guarantees of an unmodified system are probably "nothing". For example what set of contexts must we trust for biometric confidentiality, or the integrity of a DSP firmware blob? This will guide selective verification: proving the Wi-Fi firmware correct does not help if we still need to trust a Bluetooth modem because it shares critical memory with it. The key insight here is that we do not have to verify everything, since our model can tell us the effect of verification on our guarantees and trust assumptions.

This matters practically because (1) we cannot verify proprietary firmware and probably never will, and (2) the huge effort of verifying some contexts is likely unnecessary.

Moreover, as well as changing the software (replacing untrusted components with verified ones), we can also propose changes to the hardware side. OS architects can now propose guidelines and principles to SoC vendors which demonstrably and measurably improve the trustworthiness of their hardware platforms. We thus move stepwise towards the goal of a reliable, trustworthy OS for an entire complex hardware platform by dividing up and containing the hardware dumpster fire.

4 NEXT STEPS

Our work was motivated by implementing a high-assurance Baseboard Management Controller (BMC) based on seL4 [15] for Enzian [9], a heterogeneous research computer, which made painfully obvious the mismatch between a traditional OS and the *de facto* OS that really runs on an SoC.

We are applying our Rust-embedded language to transform hardware manuals into specifications, with our first target being a complete description of the NXP i.MX8 SoC [23], and analyzing the trust relationships between the multitude of cores and contexts.

There are aspects of hardware platforms that we do not address. Side channels pose challenges if our guarantees must include information flow control. More tractable are critical interconnects in a typical machine like Inter-Integrated Circuit (I^2C) [22] and related management buses [24, 25], where deriving trust relationships is clearer.

Nevertheless, we aim to show, for the first time, how to formally reason about a *de facto* OS, the guarantees it provides on a SoC, and what can be changed to reduce the trust assumptions needed to fulfil those guarantees.

REFERENCES

[1] Reto Achermann, David Cock, Roni Haecki, Nora Hossle, Lukas Humbel, Timothy Roscoe, and Daniel Schwyn. mmapx: Uniform memory protection in a heterogeneous world. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 159–166, New York, NY, USA, 2021. Association for Computing Machinery.

[2] Reto Achermann, Lukas Humbel, David Cock, and Timothy Roscoe. 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*, ITP'18, pages 1–19, 2018.

[3] Reto Achermann, Lukas Humbel, David A. Cock, and Timothy Roscoe. 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*, volume 244 of *EPTCS*, pages 66–116, 2017.

[4] Nitay Artenstein. Broadpwn: Remotely compromising Android and iOS via a bug in Broadcom's Wi-Fi chipsets. *Black Hat USA*, 2017.

[5] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. M3: A hardware/operating-system co-design to tame heterogeneous manycores. *SIGARCH Comput. Archit. News*, 44(2):189–203, mar 2016.

[6] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 29–44, New York, NY, USA, 2009. Association for Computing Machinery.

[7] Bryan Cantrill. I have come to bury the BIOS, not to open it: the need for holistic systems. Open Source Firmware Conference 2022, https://www.osfc.io/2022/talks/i-have-come-to-bury-the-bios-not-to-open-it-the-need-for-holistic-systems/, September 2022.

[8] Jiska Classen, Francesco Gringoli, Michael Hermann, and Matthias Hollick. Attacks on wireless coexistence: Exploiting cross-technology performance features for inter-chip privilege escalation. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1229–1245, 2022.

[9] David A. Cock, Abishek Ramdas, Daniel Schwyn, Michael Giardino, Adam Turowski, Zhenhao He, Nora Hossle, Dario Korolija, Melissa Licciardello, Kristina Martsenko, Reto Achermann, Gustavo Alonso, and Timothy Roscoe. Enzian: an open, general, CPU/FPGA platform for systems software research. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 434–451. ACM, 2022.

[10] Ferhat Erata, Shuwen Deng, Faisal Zaghloul, Wenjie Xiong, Onur Demir, and Jakub Szefer. Survey of approaches and techniques for security verification of computer systems. 19(1), January 2023.

[11] Xiling Gong, Peter Pi, and Tencent Blade Team. Exploiting Qualcomm WLAN and Modem Over the Air. *Proceedings of the BlackHat USA 2019*, page 58, 2019.

[12] Ronghui Gu, Zhong Shao, Hao Chen, Jieung Kim, Jérémie Koenig, Xiongnan (Newman) Wu, Vilhelm Sjöberg, and David Costanzo. Building Certified Concurrent OS Kernels. *Commun. ACM*, 62(10):89–99, September 2019.

[13] C. Harini and C. Fancy. A study on the prevention mechanisms for kernel attacks. In D. Jude Hemanth, G. Vadivu, M. Sangeetha, and Valentina Emilia Balas, editors, *Artificial Intelligence Techniques for Advanced Computing Applications*, pages 11–17, Singapore, 2021. Springer Singapore.

[14] Trammell Hudson and Larry Rudolph. Thunderstrike: EFI Firmware Bootkits for Apple MacBooks. In *Proceedings of the 8th ACM International Systems and Storage Conference*, SYSTOR '15, New York, NY, USA, 2015. Association for Computing Machinery.

[15] 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. SeL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[16] A. T. Markettos, Colin Rothwell, Brett F. Gutstein, A. Pearce, P. Neumann, S. Moore, and R. Watson. Thunderclap: Exploring Vulnerabilities in Operating System IOMMU Protection via DMA from Untrustworthy Peripherals. In *NDSS*, 2019.

[17] Benoît Morgan, Éric Alata, Vincent Nicomette, and Mohamed Kaâniche. Bypassing IOMMU Protection against I/O Attacks. In *2016 Seventh Latin-American Symposium on Dependable Computing (LADC)*, pages 145–150, 2016.

[18] 0xide. https://oxide.computer/, February 2023.

[19] Timothy Roscoe. It's time for operating systems to rediscover hardware. Keynote, 15th USENIX Symposium on Operating Systems Design and Implementation, https://www.usenix.org/conference/osdi21/presentation/fri-keynote, July 2021.

[20] seL4 Foundation. Frequently Asked Questions on seL4. https://docs.sel4.systems/projects/sel4/frequently-asked-questions.html, 2023. accessed on 2013-01-26.

[21] Denis Selyanin. Researching Marvell Avastar Wi-Fi: from zero knowledge to over-the-air zero-touch RCE. *ZeroNights*, 2018.

[22] NXP Semiconductors. I2C-bus specification and user manual. https://www.nxp.com/docs/en/user-guide/UM10204.pdf, April 2014. Rev. 6.

[23] NXP Semiconductors. *i.MX 8DualX/8DualXPlus/8QuadXPlus Applications Processor Reference Manual*, June 2019.

[24] System Management Interface Forum. System Management Bus (SMBus) Specification. http://www.smbus.org/specs/index.html, March 2018. v3.1.

[25] System Management Interface Forum (SMIF), Inc. PMBus^{TM} Power System Management Protocol Specification, revision 1.2. http://www.powersig.org/, September 2020.