Lightweight Hypervisor Verification: Putting the Hardware Burger on a Diet

Charly Castes
EPFL
Switzerland

François Costa
ETH Zürich
Switzerland

Nate Foster
Cornell and Jane Street
USA

Thomas Bourgeat
EPFL
Switzerland

Edouard Bugnion
EPFL
Switzerland

ABSTRACT

Hypervisors are an essential part of our computing infrastructure, yet ensuring their correctness remains a significant challenge for the community. While several hypervisors have been formally verified using traditional methods, they have typically required a huge effort and significant input from verification experts. With the increasing diversity of hypervisors, driven by open hardware and custom ISAs, there is a growing need for more accessible approaches that can be used by non-experts.

This paper advocates for the use of lightweight formal methods for verifying hypervisors. We conduct a top-down analysis of hypervisors and simple correctness criteria on the lock-step execution of the virtual and host machines. By relating the two executions, these criteria transform the task of verifying higher-level properties, such as memory isolation, into simpler conditions that can often be discharged automatically.

We demonstrate the applicability of our approach by developing a verification framework for a RISC-V hypervisor, leveraging the Kani Rust model checker and a Sail specification of the RISC-V architecture. Using our tool, we identified and corrected 21 bugs and proved several properties, including memory isolation, with minimal human effort.

CCS CONCEPTS

• **Security and privacy → Trusted computing**; **Virtualization and security**; *Logic and verification.*

HOTOS 25, May 14–16, 2025, Banff, AB, Canada

ACM ISBN 979-8-4007-1475-7/25/05
https://doi.org/10.1145/3713082.3730373

ACM Reference Format:
Charly Castes, François Costa, Nate Foster, Thomas Bourgeat, and Edouard Bugnion. 2025. Lightweight Hypervisor Verification: Putting the Hardware Burger on a Diet. In *Workshop in Hot Topics in Operating Systems (HOTOS 25), May 14–16, 2025, Banff, AB, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3713082.3730373

1 INTRODUCTION

Virtualization has become ubiquitous over the past few decades. Today hypervisors run on every server in the cloud, and are frequently installed on laptops [32], mobile phones [1], and even IoT and safety-critical devices [20]. But with great success comes great responsibility. With virtualization becoming a de facto standard, hypervisors are playing an increasingly important role in ensuring the security and reliability of our computing infrastructure.

Prior work has shown that it is possible to verify high-level properties for hypervisors using traditional formal methods [5, 18, 19, 21, 27, 31]. But while these results are impressive, they have typically required a huge effort and significant input from verification experts. To scale verification up to the growing number of production-grade hypervisors [2, 3, 9, 11, 13, 20, 32] there is a need for more accessible approaches than can be used by non-experts.

This paper advocates for the use of lightweight formal methods [4, 10, 17, 22, 33] for verifying hypervisors. We do not aim at full verification of general properties, but rather a pragmatic, highly automated, and non-disruptive approach that can verify critical properties and help find bugs.

Our approach takes advantage of two important trends: First, essentially all major architectures in use today are effectively *virtualizable* [24], either because they were designed with virtualization in mind, or because they have been extended to support virtualization [6, 29]. Second, a growing number of architectures come with *high-quality*, *machine-readable* specifications of their semantics [8, 26].

Hence, to lower the verification burden for developers, we can leverage these facts and the unique structure of

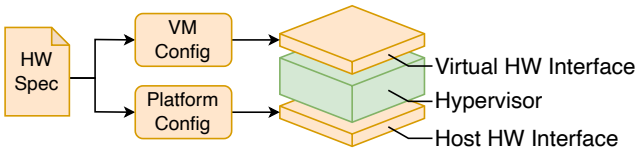

Figure 1: Illustration of the hardware burger. The hypervisor sits in between two well defined hardware interfaces.

hypervisors—which we call the hardware burger, as illustrated in Figure 1—to lift specifications for architectures into specifications for hypervisors. More precisely, building on Popek and Goldberg's classic formalization of hypervisors [24], we identify two criteria, *faithful emulation* and *faithful execution*, that are expressive enough to capture high-level guarantees, such as memory isolation, and yet simple enough to be verified using lightweight methods.

With these criteria fixed, we then develop an automated verification framework for Miralis [12], a RISC-V Rust hypervisor, and report on our experiences. Our framework uses off-the-shelf verification tools and architecture specifications and is exposed to developers as traditional unit tests. It is fully automated and requires no special expertise in formal methods to use. We report on 21 bugs that we found using our framework, none of which had not been detected previously. We also discuss our experiences proving three higher-level properties: correct emulation of privileged instructions, proper delivery of interrupts, and memory isolation.

2 CONTEXT AND BACKGROUND

This section gives background on the two key technologies that underpin our approach: virtualizable architectures and machine-readable architecture specifications.

2.1 Trap & Emulate Hypervisors

In their seminal 1974 paper on formal requirements for virtualization [24], Popek and Goldberg proved that, on virtualizable architectures, a *trap & emulate* hypervisor can provide resources control, establish equivalence, and ensure efficiency. Today nearly all architectures are virtualizable, and all major production hypervisors rely on trap & emulate for virtualization. Rather than focus on a particular implementation, we take a top-down approach and simply assume a trap & emulate hypervisor as described by Popek and Goldberg. Under this model the VM can execute each of its instructions in one of two modes: *emulation* for privileged instructions and *direct execution* for the others. We study each mode in Section 4 when we identify criteria for faithful emulation and execution.

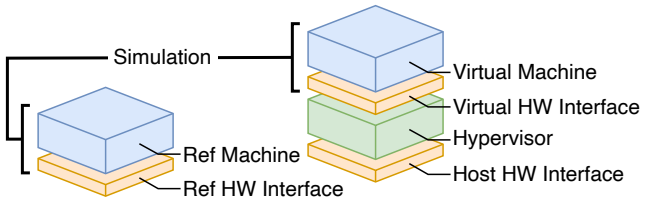

Figure 2: The reference (left) and host (right) machines.

2.2 Lifting Architecture Specifications

Formal specifications are the crux of any verification effort; they constitute the foundation that supports any formal claims. The smallest mismatch between a specification and the real system can void the proofs altogether, or at least severely limit their applicability. As a consequence, writing correct specifications remains a challenge even for lightweight formal methods.

In general, there is no way to escape the need to write specifications—at a minimum, one must at least specify the exposed interfaces of the system being verified [15, 16, 22]. But hypervisors are different. Indeed, the interfaces to a hypervisor are well defined, as it executes on hardware and exposes virtual hardware—this is the hardware burger from Figure 1. Fortunately, in recent years we have seen the emergence of high-quality, well-maintained, and machine-readable specification for most architectures [26]. For instance, the ARM architecture is formalized in ASL [25] while RISC-V has an official model in Sail [8]. This is not only an opportunity to free hypervisor developers from the burden of writing formal specifications, but also a guarantee that the specifications will stay up to date as the hardware evolves. In the rest of this paper, we therefore assume the availability of a machine-readable architecture specification.

3 A SIMULATION PROBLEM

Many of the most important properties that a hypervisor might satisfy, such as correct emulation and memory isolation, are the consequences of a simpler simulation problem—i.e., proving that the virtualized program, *i.e.,* the *virtual machine* (VM), is a simulation of a separate *reference machine*. Figure 2 illustrates the host, virtual, and reference machines. The main complication is that while the host and reference machine usually share the same architecture, they often have different *configurations*, *e.g.,* memory, number of cores, interrupt IDs, or hardware extensions. To illustrate, consider memory isolation: assuming the reference machine is configured with a subset of the host memory, then proving that the VM is a simulation of the reference machine implies that it cannot access the rest of the host memory. In particular, because any such access would trap (for invalid address) on the reference machine, it will also trap on the VM.

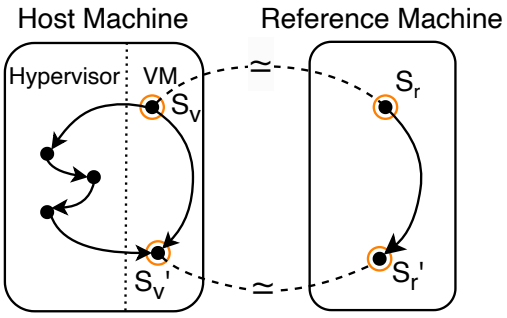

Figure 3: Lock-step execution between a VM and the corresponding reference machine. For each instruction, the VM can either execute the step directly or trap to the host hypervisor for emulation.

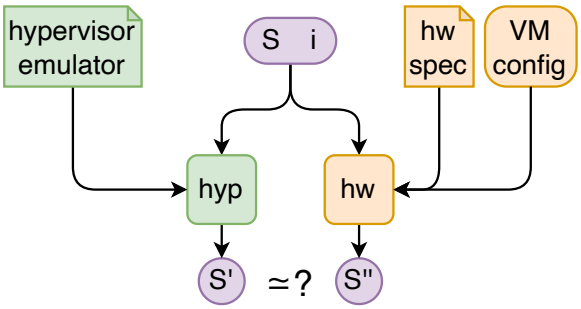

Figure 4: The faithful emulation criteria. A hypervisor implementation properly emulates the virtual hardware interface if for any state and instruction the resulting state is equivalent to what a reference machine would produce.

To prove the simulation relation, we first need a notion of *observability*, after which it suffices to prove that observable states evolve in lock-step. The Popek and Goldberg formalization offers a natural notion in the form of *privileged state* that is useful for defining observability. The privileged state is all architectural state that can only be accessed through privileged instructions[1], and by opposition we refer to all other state as *unprivileged*. Because unprivileged instructions are executed directly by the VM, without interference from the hypervisor, the unprivileged state is what can effectively be observed by the VM. Parts of the privileged state can selectively be made observable during privileged instruction emulation by the hypervisor, for instance by loading the content of a virtual privileged register into a general-purpose register. With this definition of observability, we define lock-step execution as follow:

Definition (Lock-step execution). *The virtual and reference machines execute in lock-step if (i) their initial observable states are equal, and (ii) after each executing an instruction, their observable states remain equal.*

To prove lock-step execution we need to proceed disjunctively by cases, as illustrated in figure 3. If the instruction is privileged it traps to the hypervisor for software emulation. Note that the whole emulation process counts only for a single instruction executed in the VM. Otherwise, it executes directly on the hardware.

4 CORRECTNESS CRITERIA

The difficult problem of verifying high-level properties such as memory isolation is now reduced to the simpler problem of proving lock-step execution. In this section, we formalize two criteria, *faithful emulation* and *faithful execution*, that guarantee lock-step execution during privileged instructions emulation and direct execution of unprivileged instructions. Table 1 summarizes the notations used in this section.

[1]In this context we refer to instructions that trap out of the VM environment to the hypervisor. With virtualization extensions kernel-level instructions are not necessarily privileged.

4.1 Modelling the Architecture

An instruction set defines the transition function of the system's state machine. The transitions depend on the configuration $c \in C$ of the platform (accessible memory ranges, available hardware extensions, number of cores, interrupts IDs, *etc.*) and the current state $s \in S$ of the machine (registers and memory). For the purpose of verification, we also make the next instruction $i \in I$ explicit—i.e., we encode the instruction fetch in the model. Note that interrupts can be modelled as special instructions. We can formalize the transition function as follows:

$$hw : C \times S \times I \to S.$$

Here, the *hw* function encodes the whole architecture—e.g., for the official RISC-V Sail model, it runs to 16k lines of code. If we fix a configuration c, it can also be used as a simulator. Indeed, C and OCaml simulators can be generated from the official RISC-V Sail model. In short, the *hw* function is a high-quality specification of the architecture that already exists. In the following we leverage *hw* to specify the host, virtual, and reference hardware interfaces (see Figure 2), instead of using custom specifications.

4.2 Faithful Emulation

Privileged instructions (and interrupts) executed by the VM trap to the hypervisor for software emulation. The virtual hardware interface constitutes the biggest attack surface that is directly exposed to the software running in the VM. Bugs in the emulation logic can at best disrupt the VM workload, and at worst lead to privilege escalation. We designate the

Table 1: Symbols definitions

Symbol	Definition	Symbol	Definition
C	Set of platform configurations	S_u	Set of unprivileged machine states
c_r	Configuration of the reference machine	s	Machine state
c_h	Configuration of the host machine	p_v	Privileged VM state
I	Set of instructions	u	Unprivileged machine state
I_p	Set of privileged instructions	hyp	Hypervisor privileged instruction emulator
i	Instruction	hw	Executable hardware specification
S	Set of machine states	$hw\|_u$	Restriction of the hardware specification to S_u
S_p	Set of privileged machine states	cfg	Hypervisor hardware configuration function

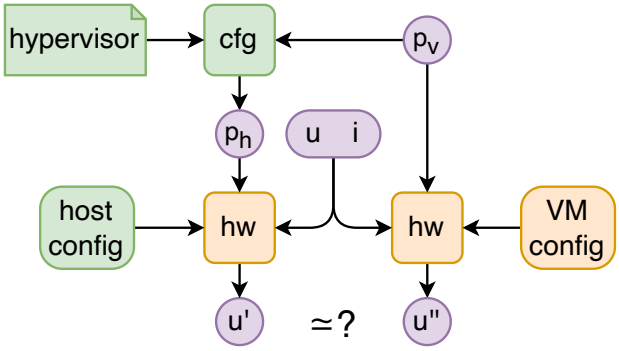

Figure 5: The faithful execution criteria. A hypervisor must configure the host machine such that direct execution produces observable results similar to a reference machine.

emulation function of a hypervisor as hyp, where $I_p \subset I$ is the set of privileged instructions:

$$hyp : S \times I_p \to S$$

The hyp function corresponds to one iteration of the trap, emulate, resume loop that is commonly found in hypervisors. Note that the function can include arbitrary side effects, including scheduling other VMs. Using these definitions, we can define faithful emulation, which states that the hypervisor implements a correct virtual hardware interface:

Definition (Faithful emulation).

$$\exists c_r \in C,\ \forall (s,i) \in S \times I_p,\ \ hyp(s,i) \simeq hw(c_r, s, i)$$

In plain English, a hypervisor should be an accurate emulator of a reference machine, at least for the privileged instructions. Figure 4 illustrates the criteria. The comparison ($\simeq$) might need to take into account differences in internal representation between the hypervisor and the specification.

4.3 Faithful Execution

To enforce lock-step execution during direct execution the hypervisor must configure the host hardware to behave like the reference machine. The difficulty comes from the difference in configuration between the host and reference machine, as well as the need for the hypervisor to maintain its own privileged state. To illustrate this with loads and stores, *i.e.,* prove memory isolation, we present a concrete example involving the RISC-V PMP in Section 6.3.

To make reasoning easier it is helpful to partition the machine's state between privileged and unprivileged state, *i.e.,* $S = S_p \times S_u$. Further, as per the virtualization requirements, the privileged state cannot be modified by unprivileged instructions, thus we consider a restriction of hw to the unprivileged state:

$$hw|_u : C \times S_p \times S_u \times I \to S_u.$$

Faithful execution is linked to a notion of configuration of the host privileged state. During privileged instruction emulation the hypervisor might update the VM's privileged state $p_v \in S_p$, which is often kept in in-memory data structures, and gets a chance to modify the host's own privileged state $p_h \in S_p$, which is installed in the hardware during direct execution. We use $cfg : S_p \to S_p$ to denote the abstract hypervisor function which given a virtual privileged state returns the host privileged state. In practice this function is often incremental: given a change in one virtual privileged register the hypervisor updates the corresponding physical register. For the purpose of verification, capturing the incremental changes is enough to reconstruct the cfg function. With this notation, we define faithful execution:

Definition (Faithful execution).

$$\exists (c_h, c_r) \in C \times C,\ \forall (p_v, u, i) \in S_p \times S_u \times I,$$
$$hw|_u(c_h, cfg(p_v), u, i) \simeq hw|_u(c_r, p_v, u, i)$$

In plain English, the host hardware must be programmed to execute as if the VM was running on a machine with a different configuration. The verification of the host machine programming therefore requires instantiating two hardware interfaces (the two "buns" of the hardware burger), one with the VM platform configuration and privileged state, and another with the host platform configuration and the privileged

state derived from the VM virtual state by the hypervisor. We illustrate the criteria in Figure 5.

While the faithful emulation criterion often catches standard implementation bugs, faithful execution can detect more subtle hardware misconfiguration than cannot be easily caught by language-level analysis or with an incomplete hardware specification. In practice it might be impractical to verify faithful execution for all instructions, but verifying even a subset of instructions can be sufficient to derive desirable properties. For instance, verifying faithful execution of loads and stores is sufficient to ensure memory isolation.

5 CHECKING THE CRITERIA

Section 4 presented the two correctness criteria against an abstract hypervisor model. The rest of this paper discusses how to apply the criteria to verify real systems. The first step is to embed both the specification and the hypervisor implementation into a common representation. Hardware specification languages often support compilation into other languages, for instance the Sail compiler can generate C, OCaml, Coq, and Isabelle, and is easily extensible. The next step is to develop adapters to convert between the internal representation of the machine state of the hypervisor and the specification in order to check for equality. In our experience the adapters tend to be straightforward mappings from one structure layout to another. Finally, the choice of verification framework can be adapted to the needs of each specific project. We found software model checking to be especially effective because architecture specifications and hypervisor implementations tend to avoid complex code patterns that complicate model checking, such as unbounded loops. However, we believe other approaches such as symbolic execution or even fuzzing would also yield good results.

6 AUTOMATED VERIFICATION OF A RISC-V HYPERVISOR

As a case study, we verified faithful emulation and partial faithful execution (memory isolation) of Miralis [12], a RISC-V M-mode hypervisor. Miralis is a relatively small hypervisor (5.6k lines of code) written in Rust whose purpose is to isolate untrusted M-mode software from trusted execution environments. Notably, Miralis was *not* designed with verification in mind; verification was only performed after the fact.

We built an automated verification framework for Miralis using the Kani [30] model checker for Rust. By translating the Sail specification to Rust, the faithfulness criteria are written as simple unit tests that can be executed either with concrete or symbolic values using Kani. This makes verification simple to set-up and comprehend for Rust software engineers. We found and corrected 21 in the Miralis code base, and proved critical properties such as memory isolation.

6.1 Preparing the Architecture Specification

To analyze the specifications of the architectures using Kani, we needed to translate the Sail code to Rust. Unfortunately, the Sail compiler does not yet support compiling to Rust, therefore we developed a new Sail back-end in 2K lines of OCaml. In the future we hope that support for generating Rust will exist upstream and be maintained in the same way as the C back-end is. Using our Rust back-end we generated 6k lines of Rust code from the Sail specification, with full support for all privileged instructions, interrupts, and memory protection. The model is configured by importing external functions, for instance the number of PMP registers is controlled by providing a `sys_pmp_count` function. We selected one set of parameters and fixed them for the purpose of the verification. Finally, we wrote adapters to convert between the internal representation of the RISC-V state of Miralis and the model, as well as checks for equality. In total the TCB includes the Sail model and compiler, Kani, and the Rust compiler, all maintained externally, as well as our Rust back-end and the adapters.

6.2 Checking Faithful Emulation

First, we proved faithful emulation of all privileged instructions and of interrupt delivery. We used the faithful emulation criteria from Section 4.2 with arbitrary (symbolic) state and privileged instruction, executing the instruction in both the Miralis emulator and the reference Rust model. The symbolic execution covers instruction decoding, emulation, and virtual interrupt delivery, but ignores the assembly trap handler which is part of the TCB. This amount for a total coverage of 2.4k lines of code, or just above 40% of the hypervisor code base. We found bugs throughout the whole emulation pipeline, with the biggest concentration in CSR emulation. Thanks to Rust we did not find security flaws due to undefined behavior, although we found multiple opportunities for the VM to crash the hypervisor and mismatch with the RISC-V specification.

During verification we found it especially useful to progressively increase the coverage, making the process of retrofitting formal methods to an existing system much simpler. Indeed, while we can now verify end-to-end emulation correctness, from arbitrary instruction bytes, to the decoder and emulator, we started by verifying instructions one at a time. There are two levers that enable this: the first is simply to verify smaller functions, for instance the decoder only, the second is to constrain the (symbolic) input. We report the verification time for some of the components of the emulation pipeline in Table 2, reported on a M3 MacBook Pro. End-to-end verification takes 118 minutes, making it practical as a nightly CI job to catch regressions, while verifying

Table 2: Verification time of the emulation pipeline

Verification task	Time	Verification task	Time
`mret` instruction	68s	`wfi` instruction	28s
`sret` instruction	56s	instruction decoder	45s
`CSR` read	99s	interrupt virtualization	94s
`CSR` write	9min	end-to-end emulation	118min

components individually only takes a few minutes, allowing interactive debugging sessions.

As a caveat, we did have to slightly modify the RISC-V model to verify Miralis. In total we changed 43 lines of code, or 0.25% of the code base. The two reasons for these changes are the discrepancy between extensions available on the hardware platform and in the model, and the lack of configurability for some aspects of the model. We expect these limitations to disappear as the RISC-V model improves.

6.3 Checking Faithful Execution of Loads and Stores

Second, we proved memory isolation using the faithful execution criteria—i.e., we proved faithful execution for load and store instructions. Miralis relies on PMP (physical memory protection) registers for its own memory protection but also exposes virtual PMP to the VM. As there can be at most 64 PMP registers, Miralis exposes fewer than the host machine. To proceed, we instantiated two hardware interfaces, one with the reference configuration (*i.e.,* fewer PMP registers) and one for the host. We also removed the memory used by Miralis and MMIO from the reference machine, as those should not be accessible from the VM.

The main difficulty in proving faithful execution is to reconstruct the configure function *cfg* from the hypervisor source code. As a reminder, the *cfg* functions returns the target host configuration from a virtual machine state. Indeed, while the *hyp* function is often directly implemented in hypervisors, the host state is usually configured incrementally. To reconstruct the *cfg* function in Miralis we recorded the writes to privileged registers—functionality that already existed for the purpose of unit-testing. We first recorded the writes during hypervisor initialization, then we provided an arbitrary (symbolic) VM state and recorded the writes that occur in response. The first step was necessary as some of the privileged registers are modified at initialization-time only—e.g., the PMP entries protecting the Miralis and MMIO memory. We then installed the VM symbolic state in the reference hardware interface and the state produced by Miralis in the host interface, performing a memory access at a symbolic address with both interfaces and querying Kani for equality. We notably discovered an exploitable integer overflow in the computation of memory ranges. In total memory isolation took 23 minutes to verify.

7 RELATED WORK

This work takes inspiration from previous research in hypervisor testing and automated verification. In particular, the idea of applying hardware development tools to hypervisor development has been explored in [7] where a CPU vendor test suite has been ported to test the KVM implementation. This work additionally leverages the finding of past research that the verification of systems with *finite interfaces* [22, 23] can often be fully automated by SMT solvers [14, 28]. Our method additionally builds on the design of hypervisors to automate verification one step further: by automatically generating the specification.

8 CONCLUSION

This paper demonstrates how to apply lightweight formal methods to hypervisors with minimal human effort. We observe that many high-level hypervisor properties, such as isolation and platform compatibility, are consequences of proper simulation of a reference machine. We propose two criteria, faithful emulation and faithful execution, to model correct simulation in hypervisors. Not only can the criteria be efficiently checked by existing lightweight verification tools but they can also use existing architecture specifications, which removes the need for manual specifications. We built a verification tool for Miralis, a RISC-V hypervisor, and used it to discover 21 bugs and prove several guarantees including memory isolation. Overall, we believe that lightweight verification is a great match for low-level systems code, offering flexibility and ease of use without compromising on formal guarantees.

ACKNOWLEDGEMENTS

The authors would like to thank Adrien Ghosn and Timothy Roscoe, as well as the anonymous reviewers for their valuable feedback. This work has received funding from the Swiss State Secretariat for Education, Research, and Innovation (SERI) under the SwissChips initiative, from the Microsoft-EPFL Joint Research Center, an ONR grant N68335-22-C-0411, a DARPA grant W912CG-23-C-0032, and gifts from Google, InfoSys, and the VMware University Research Fund.

REFERENCES

[1] Android virtualization framework (AVF). https://source.android.com/docs/core/virtualization.
[2] The BSD hypervisor. https://bhyve.org/.
[3] MacOS Hypervisor.framework. https://developer.apple.com/documentation/hypervisor.

[4] Agerholm, S., and Larsen, P. G. A Lightweight Approach to Formal Methods. In *Proceeedings of the International Workshop on Current Trends in Applied Formal Method* (1998), pp. 168–183.

[5] Alkassar, E., Hillebrand, M. A., Paul, W. J., and Petrova, E. Automated Verification of a Small Hypervisor. In *Proceedings of the 3rd International Conference on Verified Software, Theories, Tools and Experiments (VSTTE)* (2010), pp. 40–54.

[6] AMD. Secure virtual machine architecture reference manual, 2005.

[7] Amit, N., Tsafrir, D., Schuster, A., Ayoub, A., and Shlomo, E. Virtual CPU validation. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)* (2015), pp. 311–327.

[8] Armstrong, A., Bauereiss, T., Campbell, B., Reid, A., Gray, K. E., Norton, R. M., Mundkur, P., Wassell, M., French, J., Pulte, C., Flur, S., Stark, I., Krishnaswami, N., and Sewell, P. ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS. *Proc. ACM Program. Lang. 3*, POPL (2019), 71:1–71:31.

[9] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)* (2003), pp. 164–177.

[10] Bornholt, J., Joshi, R., Astrauskas, V., Cully, B., Kragl, B., Markle, S., Sauri, K., Schleit, D., Slatton, G., Tasiran, S., Geffen, J. V., and Warfield, A. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)* (2021), pp. 836–850.

[11] Bugnion, E., Devine, S., Rosenblum, M., Sugerman, J., and Wang, E. Y. Bringing Virtualization to the x86 Architecture with the Original VMware Workstation. *ACM Trans. Comput. Syst. 30*, 4 (2012), 12:1–12:51.

[12] Castes, C., Kalani, N. S., Saltovskaia, S., Terrier, N., Wilkinson, A. V., and Bugnion, E. Kicking the Firmware Out of the TCB with the Miralis Virtual Firmware Monitor. In *Proceedings of the 2nd Workshop on Kernel Isolation, Safety and Verification (KISV)* (2024), pp. 8–15.

[13] Community, T. L. K. Linux kernel virtual machine. https://linux-kvm.org/page/Main_Page, 2007.

[14] de Moura, L. M., and Bjørner, N. S. Z3: An Efficient SMT Solver. pp. 337–340.

[15] Ferraiuolo, A., Baumann, A., Hawblitzel, C., and Parno, B. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)* (2017), pp. 287–305.

[16] Gu, R., Shao, Z., Chen, H., Wu, X. N., Kim, J., Sjöberg, V., and Costanzo, D. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)* (2016), pp. 653–669.

[17] Jackson, D. Lightweight Formal Methods. In *Proceedings of the 2001 International Symposium on Formal Methods Europe* (2001), p. 1.

[18] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D. A., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. seL4: formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)* (2009), pp. 207–220.

[19] Leinenbach, D., and Santen, T. Verifying the Microsoft Hyper-V Hypervisor with VCC. In *Proceedings of the 16th International Symposium on Formal Methods (FM)* (2009), pp. 806–809.

[20] Li, H., Xu, X., Ren, J., and Dong, Y. ACRN: a big little hypervisor for IoT development. In *Proceedings of the 15th International Conference on Virtual Execution Environments (VEE)* (2019), pp. 31–44.

[21] Li, S.-W., Li, X., Gu, R., Nieh, J., and Hui, J. Z. Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor. In *Proceedings of the 30th USENIX Security Symposium* (2021), pp. 3953–3970.

[22] Nelson, L., Bornholt, J., Gu, R., Baumann, A., Torlak, E., and Wang, X. Scaling symbolic evaluation for automated verification of systems code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)* (2019), pp. 225–242.

[23] Nelson, L., Sigurbjarnarson, H., Zhang, K., Johnson, D., Bornholt, J., Torlak, E., and Wang, X. Hyperkernel: Push-Button Verification of an OS Kernel. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)* (2017), pp. 252–269.

[24] Popek, G. J., and Goldberg, R. P. Formal Requirements for Virtualizable Third Generation Architectures. *Commun. ACM 17*, 7 (1974), 412–421.

[25] Reid, A. Trustworthy specifications of ARM® v8-A and v8-M system level architecture. In *Proceedings of the 2016 Formal Methods in Computer-Aided Design Conferenc (FMCAD)* (2016), pp. 161–168.

[26] Sammler, M., Hammond, A., Lepigre, R., Campbell, B., Pichon-Pharabod, J., Dreyer, D., Garg, D., and Sewell, P. Islaris: verification of machine code against authoritative ISA semantics. In *Proceedings of the ACM SIGPLAN 2022 Conference on Programming Language Design and Implementation (PLDI)* (2022), pp. 825–840.

[27] Tao, R., Yao, J., Li, X., Li, S.-W., Nieh, J., and Gu, R. Formal Verification of a Multiprocessor Hypervisor on Arm Relaxed Memory Hardware. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)* (2021), pp. 866–881.

[28] Torlak, E., and Bodík, R. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* (2013), pp. 135–152.

[29] Uhlig, R., Neiger, G., Rodgers, D., Santoni, A. L., Martins, F. C. M., Anderson, A. V., Bennett, S. M., Kägi, A., Leung, F. H., and Smith, L. Intel Virtualization Technology. *Computer 38*, 5 (2005), 48–56.

[30] VanHattum, A., Schwartz-Narbonne, D., Chong, N., and Sampson, A. Verifying Dynamic Trait Objects in Rust. In *ICSE (SEIP)* (2022), pp. 321–330.

[31] Vasudevan, A., Chaki, S., Jia, L., McCune, J. M., Newsome, J., and Datta, A. Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework. In *IEEE Symposium on Security and Privacy* (2013), pp. 430–444.

[32] Velte, A., and Velte, T. *Microsoft virtualization with Hyper-V*. McGraw-Hill, Inc., 2009.

[33] Zave, P. Using lightweight modeling to understand chord. *Comput. Commun. Rev. 42*, 2 (2012), 49–57.