Systematic Mapping Study: Formal Verification of Hypervisors

Lucas Ransan[1,2], Nikolai Kosmatov[1], and Frédéric Loulergue[2]

[1] Thales Research and Technology, cortAIx Labs, Palaiseau, France
{lucas.ransan,nikolai.kosmatov}@thalesgroup.com
[2] Université d'Orléans, INSA Centre Val de Loire, LIFO UR 4022, France
frederic.loulergue@univ-orleans.fr

Abstract. Hypervisors are a fundamental building block of modern computing, supporting cloud infrastructures, virtualization platforms, and security-critical systems. They enable multiple operating systems to run on shared hardware while enforcing strong isolation. At the same time, their low-level position in the software stack makes them particularly sensitive: a single flaw can expose all hosted virtual machines and impact confidentiality, integrity, and availability. Establishing their correctness is therefore essential. In this context, formal verification provides a rigorous means to prove correctness and security properties, relying on mathematically sound foundations. This state-of-knowledge paper surveys existing work on the formal verification of hypervisors. It examines representative projects and approaches, comparing their verification scope, tools, and guarantees. The paper further identifies key challenges, including concurrency, low-level implementation details, and hardware interactions, and highlights current trends and future directions.

Keywords: formal verification · hypervisors · state of knowledge · systematic mapping study

1 Introduction

Hypervisors have become a key component of modern computing systems, underpinning cloud infrastructures, virtualization platforms, and security-critical environments. By enabling multiple operating systems to share the same hardware while providing strong isolation guarantees, hypervisors play a central role in sharing resources, scalability, and system security. Their position at the lowest layer of the software stack, however, also makes them an attractive target for attackers: any vulnerability can compromise all hosted virtual machines, leading to dramatic consequences for confidentiality, integrity, and availability. Type-1 hypervisors, also known as bare-metal hypervisors, run directly on the host hardware and provide high performance and isolation, making them common in enterprise and cloud environments, whereas Type-2 hypervisors run on top of a conventional operating system and are typically used for desktop virtualization, testing, and development due to their ease of deployment and flexibility.

Given this critical role, ensuring the correctness of hypervisors is of paramount importance. Traditional testing and validation techniques are often insufficient to provide the strong guarantees required in such high-assurance contexts. Formal verification has therefore emerged as a promising approach to rigorously establish the correctness and security properties of hypervisors, offering mathematically grounded assurances about their behavior.

This state-of-knowledge paper presents a concise survey of existing efforts in the formal verification of hypervisors. It reviews key projects and methodologies, highlighting their verification scope, applied tools, and achieved guarantees. By synthesizing these contributions, the paper identifies current challenges, such as concurrent code, real-life low-level code, and interactions with hardware, as well as emerging trends and promising directions.

The survey aims to provide both researchers and practitioners with a structured synthesis of the field, offering insights into how formal methods can further strengthen the trustworthiness of next-generation hypervisors.

Outline. The remainder of this paper is organized as follows. Section 2 presents the research questions and the methodology we used to select and classify relevant articles. Section 3 reports the results of our analysis of the selected papers across various hypervisors. Section 4 discusses the lessons learned. Threats to the validity of our findings are addressed in Sect. 5. Finally, Sect. 6 provides a conclusion and an index of acronyms used in the paper.

2 Methodology

We conducted a systematic mapping study (SMS)—following the methodology of Petersen et al. [44, 45]—to identify, classify, and compare published work on the formal verification of hypervisors. The objective of the mapping is to obtain a structured view of a research area whose contributions differ in target hypervisor, verification scope, formal method, proof tool, and treatment of low-level execution aspects.

2.1 Research Questions

The goal of this work is to characterize how formal verification has been applied to hypervisors and hypervisor components. We therefore defined research questions that cover both the verification target and the verification method:

- RQ1: Which hypervisor is verified?
- RQ2: Which hypervisor components are verified?
- RQ3: Is concurrency taken into account during verification?
- RQ4: Which formal methods are used?
- RQ5: Which verification tools are used?

RQ1 identifies the concrete artifact considered by each study, such as an existing hypervisor, a research prototype, or an abstract hypervisor model. RQ2 records the part of the hypervisor covered by the verification effort, for instance memory management, interrupt handling, scheduling, initialization, hypercall

handling, or a complete (small) code base. RQ3 separates sequential verification efforts from those that model interleavings, multicore execution, shared-memory effects, weak memory, or concurrent hypervisor services. RQ4 classifies the verification approach, including deductive verification, model checking, symbolic execution, abstract interpretation, and refinement-based verification; when relevant, we also record proof approaches such as simulation, bisimulation, and forward refinement. RQ5 identifies the concrete verification tool, such as an interactive proof assistant, an automatic verifier, an SMT solver, a model checker, or a combination of these tools.

2.2 Search Strategy

The search was designed to cover peer-reviewed work written in English, without restricting the publication year. We did not impose a lower date bound because the subject is recent and because early work on verified kernels, separation kernels, and virtual machine monitors remains relevant for understanding later hypervisor-verification projects.

We searched major publisher and open-access repositories: ACM Digital Library, SpringerLink, IEEE Xplore, arXiv, and HAL. The search interfaces of these repositories do not provide the same metadata filters. We therefore used two search strings. For ACM Digital Library and SpringerLink, where the search was applied to the full text, we used:

> *"hypervisor" AND ("formal verification" OR "functional verification" OR "model checking" OR "abstract interpretation").*

For IEEE Xplore, arXiv, and HAL, where the search was applied to metadata fields, we used the broader query:

> *"hypervisor" AND ("formal" OR "verification" OR "verified" OR "model checking" OR "abstract interpretation").*

The initial searches returned 920 entries. After merging duplicate records and screening titles and abstracts, 66 entries remained for full-text analysis. Figure 1 summarizes the selection process.

2.3 Study Selection

We defined inclusion and exclusion criteria before the final screening step. A paper was included when it reports the use of formal methods to specify, model, verify, or validate a hypervisor or one of its components. We also included work on separation kernels when the artifact provides virtualization or hypervisor-like isolation services, or when the paper is explicitly positioned as part of a hypervisor-verification effort. This decision reflects the fact that the boundary between hypervisors, virtual machine monitors, and separation kernels is sometimes technical rather than terminological: several verified systems provide memory isolation, partitioning, interrupt control, and controlled communication without using exactly the same vocabulary.

We excluded papers that only use a verified hypervisor as an execution platform, without contributing a verification result about the hypervisor itself. We

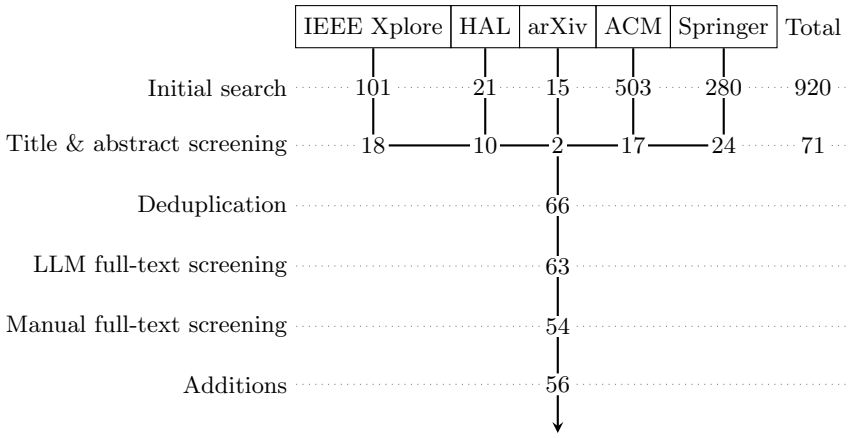

Fig. 1. Paper selection methodology.

> You are an academic research assistant. Determine if the following paper is about the formal verification of hypervisors.
>
> Title: *title inserted here*
> Abstract: *abstract inserted here*
>
> Answer strictly with 'Yes' if it is directly about the formal verification of hypervisors. Answer 'No' if it is unrelated, or about a different topic. Only output 'Yes' or 'No'.

Fig. 2. Prompt for LLM-assisted abstract screening.

also excluded papers outside the scope of hypervisors or hypervisor-like isolation mechanisms, unavailable papers, non-peer-reviewed material, and theses when the relevant peer-reviewed publication was already included in the corpus. When several papers described successive stages of the same verification project, we retained the papers needed to represent the scope and evolution of the project, and grouped them during the mapping phase.

2.4 Screening Procedure

The first screening step was performed on titles and abstracts. We used a procedure assisted by large language models (LLMs) to reduce the cost of screening the 920 initial entries. A Python script submitted the title and abstract of each paper to Xiaomi's MiMo-V2-Flash model with the prompt shown in Fig. 2. Manual screening of the titles and abstracts of the papers filtered out by an LLM was done only selectively due to their large number.

This step reduced the data set to 71 entries and after removing duplicate entries, 66 papers remained. We then conducted a full-text screening. For this step, the PDF files were submitted to Google's Gemini 3 Flash with instructions

to determine whether the paper concerns the formal verification of hypervisors and to extract preliminary answers to the research questions. The resulting summaries provided a first pass over the papers and led to the exclusion of three papers. These papers were also double-checked manually.

The final inclusion decision and the final answers to the research questions were established by reading the papers. During this manual validation step, we corrected and completed the LLM-produced information and excluded eight additional papers. The reasons for exclusion were the following:

- two papers were not available to us;
- four theses contained material already covered by included peer-reviewed papers;
- four papers used a verified hypervisor but did not verify any part of that hypervisor;
- two papers were not about hypervisors.

Finally, we added two papers we knew, relevant to the study, but not present in the obtained 54 papers after the previous steps.

The use of LLMs is therefore limited to screening support and preliminary extraction. The claimed classification is based on manual inspection of the selected papers. This distinction is important because several papers use adjacent terminology, such as virtual machine monitor, separation kernel, secure monitor, or partitioning kernel, and require a reading of the technical content to decide whether they belong to the mapping.

2.5 Mapping of Studies

For each included paper, we recorded the answers to the five research questions in a structured extraction sheet. The extraction focused on the verified artifact, the verified components, the concurrency model, the formal method, and the tool chain. When the paper reported proof metrics, proof automation, or trusted assumptions, we also recorded this information to support the discussion, although these data were too heterogeneous and therefore were not used as independent research questions in the classification table.

Some verification efforts are reported across several papers. This is the case, for example, when an initial model is later extended to machine code, when a memory-management proof is followed by an interrupt-handling proof, or when the same hypervisor is considered at different abstraction levels. We therefore group related papers by verification project in the results section. This grouping avoids counting a long-running verification effort as unrelated studies, and provides readers with a more structured view by explicitly indicating which components and proof techniques belong to the same verified artifact.

3 Results

In Table 1, we directly answer the five research questions.

In the remainder of this section, we present a brief summary of verification efforts for each hypervisor.

Table 1: Classification of the eligible studies

Refs	Hypervisors	Components	Methods	Tools	C†
[36, 31, 11]	Anaxagoros	Memory management	Deduct. verif.	Frama-C	●
[55]	ARLX[1]	Scheduling	Deduct. verif.	ACL2	○
[3, 5, 4, 43, 50]	Baby hypervisor	Whole hypervisor	Deduct. verif.	VCC	●
[21]	Bao	Cache coloring	Deduct. verif.	Frama-C	○
[37]	BedRock HyperVisor[2]	Hypervisor's core	Refinement	Coq, BRiCk	●
[58, 60]	CASMonitor[1]	Whole hypervisor	Model checking	Spin, PROMELA	◐
[59]	CertiKOS	Interrupt injection	Deduct. verif.	Coq	○
[35]	Hafnium	Hypercall interface	Deduct. verif.	Coq, Iris	○
[7, 8]	HASPOC hypervisor	Hypervisor's core	Deduct. verif.	HOL4, BAP	◐
[32, 15]	Hyper-V	Whole hypervisor	Deduct. verif.	VCC	●
[18]	HyperEnclave	Memory management	Deduct. verif., Refinement	Coq, mirlightgen	◐
[41, 26, 40]	KVM	Memory management	Refinement	HOL4	○
[17, 38]	MinVisor	Memory management	Deduct. verif., Symb. exec.	ACL2	○
[13]	Miralis	Instr. emulation, mem. management	Model checking, Symb. exec.	Kani, Sail	○
[29]	Muen Separation Kernel	Memory management, scheduling	Refinement	Ada SPARC, GNAT	○
[54]	NOVA	Memory management	Deduct. verif.	PVS	○
[10]	NOVA	Hypercall interface	Model based testing	Coq	○
[42]	Phidias	Whole hypervisor	Symb. exec.	Custom	◐
[47]	pKVM	Memory management	Deduct. verif.	CN	○
[14]	pKVM	Memory management	Symb. exec.	Tpot	○
[19, 20]	PROSPER	Whole hypervisor	Deduct. verif.	STP, BAP	○
[22]	Realm Management Monitor	ABI, Memory management	Deduct. verif., Model checking	HOL4, CBMC	◐

Continued on next page

Table 1: Classification of the eligible studies (Continued)

Refs	Hypervisors	Components	Methods	Tools	C†
[1]	Security MicroVisor	TCM boundaries	Deduct. verif.	Isabelle-/HOL, BAP	○
[12]	SecT's hypervisor	Memory management	Refinement	Prove & Run	○
[24]	SecVisor, sHype[3]	Memory management	Model checking	Murphi (Muro)	●
[52]	ShadowVisor, SecVisor, sHype[3]	Mem. manag., access control	Model checking	UCLID, Plingeling	○
[34, 53, 33]	SeKVM	Hypervisor's core	Deduct. verif.	Coq	●
[28]	TbaaS's hypervisor	Hypervisor's core	Model checking	PAT	●
[6]	Unnamed	Memory management	Deduct. verif.	Coq	○
[46]	Unnamed	Memory management, I/O	Refinement	AtelierB	○
[49, 48]	Unnamed	Whole hypervisor model	Deduct. verif., Symb. exec.	Coq	○
[51]	Xen	Access control	Model checking	UVHM	○
[2]	Xen	Access control	Model checking	SAL, PVS	●
[57]	Xen	Userspace utilities	Deduct. verif.	Isabelle-/HOL	○
[16]	Xen	Hypercall interface, mem. manag.	Model checking	CBMC	○
[23]	Xen, ShadowVisor	Memory management	Model checking	CBMC	●
[39, 25]	Xenon[1]	Hypercall interface	Refinement	Circus	●
[56]	XMHF	Hypervisor's core	Model checking	CBMC	○
[30]	Xvisor, Xen, Bao	Interrupt injection	Deduct. verif.	Isabelle-/HOL	○

† Concurrency [1] Based on Xen [2] Based on NOVA [3] Xen extension

Anaxagoros. Anaxagoros is a secure microkernel that also functions as a cloud-oriented hypervisor. Three studies [36, 31, 11] focus on the virtual-memory subsystem, which is responsible for (i) enforcing that a guest may access only pages it owns, (ii) guaranteeing that pages are used according to their declared type, and (iii) maintaining per-page reference counters that bound the number of active mappings. All works employ deductive verification on C code annotated in ACSL and processed with the Frama-C toolset. Loulergue et al. [36] mention an ongoing effort based on the Jessie and WP plugins, while Kosmatov et al. [31] augment

this approach with structural all-path testing (PathCrawler) to gain confidence on the small portion of code that automatic provers cannot discharge. Blanchard et al. [11] further leverage the WP plugin together with SMT solvers and use the Coq[3] proof assistant to discharge residual lemmas concerning counting predicates and concurrency-simulation constructs. The combined methodology isolates unproved fragments, simulates interleavings of atomic steps to handle multi-threaded execution, and proves that the global invariant—stating that the actual number of mappings to any valid page is never larger than its counter and never exceeds the maximum allowed value—is preserved.

Baby Hypervisor. In the Verisoft XT project, the Baby Hypervisor was built as a minimal hypervisor for a simplified 32-bit RISC architecture (Baby VAMP). Alkassar et al. [5] gave the first functional verification of this hypervisor using the automatic C verifier VCC; they proved that the hypervisor correctly simulates each guest by establishing coupling invariants between the concrete host state and the abstract guest state, covering initialization, guest scheduling and the core page-fault handling path. The same team also formalized the shadow-page-table algorithm and proved correctness of its memory-virtualization invariants [3][4]. A further advance was made by Alkassar et al. [4], who modeled a translation lookaside buffer (TLB) in VCC and verified the shadow page table algorithm against a detailed hardware memory management unit (MMU) specification, showing that the virtual TLB supplied to a guest follows the prescribed hardware transition relation. The original verification only considered the C code; the assembler portions (guest register save/restore, interrupt entry/exit) were left unverified. Paul, Schmaltz, and Shadrin [43, 50] filled this gap by defining an integrated semantics of a C intermediate language and a Macro-Assembler, translating the Macro-Assembler code to C, and re-verifying it with VCC, thus attaining full code coverage of the hypervisor.

Bao. Bao is a light open-source static hypervisor, intended for embedded and real-time systems. Férreol et al. [21] focus on verifying the functional correctness of the hypervisor's memory-management subsystem that implements cache coloring: the code that computes the color of a page, searches for a consecutive set of free pages whose colors belong to a user-specified color vector, and marks those pages as allocated. The authors use deductive verification: the C code is annotated in ACSL and processed with the Frama-C/WP plugin, which generates verification conditions discharged automatically by SMT solvers (Alt-Ergo, CVC5, Z3) and, when necessary, interactively in the Coq proof assistant. The main contributions are a complete formal specification of Bao's cache-coloring mechanism, a machine-checked proof of the listed properties, and the identi-

[3] Recently renamed to Rocq.

[4] Although Alkassar et al. [3, 4] do not explicitly name the hypervisor they verify, we attribute this work to the Baby Hypervisor based on its inclusion in the Verisoft XT project and the chronological alignment with the Baby Hypervisor verification effort reported in [5].

fication of two subtle bugs in the original implementation together with their fixes.

Huang et al. [30] focus on Bao's interrupt-virtualization subsystem. The work models the hypervisor's handling of virtual interrupts in the list registers and the associated in-memory "spilled list", together with the interrupt handlers that govern overflow and preemption. Using deductive verification in the interactive theorem prover Isabelle/HOL, the authors encode the state of the virtual generic interrupt controller (GIC), the list register cache, and the pending list as higher-order logic records and define transition relations for a single interrupt-injection step. Within this framework they prove a lemma showing that, under the naïve injection policy (insert into the pending list when list registers are full), a higher-priority interrupt can be delayed behind lower-priority ones—a priority-inversion property that the model captures and that reflects a potential correctness issue in Bao's design.

CertiKOS. CertiKOS is a research hypervisor built for the x86 architecture that relies on Intel VMX (Virtual Machine Extensions) and AMD SVM (Secure Virtual Machine, also called AMD-V) extensions. Zhang and Guo [59] focused on the interrupt-injection subsystem of CertiKOS: they modeled the non-virtualized interrupt behavior (including the previously unmodeled interrupt-shadow) and the virtualized interrupt controller, and then proved two key properties: safety (the hypervisor never gets stuck while injecting) and reliability (injected interrupts have the same effect as on bare metal). The verification is carried out deductively in the Coq proof assistant; the abstract machines M_0 (bare-metal) and M_1 (virtualized) are defined in Coq, and a simulation relation is proved between them.

Hafnium. Hafnium is a thin hypervisor that implements the ARM Firmware Framework for A-profile (FF-A) and is used to isolate security-critical components such as keychains and digital wallets from the main operating system. Liu et al. [35] formalized a subset of the FF-A hypercall application binary interface (ABI) as realized by Hafnium and verify the correctness of the memory-management and scheduling primitives that the hypervisor provides. The verification is carried out by deductively proving properties in the Iris proof assistant: functional correctness of the hypercall steps, preservation of page-ownership invariants, and a robust-safety guarantee that even a compromised guest virtual machine (VM) cannot breach the integrity of pages belonging to other VMs. The main contribution of the work is the development of VMSL, a novel higher-order separation logic that enables modular reasoning about individual VMs and their interactions, together with a mechanized soundness proof in Coq.

HASPOC. The HASPOC hypervisor is a lightweight static hypervisor that runs at EL2 on ARMv8-A processors and forces inter-guest communication through prescribed unidirectional channels. Verification effort by Baumann et al. [7, 8] focuses on the memory-management subsystem, the interrupt controller configuration, peripheral ownership rules and the overall integrity of the hypervisor

state. The approach combines deductive verification in the HOL4 theorem prover with a machine-code level proof: an ARMv8 model is formalized, the hypervisor design is captured as a refined HOL4 specification, and a bisimulation with an idealized physically isolated model is proved; binary correctness of the compiled code is then discharged using the Binary Analysis Platform (BAP) to generate proof obligations that are solved by an SMT solver.

Hyper-V. Microsoft Hyper-V is a widely used type-1 hypervisor that runs on x86-64 hardware. The verification effort reported by Leinenbach and Santen [32] focuses on the kernel stratum of Hyper-V, covering the memory manager (including the shadow-page-table algorithm), the multiprocessor scheduler, and the basic hypercall interface. Functional correctness (e.g., that the guest view of memory matches the composition of guest and host page tables) and safety properties (such as memory safety and absence of data races) are proved using deductive verification with the VCC tool chain, which translates annotated C (and a small amount of assembly) into Boogie verification conditions that are discharged by the Z3 SMT solver. Cohen et al. [15] complement this work by developing a theoretical foundation for multi-core hypervisor verification. They model a concurrent instruction set architecture (ISA) (including store buffers, caches and MMUs), define an ownership-based programming discipline, and prove reduction theorems that relate the concrete multi-core execution (ISA-sp) to an abstract sequential model (ISA-u). Their theory also justifies the sound use of VCC for reasoning about concurrent C code, assembly fragments, and device interaction, and provides a simulation theorem for the shadow-page-table algorithm on processors without hardware two-level translation.

HyperEnclave. HyperEnclave is a trusted execution environment (TEE) implemented in Rust as an hypervisor. The verification effort described by Dai et al. [18] concentrates on the hypervisor's memory-management stack, namely the allocation of physical frames, the manipulation of page-table entries, and the construction of address-space structures that guarantee spatial isolation between enclaves, the primary OS, and applications. The authors prove two kinds of properties: functional correctness of the paging subsystem (the code correctly implements the intended page-table updates) and an information-flow non-interference property (an untrusted domain cannot read or overwrite enclave memory). To obtain these results they adopt a deductive, layered verification methodology based on the CCAL[5] framework: a lightweight operational semantics for Rust's Mid-level IR (MIR) is formalized in Coq, the hypervisor's MIR is automatically emitted by the tool mirlightgen, and a series of refinement proofs relates the concrete MIR code to an abstract functional specification and finally to the high-level security model.

KVM & Derivatives. KVM (Kernel-based VM) is a virtualization kernel module that allows Linux to function as a hypervisor. Nemati et al. [41] formalized the

[5] Certified Concurrent Abstraction Layers

memory-management component of a minimal ARMv7 hypervisor using direct-paging and proved memory isolation and information-flow security in HOL4, refining an abstract model to C code against an ARMv7 ISA model. Guanciale et al. [26] demonstrated that this cache-free model is unsound: virtual aliases with mismatched cache attributes or self-modifying code induce cache incoherence, breaking verified isolation. Building on this, Nemati et al. [40] provided a HOL4 framework for verifying integrity-preserving countermeasures by composing hardware obligations (MMU domain, cache-coherency) with existing proofs.

SeKVM retrofits Linux KVM by restricting the trusted computing base to a small core (KCore) in EL2, while remaining services (KServ) run under the host kernel in EL1. Li et al. [34, 33] verified isolation components—stage-2 page tables, TLB maintenance, cache flush, SMMU handler, and hypercall dispatcher—using non-interference invariants in Coq with CompCert-parsed code and transparent-refinement layers. Tao et al. [53] extended this with a relaxed-memory layer, proving SeKVM satisfies six weak-data-race-free conditions under Arm's Promising-Arm model.

Protected KVM (pKVM) powers Android's Virtualization Framework, with verification targeting shadow page-table management, VM-exit handling, and vCPU scheduling to ensure memory safety, strong isolation, functional correctness, and deterministic scheduling. Pulte et al. [47] applied deductive verification in Coq using VST and CompCert, refining abstract specifications to C implementations. Cebeci et al. [14] complemented this with model checking (CBMC) on assembly fragments and symbolic execution (KLEE) on device I/O paths.

MinVisor. MinVisor is a minimal type-1 hypervisor for the x86 platform that uses AMD-V's nested paging to isolate the hypervisor's own memory, the VM Control Block and selected model-specific registers (MSRs) from a potentially malicious guest. The verification effort reported by Dahlin et al. [17] focuses on the code that creates the nested page tables, proving that the resulting translation is an identity map for all addresses outside the protected region and forces a page-fault for any address that belongs to MinVisor's memory image. McCoyd et al. [38] extend this work by showing that, once the page tables are established, the protected resources remain unchanged during arbitrary guest execution, i.e., the hypervisor's memory and the guarded MSRs (control registers) are invariant under all guest instructions that do not trigger an intercept. Both papers employ a deductive, bottom-up approach: the implementation is translated to a simplified Y86++ instruction set, annotated with compositional cut-points (inductive assertions), and the resulting verification conditions are discharged automatically by the ACL2 theorem prover.

Miralis. Miralis is a small, open-source RISC-V hypervisor written in Rust that runs in M-mode (Machine mode, i.e. the highest privilege level) and isolates untrusted M-mode code from a trusted execution environment. In Castes et al. [13], the verification effort focused on two major subsystems: the instruction-emulation path for privileged instructions and interrupts, and the memory-protection mechanism that underlies isolation. Using a model-checking approach,

the authors instantiated the official Sail specification of RISC-V as a Rust library and employed the Kani Rust model checker to symbolically execute the hypervisor code against the reference model. This setup allowed them to state the "faithful emulation" and "faithful execution" criteria as unit-test-like assertions, proving that Miralis correctly emulates all privileged instructions and delivers virtual interrupts, and that loads and stores respect the intended memory isolation guarantees.

Muen Separation Kernel. The Muen separation kernel is a small, generative hypervisor that runs on x86-64 processors with Intel VT-x support and produces a tailor-made kernel, page tables and subject binaries from an XML-based policy describing a fixed set of subjects, their memory channels and a cyclic schedule. In Haque et al. [29], the verification effort focuses on the kernel's memory-management and scheduling infrastructure: the page-table construction, the enforcement of read/write/executable permissions, the handling of the VT-x virtual-machine control structures, and the timer-driven scheduler that switches subjects according to the major/minor-frame schedule. To reason about these components the authors introduce a conditional parametric refinement methodology and carry out the proof in the SPARK Ada environment, discharging verification conditions with automated SMT solvers. The main contribution is a two-step approach that first proves, once for all, that the parametric kernel template refines an abstract specification under a set of natural assumptions on the parameter values (injectivity of page tables, correct permission encoding, etc.), and then checks those assumptions for each generated configuration using a lightweight condition-checking tool. This yields machine-checked conformance proofs for dozens of policies and also uncovers latent bugs such as undeclared shared memory channels.

NOVA & Derivatives. NOVA is a type-1 micro-hypervisor for x86 that is written in C++ with a small amount of inline assembly. The verification effort described by Tews et al. [54] concentrates on the low-level memory subsystem: a stack of three formal memory models (physical RAM, memory-mapped devices, and page-table based virtual memory) is developed in the interactive theorem prover PVS and shown to satisfy a plain-memory abstraction. Using this stack, the authors verify that accesses to blessed memory regions preserve values, that page-table updates respect alignment and reserved-bit constraints, and that reads/writes to device registers respect type-, alignment- and access-mode restrictions.

Becker et al. [10] adopt a complementary, model-based approach for the same hypervisor. An abstract Coq model of NOVA's state (processes, threads, capabilities, scheduling objects, etc.) is defined, and high-level security properties such as authority confinement and memory confinement are proved by induction on execution traces. The Coq development is extracted to OCaml, which is then used as an executable oracle in a conformance-testing framework that runs generated traces on the actual NOVA implementation and compares final states.

This combination of deductive reasoning (Coq proofs) and systematic testing bridges the gap between the abstract model and the low-level C++ code.

Malecha et al. [37] extend the verification frontier to the commercial BedRock HyperVisor, which builds on NOVA's micro-kernel. The BedRock HyperVisor is verified using the BRiCk program logic for C++ (implemented on top of the Iris separation-logic framework in Coq). Verification targets include lock-free data structures, the inter-process communication subsystem, and the hypervisor's round-up protocol; proofs are expressed as Hoare triples over C++ code and rely on Coq's automation, custom hints, and extraction of verified code.

Phidias. Phidias is a deliberately minimal static type-1 hypervisor intended for embedded systems. Nordholz [42] verify two principal safety properties of this static kernel: (1) hypervisor integrity—i.e., the immutability of the hypervisor's code, page tables and internal data structures—and (2) guest non-interference, ensuring that no hypervisor code path can leak information from one guest to another. The verification is carried out by a custom symbolic-execution engine that directly interprets the compiled bootable image; symbolic states are fed to the Z3 SMT solver to check invariants on each explored path.

PROSPER. PROSPER is a minimal separation kernel for the ARMv7-A architecture that runs two statically-allocated guest partitions, enforces isolation via the MMU, schedules the guests with a static round-robin policy, and provides asynchronous message-passing through hypercalls. In Dam et al. [19, 20], the verification effort covers the MMU configuration, the scheduler, and the hypercalls that implement message delivery. The main security property proved is a trace-equivalence between the concrete implementation and a top-level ideal specification, which yields the usual non-interference style guarantees of no-exfiltration and no-infiltration. To obtain these results the authors used deductive verification in the HOL4 theorem prover for the architectural lemmas and the overall bisimulation proof, and they combined this with binary-code verification of the kernel handlers: the ARMv7 assembly was lifted to BAP's intermediate language, weakest preconditions were generated, and the resulting formulas were discharged by an SMT solver (STP).

Security MicroVisor. Security MicroVisor (SμV) is a lightweight software-based hypervisor intended for IoT microcontrollers that lack a hardware MMU. It partitions the device memory into a trusted computing module (TCM) and an untrusted application region, enforcing that code residing in the application region may only read or write its own data and may invoke TCM code solely through designated entry points. Ahmadi et al. [1] model the memory-management and control-flow mechanisms of SμV and verify that the following properties hold: (i) an untrusted application cannot read from or write to the TCM, and (ii) it cannot jump to or call arbitrary TCM instructions except at the safe entry points. The verification is carried out by a deductive approach in Isabelle/HOL: assembly programs generated from C source via the Binary Analysis Platform

are encoded as Isabelle theories, an adversary model is defined, and a suite of conformance predicates is proved to be satisfied for all reachable states.

Xen & Derivatives. Xen is an open-source, dynamic type-1 hypervisor that runs a privileged control domain (Dom0) and multiple guest domains (DomU). She et al. [51] introduced a lightweight binary-level model-checking pipeline that proves a safety invariant of the access-control module, called XSM-ACM, directly from the compiled image. Alexander et al. [2] built a configurable model in the SAL model checker for Xen's inter-VM communication and mandatory access control (MAC) mechanisms, verifying both confidentiality/integrity soundness properties and boot-progress completeness theorems for systems with up to nineteen domains. Verbeek et al. [57] combined automated invariant generation, symbolic execution of the x86-64 instruction set, SMT solving, and Isabelle/HOL to certify memory-usage theorems for hundreds of functions extracted from production Xen binaries, showing that return-address integrity can be proved with modest manual effort. Cook et al. [16] applied bounded model checking (using the CBMC tool) together with aggressive slicing and assembly modeling to the hypercall interface and page-table checks, enabling rapid reachability analysis of real security advisories. Franklin et al. [23] presented a parametric guarded-command model of Xen's shadow-paging and context-caching subsystem and proved address-space separation for arbitrarily many VMs using small-model reduction and CBMC. Finally, Huang et al. [30] formalized Xen's ARM interrupt-injection policy in Isabelle/HOL and showed that higher-priority virtual interrupts are never lost, providing a machine-checked guarantee for this component.

ARLX (ARINC 653 Real-time Linux on Xen) extends the open-source Xen hypervisor to the ARINC 653 standard for safety-critical avionics systems, implementing time and space partitioning along with I/O bandwidth management. In Vanderleest et al. [55], the formal verification effort, conducted by Rockwell Collins, focuses specifically on the ARINC 653 scheduling subsystem. The analysis targets security properties related to information flow control and resource isolation between partitions. The methods employed include structural analysis for call tree construction, classification of 195 global variables and 375 data structure types, and information flow analysis using the Data Flow Logic (DFL) framework developed by Rockwell Collins. DFL leverages a C intermediate language as a frontend and ACL2 as the underlying theorem prover, enabling automated verification of information flow properties specified through C annotations.

CASMonitor is a secure, high-assurance hypervisor prototype derived from the open-source Xen hypervisor, designed to meet the B3 level or higher of the TCSEC (Trusted Computer System Evaluation Criteria) standard[6]. Wang et al. [58] use the Spin model checker to verify design-level properties, including the correctness of the Chinese-Wall security policy (ensuring mutually exclusive domains cannot run simultaneously), consistency of hypercall communication (a domain waiting for a message remains blocked), and memory isolation

[6] predecessor of Common Criteria for Information Technology Security Evaluation

between guest domains and the hypervisor (domains run at RING 1 privilege level without direct access to RING 0 resources). The hypervisor components such as domain control, scheduling, memory management, hypercalls, and the Chinese-Wall policy are modeled in PROMELA (Process Meta Language), and properties are verified using linear temporal logic formulas. Zhang and Zhao [60] combine the B Method with model checking to verify the consistency between the security policy model and its formal top-level specification. This approach constructs an abstract specification of the Chinese-Wall policy as a B abstract machine, then refines it into a more concrete formal specification that includes detailed models of domain management, virtual CPU scheduling (using round-robin), grant tables, event channels, and MAC. Correspondence analysis and conformance testing are performed using the Event-B tool to ensure semantic consistency between the abstract security policy model and the refined model.

Xenon is a high-assurance separation hypervisor that has been obtained by re-engineering Xen so as to enforce a strict information-flow policy between the domains that it hosts; the policy is expressed in a CSP-style independence condition that guarantees that actions performed by a high-privilege domain cannot influence the observable behavior of a low-privilege domain. McDermott and Freitas [39] introduce a formal security policy written in the Circus language—a combination of Z, CSP and the refinement calculus—and they show that the policy is preserved by refinement, which makes it possible to reason about successive implementation stages of the hypervisor without losing the security guarantee. In a later work, Freitas and McDermott [25] build on this foundation by giving a detailed Circus specification of the Xenon kernel, covering the hypercall interface, the management of virtual interrupts (event channels), the grant-table mechanism for shared memory, and a round-robin scheduler that is modeled as an abstract state transition system. The verified properties include non-interference of the scheduler, the absence of illegal memory accesses through the grant tables, and the correctness of the hypercall operations that mediate access to virtual CPUs, memory pages and event channels, all of which are shown to be deterministic under the lazy-abstraction construction that captures the behavior of an arbitrary high-level attacker. Verification is carried out deductively: the Circus models are type-checked and the associated proof obligations—precondition feasibility, invariant preservation and refinement steps—are discharged with the Z/Eves theorem prover and the CZT tool chain that supports Circus, while the independence definition provides the logical basis for the security proofs.

sHype is a MAC extension of the Xen hypervisor that enforces a Chinese-Wall separation policy among virtual machines. The hypervisor mediates memory accesses through shadow page tables and checks each VM's request against a policy that groups workloads into Conflict-of-Interest (CoI) classes; a VM may access at most one workload of a given class, which guarantees that no VM can observe data from competing classes. Franklin et al. [24] use a custom parametric temporal logic to express the security property stating that if a VM has access to a workload, then all previously accessed workloads belong to the same CW-type or to different CoI classes. The authors instantiate the

model in the Muro model checker and obtain a cut-off of one VM, and prove that the Chinese-Wall policy holds for an arbitrary number of VMs using a small-model theorem. Sinha et al. [52] introduce a complementary methodology, called Small-Short-World (S^2W), which combines inductive invariant checking, localization-abstraction ("small world") and bounded model checking ("short world") to scale verification of hypervisors that manipulate large data structures such as page tables and caches. For sHype they extract from the security property a dependence set containing only the per-VM access bits that must be modeled precisely; the remaining entries of the policy matrix are abstracted to nondeterministic values. The abstract model is then analyzed with the UCLID tool (an SMT-based bounded model checker) after computing a short diameter bound (nine steps in the reported experiments). The approach validates the same Chinese-Wall property as the parametric method but does so without relying on a cut-off theorem, showing that the property can be verified automatically after a few abstraction-guided steps. Both papers demonstrate that formal verification of MAC-enforced hypervisors is practically achievable despite the presence of unbounded data structures, also applying these methodologies to the SecVisor hypervisor.

XMHF. The eXtensible and Modular Hypervisor Framework (XMHF) is a type-1 hypervisor that provides a small trusted core together with a thin library of APIs intended for the construction of security-oriented hypervisor applications (hypapps). Vasudevan et al. [56] concentrate on verifying a memory integrity property, enforced through the proper initialization of hardware page tables, direct memory access (DMA) protection tables, and the preservation of intercept entry points during runtime. The authors introduced the DRIVE methodology, which enumerates six design properties: modularity, atomicity of initialization and intercept handling, memory-access-control protection, correct initialization, proper mediation of guest and device accesses, and safe state updates. Most of the 6,018 lines of core code were subjected to automated verification using the bounded model checker CBMC, which checks inserted assertions that encode the DRIVE conditions; 5,208 lines of C code were proved correct in under two minutes with modest memory consumption, while the remaining stable C and assembly fragments were inspected manually.

Other Verification Efforts. Bolignano [12] conducted a verification study on a type-1, ARMv7 hypervisor developed at TU Berlin, which isolates multiple guest operating systems using shadow page tables. The authors focused on the memory-management subsystem, specifically the `map` and `unmap` primitives, and proved two security properties for each guest: integrity and confidentiality. Using a refinement-by-abstraction methodology, they introduced an abstract model in which the concrete page-table structures are replaced by abstract segments, and proved that every concrete transition is simulated by the corresponding abstract transition via commutation lemmas. All proofs were mechanized in the Prove&Run tool chain (Smart language with interactive theorem prover and SMT solving).

Hao et al. [28] modeled the TBaaS (Trusted Block as a Service) hypervisor[7]—a minimal cloud hypervisor handling only two hypercalls (data-arrival from the management VM and job-done from the Trusted Block) and relying on paged memory management, trap-and-emulate execution, and TPM-based measurement—using their vTRUST framework. They encoded the hypervisor, MMU, device drivers, and TPM in CSP# and subjected the resulting transition system to the PAT model checker. The verification targeted confidentiality, verifiability, strong and weak isolation, and PCR-consistency, expressed as reachability properties. A notable contribution was the fully automated discovery of a boot-loader/hypervisor relocation bug that violates all security properties, demonstrating that model checking can scale to realistic cloud hypervisors.

The Realm Management Monitor is a firmware component of Arm's Confidential Computing Architecture that executes at privilege level EL2 and functions as a separation kernel, mediating between the untrusted host hypervisor and protected execution environments called Realms. The verification effort exposed by Fox et al. [22] addressed both the specification and a prototype C implementation, targeting properties such as the preservation of key invariants governing Realm state consistency and page table structure, memory isolation between distinct Realms, and deadlock freedom for concurrent operations. The methodology combined interactive theorem proving using HOL4 to validate the specification-level properties and derive the isolation guarantees, model checking with CBMC to verify that the implementation correctly refines the specification through automatically generated verification harnesses extracted from the machine-readable specification.

Barthe et al. [6] formalized a hypervisor responsible for memory sharing, TLB management, data cache control, and privilege mode transitions required for context switches. The authors proved a system-level non-interference theorem: for any two initial states differing only in a victim OS's secret data, an attacker controlling the scheduler cannot distinguish executions by observing cache evictions, provided the victim runs constant-time code. The development (approximately 30 k lines of Coq) captures virtual-address translation, TLB, VIP-cache, and hypercall handling, and provides a reusable framework instantiable with concrete replacement and write policies.

Plouviez et al. [46] presented a small, static hypervisor for the many-core TSAR chip, where each VM occupies a contiguous set of clusters and accesses hardware only through hardware "doors" that enforce spatial isolation. The verified software initializes these doors, mediates I/O commands via registration buffers, and performs memory-partition checks before permitting VM interaction with physical storage. Using the B Method, the authors built an abstract specification, refined it to C code, and discharged over two thousand proof obligations with AtelierB. The verification targets VM-to-VM memory isolation, DMA address validation, absence of buffer- and integer-overflows, and guaranteed termination of the main loop. The effort is limited to a static configuration

[7] We believe it is SandVisor[27].

where VMs are fixed at power-up, assumes the underlying hardware doors operate correctly, and leaves a small amount of low-level glue code unverified.

Rusu et al. [49, 48] proposed a formal model of a small full-virtualization monitor for ARM machine code, which alternates between a static analysis phase (scanning and optionally instrumenting instructions) and a dynamic execution phase (running blocks of checked code). The hypervisor is represented as a transition system with modes and components tracking the next instruction to analyze and execute. The verification targeted two functional properties: an invariance property ensuring that no special (potentially unsafe) instructions are executed in processor mode, and a partial-correctness property guaranteeing that the effect of virtualized execution on processor state matches that of unvirtualized execution upon termination. The authors generalized Reachability Logic to arbitrary transition systems, embedded the proof system in Coq, and used abstract symbolic execution with incremental invariant strengthening to construct inductive invariants.

4 Discussion

Techniques and Tools. The most used techniques for hypervisor verification are *deductive verification and model checking,* in some cases using refinement to connect abstract specifications and concrete implementations. Model checking is the most widely used *automatic* technique, e.g. with UCLID [52], Spin [58, 60], CBMC [23, 56, 16, 22], Kani [13], and PAT [28].

Deductive verification, on the other hand, is the workhorse for the majority of *deep, property-preserving proofs.* VCC (used on the Baby Hypervisor [3, 5, 4, 43, 50] and Hyper-V [32, 15]) and Frama-C/WP (used on Anaxagoros [36, 31, 11] and Bao [21]) operate directly on annotated C code, generating proof obligations that are discharged by automatic SMT solvers with occasional interactive proofs. The overall dependency on interactive provers is significant: more than two thirds of the surveyed works eventually resort to Coq [59, 10, 49, 6, 48, 34, 53, 33, 37, 35, 18], Isabelle/HOL [57, 1, 30], HOL4 [41, 7, 26, 40, 8, 22], or PVS[54, 2]. This reflects the *inherent difficulty of reasoning about hypervisor components:* page-table management, TLB handling, and interrupt injection involve intricate layered invariants, ownership predicates, and simulation relations.

Notably, *refinement* (e.g. based on Z or B approaches) is used both as a forward methodology and as a backward validation. In forward usage (e.g., Muen [29], SeKVM [34, 53, 33], pKVM [47]), a high-level abstract specification is progressively refined to C code, yielding a single end-to-end proof chain. In backward usage (e.g., NOVA/BedRock [10]), an abstract model is built from existing code and compared to it, allowing the proof to stay at a higher level while relying on testing or translation toolchains (CompCert, BAP) to establish the code-to-model link.

Maintainability and Reproducibility. A practical concern is that the *longevity of verification tools is sometimes below that of the target hypervisors.* VCC, for instance, is not supported today, while Hyper-V is still actively used. This asymmetry raises questions about the *long-term maintainability of machine-checked*

guarantees. The recently established practice of rigorously preparing and archiving verification artifacts at major formal methods conferences should ensure at least the *reproducibility of proofs* and deserves to be generalized.

Verified Components. Memory management consistently emerges as the most frequently verified subsystem. This is hardly surprising: page-table structures, TLB handling, physical frame allocation, cache coloring, and DMA protection are the primary mechanisms by which a hypervisor enforces spatial isolation between domains. The second most common target is *interrupt and scheduling subsystems,* where correctness requirements demand precise modeling of hardware-assisted virtualization (VMX, SVM, GIC) and of the processor's interrupt-delivery pipeline. *Hypercall interfaces* are another relatively well-covered component. In contrast, components such as device emulation, I/O virtualization, cryptography subsystems, and multi-core synchronisation receive comparatively little formal attention in this context.

Hardware Constraints. A notable deficiency across the surveyed literature is the limited treatment of *realistic hardware constraints and concurrency.* Most works model only a simplified memory hierarchy, and even when a full ISA model is used, components such as the instruction cache, store buffers, cache-coherency protocols, and speculative execution are typically omitted. Notable exceptions include [26], [40]. Multi-core aspects are addressed in some efforts, such as SeKVM's relaxed-memory layer [53] and Hyper-V's theoretical multi-core framework [15]. Yet, the vast majority of the surveyed articles confine verification to a sequential or single-core model, a limitation that is explicitly acknowledged by their authors.

Recent Trends. Since 2021, the community has increasingly turned to *Rust as the implementation language* of choice for new hypervisor projects such as HyperEnclave [18] and Miralis [13]. Another emerging trend is the *combination of deductive verification with large-scale automated analysis.* The pKVM verification effort [47, 14] exemplifies this hybrid methodology: deductive verification in Coq/VST proves deep invariants about shadow page tables and VM-exit handling, while CBMC and KLEE provide complementary coverage on assembly fragments and I/O paths. *Verification of real-life code* attracts increasing attention (e.g. [37, 47, 14, 18, 13]) but still remains a challenge.

5 Threats to Validity

This survey is subject to several threats to validity. The main threat is related to the representativity of the selected papers. The selection of primary studies may be incomplete, as relevant work on hypervisor verification could have been omitted due to overfiltering by the LLMs, limited search queries, publication venues, or accessibility constraints. To mitigate this risk, we attempted to cover major publication databases and perform automatic selection of papers with relevant keywords followed by a careful screening. The rapidly evolving nature

of this research domain also increases the risk of missing recent or less visible contributions. Moreover, some verification projects are published on the web rather than in peer-reviewed papers, e.g. the AWS Nitro System [9].

Second, the comparison of approaches may be affected by subjective interpretation, particularly when assessing the scope of verification, the strength of guarantees, or the level of automation provided by different tools. Differences in reporting practices across papers further complicate direct comparisons. The presentation methodology has significantly evolved and become more structured over years, with a clear statement of contributions along with a companion artifact in recent papers, which was not general practice in earlier papers. We tried to carefully take into account this risk during our study.

Third, the evaluation of verification scope and assumptions relies on the information provided by the original authors, which may vary in terminology, precision or completeness. Some relevant work may use terms such as virtual machine monitor, separation kernel, secure monitor, partitioning kernel, or virtualization layer without mentioning hypervisors in the searchable metadata. We declare that our selection is based on explicit statements made by the authors in their papers. The manual validation step mitigates misclassification inside the retrieved set, but it cannot guarantee that all relevant papers outside the search results were found.

Finally, the generalizability of the findings can be limited. Indeed, the selected works present results for specific hypervisors, hardware models, or threat assumptions, which may not extend to other systems or deployment contexts. This limitation is reported explicitly: the purpose of our study is to provide a structured view of the field, without claiming exhaustiveness.

6 Conclusion

This paper presents a systematic mapping study on formal verification of hypervisors. It surveys a selection of 56 papers from archival repositories and emphasizes their verification scope, used techniques and tools. Key challenges and future work directions include scalable concurrency verification, comprehensive hardware modeling, and end-to-end proofs of real-life hypervisor code bases. Aligning formal verification efforts of hypervisors with certification standards, such as Common Criteria or DO-178C, will facilitate their adoption in safety-critical domains.

Acknowledgment. Part of this work was supported by ANR (grants ANR-22-CE39-0014, ANR-22-CE25-0018, ANR-24-ASM2-0001). We thank the anonymous referees for helpful comments.

Acronyms. Here is the list of acronyms and the page on which they are defined.

References

1. Ahmadi, S., Dongol, B., Griffin, M.: Proving Memory Access Violations in Isabelle/HOL. In: Proceedings of the 8th ACM SIGPLAN International Workshop on Formal Techniques for Safety-Critical Systems. pp. 45–55. ACM (2022). https://doi.org/10.1145/3563822.3568010
2. Alexander, P., Pike, L., Loscocco, P., Coker, G.: Model Checking Distributed Mandatory Access Control Policies. ACM Transactions on Information and System Security **18**(2), 1–25 (2015). https://doi.org/10.1145/2785966
3. Alkassar, E., Cohen, E., Hillebrand, M.A., Kovalev, M., Paul, W.J.: Verifying shadow page table algorithms. In: Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2010). pp. 267–270. IEEE (2010), https://ieeexplore.ieee.org/document/5770958/
4. Alkassar, E., Cohen, E., Kovalev, M., Paul, W.J.: Verification of TLB Virtualization Implemented in C. In: Verified Software: Theories, Tools, Experiments, vol. 7152, pp. 209–224. Springer (2012). https://doi.org/10.1007/978-3-642-27705-4_17
5. Alkassar, E., Hillebrand, M.A., Paul, W., Petrova, E.: Automated Verification of a Small Hypervisor. In: Verified Software: Theories, Tools, Experiments, vol. 6217, pp. 40–54. Springer (2010). https://doi.org/10.1007/978-3-642-15057-9_3
6. Barthe, G., Betarte, G., Campo, J.D., Luna, C.: System-Level Non-interference of Constant-Time Cryptography. Part I: Model. Journal of Automated Reasoning **63**(1), 1–51 (2019). https://doi.org/10.1007/s10817-017-9441-5
7. Baumann, C., Naslund, M., Gehrmann, C., Schwarz, O., Thorsen, H.: A high assurance virtualization platform for ARMv8. In: 2016 European Conference on Networks and Communications (EuCNC). pp. 210–214. IEEE (2016). https://doi.org/10.1109/EuCNC.2016.7561034
8. Baumann, C., Schwarz, O., Dam, M.: On the verification of system-level information flow properties for virtualized execution platforms. Journal of Cryptographic Engineering **9**(3), 243–261 (2019). https://doi.org/10.1007/s13389-019-00216-4
9. Bean, J., Chong, N., Mulligan, D., Saidi, A.: Nitro isolation engine whitepaper. Tech. rep., Amazon Web Services (2026), https://d1.awsstatic.com/onedam/marketing-channels/website/aws/en_US/whitepapers/compliance/nitro-isolation-engine-whitepaper.pdf
10. Becker, H., Crespo, J.M., Galowicz, J., Hensel, U., Hirai, Y., Kunz, C., Nakata, K., Sacchini, J.L., Tews, H., Tuerk, T.: Combining Mechanized Proofs and Model-Based Testing in the Formal Analysis of a Hypervisor. In: FM 2016: Formal Methods, vol. 9995, pp. 69–84. Springer (2016). https://doi.org/10.1007/978-3-319-48989-6_5
11. Blanchard, A., Kosmatov, N., Lemerre, M., Loulergue, F.: A Case Study on Formal Verification of the Anaxagoros Hypervisor Paging System with Frama-C. In: Formal Methods for Industrial Critical Systems, vol. 9128, pp. 15–30. Springer (2015). https://doi.org/10.1007/978-3-319-19458-5_2
12. Bolignano, P.: Formal Models and Verification of Memory Management in a Hypervisor. Ph.D. thesis, Université de Rennes ; Prove & Run (2017), https://theses.hal.science/tel-01637937
13. Castes, C., Costa, F., Foster, N., Bourgeat, T., Bugnion, E.: Lightweight Hypervisor Verification: Putting the Hardware Burger on a Diet. In: Proceedings of the Workshop on Hot Topics in Operating Systems. pp. 27–33. ACM (2025). https://doi.org/10.1145/3713082.3730373

14. Cebeci, C., Zou, Y., Zhou, D., Candea, G., Pit-Claudel, C.: Practical Verification of System-Software Components Written in Standard C. In: Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles. pp. 455–472. ACM (2024). https://doi.org/10.1145/3694715.3695980
15. Cohen, E., Paul, W., Schmaltz, S.: Theory of Multi Core Hypervisor Verification. In: SOFSEM 2013: Theory and Practice of Computer Science, vol. 7741, pp. 1–27. Springer (2013). https://doi.org/10.1007/978-3-642-35843-2_1
16. Cook, B., Döbel, B., Kroening, D., Manthey, N., Pohlack, M., Polgreen, E., Tautschnig, M., Wieczorkiewicz, P.: Using model checking tools to triage the severity of security bugs in the Xen hypervisor. In: 2020 Formal Methods in Computer Aided Design (FMCAD). pp. 185–193. TU Wien Academic Press (2020). https://doi.org/10.34727/2020/ISBN.978-3-85448-042-6_26
17. Dahlin, M., Johnson, R., Krug, R.B., McCoyd, M., Young, W.: Toward the Verification of a Simple Hypervisor. Electronic Proceedings in Theoretical Computer Science **70**, 28–45 (2011). https://doi.org/10.4204/EPTCS.70.3
18. Dai, Z., Liu, S., Sjoberg, V., Li, X., Chen, Y., Wang, W., Jia, Y., Anderson, S.N., Elbeheiry, L., Sondhi, S., Zhang, Y., Ni, Z., Yan, S., Gu, R., He, Z.: Verifying Rust Implementation of Page Tables in a Software Enclave Hypervisor. In: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. pp. 1218–1232. ACM (2024). https://doi.org/10.1145/3620665.3640398
19. Dam, M., Guanciale, R., Khakpour, N., Nemati, H., Schwarz, O.: Formal verification of information flow security for a simple arm-based separation kernel. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security - CCS '13. pp. 223–234. ACM Press (2013). https://doi.org/10.1145/2508859.2516702
20. Dam, M., Guanciale, R., Nemati, H.: Machine code verification of a tiny ARM hypervisor. In: Proceedings of the 3rd International Workshop on Trustworthy Embedded Devices. pp. 3–12. ACM (2013). https://doi.org/10.1145/2517300.2517302
21. Ferréol, A., Corbin, L., Kosmatov, N.: Prove your Colorings: Formal Verification of Cache Coloring of Bao Hypervisor. In: Fundamental Approaches to Software Engineering, vol. 15693, pp. 214–235. Springer (2025). https://doi.org/10.1007/978-3-031-90900-9_11
22. Fox, A.C.J., Stockwell, G., Xiong, S., Becker, H., Mulligan, D.P., Petri, G., Chong, N.: A Verification Methodology for the Arm® Confidential Computing Architecture: From a Secure Specification to Safe Implementations. Proceedings of the ACM on Programming Languages **7**(OOPSLA1), 376–405 (2023). https://doi.org/10.1145/3586040
23. Franklin, J., Chaki, S., Datta, A., McCune, J.M., Vasudevan, A.: Parametric Verification of Address Space Separation. In: Principles of Security and Trust, vol. 7215, pp. 51–68. Springer (2012). https://doi.org/10.1007/978-3-642-28641-4_4
24. Franklin, J., Chaki, S., Datta, A., Seshadri, A.: Scalable Parametric Verification of Secure Systems: How to Verify Reference Monitors without Worrying about Data Structure Size. In: 2010 IEEE Symposium on Security and Privacy. pp. 365–379. IEEE (2010). https://doi.org/10.1109/SP.2010.29
25. Freitas, L., McDermott, J.: Formal methods for security in the Xenon hypervisor. International Journal on Software Tools for Technology Transfer **13**(5), 463–489 (2011). https://doi.org/10.1007/s10009-011-0195-9

26. Guanciale, R., Nemati, H., Baumann, C., Dam, M.: Cache Storage Channels: Alias-Driven Attacks and Verified Countermeasures. In: 2016 IEEE Symposium on Security and Privacy (SP). pp. 38–55. IEEE (2016). https://doi.org/10.1109/SP.2016.11
27. Hao, J., Cai, W.: Trusted Block as a Service: Towards Sensitive Applications on the Cloud (2011). https://doi.org/10.1109/TrustCom.2011.13
28. Hao, J., Liu, Y., Cai, W., Bai, G., Sun, J.: vTRUST: A Formal Modeling and Verification Framework for Virtualization Systems. In: Formal Methods and Software Engineering, vol. 8144, pp. 329–346. Springer (2013). https://doi.org/10.1007/978-3-642-41202-8_22
29. Haque, I., D'Souza, D., Habeeb, P., Kundu, A., Babu, G.: Verification of a Generative Separation Kernel. In: Automated Technology for Verification and Analysis, vol. 12302, pp. 305–322. Springer (2020). https://doi.org/10.1007/978-3-030-59152-6_17
30. Huang, Y., Luo, J., Chen, L., Xiao, K., Luo, L., Li, Y.: Formal Modeling and Priority Handling Verification of Interrupt Injection in a Hypervisor. In: Cyberspace Simulation and Evaluation, vol. 2422, pp. 85–99. Springer (2025). https://doi.org/10.1007/978-981-96-4509-1_6
31. Kosmatov, N., Lemerre, M., Alec, C.: A Case Study on Verification of a Cloud Hypervisor by Proof and Structural Testing. In: Tests and Proofs, vol. 8570, pp. 158–164. Springer (2014). https://doi.org/10.1007/978-3-319-09099-3_12
32. Leinenbach, D., Santen, T.: Verifying the Microsoft Hyper-V Hypervisor with VCC. In: FM 2009: Formal Methods, vol. 5850, pp. 806–809. Springer (2009). https://doi.org/10.1007/978-3-642-05089-3_51
33. Li, S.W., Li, X., Gu, R., Nieh, J., Hui, J.Z.: Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor (2021)
34. Li, S.W., Li, X., Gu, R., Nieh, J., Zhuang Hui, J.: A Secure and Formally Verified Linux KVM Hypervisor. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 1782–1799. IEEE (2021). https://doi.org/10.1109/SP40001.2021.00049
35. Liu, Z., Stepanenko, S., Pichon-Pharabod, J., Timany, A., Askarov, A., Birkedal, L.: VMSL: A Separation Logic for Mechanised Robust Safety of Virtual Machines Communicating above FF-A. Proceedings of the ACM on Programming Languages **7**(PLDI), 1438–1462 (2023). https://doi.org/10.1145/3591279
36. Loulergue, F., Gava, F., Kosmatov, N., Lemerre, M.: Towards verified cloud computing environments. In: 2012 International Conference on High Performance Computing & Simulation (HPCS). pp. 91–97. IEEE (2012). https://doi.org/10.1109/HPCSim.2012.6266896
37. Malecha, G., Stewart, G., Farka, F., Haag, J., Hirai, Y.: Developing With Formal Methods at BedRock Systems, Inc. IEEE Security & Privacy **20**(3), 33–42 (2022). https://doi.org/10.1109/MSEC.2022.3158196
38. McCoyd, M., Krug, R.B., Goel, D., Dahlin, M., Young, W.: Building a Hypervisor on a Formally Verifiable Protection Layer. In: 2013 46th Hawaii International Conference on System Sciences. pp. 5069–5078. IEEE (2013). https://doi.org/10.1109/HICSS.2013.121
39. McDermott, J., Freitas, L.: A formal security policy for xenon. In: Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering. pp. 43–52. ACM (2008). https://doi.org/10.1145/1456396.1456401
40. Nemati, H., Baumann, C., Guanciale, R., Dam, M.: Formal Verification of Integrity-Preserving Countermeasures Against Cache Storage Side-Channels. In: Principles of Security and Trust, vol. 10804, pp. 109–133. Springer (2018). https://doi.org/10.1007/978-3-319-89722-6_5

41. Nemati, H., Dam, M., Guanciale, R., Do, V., Vahidi, A.: Trustworthy Memory Isolation of Linux on Embedded Devices. In: Trust and Trustworthy Computing, vol. 9229, pp. 125–142. Springer (2015). https://doi.org/10.1007/978-3-319-22846-4_8
42. Nordholz, J.: Design of a symbolically executable embedded hypervisor. In: Proceedings of the Fifteenth European Conference on Computer Systems. pp. 1–16. ACM (2020). https://doi.org/10.1145/3342195.3387516
43. Paul, W., Schmaltz, S., Shadrin, A.: Completing the Automated Verification of a Small Hypervisor – Assembler Code Verification. In: Software Engineering and Formal Methods, vol. 7504, pp. 188–202. Springer (2012). https://doi.org/10.1007/978-3-642-33826-7_13
44. Petersen, K., Feldt, R., Mujtaba, S., Mattsson, M.: Systematic Mapping Studies in Software Engineering. In: 12th International Conference on Evaluation and Assessment in Software Engineering (EASE) (2008). https://doi.org/10.14236/ewic/EASE2008.8
45. Petersen, K., Vakkalanka, S., Kuzniarz, L.: Guidelines for conducting systematic mapping studies in software engineering: An update. Information and Software Technology **64**, 1–18 (2015). https://doi.org/10.1016/j.infsof.2015.03.007
46. Plouviez, G., Encrenaz, E., Wajsbürt, F.: A Formally Verified Static Hypervisor with Hardware Support for a Many-Core Chip. In: Euro-Par 2013: Parallel Processing Workshops, vol. 8374, pp. 801–811. Springer (2014). https://doi.org/10.1007/978-3-642-54420-0_78
47. Pulte, C., Makwana, D.C., Sewell, T., Memarian, K., Sewell, P., Krishnaswami, N.: CN: Verifying Systems C Code with Separation-Logic Refinement Types. Proceedings of the ACM on Programming Languages **7**(POPL), 1–32 (2023). https://doi.org/10.1145/3571194
48. Rusu, V., Grimaud, G., Hauspie, M.: Proving partial-correctness and invariance properties of transition-system models. Science of Computer Programming **186** (2020). https://doi.org/10.1016/j.scico.2019.102342
49. Rusu, V., Grimaud, G., Hauspie, M., Serman, F.: Deductive Verification of a Hypervisor Model (2017), https://inria.hal.science/hal-01614509v1
50. Schmaltz, S., Shadrin, A.: Integrated Semantics of Intermediate-Language C and Macro-Assembler for Pervasive Formal Verification of Operating Systems and Hypervisors from VerisoftXT. In: Verified Software: Theories, Tools, Experiments, vol. 7152, pp. 18–33. Springer (2012). https://doi.org/10.1007/978-3-642-27705-4_3
51. She, Y., Li, H., Zhu, H.: UVHM: Model checking based formal analysis scheme for hypervisors. In: Information and Communication Technology, Lecture Notes in Computer Science, vol. 7804, pp. 300–305. Springer (2013). https://doi.org/10.1007/978-3-642-36818-9_31
52. Sinha, R., Sturton, C., Maniatis, P., Seshia, S.A., Wagner, D.: Verification with Small and Short Worlds. In: 2012 Formal Methods in Computer-Aided Design (FMCAD). pp. 68–77 (2012), https://ieeexplore.ieee.org/abstract/document/6462557
53. Tao, R., Yao, J., Li, X., Li, S.W., Nieh, J., Gu, R.: Formal Verification of a Multiprocessor Hypervisor on Arm Relaxed Memory Hardware. In: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. pp. 866–881. ACM (2021). https://doi.org/10.1145/3477132.3483560
54. Tews, H., Völp, M., Weber, T.: Formal Memory Models for the Verification of Low-Level Operating-System Code. Journal of Automated Reasoning **42**(2-4), 189–227 (2009). https://doi.org/10.1007/s10817-009-9122-0

55. VanderLeest, S.H., Greve, D., Skentzos, P.: A safe & secure ARINC 653 hypervisor. In: 2013 IEEE/AIAA 32nd Digital Avionics Systems Conference (DASC). pp. 7B4–1–7B4–17. IEEE (2013). https://doi.org/10.1109/DASC.2013.6712638
56. Vasudevan, A., Chaki, S., Limin Jia, McCune, J., Newsome, J., Datta, A.: Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework. In: 2013 IEEE Symposium on Security and Privacy. pp. 430–444. IEEE (2013). https://doi.org/10.1109/SP.2013.36
57. Verbeek, F., Bockenek, J.A., Ravindran, B.: Highly Automated Formal Proofs over Memory Usage of Assembly Code. In: Tools and Algorithms for the Construction and Analysis of Systems, vol. 12079, pp. 98–117. Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_6
58. Wang, S., Liu, J., Yi, Q., Zhang, X.: Model checking a secure hypervisor. In: 2010 Second World Congress on Software Engineering. pp. 119–122. IEEE (2010). https://doi.org/10.1109/WCSE.2010.115
59. Zhang, H., Guo, Y.: Formal Verification of Interrupt Injection in a Hypervisor. In: 2014 Theoretical Aspects of Software Engineering Conference. pp. 74–81. IEEE (2014). https://doi.org/10.1109/TASE.2014.26
60. Zhang, X., Zhao, X.: Formal analysis and verification on a secure microkernel. In: 2011 International Conference on Mechatronic Science, Electric Engineering and Computer (MEC). pp. 2064–2068. IEEE (2011). https://doi.org/10.1109/MEC.2011.6025897