Abstract architecture catches concrete bugs: lightweight checking of page table synchronisation

Ben Simner
University of Cambridge
UK
Ben.Simner@cl.cam.ac.uk

Thomas Fourier
University of Cambridge
UK; École polytechnique
France
thomas.fourier@polytechnique.edu

Yeji Han
Seoul National University
Korea
yeji.han@sf.snu.ac.kr

David Kaloper Meršinjak
University of Cambridge
UK
dk505@cl.cam.ac.uk

Thibaut Pérami
University of Cambridge
UK
thibaut.perami@cl.cam.ac.uk

Peter Sewell
University of Cambridge
UK
Peter.Sewell@cl.cam.ac.uk

Jean Pichon-Pharabod
Aarhus University
Denmark
Jean.Pichon@cs.au.dk

Abstract

Computer security relies on systems code to correctly manage the underlying hardware, in particular the virtual memory mechanisms used to enforce controlled sharing between processes and virtual machines. This management requires delicate synchronisation around page table accesses, to handle the subtle relaxed concurrency phenomena of virtual memory. These are not supported by conventional testing, semantics, or verification methods, so systems programmers have nothing but code review by domain experts to ensure correctness of these software components.

We present a lightweight technique to find bugs caused by insufficient virtual-memory synchronisation. Our approach is to explicitly capture a discipline of programming implicitly followed by production systems code. We do so in the form of Casemate, a novel, semantics-driven abstraction of the underlying relaxed hardware architecture semantics, focusing on Arm-A. As such, Casemate gives a clear account of the subtle but ad-hoc and informal discipline relied on by the code used in production. Casemate works by tracking the effect of accesses to page tables and relevant instructions, checking that they follow the expected discipline as they happen. We implement Casemate as a library that works both as a monitor integrated in systems code, and as a standalone offline checker over logs. We give a pen-and-paper proof of refinement, showing that, once made precise, the discipline that systems code tries to follow and that Casemate enforces is sound.

We apply Casemate to two production hypervisors: Google's pKVM for Android and FreeBSD's bhyve. Casemate detected one security-critical bug in each before release, as well as a variety of synthetic page table synchronisation bugs. We also found several other synchronisation bugs along the way.

We argue that delicate code like this, whether de novo code like pKVM, or ported from other architectures like bhyve, needs to be checked precisely, and that the design of such a precise tool can only emerge through semantics-driven design from architectural models.

CCS Concepts: • **Software and its engineering → Empirical software validation**; • **Theory of computation** → *Operational semantics*; • **Security and privacy → Virtualization and security**.

Keywords: Software Testing, Operating Systems, Programming Languages, Semantics, Computer Architecture, Virtualization

ACM Reference Format:
Ben Simner, Thomas Fourier, Yeji Han, David Kaloper Meršinjak, Thibaut Pérami, Peter Sewell, and Jean Pichon-Pharabod. 2027. Abstract architecture catches concrete bugs: lightweight checking of page table synchronisation. In *22nd European Conference on Computer Systems (EuroSys '27), April 19–23, 2027, Rabat, Morocco.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3842654.3848565

EuroSys '27, Rabat, Morocco

ACM ISBN 979-8-4007-2971-3/2027/04
https://doi.org/10.1145/3842654.3848565

1 Introduction

Computer security relies on the protections provided by systems code, especially the *controlled sharing* provided by operating systems and hypervisors. These manage the architectural virtual memory support of the underlying hardware,

to map the virtual addresses used by different processes or virtual machines to disjoint-by-default regions of physical memory, thereby bounding the potential effects of buggy or malicious code. Ensuring that they do so correctly is thus crucial, but also very challenging, in several ways.

First, it is intrinsically a racy concurrency problem. Address translation, from virtual to physical addresses, is defined by page tables in memory. Each programmer-visible memory access, to some virtual address, requires the hardware to read the currently enabled page tables in memory or in cache to compute the corresponding physical address. Systems code manipulates the page tables for its own address space which contains its own code and data, and for the address spaces of virtual machines and user processes. It would be infeasible to stop the world while it does so, and so these page table writes race with the hardware translation table walks which access the same in-memory data structure.

Second, it is intrinsically a problem of controlled divergence between the logical view of the software and the physical state of the hardware. Excessive divergence leads to memory pages being transiently visible to the wrong process, violating the core security guarantee of the hypervisor or OS. However, the aggressive pipelining, out-of-order and speculative execution, and extensive caching and buffering of memory of modern hardware make divergence prohibitively expensive to avoid completely. Controlled divergence is made not only complex but also hazardous by the specific relaxed behaviour of page tables: for normal memory accesses, caches are coherent, but for translation-table-walk accesses, the caches, called TLBs (Translation Lookaside Buffers), must be selectively cleaned by the programmer. Indeed, translation-table-walk reads are far more common than writes to those tables, so TLBs are typically not coherent by default: writing to a page table does not guarantee that subsequent memory accesses use the new address mapping. Instead, the systems programmer must use explicit TLB invalidation instructions (TLBIs), and various barriers, to ensure proper removal of old cached entries. As a concrete hazard, on Arm-A, this must be done following a discipline that Arm calls Break-Before-Make (BBM, see §2.1.2). Failure to follow BBM invalidates the very isolation that the hypervisor or OS is trying to enforce. Accordingly, as part of our broader enforcement of controlled divergence, we need to ensure that BBM is heeded.

Third, precisely defining the architecturally-allowed relaxed behaviour has, until recently, been beyond the reach of mathematical models, and not fully understood even in non-mathematical ways. Extensive research has put 'user' relaxed concurrency on a solid footing, e.g. [1–3, 5–7, 11, 18–20, 23, 24, 26, 29, 41–45, 47, 53], but did not touch on 'systems' relaxed concurrency. Recent work by Simner et al. [49, 52] developed semantics for Arm-A instruction fetch and exceptions, and Simner et al. [50] and Alglave et al. [4] developed semantics for virtual memory. These models have been developed and validated by a combination of experimental testing and discussions with the Arm chief architect. While doubtless still subject to change, they finally provide a reasonably solid foundation for these aspects of systems code – but they remain very complex, and there is no testing or verification theory above them beyond tooling for litmus tests.

All this makes it hard to gain assurance in the correctness of virtual memory management using conventional testing, conventional static analysis, or conventional verification. Conventional runtime testing will typically not reveal incorrect synchronisation bugs, as the relevant architecturally allowed relaxed hardware behaviour may be exhibited only rarely, or only on some hardware implementations, and testing that memory is not transiently incorrectly exposed requires accessing that memory in exactly the right time window. Conventional static analysis and verification techniques almost all assume sequential or sequentially consistent (SC) interleaving semantics, e.g. in the seL4 proof [46], but the Arm-A architecture (much like RISC-V and IBM Power) is very much not SC, even for 'user' concurrency, and the relaxed behaviours, including those described above, are especially problematic for reasoning.

We leverage the recent semantic understanding of relaxed virtual-memory management concurrency to develop a sound abstraction over the architecture. We use this abstraction to create a tool, *Casemate*, which tests whether software remains synchronised with the physical state when changing memory mappings. This is complementary to work on the higher-level question of the functional correctness of that software [38]. Our approach applies *in a lightweight way* during development of *production systems code*: requiring minimally invasive instrumentation, with no change to the logic of the original program. We demonstrate its efficacy by applying it to two hypervisors: (1) Google's pKVM hypervisor – developed as part of the Android kernel, deployed since Android 13, and since upstreamed to Linux [16, 25, 32, 40] – which enforces isolation between the Android Linux kernel and guest virtual machines, protecting the latter from post-initialisation kernel compromises, and vice versa; and (2), FreeBSD's hypervisor, bhyve, in particular the recent development of its Arm64 port. We do this for Android 14 and 15, and FreeBSD 15, and discover subtle-but-potentially-serious synchronisation errors in both.

Contributions:

- We design Casemate (§3), a tool for Concurrent Architectural System-level Monitoring and Testing, which monitors page tables to detect synchronisation errors in the code and warn when software and hardware views problematically diverge. Casemate defines a safe programming discipline for page table manipulation, including ownership and TLB maintenance, and checks that the program under test heeds it. Concretely, Casemate maintains a shadow copy of the page table

memory, tracking each individual page table entry with automata. To account for the relaxed behaviour described above, and for the hierarchical nature of page tables, these automata interact in subtle ways with those of their parents and children in the page tables, and depend on each entry's reachability history.
- We implement Casemate (§4) as a practical runtime monitor, in C as a standalone library which can be linked to the system. This gives sufficient performance to run, for example, pKVM in its usual QEMU development and test environment, without disrupting its execution. Since Casemate checks the intensional discipline, not just end-to-end memory-access violations, it can find bugs without massive test runs.
- We apply Casemate (§5) to two production hypervisors on Arm: Android's pKVM and FreeBSD's bhyve. In doing so, we uncovered security-critical TLB synchronisation bugs in both, which transiently exposed memory to the wrong virtual machine. Development of Casemate found other synchronisation bugs.
- We connect Casemate to the ground truth (§6) given by the virtual memory systems architecture model of Simner et al. [50] through a pen-and-paper proof of refinement.

Hypervisors and other systems code rely on programming models that embody multiple abstractions above the underlying hardware architecture, providing fictions of virtual memory (with mapping on demand, and disjoint-by-default address spaces with controlled sharing), Harvard-architecture machines (with linkable programs rather than arbitrarily self-modifying code), and high-level languages and their concurrency models. Ensuring that they do so correctly, above complex modern architectures, is a major open problem — and work of the kind we report on here is a necessary step along that path. This work also serves as an example of a lightweight formal approach to systems code assurance, using formal models for pragmatic and practical testing rather than all-or-nothing verification.

2 Background

2.1 Virtual memory

The Arm-A virtual memory architecture, defined in prose in the Arm architecture reference manual [10, Ch. D8], is complex. We summarise the key aspects for a common configuration, to understand how hypervisors enforce isolation.

Address translation. Virtual memory indirects the program's accesses through a memory management unit (MMU), which both translates *virtual* addresses (VA) used by the program into a hardware *physical* address (PA), and checks for permissions. These mappings are determined by in-memory data structures consisting of trees of individual page tables (as in Fig. 1). We sometimes refer to a tree collectively by its page tables. On Arm-A, particular system registers, Translation Table Base Registers (TTBRs), hold the physical addresses of the roots of the current mappings. Each table is an array of *page table entries* (PTEs), each of which encodes the mapping for a contiguous region of the domain. Each entry is either an *invalid* descriptor, which means that the translation function is undefined for that input region (those virtual addresses are 'not mapped'), or a *valid* descriptor, which either encodes the output physical address and permissions for that region, or points to another table that defines the translation for the fragments of that region. In a typical configuration, for example the one that Linux uses, the tree is of height at most four, with 4K pages and 48-bit addresses, but this is configurable.

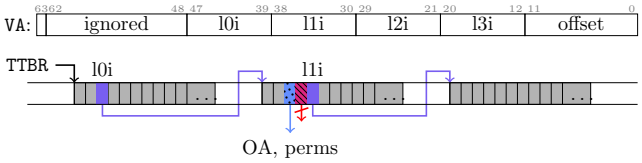

Figure 1. A sketch of a translation table walk, with an input virtual address and page tables (in grey). The input address is split into slices (`l0i`–`l3i`) which are used as indices into the nested tree of tables, plus an offset into the page. We show one entry of each kind: an *invalid* entry in hatched magenta, a valid leaf in dotted blue with output address (OA) and permissions, and *table* entries in solid purple.

For each data or instruction-fetch memory access by the program, the memory management unit (MMU) translates the virtual address which the program requested to access, through a combination of translation table walks and reading caches. This gives either: a physical address, which can be used to access that location of physical memory; or a *translation fault*, if the virtual address was unmapped; or a *permission fault*, if the translation did not give permission to do the requested access. Such faults are reported back to the processor and are taken as processor exceptions.

Process isolation. In this paper, we focus on the use of virtual memory to implement isolation with controlled sharing. Operating systems typically maintain page tables for each process. When context switching from one process to another, the kernel switches to that process's page table tree, by writing its root address into the appropriate TTBR register. This gives each process its own translation function. Processes can then be controlled by ensuring that their physical-memory ranges are disjoint except where otherwise desired.

Virtualisation. Hypervisors like pKVM and bhyve follow a similar scheme: they maintain page tables for each virtual machine. By ensuring VMs' physical memory ranges are disjoint except for controlled sharing, isolation between

VMs is guaranteed. To allow each VM to perform process isolation, Arm-A enables multiple *stages* of translation: the address of a process running in a VM is first translated into the physical address in the address space of that VM, called an *intermediate-physical* address (IPA). The IPA is then translated into a *physical* address (PA). The guest's OS manages the first stage of translation and the hypervisor manages the second. This composition of translations allows hypervisors to enforce controlled isolation between whole virtual machines and switch between them, while letting each virtual machine manage isolation among its processes.

2.1.1 TLBs. Hardware translation table walks, especially with multi-stage translation, quickly become expensive. On Arm, translation tables are typically up to depth four, and each of the four accesses of the first stage of translation accesses an IPA which must itself be translated, and the IPA resulting from this first stage must itself also be translated. In total, this gives up to twenty-four separate memory accesses per translation table walk, for every instruction fetch, read, or write, that the program performs.

To alleviate pressure on the hardware walker and memory subsystem, processors aggressively cache translation table walks in TLBs. These can cache both complete end-to-end translations, from virtual to physical, and partial translation table walks. They are indexed by virtual machine identifiers (VMIDs) and by user address-space identifiers (ASIDs), so that context switching does not require the invalidation of the whole TLB. On Arm, TLBs are only allowed to cache translations of addresses which are mapped. Translations of unmapped addresses (which result in a translation fault) thus must result from reading an invalid descriptor directly from memory. Out-of-order and speculative execution means that the architecture must allow TLB caching from any legal walk at any time, e.g. for speculative instructions and prefetchers, not just architecturally accessed virtual addresses.

2.1.2 Break-before-make. TLBs may cache multiple translations for the same input address, for example translations leading to the same output address but with different permissions. However, if a TLB contains two *conflicting* translations for the same input address (and the same VMID or ASID) according to the architectural definition, the behaviour can be unpredictable. In particular, Arm does not tightly constrain the behaviour of the machine in such a situation, instead saying the behaviour is 'ConstrainedUnpredictable'. Software must avoid this, and one therefore needs to ensure that whenever there is a write to a potentially reachable page table entry, it and its children (the entries in the subtree below it) are not in conflicting states, even transiently. The exact details of what needs to be done to ensure this are architecture-specific, and are significantly different for Armv8+, on which we focus, than for x86 or even previous versions of Arm. Concretely, it requires that page table updates follow *break-before-make*: when updating the translation, one must first *break* the mapping: write an invalid descriptor to a page table entry, and then ensure that no TLB caches an outdated translation based on the old valid descriptor, by using the provided TLBI ('TLB invalidate') maintenance instructions with enough barriers to order them. All TLBs need to be cleared of the old valid descriptor for that page table entry *before* one can *make* the new mapping: write the new valid descriptor to the page table entry.

Fine-grained TLBIs. Arm provides a family of TLBIs with different scopes along different axes, and accordingly different costs: by virtual address, by virtual machine identifier, by intermediate physical address, invalidate one or all, broadcast and non-broadcast, etc. For performance, invalidation typically involves a combination of several fine-grained TLBIs that match the specific scenario: there is no simple canned sequence of instructions used in all scenarios. For example, to invalidate compound VA-to-PA mappings, invalidating by VA (as opposed to by a TLBI all) requires, ordered between the start of the invalidation via the write of an invalid descriptor and the end of the invalidation via the write of a valid descriptor, a TLBI by IPA ordered before a TLBI by VA. Properly ordered, the TLBI by IPA ensures that the TLBs do not cache IPA-to-PA mappings from which they could reconstruct a VA-to-PA mapping; and the TLBI by VA then removes the VA-to-IPA and compound VA-to-PA mappings. These TLBIs are, by themselves, neither ordered with memory accesses, nor ordered with each other: order needs to be imposed. This typically involves inserting, in between any two elements of this sequence, an expensive Data Synchronisation Barrier (DSB) – and potentially even more events if the sequence is split across multiple cores.

This top-down explanation of how to use TLBIs is deceptively simple: in practice, systems code batches these expensive DSBs and TLBIs, which hides the specific context of each individual invalidation. This makes selecting the appropriate flavours of TLBIs challenging, and makes maintaining code which uses TLBIs significantly error-prone, even for experts.

Casemate supports a range of broadcast and thread-local TLBIs. There is no fundamental restriction on which TLBIs we support, and over time we have added support for a wider variety. For currently unsupported variants, the user can choose an over-approximation or an under-approximation.

2.1.3 Complexity of TLB maintenance. TLB maintenance requires subtle synchronisation around page table updates, for several reasons: Page tables are hierarchical, and invalidating an entry cannot be done without considering the state of its children. TLBs only store entries that were reachable by the MMU, so software needs to keep track of what has been reachable, to know what it can safely overwrite. Page tables are a priori global state, but some hypervisors, for example pKVM, treat some parts thread-locally, and this ownership needs to be tracked. On top of that, there are standard relaxed memory concerns, in particular that accesses

to page tables can be reordered with each other, which are compounded by virtual memory: writes to page tables race with the MMU reads, which cannot be stopped.

Even ignoring violations of break-before-make, the relaxed behaviour of Arm's virtual-memory systems architecture is complex: the translation table walks for different instructions may happen out-of-order, not necessarily adjacent to the associated explicit access; and threads may have multiple TLBs per MMU, caching distinct values for given addresses, enabling software to 'see' old translations even program-order-after instructions that saw more recent ones – TLBs are not *coherent*.

Translation table walks can happen even for speculative instructions, except that such speculative walks cannot read-from writes which are also speculative. TLB caching is indexed by process *address space identifiers* (ASIDs) and *virtual-machine identifiers* (VMIDs), so that context switching need only change the ASID or VMID rather than flush all the TLBs. Each of these concerns, and others, adds significant complexity to any *architectural* model of relaxed virtual memory [4, 50], which has to describe the architectural guarantees for arbitrary software.

For example, having no synchronisation between sequential writes to the same tree of translation tables does not, as long as the entries were initially invalid, require break-before-make: this is architecturally permitted, but would allow a translation table walk to see any combination of the possible trees induced by those writes [51, §A.7.2.1], a situation software would likely want to avoid. Those translation table walks may be performed out-of-order with respect to others in the same thread, potentially with other events interposing between the translation table walk of one instruction and its respective explicit access – although proper TLB maintenance ensures that this microarchitectural behaviour is not visible to the programmer [51, §A.4.1.6].

Simner et al. [50] and Alglave et al. [4] give formal semantics to Arm-A's relaxed virtual memory, covering different, but overlapping, aspects of the architecture: the first covers multiple stages and levels of translation, whereas the second covers hardware updates to access flags and dirty bits. Both are typical 'axiomatic' memory models. Casemate is agnostic to any particular choice of underlying formal model — as it should be a sound abstraction over any correct architectural definition — although we will use the former for proof, as it is the most complete w.r.t. multi-stage translation.

2.2 Hypervisors: a practical target

Casemate is primarily targeted at modern hypervisor usage. Before we explore the design of Casemate itself (§3) or evaluate it on real systems (§5), in this subsection, we first briefly explore the design of a modern hypervisor, pKVM, as a motivating example, and to contextualise the design decisions and scope.

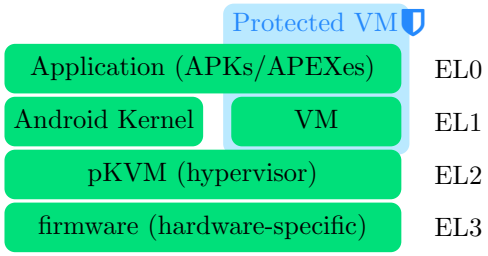

Figure 2. Architecture of pKVM [25], typical of a Type-1 hypervisor.

Protected KVM, abbreviated pKVM, is a new hypervisor originally written by Google for Android, and now upstreamed to the Linux kernel, to protect components managing sensitive data from the Android Linux kernel, and vice versa (see Fig. 2). It has been shipped with Android since version 13 in 2022, and remains in active development.

Arm-A supports multiple privilege levels, known as *exception levels*, from EL0 to EL3. EL0 is typically for user processes, and EL1 for operating-system kernel execution. Hypervisors like pKVM run at EL2, one privilege level higher than the Android kernel itself. Below pKVM, at EL3, is typically the hardware-specific firmware, and other hardware security features such as Arm's TrustZone, which are outside pKVM's domain. pKVM is an isolation kernel, and has an intentionally limited scope: it does not deal with scheduling (which it delegates to the Android kernel), nor with the file system, nor with most interrupts. It supports protected virtual machines (VMs) which are isolated, not just from each other, but also from the base Android kernel (in contrast to KVM). It also allows for explicit, controlled sharing of memory where needed, exposing a small API of hypercalls to the kernel and guest VMs to let the kernel start and manage VMs, and to share or donate pages among them.

In configuring the hardware virtualisation features pKVM enforces isolation between the kernel and VMs. This, involves managing the second-stage translation tables, responsible for translating intermediate-physical (i.e. guest-physical) addresses and performing permission checks, as well as other ancillary virtualisation state. In practice, this is achieved through software translation table visitors, which the developers write as part of the source of pKVM.

pKVM maintains a separate Stage 2 translation tree for the kernel and each VM, and one Stage 1 tree for its own translations (for convenience). These tables form proper trees, but allocate pages from a shared pool. They are dynamically reshaped when inserting or removing mappings.

This design is fairly typical, and is what one finds in many modern isolation kernel hypervisors.

3 A Protocol for Page Table Code

Hardware implementations cover a wide range of designs, supporting a variety of usages and configurations of page tables. The range of behaviours exhibited by those usages is far beyond what any sensible software would rely upon. Moreover, while there are virtual memory systems architecture (VMSA) *memory models* that describe that behaviour precisely [4, 50], they are too complex to think about directly: they consider whole-program candidate executions represented as graphs of memory accesses, enumerate potential justifications of these candidate executions, build relations that capture the emergent ordering, and filter out candidate executions in which these relations do not meet some conditions called 'axioms'.

pKVM and bhyve, and likely much systems software, avoid much of this complexity by following, in an ad-hoc way, an informal programming discipline when manipulating page tables. For example, they avoid racy writes to the page tables, and complex divisions of responsibility for maintenance of page tables. By focusing on this discipline that system software often follows, we can check compliance much more directly. Specifically, by capturing such a discipline in the form of a transition system which updates its state step-by-step, we can make this discipline precise and check whether the program is respecting it at each step, stopping early if we discover a violation.

The key observations of this paper are:

1. by capturing this discipline and the invariants it enforces, we can build a directly executable transition system which avoids these concerns; and
2. this can be achieved by monitoring of only a limited set of events: writes to potential page table memory, TLBIs, and barriers that affect page table accesses, and hence we can make these models effectively *testable*, in online or offline checkers.

Casemate is then an abstraction of the underlying Arm architecture which captures this discipline.

The Casemate protocol. At its core, Casemate maintains a *shadow* copy of the architectural translation state (page tables, translation context, etc.), annotated with additional state. This shadow state tracks the logical and physical state of the page tables, and is used to flag when the logical state becomes inconsistent or if the two diverge too far. In the rest of this section, we explain the design of Casemate informally.

The Casemate protocol is comprised of three parts: (1) reachability; (2) divergence; and (3) synchronisation. First, we track translation contexts (i.e. the collection of system registers that define the translation that's in use), imposing restrictions on those contexts ensuring consistency between context-switches, and use that to determine which trees (and thus, which individual entries) are currently 'live', and possibly being read and cached by some thread's MMU. For those entries, we then impose a per-location protocol, which requires that the code maintains consistency between the in-memory state of the page tables and the state reachable by the MMU, ensuring that the actual translations done by hardware are in sync with the logical translations defined by the in-memory data structures. Finally, we ensure that this correspondence between logical and physical state does not get corrupted by multiple cores racing while updating the page tables by requiring a strict — but realistic in practice — locking discipline. In the following subsections, we give a high-level view of the protocol as illustrated in Fig. 3, and explain its different aspects in more detail.

3.1 Reachability

Page tables are transient: a page can be used as part of a page table tree (e.g. starting a new virtual machine, allocating a new page table), then, when it is not used anymore (e.g. on killing a VM and freeing its page table tree), it may be freed and re-used elsewhere. At any point in time, one can view memory as logically partitioned in two:

- Locations which are 'potentially reachable', and thus must always define well-formed page tables.
- Locations which are 'definitely not reachable'.

Tracking which locations are reachable is not simply a property of the current state of memory: it depends on both the translation context (the current values of the system configuration registers) and the state of divergence between the register and memory state and the MMU state (discussed in §3.2). Accordingly, Casemate tracks writes to the base address register(s), marking new locations as reachable as they become visible, and unmarking locations once cleaned when no longer reachable. We assume, but do not check, race freedom of the unreachable locations.

Translation contexts. We associate with each page table root its *translation context* – the collection of system register values that makes up that translation's configuration – and apply some simplifying restrictions to those contexts. The restrictions we apply are not fundamental, but help simplify the implementation later, without disallowing typical software patterns, in particular, we restrict down to a common set of page table configurations (those used by pKVM and bhyve). We further enforce that a particular translation's context is always preserved, preventing it being changed silently between context switches, as such we forbid runtime changes to the configuration of the page tables while the page table is live.

3.2 Divergence

TLBs and other caching structures mean that, fundamentally, the logical view that software has and the physical state of the page tables are never perfectly in sync. Some divergence is permissible, and merely causes spurious page faults, but failure to properly manage these caches would

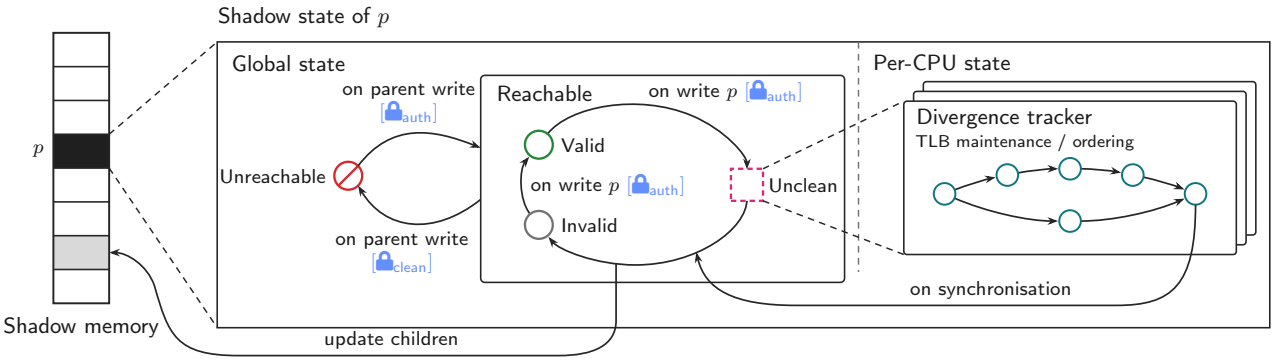

Figure 3. Schematic view of Casemate's shadow state, expanded to show the shadow state of an individual page table entry. Transitions labelled with 🔒 have an additional guard based on the current state.

mean old mappings software has removed may still be accessible in practice, breaking the very isolation guarantee the system is trying to enforce. To manage this, software must ensure that TLBs are cleaned whenever it updates the page tables, removing cached versions of the old unwanted translations. Additionally, Arm requires software to do this by going through a pattern they term 'break-before-make', whereby the entry has to go through an invalid mapping.

To aid performance, Arm provides a rich family of TLB invalidation instructions (TLBIs), including ones which broadcast TLB maintenance to other cores. Software makes use of a broad collection of these, often composing multiple instructions together with various barriers to target precisely the cached entries it needs. Moreover, software tries to amortise the cost of these expensive operations by sharing them between multiple operations, splitting up individual instructions and scattering them across different functions and files in the source, making it hard to see whether the code is correctly managing these caches. Casemate helps bridge this gap, tracking the sequence of operations, ensuring that no step is missed or cached entry overlooked.

This could be done by brute force, merely recording all events and running (our extension of) the axiomatic break-before-make check of Simner et al. [50]. However, working directly with the axiomatic memory model requires tackling a significant state space explosion: one needs to keep track of the entire execution and do an after-the-fact check. Instead, Casemate checks that the hypervisor adheres to a clean discipline which then enables a tractable check on each step, without remembering the entire execution up to that point.

At its core, Casemate enforces this discipline by tracking, for each PTE, how *clean* the physical state is with respect to the logical state. We do this by keeping a shadow copy of the page table memory, where each location is annotated with a small automaton, whose state captures: whether this location is *reachable* by an MMU (see below); if so, whether the descriptor is valid or invalid; and, if unclean (divergent from the logical state), which of the relevant TLB maintenance steps have been taken so far (sketched in Fig. 3). We then define transitions between the states of the automata, corresponding to actions on the PTE (for example, writing to it), precisely tracking how clean the entry is (i.e. whether there could be any cached versions of it).

For example, the 'Unclean' state in the automaton corresponds to a location whose memory value has been updated, but the necessary TLB synchronisation has not been performed, and so it would be a violation to attempt to write a conflicting entry to that location (as Arm requires that this be done with break-before-make).

The shadow state is hierarchical, with global state that all cores agree on, and with thread-local automata tracking local updates to the page table. On synchronisation (e.g. acquiring a lock), thread-local information about the cleanliness is propagated to other threads by updating the global automata (see §3.3).

Tracking TLB maintenance. To each unclean PTE, we associate a thread-local automaton (sketched on the right-hand side of Fig. 3) which captures, for the thread that wrote the invalid entry, how much synchronisation the thread has done so far. The states of the sub-automaton track the cumulative effect of TLB maintenance and thread-local ordering, advancing through the state machine as the software makes progress through the break sequence as required by the architecture. Once the thread-local automaton reaches the end, the location can be considered *clean* for this thread, and this is then propagated globally on the next synchronisation point.

Hierarchical page tables. Invalidating a table entry does not immediately clean the child table's entries. Software must ensure that *all* the children are clean, before the invalidated table entry can be considered cleaned away. Once all the children and the parent are clean, the child becomes *unreachable* (see §3.1).

A common pattern is to walk the tree, updating multiple entries in a row. However, an unsynchronised sequence of writes to different levels of the page table may be observed out-of-order by the hardware translation table walker. This would permit several page table shapes to be visible to the MMU simultaneously, a situation software typically wants to avoid. We therefore check that writes to the same tree are serialised enough (e.g. by locks or hardware memory ordering, see *write authorisation* in §3.3) so it does not happen.

Furthermore, the hierarchical nature of the page tables allows software to reach a number of states which, while not forbidden by the architecture, are generally avoided by software. Notably, page tables need not strictly be trees: the same page is architecturally allowed to appear in the same page table tree multiple times, or even in multiple page tables. For simplicity, we forbid this, although relaxing to allow general acyclic graphs with constant pages would be a relatively straightforward extension.

3.3 Synchronisation

In order to ensure consistency over the global state, we must ensure that the threads' accesses to the page tables themselves are data-race-free.

To ensure data race freedom over the entire page table tree, we enforce an ownership discipline: assigning to each page table tree an owner, and requiring that only the owner performs transitions on locations in that tree. For pKVM and bhyve, we can associate ownership to a lock, reducing the ownership discipline to a lock discipline for almost all cases.

However, there is an exception to reflect pKVM's behaviour. In pKVM, every thread owns one leaf page table entry of pKVM's own page table tree. This allows it to safely read or write to physical memory without having to take its own page table lock. Those PTEs are thus exempt from respecting the locking discipline, but we ensure that they are only accessed by the threads that own them, enforcing data race freedom.

To enforce intra-thread synchronisation of the writes to page tables, we introduce the notion of *write authorisation*, which ensures that the propagation to memory of the writes to the *same page table* is not reordered (as in Fig. 6a line 8). The checker for write authorisation is tightly integrated with the locking discipline to take advantage of the ordering and consequent simplicity that it imposes. Essentially, a write is authorised when it is well-synchronised (e.g. a release write), or a plain write after acquiring the lock or another release write, unless it is a plain write of zero to sibling entries (e.g. zeroing). However, this is unnecessary for page table entries that are statically assigned to a given thread. Therefore, we exempt those from this discipline, and merely check that none of the parents of those PTEs are written to.

Under-approximating synchronisation. The locking discipline we enforce in Casemate is rather strict. While we believe it is not too onerous in practice, and is sufficient for hypervisors we have studied, there are a variety of implementations enforcing data race freedom through other mechanisms. To support these, we also include an *unsound under-approximate* mode of the protocol, which removes many of the restrictions on the locking discipline. This means that the implementation, when in this mode, does not *guarantee* correctness, but still does precise tracking of the reachability and divergence management parts of the protocol.

In the future, we plan to loosen the restriction to more kinds of dynamic data race checking rather than simple locksets.

4 Implementation

We implement Casemate as a library which can be used in two ways: it can be linked to a hypervisor to monitor page table updates 'online', at runtime, or it can be combined with a simple trace reader to work as a standalone trace checker to be run 'offline' on logs. Both of them work with the same API, which exposes a simple interface of step functions for the model transitions. Depending on compile-time configuration, this API is implemented by the library, which executes the model steps, or a logging shim, which records the steps for the offline trace checker. This allows us to both perform efficient runtime checking, even in long-running executions, and extract lightweight traces to perform the check later (e.g. for regression testing).

Casemate is intended for development-time testing, not for deployment in production. Its source and documentation can be found at https://github.com/rems-project/casemate.

Instrumentation. Importantly, the API is implemented in a way which integrates into but does not disrupt the existing hypervisor, and requires only light instrumentation. One needs only to call some initialisation function, and then insert the appropriate calls into the library in the right places in the page table management code.

Concretely, we annotate each location in the hypervisors' source which reads or writes to a page table, performs a DSB instruction, reads or writes relevant system registers (namely, the TTBRs), or does TLB maintenance. For each, we add a call to the corresponding step function. For example, in the `kvm_clear_pte` function, which writes zero to one entry of a page table, we call the model write step, as in Fig. 4.

Most of the places that require annotation already use helper functions or macros in the code (kvm_clear_pte(...), WRITE_ONCE(...), dsb(), etc.), and simply adding the annotation in these places requires little effort, and crucially does not require understanding the underlying protocol.

Transition system implementation. To check that a trace of annotated events is consistent, we implement the transition system described in §3, by encoding the shadow

```
1 static void kvm_clear_pte(kvm_pte_t *ptep) {
2   WRITE_ONCE(*ptep, 0);
3 #ifdef CASEMATE
4   casemate_model_step_write(
5     WMO_plain, hyp_virt_to_phys(ptep), 0
6   );
7 #endif /* CASEMATE */
8 }
```

Figure 4. Instrumented Android pKVM source for write-invalid to page table, `arch/arm64/kvm/hyp/pgtable.c`

```
1 struct sm_location {
2   u64 phys_addr;
3   u64 val;
4   bool is_pte;
5   struct pte_state state;
6   u64 owner;
7 };
```

```
1 struct pte_state {
2   enum pte_state_kind kind;
3   union {
4     valid_state valid_st;
5     invalid_state invalid_st;
6     unclean_state unclean_st;
7   };
8 };
```

Figure 5. Checker C type for a single 64-bit memory location.

state as a C data structure and providing a set of step functions for the various instrumentation points.

The shadow state contains a map from location to a structured datatype including the automata state (Fig. 5). Additionally, we track the page table roots, locks, and (system) register state of each thread.

4.1 Transition system

The step functions implement, in an efficient way, the transitions of the model, with step functions for writes, reads, barriers, register accesses, TLB maintenance and so on. To take one transition as a concrete example, Fig. 6a contains an outline of the write transition. It (lines 4–5) reads the current state of the location being written to, from the model's shadow memory, and updates the value. Then, (6) if, in the model's view, that location is not visible to an MMU, no further checks are required. Otherwise, (7) it checks whether the lock which owns the page table for this location is held by this thread, and whether the write is authorised (see §3.3). Finally, (8–) it dispatches to specific transition functions depending on the global state of the location.

For example, if the location contained a page table entry with the valid bit set, it dispatches to the code in Fig. 6b. It (lines 2–3) reads the *previous* value at that address, and checks whether it was valid. On overwriting one valid entry with another, it (lines 3–8) checks whether this was a case that architecturally required going through an invalid mapping first, i.e. break-before-make, and if so, the model signals an error to the user. Otherwise, on a write of invalid (lines 10–) the state is updated, moving from the 'valid' state to the 'invalid-unclean' state, signifying that, globally, this location is now unmapped but the old mapping may still be cached (i.e. it is 'unclean').

Finally, unwrapping the `requires_bbm` call in Fig. 6c, we see it is essentially an encoding of the Arm architecture requirements as defined by the Arm ARM [10, D8.16.1]. Roughly, break-before-make is required when the old and new descriptors are valid, when there is a change of OA or

```
1 void step_write(struct transition t) {
2   u64 addr = t.write_data.phys_addr;
3   u64 val = t.write_data.val;
4   struct sm_location *loc = location(addr);
5   loc->val = val;
6   if (!is_reachable(loc)) return;
7   assert_has_authority_to_write(loc, t.write_data.mo);
8   switch (loc->state.kind) {
9   case VALID:
10    step_write_on_valid(loc, val, t.write_data.mo);
11    break;
12  ...
```

(a) Checker write step

```
1 static void step_write_on_valid(...) {
2   u64 old = read_phys_pre(loc->phys_addr);
3   if (is_desc_valid(val))
4     if (! requires_bbm(loc, old, val))
5       return;
6     else
7       CASEMATE_MODEL_CATCH_FIRE
8         ("BBM valid->valid");
9
10  loc->state.kind = INVALID_UNCLEAN;
11  loc->state.unclean_st.invalidator_tid = cpu_id();
12  loc->state.unclean_st.old_valid_desc = old;
13  loc->state.unclean_st.local_st = LIS_unguarded;
14 }
```

(b) Checker write transition on a valid PTE

```
1 static bool requires_bbm(...) {
2  struct descriptor before = deconstruct_pte(loc, old);
3  struct descriptor after = deconstruct_pte(loc, val);
4  /* see ARM DDI 0487 J.a D8.14.1 */
5  if (before.kind == PTE_KIND_INVALID
6      || after.kind == PTE_KIND_INVALID)
7   return false;
8  if (before.kind == PTE_KIND_TABLE
9      || after.kind == PTE_KIND_TABLE)
10  return true;
11 if (before.map_data.oa_region.range_size
12     != after.map_data.oa_region.range_size)
13  return true;
14 if (! is_only_update_to_sw_bits(before, after))
15  return true;
16 return false;
17 }
```

(c) Break-before-make check

Figure 6. Excerpts of the C source of Casemate

permissions. We overapproximate the definition, by only permitting idempotent writes or updates to the software-defined bits, marking all other writes as BBM violations.

4.2 Usability

Error reporting. For user-friendliness, Casemate does not merely report that an error has happened, but identifies a specific problem. This includes incorrect synchronisation (transition without holding the lock, write without authorisation, etc.), BBM violations, or incorrect lock usage. Other errors we report include unaligned writes to page table entries and similar hard-to-spot mistakes. This mechanism identified the bugs in pKVM and bhyve we describe in §5.3.

Monitor. Running Casemate as a monitor allows one to validate that the page tables are always consistent (by checking the code follows the Casemate discipline) at runtime. We do this by ensuring the library is written in kernel-compatible C. This means that the monitor can be built with conventional tooling, either in or linked to typical hypervisor codebases such as Linux (pKVM) or FreeBSD: these are almost bare-metal environments, without the usual C standard library (including the usual `malloc`) or any OS facilities, and so any online testing that runs as part of the hypervisor has to do the same. Integration with the kernel codebase means the checker should be accessible to conventional kernel developers without a formal-semantics background, and would likely be essential for upstreaming the monitor.

Offline checking. Casemate can also run *offline*, with a standalone tool `casemate-check`. Given a log of a trace of events, `casemate-check` runs the transition system on each event — just as the online monitor would do — and reports the result.

This gives additional flexibility, allowing one to, in a very lightweight way, collect a trace by logging the page table events to a file, and check them later. While the monitor is much more efficient, and can check long-lived executions, offline checking gives a way to debug bad traces by replaying, or just for fast prototyping of Casemate annotations.

5 Evaluation

We evaluate the effectiveness of Casemate, by:

- demonstrating that it can detect synthetically injected bugs, of a variety of different kinds, on pKVM; and
- systematically annotating pKVM's (Android 14/Linux 6.4 and Android 15/Linux 6.6) and bhyve's (FreeBSD/stable-15) page table manipulation code, detecting a subtle security-critical page table synchronisation bug in each.

Ideally, one would exercise Casemate using for example syzkaller [58] and kernel support for compiler-generated coverage. However, these do not work for pKVM or bhyve.

	default	monitor	checker	tracing
time	3.2 s	422 s	466 s	716 s
factor	1 ×	131 ×	144 ×	222 ×

Figure 7. Test-suite times with the default kernel, online monitor, offline checker, and just tracing (median of 11 runs).

Instead, we evaluate our approach systematically by running the high-coverage pKVM test battery of Memarian et al. [38]. We motivate our use of this test suite, and demonstrate it is complete and performant enough for our needs. We then describe how we use it in conjunction with Casemate to evaluate our tool's ability to detect synchronisation bugs.

5.1 Testing, Coverage and Performance

Regular pKVM development relies on large integration tests (e.g. booting the Android kernel and running regular Android tests), or occasionally tests at the kernel syscall level. These are sufficient for checking for regressions and asserting the positive (non-error) cases, but typically are not systematic. To evaluate Casemate we run the pKVM test suite from Memarian et al. [38] to exercise the page table code. This includes booting the Android kernel, 41 hand-written, self-contained and narrowly focused test cases, and random walks over hypercall invocations, which we leave running for up to 48 hours.

Coverage. These tests exercise 18 of the 26 top-level API calls of pKVM (7 of the remainder are for non-protected vCPU management in KVM, not relevant here; the last is relevant for TLB management, but called automatically from inside pKVM for protected VMs). They cover all of the pKVM page table walker code and callbacks for protected VMs, except for two callbacks pertaining to memory management.

Monitor Performance. Integrating the monitor in pKVM below Linux at runtime means that performance has a functional impact. Much of the usual kernel operation is time-sensitive, and taking too long may cause timeouts, leading to kernel calls into the hypervisor which may obscure the manipulation of page tables. At the same time, for our checker to be sound, it often has to touch every memory location that it thinks could be reachable. While we do not intend for Casemate to run in production, in order to actually exercise the scenarios we are interested in, we need Casemate to achieve reasonable performance. Accordingly, we employ a number of optimisations to improve performance to adequate levels: we cache the shadow state to reduce the number of page table walks Casemate must perform in software; we use page-granularity lookup mechanisms and pre-fetching to make contiguous access faster; we keep intrusive linked lists of various states (e.g. unclean PTEs) to minimise how many traversals are required; and others.

pKVM developers use QEMU for their normal development and testing, making it the appropriate setting in which to evaluate performance. With runtime checking, our test suite, which makes around 750 hypercalls, takes 422 seconds (sequentially on an Intel Xeon Gold 6240, Fig. 7). The time is dominated by the instrumented exceptions.

Linux RCU's CPU stall detector triggered between two and three warnings per execution of our set of tests. This means that we are likely right on the edge of allowable performance: it is usable in testing with QEMU, but if it were any slower it would start impacting the functional behaviour of Linux.

In total we allocate 900 MiB of storage for the entire Casemate state, which is sufficient for pKVM and bhyve with a couple of live VMs.

Offline checker performance. Traces can grow quite large for real systems. For example, pKVM generates roughly a million events during initialisation, and an additional 133000 events running the test suite. This means performance is a constraining factor for practical use.

We obtain some whole-execution traces by running our whole pKVM test suite. Generating the trace takes 716 seconds in QEMU. Much of this time is in extracting the trace via the emulated serial driver; one could instead use (newer) tracing infrastructure to extract traces in real time. The extracted offline checker then takes 466 seconds to check the trace, comparable to the online monitor (Fig. 7).

5.2 Detecting injected bugs

We show that Casemate can find a variety of subtle synchronisation bugs by manually injecting a number of representative bugs of different kinds into pKVM:

- eliding DSB barriers;
- removing one or more TLBI instructions;
- replacing TLB maintenance with thread-local versions, other variants, or shifting or reducing the range;
- eliding or mutating the VMID context-changes around TLB maintenance; and
- eliding memory barriers or substituting memory order on page table accesses.

This is by no means an exhaustive list of all the possible ways to get page table management wrong, but covers a reasonable variety of kinds of error: using the wrong TLB maintenance instruction, insufficient synchronisation around it, or incorrectly managing context-switching around VMIDs.

We inject 10 such bugs into the source of pKVM, covering all of the above categories. Casemate finds 9 of them: one injected bug was not found by Casemate as the Android 14 pKVM kernel does not hit it: since Casemate is a dynamic check, it can miss executions which do not occur in practice.

5.3 Uncovering real bugs in pKVM and bhyve

We demonstrate Casemate by applying it to pKVM and bhyve, and in doing so discover page table synchronisation bugs that transiently expose memory that should have been private.

We now walk through one of the security-critical TLB synchronisation bugs we found in some detail, both to show the kind of errors Casemate can catch, and as an example of the kind of bug that appears in such code.

```
1  static bool stage2_try_break_pte(...) {
2   ...
3   if (!stage2_try_set_pte(ctx, 0))
4    return false;
5   ...
6   if (kvm_pte_table(ctx->old, ...)) {
7    // ... in __kvm_tlb_flush_vmid
8    struct tlb_inv_context tlbcxt;
9    enter_vmid_context(mmu, &tlbcxt, false);
10   __tlbi(vmalls12e1is);
11   dsb(ish);
12   isb();
13   exit_vmid_context(&tlbcxt);
14  } else if (kvm_pte_valid(ctx->old)) {
15  ...
16 }
```

(a) pKVM Stage 2 'break' (`arch/arm64/kvm/hyp/pgtable.c`).

```
1  static void enter_vmid_context(...) {
2   vcpu = host_ctxt->__hyp_running_vcpu;
3   /*
4    * If we're already in the desired
     ↪ context,
5    * then there's nothing to do.
6    */
7   if (vcpu) {
8     ...
9   } else {
10   /* We're in host context */
11   if (mmu == host_s2_mmu)
12    return;
13  }
14  dsb(ish);
15  ...
16  if (vcpu)
17   __load_host_stage2();
18  else ...
19  isb();
20 }
```

(b) pKVM VMID-switch code (`arch/arm64/kvm/hyp/nvhe/tlb.c`).

Figure 8. The pKVM bug identified by Casemate

Stage 2 break. pKVM has separate code sources for the break-before-make sequences for Stage 1 and Stage 2 translations, as their requirements are slightly different. Fig. 8a contains the Android source of the 'break' part for a stage 2 break-before-make sequence. It starts by writing an invalid entry to the page table (line 3), before dispatching TLB invalidation (lines 9–13). The TLB invalidation sequence is

different depending on whether the old entry was a leaf or a table. As an example we examine only the first case, that the previous value was that of a table entry, and we inline the code of the invalidation into the function for clarity.

The key part of the break is the TLB maintenance to invalidate any cached entry of the old PTE for that VMID, which is achieved with a single TLBI VMALLS12E1IS instruction which clears all stage 1 and stage 2 cached entries for the current VMID. This needs DSB barriers before and after it to synchronise it with the other instructions in the thread. Before performing the TLB maintenance, software must ensure the target VMID is set as the current VMID in the VTTBR. Both the VMID-switch and the TLB maintenance require (at least) a DSB, and so Android re-uses the synchronisation in one for the other, in `enter_vmid_context` (Fig. 8b). So the code must do a sequence of:

dsb set-VTTBR isb tlbi dsb reset-VTTBR isb

A bug was introduced when the pKVM developers introduced an optimisation: if the right VMID is already loaded (Fig. 8b, line 11), there is no need to do the switch, and so the function could return early (line 12). However, this meant that the DSB would not be hit in that case, but as just mentioned, it was serving a dual purpose. This means that breaking a PTE for an already-loaded VM would fail to synchronise the TLB maintenance, potentially permitting that VM to continue to access memory which it logically no longer owns. This bug was missed during Google's code review process, and was found by Casemate, reported, and fixed before the release of Android 15.

We discovered an additional TLB synchronisation bug in the initialisation of pKVM when reviewing the code, although the runtime monitor of Casemate cannot run during this phase so it cannot catch this bug.

bhyve. We identified two bugs in bhyve with Casemate. The first was a failure in the protocol: a missing DSB between exception entry and the first TLBI by a stage 2 update, which could (depending on the hardware and timing) have allowed an attacker access to a page it should not have access to anymore. The second bug was in the address space allocation for EL2 data structures, which was revealed by a violation of Casemate's internal invariants. The code incorrectly assumed that the allocator returned contiguous pages. These bugs were reported, and (at the time of writing) the first one has been fixed.

6 Soundness

The design of Casemate was based on a careful abstraction from the axiomatic VMSA model of Simner et al. [50], extended to account for multi-stage BBM. This semantics-driven approach ensures the following relationship:

Theorem 1. *The axiomatic VMSA model refines Casemate.*

To state Theorem 1 precisely, we need to bridge the gap of style between the two models: the axiomatic model works on event graphs, while Casemate works on traces.

Moreover, as described earlier, the fact that Casemate is defined in C makes such a direct proof out of scope. We circumvent this by writing a version of Casemate in a more traditional mathematical style, using the Rocq specification language, and do our pen-and-paper proof over this definition. We extract, from the Rocq definition, an executable trace checker, which we cross-check with the C trace checker, boosting assurance in both.

The definition of the automata and the step functions that drive them takes 1948 lines of (non-whitespace) Rocq code. By contrast, the C implementation of the automata is 1493 lines of code and 657 lines of headers containing type definitions of the state and transitions. The length of the Rocq version reflects the underlying complexity of the discipline.

We sketch the high-level structure of this proof and its salient points in this section, and present its detail in the supplementary material [9]. To show the refinement statement above, we introduce an intermediate model, which we call the axiomatic simplified model (ASM), which is in axiomatic style, but internalises the state tracking of Casemate; we then show that the VMSA model refines the axiomatic simplified model, which itself refines Casemate. Moreover, we show that the linearisation imposed by working on traces does not imply a blind spot that would make Casemate miss synchronisation violations: there always exists a linearisation. This latter property is important for our use of Casemate on traces resulting from logging, which (by contention on the output stream) imposes extraneous ordering.

Axiomatic simplified model. The axiomatic simplified model aims to bridge the two models. It differs from the axiomatic VMSA model in the detection of synchronisation violations, and in particular in the detection of BBM, which it purposefully over-approximates. Concretely, it internalises the ownership discipline of page tables of Casemate, and tracks potential violations of break-before-make in terms of a mostly thread-local sequence of events, rather than the indirect constraints on write ordering existentially quantified in the axiomatic model. It does not have its own notion of consistency, but instead uses that of the VMSA model.

To track the page table locking discipline, we make three assumptions: (1) That we can identify events that are part of the implementation of a lock. We do this by relying on the division of memory assumed by Casemate, which requires that all locks are in a region of memory separate from page tables and VM memory. (2) That we can identify, in the lock implementation, the locking with a distinguished read acquire event, and the unlocking with a distinguished write release event, which the lock implementations of pKVM and

bhyve satisfy. (3) That the lock implementation is correct, enforcing alternation between lock and unlock events.

Erasure. Relying on these requirements, in the axiomatic simplified model, we erase events unrelated to page table accesses, keeping only events on page table entries and only lock and unlock events. Focusing on the page table events means that, unlike for some DRF-SC properties (data-race-freedom implies sequential consistency, e.g. [12, §3]), we do not need to reconstruct a counterfactual execution: it is enough for Casemate to scan the page table events and look for violations of synchronisation in situ.

Left refinement: VMSA $\sqsubseteq$ ASM. The axiomatic simplified model accepts certain BBM sequences, which we relate to actual BBM sequences of the VMSA model. The proof is by relational algebra, showing that the edges of the VMSA model subsume those of the axiomatic simplified model.

Right refinement: ASM $\sqsubseteq$ Casemate. The axiomatic simplified model still works on execution graphs as opposed to the traces of Casemate, but is quite close to it. In particular, thanks to the ownership discipline, the state transitions of Casemate correspond to clear sequences of states in the axiomatic simplified model. The main challenge is to show that, for any execution of the axiomatic simplified model, there exists a *trace* – that is, a linearisation that is compatible with program order and lock order. To do this, for every deviation from the discipline that would prevent the linearisation, we identify a clause of Casemate that makes the trace catch fire.

We also show that Casemate works for any linearisation:

Lemma 2. *Given a consistent execution X with a synchronisation violation, Casemate reports it on* ***any*** *trace of X.*

Extending the break-before-make predicate. The formal break-before-make predicate of the VMSA model of Simner et al. [50] only considers the simplest, most expensive invalidation method, using a DSB; TLBI all; DSB. In practice, systems code often uses lightweight, multi-step invalidations. Therefore, we extend the break-before-make predicate to handle two-stage invalidation by IPA and VA.

7 Related work

Improving reliability of operating systems and hypervisors despite the complexity and unclarity of the hardware support they utilise is a long-standing challenge. Efforts along this line span from the industry state of the art – integration tests, and testing the system in production on its users – to more principled approaches: from adapting code sanitisation tools to work on systems code, all the way to formal verification. Casemate, as a lightweight formal method [15, 30, 31, 39], shares aspects with both ends of the spectrum: it is a testing tool, but one tightly connected to ground truth about the architecture via our refinement theorem. Technically, it is most related to verification efforts, which we now discuss.

Microsoft's Hyper-V hypervisor was one of the targets of Verisoft XT [8, 14, 35], an ambitious, early hypervisor verification project. It is written in a mixture of C and x86-64 assembly, and was not designed for verification. Verification relied on a custom, ad-hoc semantics of combined C and x86-64 assembly [17]. While groundbreaking, this project was only able to verify a small fraction of the hypervisor, and did not properly address concurrency.

The more recent HASPOC [13] relies on an early formal instruction semantics of ARMv8 in HOL and L3 [21, 22], and is multicore, but has a static partition of memory, which means it avoids much of the complexity of virtual memory.

seL4 [33, 34]'s proof abstracts from TLB management and other low-level interaction with hardware, and assumes functions that touch those are correct, and have no effect on the behaviour of the kernel [46]. It relies on abstractions over an ad-hoc formalisation of ARMv7 virtual memory [54–57] without relaxed memory. In fact, it was designed for a sequential setting, and has so far only been ported to sequential consistency [33].

CertiKOS [27, 28, 48] is a de novo operating system designed for verification. Its design is significantly different from classical systems code, and incompatible with common C programming idioms, which makes it not applicable to codebases not written for verification. Moreover, it essentially assumes sequentially consistent concurrency.

SeKVM [36, 37] inserts itself to replace KVM with a hypervisor written for verification which is in a sense a successor to CertiKOS. Tao et al. [59] extend SeKVM to account for relaxed memory, using wDRF, a novel weak data race freedom condition they develop, to largely reduce relaxed reasoning involving page tables to SC reasoning. They do this with respect to an ad-hoc model of Armv8 virtual memory, which assumes that translation reads behave like normal reads. It has been since clarified [50] that translation reads are, in several respects, substantially weaker. We do not know whether wDRF is sound with respect to that weaker semantics, but we believe that pKVM does not adhere to wDRF in any case, as it violates the transactional requirement.

The executable specification of pKVM by Memarian et al. [38], whose instrumentation and tests we reuse, tracks the in-place mappings, but does not check synchronisation of page table accesses, making our work complementary.

8 Limitations

While Casemate hits a reasonable spot in the design space, it makes a few compromises. In particular, it focuses on stage 2 translation, ignoring hardware updates to access flags and dirty bits; it enforces sensible, but rigid, tree-like shapes of page tables; it rejects esoteric, but not architecturally forbidden, highly racy page table code; and it over-approximates ownership through locks. These limitations, although they could be lifted by enriching the transition

system, exist as a happy balance between what software relies on today and what is checkable with the performance needed to run without interfering with the system being observed (see §5). Other systems can and do operate outside the strict bounds of Casemate's protocol: Linux and FreeBSD's first-stage page table management manages access and dirty bits, for paging; BlueRock Security's NOVA microhypervisor uses finer-grained synchronisation; and other proprietary systems use other hardware virtualization features unsupported by Casemate. For these systems, Casemate can still be used (with underapproximation) to check the parts that Casemate does understand. Such restrictions are not fundamental – we simply choose a practical balance of features for the software we focus on – and we expect that they could be lifted.

The proof, described in §6, is only a pen-and-paper proof, and typically lags a few versions behind the Casemate monitor implementation. While up-to-date mechanised proofs would be the ideal, Casemate is first and foremost a testing tool, with the proofs there to help ground the implementation; soundness is not integral to the usefulness of the tool.

9 Conclusion and Future Work

Our use of Casemate on both pKVM and bhyve, and the fact that it identified a significant bug in each, gives confidence that it could be applied to other Arm-based hypervisors and OS kernels. In fact, developers associated with other major industry hypervisors have expressed interest in doing so.

The current version of Casemate strikes a balance between expressivity and simplicity, but there are several avenues for extending tracking, in particular by lifting the limitations listed above. More broadly, we could generalise Casemate to other architectures, and to other features (instruction cache management, exception handling, IOMMU, etc.).

Casemate was designed for testing, but is an important step towards foundational verification of systems code. It provides an intermediate model, firmly grounded in the Arm architectural semantics, but abstracting away from much of the complexity of the relaxed concurrency model. This could be leveraged to achieve such verification, for example by taking existing operational-semantics based approaches (such as Iris) and enriching their logics with a Casemate-derived state.

Finally, we argue that the idea of constructing abstractions of the architecture to track non-architectural state managed by software is a powerful one: it enables the construction of semantics-driven tooling which works as a robust testing infrastructure which can and should be widely used by the systems community.

Acknowledgments

We thank the pKVM development team, especially Will Deacon and Keir Fraser, for extensive discussions, and Ben Laurie and Sarah de Haas for their support. We thank John Baldwin for his input on FreeBSD. We thank Baptiste Lepers for his comments on the paper.

We thank our anonymous reviewers, and the shepherd, for their genuinely helpful insights and comments.

This work was funded in part by Google. This work was funded in part by Arm. This work was funded in part by an AUFF starter grant (Pichon-Pharabod). This work was funded in part by two Amazon Research Awards (Pichon-Pharabod; Sewell and Simner). This work was funded in part by École Polytechnique (Fourier, during an internship at Cambridge). This work was funded in part by the Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-IT2102-03 (Han). This work was funded in part by UK Research and Innovation (UKRI) under the UK government's Horizon Europe funding guarantee for ERC-AdG-2022, EP/Y035976/1 SAFER. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 789108, ERC-AdG-2017 ELVER). This work is supported by ERC-2024-POC grant ELVER-CHECK, 101189371. Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them. This work was supported in part by the Innovate UK project Digital Security by Design (DSbD) Technology Platform Prototype, 105694. The authors would like to thank the Isaac Newton Institute for Mathematical Sciences, Cambridge, for support and hospitality during the programme Big Specification, where work on this paper was undertaken. This work was supported by EPSRC grant EP/Z000580/1.

References

[1] Allon Adir, Hagit Attiya, and Gil Shurek. 2003. Information-Flow Models for Shared Memory with an Application to the PowerPC Architecture. *IEEE Trans. Parallel Distrib. Syst.* 14, 5 (2003), 502–515. doi:10.1109/TPDS.2003.1199067

[2] Jade Alglave. 2010. *A Shared Memory Poetics*. Ph. D. Dissertation. Université Paris 7 – Denis Diderot.

[3] Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. 2021. Armed Cats: Formal Concurrency Modelling at Arm. *ACM Trans. Program. Lang. Syst.* 43, 2 (2021), 8:1–8:54. doi:10.1145/3458926

[4] Jade Alglave, Richard Grisenthwaite, Artem Khyzha, Luc Maranget, and Nikos Nikoleris. 2024. Puss In Boots: on formalising Arm's Virtual Memory System Architecture (extended version). (May 2024). https://inria.hal.science/hal-04567296 working paper or preprint.

[5] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2010. Fences in Weak Memory Models. In *Proc. CAV*.

[6] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. 2011. Litmus: Running Tests Against Hardware. In *Proc. TACAS*.

[7] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2 (2014), 7:1–7:74. doi:10.1145/2627752

[8] Eyad Alkassar, Mark A. Hillebrand, Wolfgang J. Paul, and Elena Petrova. 2010. Automated Verification of a Small Hypervisor. In *Verified Software: Theories, Tools, Experiments, Third International Conference, VSTTE 2010, Edinburgh, UK, August 16-19, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6217)*, Gary T. Leavens, Peter W. O'Hearn, and Sriram K. Rajamani (Eds.). Springer, 40–54. doi:10.1007/978-3-642-15057-9_3

[9] Anonymous. 2024. Supplementary material.

[10] Arm. 2024. Arm Architecture Reference Manual: for A-profile architecture. https://developer.arm.com/documentation/ddi0487/latest. Accessed 2024-05-11. Issue K.a. 14777 pages..

[11] Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Pulte, and Peter Sewell. 2021. Isla: Integrating full-scale ISA semantics and axiomatic concurrency models. In *In Proc. 33rd International Conference on Computer-Aided Verification*. Extended version available at https://www.cl.cam.ac.uk/~pes20/isla/isla-cav2021-extended.pdf.

[12] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. The Problem of Programming Language Concurrency Semantics. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. 283–307. doi:10.1007/978-3-662-46669-8_12

[13] Christoph Baumann, Mats Näslund, Christian Gehrmann, Oliver Schwarz, and Hans Thorsen. 2016. A high assurance virtualization platform for ARMv8. In *European Conference on Networks and Communications, EuCNC 2016, Athens, Greece, June 27-30, 2016*. 210–214. doi:10.1109/EuCNC.2016.7561034

[14] Bernhard Beckert and Michal Moskal. 2010. Deductive Verification of System Software in the Verisoft XT Project. *Künstliche Intell.* 24, 1 (2010), 57–61. doi:10.1007/S13218-010-0005-7

[15] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. 2021. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 836–850. doi:10.1145/3477132.3483540

[16] David Brazdil and Serban Constantinescu. 2022. Android Virtualization Framework — Protected computing for the next generation use cases. presented at the Linux Plumbers Conference Android Microconference.

[17] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A Practical System for Verifying Concurrent C. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics* (Munich, Germany) *(TPHOLs '09)*. Springer-Verlag, Berlin, Heidelberg, 23–42. doi:10.1007/978-3-642-03359-9_2

[18] William W. Collier. 1992. *Reasoning about parallel architectures*. Prentice Hall.

[19] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (St. Petersburg, FL, USA)*. 608–621. doi:10.1145/2837614.2837615

[20] Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. 2017. Mixed-size Concurrency: ARM, POWER, C/C++11, and SC. In *The 44st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France*. 429–442. doi:10.1145/3009837.3009839

[21] Anthony C. J. Fox. 2015. ARMv8 L3 model. http://www.cl.cam.ac.uk/~acjf3/l3/isa-models.tar.bz2

[22] Anthony C. J. Fox. 2015. Improved Tool Support for Machine-Code Decompilation in HOL4. In *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*. 187–202. doi:10.1007/978-3-319-22102-1_12

[23] Kourosh Gharachorloo. 1995. *Memory Consistency Models for Shared-Memory Multiprocessors*. Ph. D. Dissertation. Stanford University.

[24] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip B. Gibbons, Anoop Gupta, and John L. Hennessy. 1990. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture, Seattle, WA, USA, June 1990*, Jean-Loup Baer, Larry Snyder, and James R. Goodman (Eds.). ACM, 15–26. doi:10.1145/325164.325102

[25] Google LLC. 2024. Android Virtualization Framework (AVF) overview. https://source.android.com/docs/core/virtualization

[26] Kathryn E. Gray, Gabriel Kerneis, Dominic P. Mulligan, Christopher Pulte, Susmit Sarkar, and Peter Sewell. 2015. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In *Proceedings of the 48th International Symposium on Microarchitecture (Waikiki)*. 635–646. doi:10.1145/2830772.2830775

[27] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. ACM, New York, NY, USA, 595–608. doi:10.1145/2676726.2676975

[28] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. 653–669. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu

[29] Intel. 2002. A Formal Specification of Intel Itanium Processor Family Memory Ordering. http://download.intel.com/design/Itanium/Downloads/25142901.pdf.

[30] Daniel Jackson and Jeannette Wing. 1996. *IEEE Computer*. Chapter Formal Methods Light §Lightweight formal methods, 21–22. doi:10.1109/MC.1996.10038

[31] Cliff B. Jones. 1996. *IEEE Computer*. Chapter Formal Methods Light §A rigorous approach to formal methods, 20–21. doi:10.1109/MC.1996.10038

[32] Dave Kleidermacher. 2024. Why AVF? article in Dave Kleidermacher's blog. https://davek.substack.com/p/why-avf

[33] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David A. Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2010. seL4: formal verification of an operating-system kernel. *Commun. ACM* 53, 6 (2010), 107–115. doi:10.1145/1743546.1743574

[34] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive Formal Verification of an OS Microkernel. *ACM Transactions on Computer Systems* 32, 1 (Feb. 2014), 2:1–2:70. doi:10.1145/2560537

[35] Dirk Leinenbach and Thomas Santen. 2009. Verifying the Microsoft Hyper-V Hypervisor with VCC. In *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*. 806–809. doi:10.1007/978-3-642-05089-3_51

[36] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. 2021. Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 3953–3970. https://www.usenix.org/conference/usenixsecurity21/presentation/li-shih-wei

[37] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. 2021. A Secure and Formally Verified Linux KVM Hypervisor. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 839–856. doi:10.1109/SP40001.2021.00049

[38] Kayvan Memarian, Ben Simner, David Kaloper-Mersinjak, Thibaut Pérami, and Peter Sewell. 2025. Ghost in the Android Shell: Pragmatic Test-oracle Specification of a Production Hypervisor. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles, SOSP 2025, Lotte Hotel World, Seoul, Republic of Korea, October 13-16, 2025*, Youjip Won, Youngjin Kwon, Ding Yuan, and Rebecca Isaacs (Eds.). ACM, 685–700. doi:10.1145/3731569.3764817

[39] Scott Owens, Peter Böhm, Francesco Zappa Nardelli, and Peter Sewell. 2011. Lem: A Lightweight Tool for Heavyweight Semantics. In *Interactive Theorem Proving - Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22-25, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6898)*, Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk (Eds.). Springer, 363–369. doi:10.1007/978-3-642-22863-6_27

[40] Sandeep Patil and Irene Ang. 2023. Virtual Machine as a core Android Primitive. article in the Android Developers blog. https://android-developers.googleblog.com/2023/12/virtual-machines-as-core-android-primitive.html

[41] Christopher Pulte. 2018. *The Semantics of Multicopy Atomic ARMv8 and RISC-V*. Ph. D. Dissertation. University of Cambridge. https://www.repository.cam.ac.uk/handle/1810/292229.

[42] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL (2018), 19:1–19:29. doi:10.1145/3158107

[43] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. 2012. Synchronising C/C++ and POWER. In *Proceedings of PLDI 2012, the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (Beijing)*. 311–322. doi:10.1145/2254064.2254102

[44] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER Multiprocessors. In *Proceedings of PLDI 2011: the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation*. 175–186. doi:10.1145/1993498.1993520

[45] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus Myreen, and Jade Alglave. 2009. The Semantics of x86-CC Multiprocessor Machine Code. In *Proceedings of POPL 2009: the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*. 379–391. doi:10.1145/1594834.1480929

[46] seL4 Project. 2026. seL4: What the Proofs Assume. https://sel4.systems/Verification/assumptions.html

[47] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Commun. ACM* 53, 7 (July 2010), 89–97. (Research Highlights).

[48] Xiaomu Shi, Jean-François Monin, Frédéric Tuong, and Frédéric Blanqui. 2011. First Steps towards the Certification of an ARM Simulator Using Compcert. In *Certified Programs and Proofs*, Jean-Pierre Jouannaud and Zhong Shao (Eds.). Lecture Notes in Computer Science, Vol. 7086. Springer Berlin Heidelberg, 346–361. doi:10.1007/978-3-642-25379-9_25

[49] Ben Simner, Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Ohad Kammar, Jean Pichon-Pharabod, and Peter Sewell. 2025. Precise exceptions in relaxed architectures. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture, ISCA 2025, Tokyo, Japan, June 21-25, 2025*. ACM, 211–224. doi:10.1145/3695053.3731102

[50] Ben Simner, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell. 2022. Relaxed virtual memory in Armv8-A. In *Proceedings of ESOP 2022: 31st European Symposium on Programming, held as part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany*.

[51] Ben Simner, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell. 2022. Relaxed virtual memory in Armv8-A (extended version). arXiv:2203.00642 [cs.AR] https://arxiv.org/abs/2203.00642

[52] Ben Simner, Shaked Flur, Christopher Pulte, Alasdair Armstrong, Jean Pichon-Pharabod, Luc Maranget, and Peter Sewell. 2020. ARMv8-A system semantics: instruction fetch in relaxed architectures (extended version). In *Proceedings of the 29th European Symposium on Programming*.

[53] P. S. Sindhu, J.-M. Frailong, and M. Cekleov. 1991. Formal Specification of Memory Models. In *Scalable Shared Memory Multiprocessors*. Kluwer, 25–42.

[54] Hira Syeda and Gerwin Klein. 2017. Reasoning about Translation Lookaside Buffers. In *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017*. 490–508. http://www.easychair.org/publications/paper/340347

[55] Hira Taqdees Syeda. 2019. *Low-level program verification under cached address translation*. Ph. D. Dissertation. University of New South Wales, Sydney, Australia. http://handle.unsw.edu.au/1959.4/63277

[56] Hira Taqdees Syeda and Gerwin Klein. 2018. Program Verification in the Presence of Cached Address Translation. In *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*. 542–559. doi:10.1007/978-3-319-94821-8_32

[57] Hira Taqdees Syeda and Gerwin Klein. 2020. Formal Reasoning Under Cached Address Translation. *J. Autom. Reason.* 64, 5 (2020), 911–945. doi:10.1007/s10817-019-09539-7

[58] Syzkaller contributors. [n. d.]. Syzkaller: An unsupervised coverage-guided kernel fuzzer. https://github.com/google/syzkaller.

[59] Runzhou Tao, Jianan Yao, Xupeng Li, Shih-Wei Li, Jason Nieh, and Ronghui Gu. 2021. Formal Verification of a Multiprocessor Hypervisor on Arm Relaxed Memory Hardware. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 866–881. doi:10.1145/3477132.3483560