Abstract architecture to catch concrete bugs: checking Android hypervisor TLB 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, based on a novel sound abstraction of the underlying relaxed hardware architecture semantics, specialised to a particular programming discipline used by systems code that is efficiently checkable at runtime. We implement this technique in a tool, Casemate, which can be integrated as a monitor into systems code, or as a standalone offline checker over logs. We apply Casemate to production systems code: Google's pKVM hypervisor for Android, systematically covering its virtual memory management code. Casemate successfully finds a variety of synthetic TLB synchronisation bugs, and (to date) one subtle security-critical real bug, which was fixed before release. We found another real TLB synchronisation bug while working on the system, also fixed. We give a pen-and-paper proof of the soundness of the Casemate model with respect to the underlying Arm-A semantics.

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 translations, from virtual to physical addresses, are 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 relaxed systems concurrency problem. Modern hardware is aggressively pipelined, with out-of-order and speculative execution, and with extensive caching and buffering of memory. For normal memory accesses, caches are coherent, but for translation-table-walk accesses, the caches, called TLBs (Translation Lookaside Buffers) must be kept in sync by the programmer. 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. The systems programmer must use explicit TLB invalidation instructions (TLBIs), and various barriers, to ensure proper removal of old cached entries. In the Arm-A architecture, this must be done before creating new mappings, following a Break-Before-Make (BBM) discipline. A failure to properly clean the TLB can lead to memory pages being transiently visible to the wrong process, violating the core security guarantee of the hypervisor or OS.

Third, defining precisely 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, 12, 20–22, 25, 26, 28, 31, 43–48, 54], but that did not touch on 'systems' relaxed concurrency. Recent work by Simner et al. [53] developed semantics for Arm-A instruction fetch and instruction-cache maintenance, and Simner et al. [51] and Alglave et al. [4] developed semantics for Arm-A 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 concurrent systems code – but they are 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 [49]. The Arm-A architecture (much like RISC-V and IBM Power) is very much not SC, even for 'user' concurrency: it allows relaxed behaviours that are especially problematic for reasoning. Previous efforts in kernel and hypervisor verification have made some progress towards this goal, e.g. SeKVM [39, 40] appeals to a *write data-race-free* (wDRF) property, which gives a bound to the behaviour of the virtual memory system under some race freedom assumptions over pagetables.

We leverage the recent semantic understanding of relaxed virtual-memory management concurrency to develop a broadly similar property that is a provably sound abstraction over the architecture. We use this abstraction to create a runtime tester, *Casemate*. This tool *tests* whether software follows the BBM discipline required by the architecture when changing the memory mapping. We do this in a way that applies *in a lightweight way* to *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 Google's *pKVM* hypervisor, used to enforce isolation between the Android Linux kernel and guest virtual machines – protecting the latter from post-initialisation kernel compromises, and vice versa. pKVM is developed in the Android Linux kernel tree, in C and Arm assembly, and has been deployed since Android 13 [18, 27, 35, 42].

Contributions:

- We develop Casemate, a tool for Concurrent Architectural System-level Monitoring and Testing, which detects synchronisation errors in the management of page tables, and in particular BBM violations. Casemate is a monitor, checking that the program sticks to some safe set of behaviour, including BBM and locking disciplines. Internally, Casemate tracks the state of each individual entry of the page tables with a simple automaton. It relies on the fact that well-behaved systems code follows a particular discipline in managing the hardware mechanisms. However, 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, and depend on each entry's reachability history.
- We implement Casemate as a practical runtime online monitor, in C and integrated with (but not disrupting) the code of the hypervisor, and with sufficient performance to run in pKVM's usual QEMU development and test environment, without disrupting its execution. We exercise it over a custom suite of tests (§4). Because Casemate checks the intensional discipline, not just end-to-end memory-access violations, it can find bugs without massive test runs.
- This checking found a security-critical TLB synchronisation bug in pKVM, which transiently exposed memory to the wrong virtual machine (§5.3). This was reported to the Google development team and fixed before release. Development of Casemate also found another synchronisation bug, during the initial hand off between Linux and pKVM.
- We give a precise, self-contained definition of Casemate by mechanising it in Rocq in a way that supports extraction to an executable *offline* checker, which checks logs from the production hypervisor instrumentation (§4.2). This identifies the same bugs.
- We ground Casemate more formally by giving a paper proof that the mechanised version of Casemate is sound with respect to the axiomatic relaxed virtual memory model of Simner et al. [51] (§6) and the definition of BBM.

Hypervisors and other system code rely on programming models that embody multiple simplifications above the underlying hardware architecture, providing fictions of virtual memory (with mapping on demand, and disjoint-by-default address spaces with controlled sharing between different virtual machines), 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.

Note: the testing infrastructure we use here was developed primarily for a related paper under submission, and reported there [10]. Both papers target the same hypervisor, but are otherwise distinct: this focusses on TLB synchronisation disciplines, while that on top-level functional specification.

2 Background

2.1 Arm virtual memory

The Arm-A virtual memory architecture, defined in prose in the Arm architecture reference manual [11, Ch. D8], is complex. Here, we summarise the key aspects for a common configuration, to understand how pKVM enforces isolation.

Address translation. Virtual memory indirects the program's accesses use through a memory management unit (MMU), which both translates *virtual* addresses (VA) used by the program into a hardware *physical* addresses (PA), and checks for permissions. These mappings are determined by in-memory data structures consisting of trees of tables (as in Fig. 1), the *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 addresses 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.

For each data or instruction-fetch memory access by the program, the hardware does a translation table walk (if not cached) to translate the virtual address which the program requested to access. 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 not in the domain of the translation, and thus is *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.

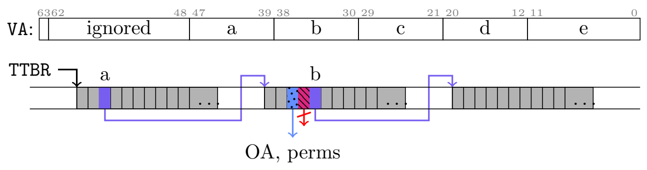

Figure 1. A sketch of a translation table walk, with an input virtual address and page tables (in grey). The input address is split into slices (a–e), which are used as indices into the nested tree of tables. We show one entry of each kind: an *invalid* entry in hatched magenta, a valid leaf in dotted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

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, by writing its root address into the appropriate TTBR register. This gives each process its own translate function. Processes can then be controlled by ensuring that their physical-memory ranges are disjoint except where otherwise desired.

Virtualisation. Hypervisors like pKVM follow a similar scheme: they maintain page tables for each virtual machine. By ensuring VM's physical memory ranges are disjoint except for controlled sharing, isolation between VMs is guaranteed. To allow each VM to perform proccess 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 *intermediate-physical* address (IPA). The IPA is then translated into *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 switching 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. The caches 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 addresses which are not mapped (and thus 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 can cache multiple *non-conflicting* 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), for example resulting in two different output addresses, the behaviour of the process or VM becomes 'constrained unpredictable'. This means the architecture guarantees almost nothing on what it is allowed to do. In particular, its normal memory accesses are not constrained by its page table, and are not even guaranteed to be coherent. Software has to avoid this at all costs, and one therefore needs to ensure that whenever there is a write to a potentially reachable page table entry, it and its children are not in conflicting states, even transiently. The exact details of what need to be done to ensure this are architecture-specific, and are significantly different for Armv8+, on which we focus, than for x86 and 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.

The basic idea of Casemate is to track, for each page table entry, how far through the break-before-make protocol it is. However, there are many subtleties one has to deal with.

Fine-grained TLBIs. Arm provide 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 or non-broadcast (for comparison, x86 only has 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, and so 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 to account for change of responsibilities within the codebase significantly error-prone, even for experts – as illustrated by the bugs we identified (§5.3).

In this paper, we only consider broadcast TLBIs, which affect all cores.

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 hypervisors like 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 order, 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 micro-TLBs, 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

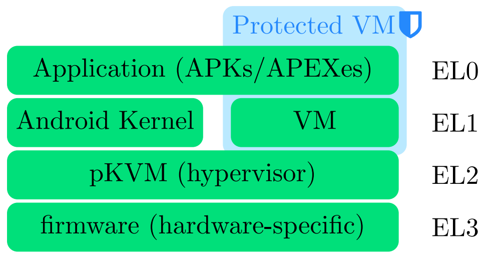

Figure 2. Architecture of pKVM (https://source.android.com/docs/core/virtualization) .

relaxed virtual memory [4, 51], 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 mean a translation table walk to see any combination of the possible trees induced by those writes [52, §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 with its respective explicit access – although proper TLB maintenance ensures that this microarchitectural behaviour is not visible to the programmer [52, §A.4.1.6].

Simner et al. [51] 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 An example modern hypervisor: pKVM

Protected KVM, abbreviated pKVM, is a new hypervisor written by Google for Android, 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. pKVM runs at EL2, one privilege level higher than the Android kernel itself. Below pKVM, at EL3, is typically the the hardware-specific firmware, and other hardware security features such as Arm's TrustZone, which are outside pKVM's domain. pKVM 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. pKVM is implemented as part of the Arm port of the Linux kernel, in Linux-kernel C code with Arm embedded assembly for TLBI instructions and barriers.

pKVM maintains Stage 2 translation mappings, for the kernel and for each VM, and a Stage 1 translation mapping for its own execution. The hypercalls and MMU fault handlers modify these mappings. To do this safely, the code must follow the break-before-make discipline discussed informally above. Failure to do so would allow, for example, a virtual machine to access memory it does not own, resulting in a security breach.

3 Casemate

The full architectural virtual memory models are written in a form that makes them challenging to check directly. They take a complete whole-program candidate execution, construct multiple large relations between the events of that execution, and then perform acyclicity and emptiness checks over those relations.

pKVM, and likely much systems software, avoids much of the complexity of the virtual memory systems architecture (VMSA) by following a clean programming discipline when manipulating page tables. For example, it avoids racy writes to the page tables, and complex divisions of responsibility for maintenance of page tables.

By enforcing such a discipline we can construct a simpler model, which is operational in nature, updating its state incrementally step-by-step. We can then check that the program is respecting the discipline at each step, stopping early if a violation is discovered.

The key observations of this paper are:

1. by capturing a reasonable programming discipline and the invariants it enforces, we can build a considerably simpler model which avoids these concerns; and
2. this requires one to keep track only of a limited subset of the most-recent writes, TLBIs, and barriers to the translation tables, and hence we can make these models effectively *testable*, in on-line or off-line checkers.

Casemate's model is then a sound abstraction of the underlying Arm architecture, specialised to the usage discipline implicitly followed by systems code. At its core, Casemate keeps track of the *logical state* of page tables to ensure they are edited as expected, and in particular that there are no break-before-make violations, flagging when one occurs.

We explain Casemate informally in the rest of this section. We describe its implementation as an online checker in C

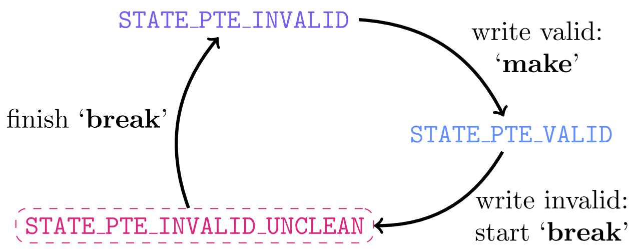

Figure 3. Per-PTE global TLB state automaton.

in (§4). We illustrate its ability to uncover bugs, applying it to pKVM (§5.3). We sketch its formalisation (which also generates an off-line checker) (§4.2) and its pen-and-paper soundness proof (§6). Finally, we describe our testing and coverage infrastructure (§5).

3.1 Tracking break-before-make

Looking at the events comprising an individual instance of break-before-make for an individual page table entry in isolation is already challenging. Figuring out which TLBIs are appropriate for which part of which method of invalidation (as described in §2.1.2) is error-prone, and spotting mistakes when updating with changes to which code is responsible for which synchronisation is doubly so.

Moreover, when looking at the code, the components of break-before-make are typically shared between multiple entries (to amortise the cost of these expensive operations), and spread over multiple functions; this leads to bugs like those we describe in §5.3. In a sense, what Casemate does is bridge this gap between the events and the code. 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. 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 break-before-make (BBM) by tracking for each PTE, where in the BBM protocol it logically is. We do this by keeping a shadow copy of the page table memory, where each location is annotated with a small automaton (whose high-level structure we sketch in Fig. 3), 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, which of the relevant TLB maintenance operations steps have been taken. This allows us, when an action pertaining to that PTE is performed (for example a write to it) to determine whether it is a potential BBM violation (either a true positive, or because our model is an over-approximation).

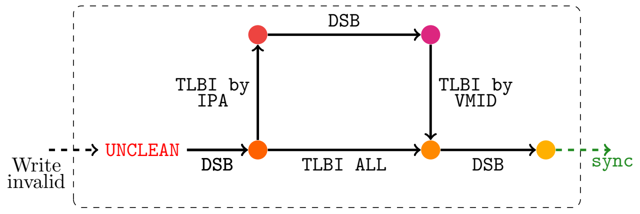

Figure 4. Per-PTE thread-local TLB state automaton.

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 BBM violation to attempt to write a conflicting entry to that location. This is a global state, but a thread-local sub-automaton tracks the local progress of any cleaning, shown as a dashed box in the figure (and expanded upon later).

For all threads to agree on this global state, all accesses to the page tables must be serialised; we discuss this in §3.3.

Tracking TLB maintenance. To simplify checking, we assume it is the responsibility of the thread that writes an invalid entry to clean that entry. Architecturally, this is not required, and it could be relaxed in the checker in future, but current systems code seems to conform to this. For each unclean PTE, we annotate it with a thread-local automaton, sketched in Fig 4: it tracks, 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 automata 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. 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. First, 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. Secondly, 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 sufficiently serialised (either e.g. by locks or hardware memory ordering) so that this cannot happen. Additionally, invalidating a table entry does not require all the children to be clean. Software must ensure

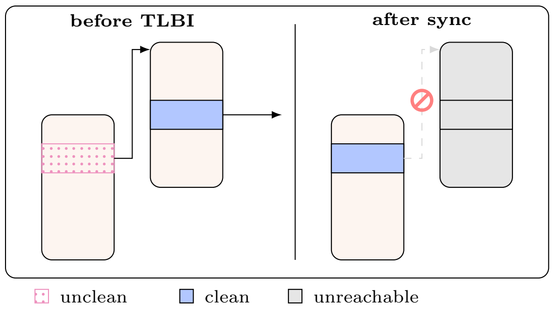

Figure 5. Cleaning an invalid parent.

that *all* the children are clean, before the invalidated table entry can be considered cleaned away. Finally, invalidating a table entry may make the children *unreachable* (see §3.2).

3.2 Reachability and ownership

Page table memory can be transient: a page can be used as part of a page table (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), it may be used elsewhere without triggering BBM violations, or become part of a page table again later on. Therefore, we logically partition memory in two:

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

We track the automata for locations which are reachable, and update this partition as locations become, or stop being, reachable. Tracking which locations are reachable is not simply a property of the current state of memory: children of a node remain visible to the MMU, up until each of those entries are individually cleaned. Fig. 5 demonstrates a case where this is relevant: cleaning an invalid parent makes the previously-reachable children now unreachable, as long as they were also clean to start with. Additionally, 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.

3.3 Ownership discipline

In theory, our model could track a per-thread 'view' of the page tables that can lag behind its latest state, in the style of the Promising semantics [34], and update it selectively on synchronisation with other threads. However, this would significantly increase the complexity of the model and require keeping a large chunk of the history, even though we believe software typically does not exhibit data races on page table locations. Therefore, we assign a single global automaton state per location, with atomic transitions on memory accesses, barriers, and TLB maintenance.

To ensure data race freedom over the entire page table tree, we enforce an ownership discipline: assigning to each page table an owner, and requiring only the owner performs any transitions over any locations in that page table. For pKVM, 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. 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 owns them, enforcing data race freedom.

We also need the different writes to PTEs and their parents/children to be ordered. For this purpose, we introduce the notion of *write authorisation* which ensures that the propagation to memory of the writes to the *same page table* are not reordered (as in Fig. 8a line 8). Here again, we enforce more ordering than necessary. The checker for write authorisation is tightly integrated with the locking discipline to take advantage of the ordering and consequent simplicity that it imposes. 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.

4 Monitoring and Offline Checking

We implement Casemate in two ways: as a monitor, which test by integrating in the build and execution of pKVM, and as a 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 monitor, which executes the model steps online, or a logging shim, which records the steps for the offline trace checker. This enables us to both extract lightweight traces to perform the check later (e.g. for regression testing) with only minimally invasive changes to the kernel as well as to perform efficient checking at runtime, even in long-running executions.

Casemate, whether online or offline, executes the steps of a concrete trace in the model, and reports violations to the user. Good error reporting is essential in developing and in using the checker, so we invested considerable effort into reporting e.g. diffs of abstract page table changes.

Importantly, the API is implemented in a way which integrates into, but does not disrupt the existing hypervisor, and requires only light instrumentation. Our example integration in pKVM requires just an additive patch to the Android kernel tree, adding the (basically) stand-alone library as a separate compilation unit, and calls to the checker initialisation and step functions guarded by `#ifdef`s at specific places in the pKVM page table management code.

```
static void kvm_clear_pte(kvm_pte_t *ptep) {
  WRITE_ONCE(*ptep, 0);
#ifdef CASEMATE
  casemate_model_step_write(
    WMO_plain, hyp_virt_to_phys(ptep), 0
  );
#endif /* CASEMATE */
}
```

Figure 6. Instrumented Android pKVM source for write-invalid to page table, `arch/arm64/kvm/hyp/pgtable.c`

```
struct sm_location {
 u64 phys_addr;
 u64 val;
 bool is_pte;
 struct pte_state state;
 u64 owner;
};
```

```
struct pte_state {
 enum pte_state_kind kind;
 union {
  valid_state valid_st;
  invalid_state invalid_st;
  unclean_state unclean_st;
 };
};
```

Figure 7. Checker C type for a single 64-bit memory location.

Concretely, we annotate each location in the pKVM source which reads or writes to 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 appropriate checker `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. 6.

4.1 Monitor

We implement the runtime Casemate monitor in kernel-compatible C. This means that it can be built with the normal Android Linux-kernel build process, without needing exotic tooling, and it avoids the need for any other language runtime. The pKVM hypervisor necessarily runs in an almost bare-metal environment, 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 checker. Using C also facilitates making the checker sufficiently high-performance, as we discuss below. All this said, C of course has downsides as a specification language, which we have to work around: it does not have a reasonable sublanguage of pure functional computation, it requires manual memory management in the checker implementation, and formally relating the C checker to the mathematical relaxed virtual memory semantics would be a major research project in itself. We contrast this with our experience building the Rocq version in §4.2.

4.1.1 The checker ghost state. We keep a global copy of the model state, which each transition mutably borrows. The state keeps a 'ghost' copy of the memory, where locations are not just the byte values but a structured datatype including the automata state (Fig 7). Additionally, the state tracks the visible page table roots, and any locks and their current state.

Concretely, this state is kept as a large array in memory, and accesses to it are locked to prevent corruption, which imposes significant extra synchronisation. Similarly, for the logging-only instrumentation (for offline checking §4.2), contention on the output channel imposes synchronisation. We discuss these performance implications in §5.1.

Error reporting. For user-friendliness, Casemate does not merely report that an error has happened, but identifies a specific problem. The central problems are BBM violations: either a valid-on-invalid-unclean, valid-on-valid, or release-unclean. In support of tracking BBM, we rely on a locking discipline but also partly check it, and thus report incorrect lock usage or incorrect synchronisation (transition without holding the lock, write without authorisation, etc.). Other errors we report include unaligned writes to page table entries and similar hard-to-spot mistakes. This mechanism identified the bug in pKVM we describe in §5.3.

4.1.2 The step functions. The checker step functions implement, in an efficient way, the transitions of the model. To take one transition as a concrete example, Fig. 8a contains an outline of the write transition in the checker. It (lines 4–5) reads the current state of the location being written to, from the model's ghost 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 transition has *authority* to write to that location, i.e. there is enough hardware synchronisation to ensure this write is ordered after the previous write to the same table in the same thread. Finally, (8–) it dispatches to specific transition functions depending on whether the current state is clean.

In that case, the transition continues with the code in Fig. 8b. 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 break-before-make, and if so, the model 'catches fire' with undefined behaviour (signalling an error to the user). Otherwise, on a write of invalid (lines 10–) the state is updated, moving the automata from valid to invalid-unclean, marking this CPU as the invalidator.

Finally, unwrapping the `requires_bbm` call in Fig. 8c, we see it is essentially an encoding of the Arm architecture requirements as defined by the Arm ARM [11, D8.16.1]. Roughly, break-before-make is required when the old and new descriptor are valid, when there is a change of OA or permissions. We overapproximate the definition, by only permitting idempotent writes or updates to the software-defined bits, marking all other writes as BBM violations.

```
void step_write(struct transition t) {
  u64 addr = t.write_data.phys_addr;
  u64 val = t.write_data.val;
  struct sm_location *loc = location(addr);
  loc->val = val;
  if (!is_reachable(loc)) return;
  assert_has_authority_to_write(loc, t.write_data.mo);
  switch (loc->state.kind) {
  case VALID:
    step_write_on_valid(loc, val, t.write_data.mo);
    break;
  ...
```

(a) Checker write step

```
static void step_write_on_valid(...) {
  u64 old = read_phys_pre(loc->phys_addr);
  if (is_desc_valid(val))
    if (! requires_bbm(loc, old, val))
      return;
    else
      CASEMATE_MODEL_CATCH_FIRE
        ("BBM valid->valid");

  loc->state.kind = INVALID_UNCLEAN;
  loc->state.unclean_st.invalidator_tid = cpu_id();
  loc->state.unclean_st.old_valid_desc = old;
  loc->state.unclean_st.local_st = LIS_unguarded;
}
```

(b) Checker write transition on a valid PTE

```
static bool requires_bbm(...) {
 struct descriptor before = deconstruct_pte(loc, old);
 struct descriptor after = deconstruct_pte(loc, val);
 /* see ARM DDI 0487 J.a D8.14.1 */
 if (before.kind == PTE_KIND_INVALID
      after.kind == PTE_KIND_INVALID)
  return false;
 if (before.kind == PTE_KIND_TABLE
      after.kind == PTE_KIND_TABLE)
  return true;
 if (before.map_data.oa_region.range_size
     != after.map_data.oa_region.range_size)
  return true;
 if (! is_only_update_to_sw_bits(before, after))
  return true;
 return false;
}
```

(c) Break-before-make check

Figure 8. Excerpts of the C source of the online checker

4.2 Rocq formalisation and offline checking

To establish assurance that the Casemate discipline is sound with respect to the Arm-A architecture specification, we formalise the check it does and prove soundness (§6). The formalisation is in Rocq. From it, we also extract a practical *offline* trace checker, as an OCaml command-line tool. This lets us cross-check the formalised version and the online C checker, boosting assurance in both; it also lets us explore the trade-offs of online and offline checking. Our proof is a paper proof, and it and the formalisation were very useful in developing Casemate. In principle the proof could also be mechanised, but, because Casemate is a bug-finding tool, and any bugs found must be confirmed by inspection, very high assurance in the tool is not required.

In total, the definition of the automata and the step functions that drive it take 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 C version has to do some memory management (262 lines) and contains some printing infrastructure (421 lines), whereas the Rocq version has no memory management, and its printing is done by 208 lines of OCaml. The Rocq version is simpler to write, as it (and its surrounding OCaml) is in a high-level language, but it is still quite lengthy, reflecting the underlying complexity of the discipline.

5 Evaluation

We evaluate the effectiveness of Casemate, by:

- demonstrating that it can detect synthetically injected bugs, of a variety of different kinds; and
- systematically annotating pKVM's TLB synchronisation code and uncovering a subtle security-critical TLB synchronisation bug.

We do all this by running a custom-built battery of high-coverage tests over pKVM, designed to exercise the corner cases of pKVM's API and pagetable management. We start by motivating this test suite, and demonstrating it is complete and performant enough for our needs. We then describe how we use it in conjunction with Casemate to evaluate the 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 a collection of tests over pKVM, exercising its page table management. These tests include booting the Android kernel, 41 hand-written, self-contained and narrowly focused test case, and random walks over hypercall invocations, which we leave running for up to 48 hours. The testing and coverage infrastructure was developed for multiple purposes, and is reported elsewhere [10].

Coverage. We patch pKVM with coverage metrics, and use them to gauge how effective our test suite is. 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 TLB management, but called automatically from inside pKVM for protected VMs). We cover all of the pKVM page table walker code

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

Figure 9. Test-suite running times with the default kernel, online monitor, offline checker, and pure event tracing — median over 11 runs.

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. This creates a significant challenge: whereas typically checkers can take as long as needed, here, to actually exercise the scenarios we are interested in, we need to achieve reasonable performance.

Accordingly, our implementation uses several optimisations to achieve adequate performance: by recognising page tables always come in page-size granularities and are often contiguously allocated, we use a custom range-map implementation to efficiently store and look up whole pages of ghost model state at once; and we cache, for each PTE, the decoded entry containing the parent, level, and stage. This reduces the number of software page table walks we need.

pKVM developers routinely use QEMU in their normal development and testing, making it the appropriate setting in which to evaluate performance. With runtime checking enabled, our test suite, which makes around 750 hypercalls, takes 422 seconds (running the tests sequentially on an Intel Xeon Gold 6240, Fig. 9). 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.

Offline checker performance. Traces can grow quite large for real systems. For example, pKVM generates roughly a million events during initialisation, and 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 tracing infrastructure to extract traces real time. The extracted offline checker then takes 466 seconds to check the trace, comparable to the online monitor (Fig. 9).

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:

- 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 of memory barriers or substitution of 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 possible 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 Android14 pKVM kernel does not hit it — as Casemate is a dynamic check it can miss executions which do not occur in practice.

5.3 Uncovering a real bug in pKVM with Casemate

Casemate is strict enough to discover bugs caused by insufficient synchronisation, but liberal enough to be applicable to production systems code, admitting its intended synchronisation discipline. We demonstrate this by applying Casemate to traces of pKVM, and in doing so discover a TLB synchronisation bug, that transiently exposed memory that should have been private. Inspection of the pKVM code during development found a further TLB synchronisation bug during pKVM's initialisation, which was also reported and fixed.

We now walk through the security-critical TLB synchronisation bug 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.

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. 10a 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.

```c
static bool stage2_try_break_pte(...) {
 ...
 if (!stage2_try_set_pte(ctx, 0))
  return false;
 ...
 if (kvm_pte_table(ctx->old, ...)) {
  // ... in __kvm_tlb_flush_vmid
  struct tlb_inv_context tlbcxt;
  enter_vmid_context(mmu, &tlbcxt, false);
  __tlbi(vmalls12e1is);
  dsb(ish);
  isb();
  exit_vmid_context(&tlbcxt);
 } else if (kvm_pte_valid(ctx->old)) {
 ...
}
```

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

```c
static void enter_vmid_context(...) {
 vcpu = host_ctxt->__hyp_running_vcpu;
 /*
  * If we're already in the desired context,
  * then there's nothing to do.
  */
 if (vcpu) {
   ...
 } else {
  /* We're in host context */
  if (mmu == host_s2_mmu)
   return;
 }
 dsb(ish);
 ...
 if (vcpu)
  __load_host_stage2();
 else ...
 isb();
}
```

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

Figure 10. The pKVM bug identified by Casemate

The key part of the break is the TLB maintenance to invalidate any cached entry of the old PTE for that VMID, this 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 requires (at least) a DSB, and so Android re-uses the synchronisation in one for the other, in `enter_vmid_context` (Fig. 10b). So the code must do a sequence of:

dsb set-VTTBR isb tlbi dsb reset-VTTBR isb

The VMID-switch function has an optimisation: if the right VMID is already loaded, there is no need to do the switch. However, the synchronisation had dual responsibility, not just for the VMID-switch but also for the break-before-make sequence which uses it. 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 was reported and fixed before the release of Android15.

6 Soundness

To put Casemate on a solid basis, we relate it by paper proof to the axiomatic VMSA model of [51], slightly extended:

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.

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 some break-before-make 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 break-before-make, which it purposefully over-approximates. Concretely, it internalises the locking discipline for page tables of Casemate, and it 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 ticket lock implementation of pKVM satisfies. (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, 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 break-before-make in situ.

Left refinement: VMSA ⊑ 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 ⊑ 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 locking 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 break-before-make 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. [51] only considers the simplest, most expensive invalidation method, using a DSB; TLBI all; DSB. In practice, systems code like pKVM, use 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 [17, 32, 33, 41], 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, 38], 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 [19]. 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 [23, 24], and is multicore, but has a static partition of memory, which means they avoid much of the complexity of virtual memory.

seL4 [36, 37] relies on abstractions over an ad-hoc formalisation of ARMv7 virtual memory [55–58] without relaxed memory. In fact, it was designed for a sequential setting, and has so far only been ported to sequential consistency [49].

CertiKOS [29, 30, 50] 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 like pKVM. Moreover, it essentially assumes sequentially consistent concurrency.

SeKVM [39, 40] 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 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 [51] that translation reads are, in several respects, substantially weaker. The requirements that they make of page table manipulations to obtain a manageable state (via data-race-freedom) are, as expected, broadly similar to ours.

8 Conclusion and Future Work

The Casemate runtime monitoring of pKVM's management of the Arm VMSA architecture identified a significant bug in pKVM. Could it be applied elsewhere, and could it be extended to other architecture-related code disciplines?

We target pKVM as an example, but the machinery and model is not specialised to that particular hypervisor. Casemate should be applicable to other Arm-based hypervisors and OS kernels, including Linux, non-protected KVM, and other hypervisors for the Android AVF framework (which is designed so that the hypervisor can be swapped out [15, 16, 27]). We believe this would largely be a matter just of adding the appropriate instrumentation calls and linking to Casemate, though one would also need testing infrastructure, and one might have to modify or extend the Casemate discipline. Developers associated with other major industry hypervisors have expressed interest in applying the tool.

The current version of Casemate strikes a balance between expressivity and simplicity, but there are several avenues for extending tracking, to instruction/data cache management and to more of the VMSA. We could track cache maintenance requirements associated with cacheable or executable mappings (not of the *pagetables*); we could refine our improved break-before-make predicate to be more complete, and to account for more invalidation methods; and we could cover requirements around access flags and dirty bits.

Casemate is designed bottom-up, trying to meet the complexity of systems code from below. A different approach would be to approach it from above: starting from the axiomatic memory model, and incrementalising it so that it does not suffer from the state explosion that naïve state exploration leads to. One could also start from a more transactional property such as wDRF, proving soundness w.r.t the underlying architecture, and building runtime-checkable models above it.

Finally, Casemate is designed for testing, but it also an important step towards foundational verification of systems code, as an intermediate model that is firmly grounded on the Arm architecture semantics but abstracts from much of its relaxed-VMSA complexity. This is analogous to DRF-SC properties, which let one reason in an SC model about race-free code: Casemate is a checking model, which explicitly reports when its assumptions are violated, rather than silently behave incorrectly. The general idea would be to take a verification that assumes sequential consistency, and replace the corresponding map from location to value by our automata for page table locations.

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.

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] Anonymous. 2025. Ghost in the Android Shell: Pragmatic Test-oracle Specification of a Production Hypervisor. Under submission.

[11] 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..

[12] 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.

[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] Elliot Berman and Prakruthi Deepak Heragu. 2023. Adding Third-Party Hypervisor to Android Virtualization Framework. presented at the Linux Plumbers Conference Android Microconference.

[16] Elliot Berman and Prakruthi Deepak Heragu. 2024. Gunyah Hypervisor Software - Supporting Protected VMs in Android Virtualization Framework. https://www.qualcomm.com/developer/blog/2024/01/gunyah-hypervisor-software-supporting-protected-vms-android-virtualization-framework

[17] 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

[18] 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.

[19] 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

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

[21] 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

[22] 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

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

[24] 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

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

[26] 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

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

[28] 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

[29] 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

[30] 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

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

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

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

[34] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 175–189. doi:10.1145/3009837.3009850

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

[36] 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

[37] 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

[38] 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

[39] 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

[40] 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

[41] 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

[42] 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

[43] 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.

[44] 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

[45] 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

[46] 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

[47] 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

[48] 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).

[49] Thomas Sewell. 2018. Verifying seL4 towards Concurrency. presented at DeepSpec 2018.

[50] 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

[51] 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.*

[52] 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

[53] 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.*

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

[55] 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

[56] 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

[57] 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

[58] 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

[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