Why write address translation OS code yourself when you can synthesize it?

Reto Achermann
University of British Columbia
Vancouver, Canada

Ilias Karimalis
University of British Columbia
Vancouver, Canada

Margo Seltzer
University of British Columbia
Vancouver, Canada

ABSTRACT

Address translation hardware is at the cornerstone of modern computer systems. It provides a wide range of security-relevant features and abstractions such as memory partitioning, address space isolation, and virtual memory. Hardware designers have developed different memory protection schemes with varying features and means of configuration.

Correct configuration is mission-critical for a system's integrity. It is the operating system's task to safely and securely manage and configure the memory hardware of a compute platform – a task that operating systems developers must repeat for every new memory hardware unit.

We present a new approach that frees the OS programmer from writing system code to set up and configure translation hardware. We leverage software synthesis to automatically generate correct systems code that interfaces with translation hardware to create or modify memory mappings from a high-level, behavioral specification.

By synthesizing correct, low-level systems code from a high-level specification we make it easier to port operating systems and facilitate incorporating accelerators into existing systems. Moreover, we believe that our system can generate actual and simulated hardware components enabling research in new memory translation and protection schemes.

CCS CONCEPTS

• **Software and its engineering** → **Virtual memory**; **Automatic programming**; *Functionality*.

ACM Reference Format:

Reto Achermann, Ilias Karimalis, and Margo Seltzer. 2023. Why write address translation OS code yourself when you can synthesize it?. In *Workshop on Hot Topics in Operating Systems (HOTOS '23), June 22–24, 2023, Providence, RI, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3593856.3595895

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

HOTOS '23, June 22–24, 2023, Providence, RI, USA

© 2023 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0195-5/23/06.
https://doi.org/10.1145/3593856.3595895

1 INTRODUCTION

Enforcing isolation between two tenants, e.g., virtual machines, processes, even threads, is a critical function of operating systems and hypervisors. Each process or virtual machine runs in its own address space, which provides isolation and abstracts the underlying physical memory. Virtualizing memory not only provides isolation but also enables system software to optimize memory usage, oversubscribe memory through demand-paging or swapping, and support features such as memory-mapped files or relocation.

These features of memory virtualization and protection are enabled and enforced by specific hardware components, such as memory protection units (MPUs), memory management units (MMUs), and other address translation hardware. The features provided by those hardware components can be classified into two categories: *translation* and *protection*.

Translation defines how the address of a memory request (e.g., load or store) is transformed. The translation fails if the address is not valid (i.e., the translation is undefined). This translation can be configurable (e.g., the virtual-to-physical address translation of the MMU) or static (e.g., bus-remappings on a system-on-chip (SoC)).

Protection checks whether a given memory access *is allowed*, but it does not change the address. This includes checking if the protection attributes match the access mode (e.g., read/write), privilege level (e.g., user/supervisor mode), or the source (e.g., a certain device) of the memory access. For example, hardware firewalls might allow memory access only from certain devices. Similarly to translation, protection can be configurable (e.g., the permission bits of a page-table) or fixed (e.g., KSeg0 of the MIPS TLB [12]).

Contemporary address translation hardware combines address translation and protection. We call this combination *remapping*. For example, the x86 MMU translates a virtual address to the corresponding physical address using the page table, and then, if there is a matching valid entry, it checks the permission bits and matches them against the mode of the memory access. Moreover, hardware extensions might perform further permission checks (e.g., Intel's MPK).

Today, there exists a wide variety of MMUs and MPUs providing a heterogeneous set of features and capabilities and offering configurable protection and translation of memory accesses. To ensure system integrity and correct operation,

the OS must correctly configure all MMUs and MPUs of the system. Thus, the system software developer needs to understand the features offered by the translation hardware then *manually* write correct software that interfaces with the address translation hardware of the compute platform.

This tedious and error-prone task is not a one-off effort: different systems use a variety of MMUs and MPUs providing a distinct set of features and different ways to configure them. System software developers spend time and effort repeating this similar task for each MMU and MPU. Any mistake in the code can lead to severe consequences in terms of system security and correct operation, ranging from incorrect protection enforcement to data corruption. For example, bugs in the IOMMU/System MMU configuration code [14–17] or the Linux memory subsystem [11] (ignoring holes in huge pages (CVE-2017-16994), miscalculation of the number of affected pages (CVE-2014-3601), and too permissive access rights to data pages (CVE-2014-9888)) have led to security vulnerabilities.

We propose to free OS developers from writing low-level system code interfacing with translation hardware. Instead, developers or hardware vendors write a specification of the translation hardware that describes *how* an MMU or MPU performs address translation and protection. Our synthesis toolchain then automatically converts the specification into correct low-level system code for configuring the memory hardware of a platform using software synthesis. Specifically, we synthesize functions to set up new mappings (map), modify permissions (protect), and remove mappings (unmap).

The focus on MMUs and MPUs is important, as they are used to isolate memory from processors and devices. Moreover, translation hardware, as opposed to devices in general, has a well-defined scope and is amenable to breaking down the problem into small, composable pieces that make MMUs and MPUs a well-suited target for software synthesis. We believe that the insights from this work can be applied to device drivers in general, but we consider that a separate area of investigation.

We present a methodology that makes the synthesis of the low-level system code tractable by decomposing translation hardware into a combination of *basic building blocks* leveraging the divide-and-conquer principle. We further describe techniques to reduce the search space of candidate programs making the software synthesis approach applicable in the development of correct operating systems. Finally, we propose a new specification language that allows developers to write specifications of MMUs and MPUs, or even have hardware vendors provide them as part of their technical manuals as Arm did with their system level architecture specification [18, 19].

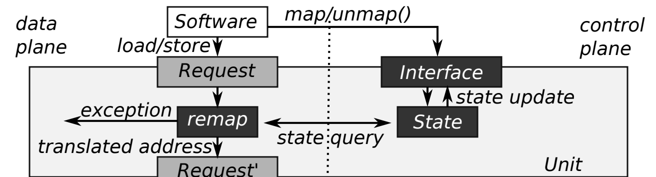

Figure 1: Illustration of the translation unit model.

2 METHODOLOGY: ABSTRACTING MEMORY TRANSLATION HARDWARE

The landscape of MMUs and MPUs is heterogeneous in terms of features and means of configuration. One of the insights of this work is that we can *decompose* complex memory translation and protection hardware into a small number of basic building blocks we call *units*, each with well-defined semantics. We can even reuse the same unit specifications to express another MMU or MPU by combining and instantiating them in different ways. We begin by describing the methodology behind this decomposition.

2.1 Translation Unit Model

We use the term *translation unit* to refer to any hardware component that can translate or filter memory accesses. Abstractly, the translation unit operates similarly to a network switch: it receives a memory access request (e.g., the processor executes a load or store instruction), then it transforms the request by translating its address based on the state (e.g., page table walk or TLB lookup), and finally it forwards the request with the translated address to the next hardware component (e.g., the memory controller, or the system bus), or raises an exception if the remapping fails due to a permission violation or an invalid address.

The translation unit model consists of three main components (dark boxes in Figure 1) that can be further categorized into data plane and control plane components.

Remap Function. The data plane of the translation units remaps memory requests (e.g., loads and stores issued by the processor or devices of a system) or it raises an exception if remap fails. Recall, remapping consist of two steps: performing *address translation* and *checking of access permission*. This division becomes important later when we formulate the synthesis problem of finding the correct system software code that configures the unit. Thus, the `remap` function succeeds only if the translation produces a valid address *and* the access mode of the memory access is allowed by the permission check, otherwise remapping fails.

Configuration State. The way a translation unit remaps memory accesses depends on its *state*. Thus, the state includes all relevant bits that define the remapping behavior,

including any registers and in-memory data structures. Abstractly, the state of a unit is a collection of bits and, to some extent, can be viewed as the private data members of a class. For example, when using MPK on x86_64 platforms, remapping succeeds only if both the page table entries and the memory protection key registers are set up correctly.

Control Plane Interface. The *control interface* defines the OS-visible interface that changes the state of the translation unit and thus its remapping behavior. System software changes the state of the unit by reading or writing memory locations or registers or executing special instructions. The interface corresponds to the public API of the translation unit and therefore the *grammar* of the possible configuration programs (see Section 4.2).

2.2 Unit (de)composition

Directly expressing translation hardware using the unit model above can be challenging. Instead, we logically *decompose* complex translation hardware into a combination of simpler units that we call *basic building blocks*. We use the x86 page directory (PDir) as our running example.

Three kinds of units. In our investigation of existing MMUs and MPUs, we found that there are parts where the translation behavior is fixed and other parts where it is configurable. In addition, some MMUs and MPUs offer alternative translation behavior depending on their state. This has led us to the insight that translation hardware can be expressed as combinations of three basic building block types:

1. *Configurable* units have state, so their remapping behavior can be changed by the OS (e.g., a single PDir entry).
2. *StaticMap* units have a defined, static remapping behavior (e.g., the static selection of the PDir entry depending on the virtual address).
3. *Enum* units enumerate a list of configurable units that can translate the virtual address (e.g., a PDir entry that either references a page table or maps a large page).

Decomposition. Figure 2 shows the decomposition of the page directory level of the x86_64 linear-address translation using 4-level paging into a combination of static and configurable units. The PDir has 512 entries, each of which can either reference a page table or a large page. The virtual address of the memory access directly identifies the entry used for translation, i.e., there is a static mapping from virtual address to the PDir entry. The PDir entry itself is configured by writing a certain bit pattern to it.

2.3 Type Hierarchy

Each defined unit corresponds to a type like a class or struct definition in a programming language. Using the type references of the unit, we can extract a type hierarchy. This lets

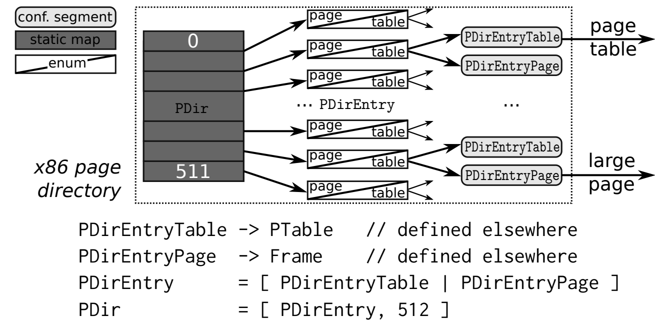

Figure 2: Decomposition of the page directory level of the x86 linear address translation with 4-level paging.

us construct hierarchical translation structures such as the multi-level radix trees used by page tables.

3 SPECIFICATION LANGUAGE

Developers specify translation units using a rust-like domain-specific language (DSL) that closely follows the concepts outlined in the previous section. The core of the language is the syntax to express each of the three unit kinds, each with its state and control interface as outlined in Section 2. We use methods to define behavior and specify constraints.

Listing 1 shows the simplified specification of the example of Figure 2. For brevity, we left out the `protect` and `unmap` operations and parts of the state and control interface. The page directory (`PDir`, line 44) is a fixed-sized array of 512 entries specified as a **`staticmap`** using the list comprehension notation. A page directory entry (`PDirEntry`, 39) is an **`enum`** listing the two alternative interpretations that either reference a page table or a large page. The two **`segment`** units `PDirEntryTable` line 2 and `PDirEntryPage` line 7 specify the configurable entries. Note that the definition of `PDirEntryTable` reuses the common `PTableDescriptor` specification through unit derivation.

State Definition. Conceptually, the state of a unit is a sequence of bits. The state definition assigns names to groups of these bits by separating the state into one or more *fields* that either reside in memory (**`mem`**) or in registers (**`reg`**). Each field can be further divided into *bit slices* allowing the developer to refer to specific bits. Line 8 shows the state definition of a page directory entry as an 8-byte memory location with three bit-slices. One can view the state definition as declaring private data members of a class.

Control Interface Definition. Similar to the state definition, the control interface definition has fields with optional bit slices. Each field can be written to or read from atomically, which triggers the state updates as specified in the `WriteActions` (18) and `ReadActions` (19). The action blocks contain one or more assignments to state or control

```
// derive from the PTableDescriptor
segment PDirEntryTable(base: addr) : PTableDescriptor {
  synth fn map(va:vaddr, sz:size, flgs:flags, pa: PTable)
    requires // removed for space
}

segment PDirEntryPage(base: addr) {
  state = StateDef(base: addr) {
    mem entry [ base, 0, 8 ] {
       0 ..  1 present,
       1 ..  2 rw,        // bits 2..21 removed for space
      21 .. 48 address,  // bits 49..64 removed for space
  } };

  interface = InterfaceDef(base: addr) {
    mem entry [ base, 0, 8 ] {
      Layout { /* same as state */ }
      WriteActions { interface.entry -> state.entry; }
      ReadActions { state.entry -> interface.entry; }
  } };

  fn matchflags(flgs : flags) -> bool
    requires state.entry.present==1 && state.entry.ps==1
  {
    flags.writable ==> state.entry.rw
  }

  fn translate(va: vaddr) -> paddr
    requires state.entry.present==1 && state.entry.ps==1
  {
    va + (state.entry.address << LARGE_PAGE_BITS)
  }

  synth fn map(va:vaddr, sz:size, flgs:flags, pa:paddr)
      requires va == 0 && sz == LARGE_PAGE_SIZE
      requires pa & (LARGE_PAGE_SIZE - 1) == 0
} // end of PDirEntryPage

enum PDirEntry(base: addr) {
  PDirEntryTable(base),
  PDirEntryPage(base)
}

staticmap PDir(base : addr) {
  mapdef = [ PDirEntry(base + i * 8) for i in 0..512 ];
}
```

Listing 1: Simplified x86 Page Directory Specification

interface fields or their bit slices. In the example, writing the interface field entry copies its values into the entry field of the state. The control interface can be seen as a public API consisting of getter/setter methods for the interface fields.

Methods. The specification uses methods to define translation behavior. The method `remap()` is separated into the methods `translate()` and `matchflags()`. Methods may have `requires` clauses that define pre-conditions that must be satisfied. The `translate()` method on line 28 requires the present-bit and the `PS`-bit to be 1. Method body and pre-conditions may refer to state, parameters or constants.

Synth Methods. Three special methods `map()`, `protect()` and `unmap()` (the latter two are omitted for brevity) provide the synthesis targets for which we want to find a low-level implementation. They are marked with the `synth` keyword and do not have a body. In the example, `map()` requires the physical address to be page aligned, the virtual address to be zero, and the size to match the page size.

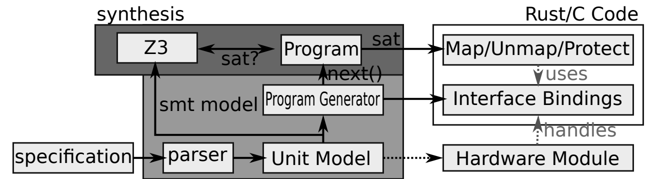

Figure 3: High-level Synthesis Overview

4 SYNTHESIZING CONFIGURATION PROGRAMS

The synthesis procedure (Figure 3) takes the unit specifications as input and searches for an implementation for each of the synthesis targets (i.e., `map()`, `protect()` and `unmap()`) that satisfies the operation's post-condition that are implicitly defined by the model.

We present the synthesis procedure in three parts: 1) define the synthesis problem we need to solve, 2) present the semantics of synthesis targets we need to implement, and 3) describe the search space reduction techniques we employ.

4.1 The Synthesis Problem

With the model described above, we can now formulate the problem of synthesizing the sequence of state transitions that achieve a certain remapping behavior as follows:

> *"Given a unit model with a state (S), interface (I) and remap function (R), what is the sequence of interface invocations such that remap produces the desired outcome."*

The desired outcome is defined as the post-condition of the synthesis targets `map()`, `protect()` or `unmap()` respectively. In the case of `map()`, we ensure that the unit remaps addresses within the given input address range $[va, va+size)$ that match the permission flags to the corresponding output address range $[pa, pa+size)$, expressed as the following predicate as its post-condition:

$$\forall a :: 0 \leq a < size \Rightarrow remap\ (va + a)\ flags == pa + a$$

4.2 Constructing Candidate Programs

The interface definition of the unit defines the *grammar* of the configuration program we want to synthesize. Intuitively, the fields of the interface define the possible API calls and the values of their bit slices define its arguments. We draw possible values for the bit slices from the parameters to the synthesized function, constant values, or expressions such

as bitwise operations (e.g., shift, and, or, ...) or arithmetic operations (e.g., +, -, ...).

4.3 Synthesis Step

During the synthesis step, the toolchain constructs candidate programs and asks the Z3 SMT solver [9] whether the program satisfies the post-condition of the synthesis targets (e.g., `map()`) that we want to synthesize. The toolchain transforms the state and interface definitions and the program to verify into SMT formulas. For each program, the solver needs to check whether the post-condition holds for all possible input values and translation unit states.

4.4 Search Space Reductions

The space of candidate programs grows exponentially in the size of the state and interface of the unit. We need to find the correct value for each bit in the state. Each field of the interface conceptually corresponds to an API call that can change many bits in the state (`ReadActions`, `WriteActions`). For each bit slice we have to consider many possible expressions (Section 4.2). This quickly results in a search space in the order of billions of programs even for a simple unit. We reduce the search space by dividing the synthesis problem into many, much smaller sub-problems and then find a global solution by combining the results from the synthesis step of the sub-problems. This divide-and-conquer approach also allows us to synthesize the sub-problems in parallel.

1. Reduction through Program Construction. We leverage the atomicity of interface invocations to reduce the search space by constructing programs that have a "read-modify-write" structure, i.e., reading a value from the interface, updating it, and then writing it back, Thus, the program for which we are searching is one-or-more RMW sequences each operating on a single interface field. An RMW sequence is an optional read of the current value, construction of the updated value, followed by a write-back of the updated value to the field.

2. Reduction through Unit Decomposition. By design, the unit decomposition methodology reduces the search space by breaking down a more complex unit (e.g., a full PDir of 32k bits) into much smaller units (e.g., one PDir entry of 64 bits). Moreover, only configurable units have state and thus require synthesis.

3. Reduction through Remap Decomposition. We leverage the distinction between the memory address translation and checking of the access permissions to reduce the search space even further. For example, the RW-bit is irrelevant for address translation, and the address field is irrelevant for checking the access mode. Thus we can find programs for the two functions independently and then combine them into a single program that satisfies both functions.

4.5 Implementation and Evaluation

We implement a prototype of the toolchain in Rust and use the Z3 SMT solver v4.8.12 for checking the satisfiability of the synthesized programs. The toolchain leverages the independence of the goals by running the synthesis in parallel and combining the results. We have completely independent synthesis tasks and therefore we can run as many as we have cores in parallel. The prototype supports emitting the synthesized programs in C and Rust. We are working on making the code generation more flexible by using an OS environment specification.

Currently, we have specifications for existing memory hardware (e.g., 32-bit and 64-bit x86 page tables and ARMv8 translation tables), the Xeon Phi system memory page tables, and x86 segmentation. Moreover, we have descriptions of hypothetical memory translation hardware (e.g., direct segments). While not yet feature complete, the prototype is capable of synthesizing configuration programs for the page table of Listing 1 in 540ms and all four levels in 2400ms on an Intel Xeon W-2275. The full specification is roughly 370 lines. In comparison, verifying the x86_64 paging code consisted of 600 lines of specification and 5000 lines of proof with a verification time of 48 seconds [7]. Note, this is just for the x86_64 page tables, and the efforts would need to be repeated for other translation hardware.

5 USE CASES

The availability of the specification language and the synthesis tool enables a variety of use cases in system software that makes the life of systems programmers easier.

Machine-Readable Specification. Conventionally, the semantics of translation hardware are described in prose that often leads to ambiguity. We can use a machine-readable description of the translation semantics as ground truth, especially when the specification is provided by the hardware vendor as part of their documentation.

System Software Verification. The specification language and its conversion into SMT queries can provide the foundation for system software verification by emitting a model for tools such as Isabelle/HOL.

Porting. Our toolchain makes porting operating systems to new hardware platforms easier. We envision that operating systems developers write a small OS-specific spec or abstraction layer that describes the environment the OS provides. This frees OS developers from writing the low-level systems code interfacing with memory translation hardware. As a bonus, developers can simply reuse their OS specifications and the translation specifications written by other users or hardware vendors.

6 THE ROAD AHEAD

Our prototype shows promising results. However, there are still many open questions and challenges.

Cache operations and Barriers. Interfacing with hardware, especially memory translation hardware, is always a bit of a challenge to get right, especially on platforms with weak memory models such as Arm or Power [5]. Memory accesses can be re-ordered, making a later write visible before an earlier one. Moreover, writes end up in write buffers or caches and are not visible to the translation hardware until the cache is written back. We are currently working on integrating support to express caches and cache management operations and barriers into the model.

Hardware Generation. Finally, we imagine that the specification can also be used for generating the hardware itself, because our specification language includes the full address translation semantics. Our prototype is already capable of generating a component for the Arm FastModels simulator, and we are looking into generating code for FPGAs. This enables us to experiment with new translation schemes that have precisely the features we desire.

Customizable Remap Behavior. Translation hardware is complex offering many additional security features beyond simple address translation with read, write and execute permissions. Features such as Arm's pointer authentication or Intel MPK allow additional checks before a remap succeeds. We are working on extending our prototype to support a customizable remap specification that allows developers to express more complex behaviors.

TLB Management. To speed up the translation process, frequently translated memory addresses are cached in TLBs. The operating system needs to ensure consistency of the TLB when updating the translation configuration. This is a non-trivial, security-relevant task, as it may involve a distributed shootdown protocol. While we can specify a TLB, we do not have support for specifying distributed TLB management protocols yet.

Applicability to other OS components. By using the divide-and-conquer strategy, we managed to break down the search space into small, tractable sub-problems that enabled efficient synthesis. While this has worked well for the problem at hand, it may also apply to other parts of the operating system such as general device drivers, memory allocators, or schedulers. We see this to be relevant for customizable embedded operating systems with resource constraints.

7 RELATED WORK

Generating translation configuration has been done before [1]. However, the focus of prior work was handling complex memory hierarchies by leveraging the decoding net model [3, 4] and the MMU configuration was static and generated at compile time. Termite [22, 23] used a game-theoretic approach to generate device drivers, but they do not support in-memory data structures. The Dingo framework [21] provides a language for specifying driver protocols to reduce protocol violations and concurrency-related bugs when interfacing with hardware. Neither Termite nor Dingo supports the generation of the device itself.

Aquarium's hardware description language, Cassiopeia [10], and its corresponding toolchain operates on the operational semantics of assembly instructions requiring a partial machine model to be implemented. This approach is not amenable t our divide-and-conquer approach and thus synthesis is significantly slower.

Chen et al. [8] support device driver generation for multiple operating systems and driver architectures. They show that by using their tool they can drastically reduce development time by achieving around 70% generation based on device classes and features. However, their approach requires defining register programming sequences, whereas our approach synthesizes this sequence.

Capability-based operating systems, e.g., Barrelfish [6] and seL4 [13] and academic proposals such as as mmapx [2], provide user-space with more control over address space management by effectively separating writing the bits in a page table from constructing the multi-level page table. Our work ties into this by generating code for the capability invocation handlers and the user-space library managing the page tables.

The Arm Specification Language (ASL) [18] provides a machine-readable description of the Arm architecture with its description providing the basis for documentation generation, simulators and test suites [19]. Reid further wrote an Arm instruction interpreter [20] based on ASL. This shows the applicability of a specification-based hardware model.

8 CONCLUSION

We presented a methodology that breaks down translation hardware into a composition of small building blocks that describe its translation behavior. We then outlined how we can efficiently synthesize low-level systems code for these building blocks independently and then combine the synthesized programs to obtain the final operating system implementation of the translation hardware driver.

ACKNOWLEDGMENTS

This work has been supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Huawei Technologies Co., Ltd.

REFERENCES

[1] Reto Achermann, David Cock, Roni Haecki, Nora Hossle, Lukas Humbel, Timothy Roscoe, and Daniel Schwyn. 2021. Generating Correct Initial Page Tables from Formal Hardware Descriptions. In *Proceedings of the 11th Workshop on Programming Languages and Operating Systems* (Virtual Event, Germany, 2021) *(PLOS '21)*. Association for Computing Machinery, New York, NY, USA, 69–75. https://doi.org/10.1145/3477113.3487270

[2] Reto Achermann, David Cock, Roni Haecki, Nora Hossle, Lukas Humbel, Timothy Roscoe, and Daniel Schwyn. 2021. Mmapx: Uniform Memory Protection in a Heterogeneous World. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Ann Arbor, Michigan, 2021) *(HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 159–166. https://doi.org/10.1145/3458336.3465273

[3] Reto Achermann, Lukas Humbel, David Cock, and Timothy Roscoe. 2017. Formalizing Memory Accesses and Interrupts. In *Proceedings 2nd Workshop on Models for Formal Analysis of Real Systems, MARS at ETAPS 2017, Uppsala, Sweden, 29th April 2017.* (2017) *(MARS'17)*. 66–116. https://doi.org/10.4204/EPTCS.244.4

[4] Reto Achermann, Lukas Humbel, David Cock, and Timothy Roscoe. 2018. Physical Addressing on Real Hardware in Isabelle/HOL. In *Proceedings of the 9th International Conference on Interactive Theorem Proving, 2018, Held as Part of the Federated Logic Conference, FloC 2018* (Oxford, UK, 2018) *(ITP'18)*. 1–19. https://doi.org/10.1007/978-3-319-94821-8_1

[5] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. 2009. The Semantics of Power and ARM Multiprocessor Machine Code. In *Proceedings of the 4th Workshop on Declarative Aspects of Multicore Programming* (Savannah, GA, USA). ACM, New York, NY, USA. https://doi.org/10.1145/1481839.1481842

[6] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 29–44. https://doi.org/10.1145/1629575.1629579

[7] Matthias Brun. 2022. *Verified Paging for x86-64 in Rust.* Master's thesis. ETH Zurich.

[8] H. Chen, G. Godet-Bar, F. Rousseau, and F. Petrot. 2014. Device driver generation targeting multiple operating systems using a model-driven methodology. In *2014 25nd IEEE International Symposium on Rapid System Prototyping*. 30–36. https://doi.org/10.1109/RSP.2014.6966689

[9] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *2008 Tools and Algorithms for Construction and Analysis of Systems*. Springer, Berlin, Heidelberg, 337–340. https://www.microsoft.com/en-us/research/publication/z3-an-efficient-smt-solver/

[10] David A. Holland, Jingmei Hu, Eric Lu, Ming Kawaguchi, Stephen Chong, and Margo I. Seltzer. 2019. *Aquarium: Cassiopea and Alewife Languages.* Technical Report. arXiv:1908.00093 [cs.PL] https://arxiv.org/abs/1908.00093

[11] Jian Huang, Moinuddin K. Qureshi, and Karsten Schwan. 2016. An Evolutionary Study of Linux Memory Management for Fun and Profit. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference* (Denver, CO, USA) *(USENIX ATC '16)*. USENIX Association, Berkeley, CA, USA, 465–478. http://dl.acm.org/citation.cfm?id=3026959.3027002

[12] Integrated Device Technology, Inc. 1995. *IDT79R4600 TM and IDT79R4700 TM RISC Processor Hardware User's Manual* (revision 2.0 ed.).

[13] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. SeL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

[14] A Theodore Markettos, Colin Rothwell, Brett F Gutstein, Allison Pearce, Peter G Neumann, Simon W Moore, and Robert NM Watson. 2019. Thunderclap: Exploring Vulnerabilities in Operating System IOMMU Protection via DMA from Untrustworthy Peripherals. In *NDSS*.

[15] Alex Markuze, Adam Morrison, and Dan Tsafrir. 2016. True IOMMU Protection from DMA Attacks: When Copy is Faster Than Zero Copy. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. ACM, New York, NY, USA, 249–262. https://doi.org/10.1145/2872362.2872379

[16] Benot Morgan, Eric Alata, Vincent Nicomette, and Mohamed Kaaniche. 2016. Bypassing IOMMU Protection against I/O Attacks. In *2016 Seventh Latin-American Symposium on Dependable Computing (LADC)*. 145–150. https://doi.org/10.1109/LADC.2016.31

[17] Benot Morgan, Eric Alata, Vincent Nicomette, and Mohamed Kaaniche. 2018. IOMMU Protection Against I/O Attacks: A Vulnerability and a Proof of Concept. *Journal of the Brazilian Computer Society* 24, 1 (09 Jan 2018), 2. https://doi.org/10.1186/s13173-017-0066-7

[18] Alastair Reid. 2016. ARM's Architecture Specification Language. Online. https://alastairreid.github.io/specification_languages/.

[19] Alastair Reid. 2016. Trustworthy Specifications of ARM v8-A and v8-M System Level Architecture. In *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design* (Mountain View, California) *(FMCAD '16)*. FMCAD Inc, Austin, Texas, 161–168.

[20] Alastair Reid. 2020. Using ASLi with Arm's v8.6-A ISA specification. Online. https://alastairreid.github.io/using-asli/.

[21] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. 2009. Dingo: Taming Device Drivers. In *Proceedings of the 4th ACM European Conference on Computer Systems* (Nuremberg, Germany) *(EuroSys '09)*. Association for Computing Machinery, New York, NY, USA, 275–288. https://doi.org/10.1145/1519065.1519095

[22] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. 2009. Automatic Device Driver Synthesis with Termite. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 73–86. https://doi.org/10.1145/1629575.1629583

[23] Leonid Ryzhyk, Adam Walker, John Keys, Alexander Legg, Arun Raghunath, Michael Stumm, and Mona Vij. 2014. User-Guided Device Driver Synthesis. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 661–676.