mmapx: Uniform memory protection in a heterogeneous world

Reto Achermann[*]
University of British Columbia
Vancouver, Canada

David Cock[*]
ETH Zurich
Zurich, Switzerland

Roni Haecki[*]
ETH Zurich
Zurich, Switzerland

Nora Hossle[*]
ETH Zurich
Zurich, Switzerland

Lukas Humbel[*]
ETH Zurich
Zurich, Switzerland

Timothy Roscoe[*]
ETH Zurich
Zurich, Switzerland

Daniel Schwyn[*]
ETH Zurich
Zurich, Switzerland

ABSTRACT

Modern Systems-on-Chip (SoCs) are networks of heterogeneous cores, intelligent devices, and memory, connected through multiple configurable address translation and protection units like IOMMUs and System MMUs.

Modern OS kernels like Linux are based on traditional MMUs and have no clear abstractions to represent this complexity, mostly leaving IOMMU configuration to device drivers. This has led to a recent spate of serious bugs, and increasing concern over “cross-SoC” attacks on memory security.

To address this, we propose a new kernel primitive, mmapx, based on a *decoding net* a rich and detailed representation of the memory addressing semantics of a complex SoC from the recent formal methods literature. mmapx provides a uniform facility for securely configuring all the address translation facilities in a system.

mmapx leverages existing Unix facilities wherever possible: the file system for naming, discovery, and coarse-grained access control, and file descriptors for fine-grained authorization. We show how mmapx can eliminate bugs caused by device drivers programming IOMMUs directly, but also the detail captured by the underlying model has further benefits while incurring minimal overhead.

CCS CONCEPTS

• **Software and its engineering → Operating systems**; **Memory management**; **Virtual memory**.

KEYWORDS

operating systems, memory management, address spaces

ACM Reference Format:
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 *Workshop on Hot Topics in Operating Systems (HotOS ’21), May 31–June 2, 2021, Ann Arbor, MI, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3458336.3465273

1 INTRODUCTION

Memory management in a modern operating system provides protection via a simple model: client software (either a user-space process or an in-kernel driver) requests access to a physical resource (either memory or device registers) by identifying its location in the system-wide physical address space. The OS checks authorization, and then maps the appropriate physical region into the client’s virtual address space. Unfortunately, these days this appealingly simple model just doesn’t work.

Modern SoCs have complex interconnects and peripheral devices, and much of what Linux models as devices are really complete processors with their own firmware or independent operating systems, often incorporating their own memory translation units. For protection to be useful, these devices must be sandboxed behind a correctly-configured IOMMU (or System MMU).

However, programming IOMMUs is complex and error-prone, and is often delegated to individual device drivers, which have ambient authority in the kernel. The task is made

[*] Authors are in alphabetical order

HotOS ’21, May 31–June 2, 2021, Ann Arbor, MI, USA
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8438-4/21/05.
https://doi.org/10.1145/3458336.3465273

worse by the need to enforce a changing partial correspondence between the virtual address space seen by the device, and that seen by a process, since the OS needs to share datastructures with devices as much as protect itself from them. The result is that buggy, compromised, or just plain malicious devices or drivers can do an end-run around the OS protection model by exploiting holes in the IOMMU-based protection domain [14, 22].

Surprisingly, modern OSes provide no good abstractions for *uniformly* handling this problem, leaving low-level configuration of protection up to individual device drivers. This is in contrast to, e.g. access control in file systems or authority over process address space, where well-established subsystems enforce OS policy.

We propose a new primitive, `mmapx`, for clients to request general memory mappings. Unlike existing interfaces, `mmapx` is explicit about which address space it is mapping a region from, *and* which address space it is mapping a region into. This allows clients to be precise in specifying what memory is exposed to devices or coprocessors via IOMMUs. Like `mmap()`, `mmapx` refers to memory regions using file descriptors, which provide capability-like protection. However, regions for `mmapx` are acquired using a file system (`/dev/as`), which captures the memory topology of the machine in detail, and allows basic authorization to leverage the full Unix file system protection model via two different rights on an address space: `map` and `grant`.

We describe `mmapx` here from the point of view of user-space Linux programs but the basic model works in a microkernel architecture, or within a monolithic kernel. In the latter case, protection within the kernel is only advisory, unless a mechanism like Nooks [28] is available. Crucially, however, even in this case `mmapx` provides a policy framework for protecting the kernel itself from malicious or buggy drivers, device firmware, or other cores not running the kernel itself by ensuring that IOMMUs and other translation units are correctly programmed.

Moreover, while `mmapx` resembles a high-level primitive like `mmap()`, this belies its true power. `mmapx` builds on our existing work on formalizing address translation and decoding, and its concept of an "address space" region is flexible and powerful enough to capture the functionality of the full range of TLBs, and even individual levels of a multi-level page table. This allows `mmapx` to express "delegation" of MMU page table structures to virtual machines, for example, as in Arrakis [24] and Ix [6].

2 MOTIVATION

The "QualPwn" exploit [14] is emblematic of the problem we address in this paper. It makes brutally clear the mismatch between the model of hardware behavior baked into the OS kernel, and the reality of modern SoC platforms.

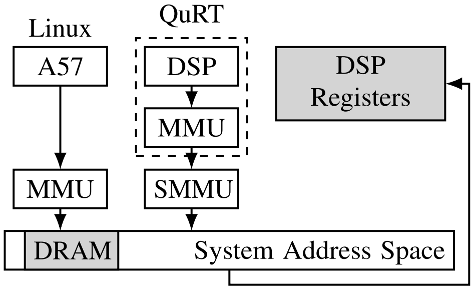

Figure 1: Relevant actors (A57 and DSP), translation units (MMU, SMMU) and memory regions (gray) of the Qualcomm SoC. Note that the DSP MMU is not controlled by the host Linux, but the SMMU is.

QualPwn affects mobile SoCs running Android and starts with a bug in the WLAN process running not on the CPU but a DSP core on the chip, which runs the proprietary QuRT OS. A series of exploits allows compromise of another process on the DSP, which itself communicates with its corresponding device driver in the Linux kernel on the application cores using DMA. Since the driver trusts the device, it can be tricked into granting the device full access to application memory by reprogramming the system MMU.

Our focus in this paper is preventing incorrect granting of memory access rights to devices and drivers. A simplified view of the hardware is shown in Figure 1: two processors, running a different OS, with different MMUs, but sharing the same memory. The Linux kernel driver is tasked with configuring the SMMU to only allow legitimate access to buffers shared between the two cores, and it fails to do this.

Linux offers little functionality to help with this task. Instead, it relies on a naive model where a set of process virtual address spaces are mapped to a single physical address space, and protection against DMA-capable devices using an IOMMU or SMMU is delegated to drivers. Indeed, the SMMU is often programmed to give a device *the same* view of memory as that of the corresponding software process, whether in user space or the kernel.

For example, the recent Linux Heterogeneous Memory Manager [29] attempts to unify device memory management using a specialized page structure to replicate translation across device and CPU address spaces, in order to simplify programming with GPU and FPGA accelerators.

We argue this is inappropriate for devices: access by the device should be restricted as much as possible, rather than giving the device free rein over application memory [22]. However, Linux provides no help in maintaining partially replicated mappings between heterogeneous devices or cores:

there are simply no abstractions for explicitly changing SMMU mappings.

We are not first to point this out [13], and so-called "cross-SoC" attacks [4, 14, 26] and other DMA-related vulnerabilities [18, 21, 23, 25, 32] which exploit the fact that the OS is not in control of software running on other cores on the SoC is becoming a serious security concern. Neither does moving the device driver out of the kernel solve the problem, whether into Linux user space or using a microkernel-oriented design.

Our solution to this is to provide a single primitive for configuring *any* translation hardware in the system, with a clear and secure authorization framework that leverages existing mechanisms in the kernel.

As a useful side-effect, this primitive also subsumes the underlying functionality of newer memory managers like Linux HMM [29], and VFIO/DPDK [19] (which could usefully be implemented over mmapx), while providing more fine-grained protection.

3 THE MMAPX PROGRAMMING MODEL

To provide principled control over address protection and translation in any part of the system (including co-processors and IOMMUs), we need abstractions that go beyond the traditional Linux model – that is,a single physical address space and multiple processes, all of which map virtual addresses to this one physical space.

We leverage our recent work on representing complex addressing in a modern SoC as a *decoding net* [2]. In our work, an *address space* or AS is simply a function from "virtual" addresses to either physical resources or another address space, and the system is represented by a directed graph whose nodes are ASes and whose arcs are mappings to other ASes. Leaf nodes in the graph are banks of memory (e.g. DRAM) or device registers (e.g. the DSP registers in Figure 1), whereas interior nodes are the views on memory created by fixed or configurable translation units (e.g., an MMU, SMMU, or system address space in Figure 1). The root nodes are the complete set of cores or DMA-capable devices (e.g., the A57 and DSP in Figure 1).

A decoding net for a system has a different AS (node) for each distinct view of what an address means. For example, every core in a system has its own physical AS containing local devices like interrupt controllers, but most of whose addresses map to a shared "system address space", which itself leads to distinct nodes representing further buses and interconnects. Decoding nets have been shown to be sufficient to capture caches, MMUs, lookup tables, and a variety of other addressing hardware. Essentially, mmapx is a primitive that changes memory protection in a system by editing this decoding net at runtime. The kernel assembles its initial decoding net representation of the hardware using standard device discovery and Devicetree representations [20].

$$\overbrace{\texttt{/dev/as}}^{base}\ \overbrace{\texttt{/0}}^{asid}\ \overbrace{\texttt{/\{grant,map,info\}}}^{right}$$

Figure 2: The path structure of /dev/as

For example, the Linux kernel running on the A57 in Figure 1 would detect the presence of the modem coprocessor from a Devicetree entry, and would create the corresponding decoding net elements: the DSP register bank as a leaf node, plus a mapping between the System address space whose exact location is determined by the reg field in the Devicetree.

In this example, additional information (not available in Devicetree) is required to infer that a DMA capable device or coprocessor (the DSP) exists and must be added to the decoding net, together with its own associated MMU connected to the System MMU; it would seem useful for a future version of Devicetree to provide this important information.

In addition to representing the already complex memory topology in our example, the kernel would instantiate drivers for all the translation units involved (including the DSP MMU, the System MMU, and any associated lookup tables).

The decoding net is also augmented with a virtual ASes for each process.

The /dev/as file system: The kernel internal decoding net representation is exposed to clients via a file system interface. All nodes in the decoding net are enumerated and become directories in a virtual file system called /dev/as, which provides naming and access control using standard Linux access control lists. We use standard Linux udev to add links from easily determinable names into /dev/as. There exists for example a static name that links to the system address space.

Each node directory in /dev/as contains a file called info, which describes the relation of this AS to others. Each region of the AS which maps to another is listed with the filename of the destination AS, plus whether the mapping is static (fixed), or can be configured, and if so, what constraints exist on this mapping (e.g., page size).

For example, in Figure 1 the mapping from the system address space to the DSP register leaf node would be described in the info of the system address space as a *static* mapping, because it can not be reconfigured.

Regions that simply contain RAM or memory-mapped IO are similarly listed. /dev/as thus provides a flat, persistent namespace for ASes with cross-references, which is needed since the decoding net for real systems is not a tree, and indeed often contains cycles.

```
int mmapx(int src_fd,  off_t src_offset,
        int dst_fd, off_t dst_offset,
        size_t length, int flags);
int munmapx(int src_fd, off_t src_offset,
            size_t length);
```

Figure 3: `mmapx` and `munmapx` signatures

`grant` and `map`. : Each AS directory can also contain two further files, `map` and `grant` (Figure 2). Access to `grant` confers the right to map parts of this AS into another which has an arc to it in the decoding net (e.g. by configuring an MMU between the two), and can be viewed as a generalization of traditional `mmap()`.

`map`, however, is different and has no analogue in traditional Unix systems. Access to `map` is required to map a region of another AS *into this one*. Creating a mapping from a region r_1 in AS A_1 (e.g., a process address space) into region r_2 of AS A_2 requires access to both `grant` for A_2 *and* `map` for A_1, plus there to be an arc from A_1 to A_2 which includes the region r_1.

Leaf nodes in the decoding net, i.e. RAM or device registers, have a `grant` but no `map`. Conversely, process virtual address spaces have a `map` but no `grant` (since they cannot be mapped into some other AS). Intermediate translation steps lead to ASes with both `grant` and `map` files. In the example of Figure 1 the SMMU is such an intermediate AS. Initially it contains only `map`. When a mapping is installed by calling `mmapx` with a `grant` handle to the DRAM and a `map` handle to the SMMU, it appears in the SMMU `grant`.

There are therefore *two* fundamental rights involved in mapping: the right to change the mapping of an address space (`map`) and the right to map to something (`grant`). This combination of rights is key to how `mmapx` works.

`mmapx` itself: Figure 3 shows the `mmapx` syscall. It attempts to create a mapping of a region in a source AS given by a file descriptor `src_fd` to a target AS given by `dst_fd`. The size and location of the region in each AS is also supplied as arguments.

The source file descriptor is obtained by opening the `map` of the appropriate AS, and the destination file descriptor is obtained by opening the `grant` of the target AS. By way of contrast, in traditional `mmap()` the source is implicitly the calling process' virtual address space; only the target region (segment) is given.

Access control on mapping is therefore enforced when the `map` and `grant` are opened; thereafter the file descriptors function as short-term *capabilities*. This is by design, and not by accident: capabilities give the bearer certain rights over a specific resource. Nevertheless, `mmapx` can still fail for a variety of reasons: the requested mapping may not be meaningful given the topology, hardware granularities like page sizes constrain the region offset and length, or a conflicting mapping might already exist.

ASes in decoding nets tend to be more fine-grained than the traditional single physical address space (device registers often occupy their own small AS, for example), but `mmapx` as described so far is not quite sufficient for fine-grained (page-sized) least-privilege authorization.

For this we allow operations on the file descriptors (such as `truncate`) which limit the range of authorized addresses in the AS, analogous to *refining* a capability (deriving a new capability with less or equal rights, such as covering a smaller region of memory). This means that we can augment coarse-grained file system-based access control with system "monitor" processes with the rights to open some `map` and `grant` files and then restrict the descriptors before passing them to client processes. An allocator (either in-kernel or at user level) would be such a process: The right to allocate (e.g. DRAM) is given by `grant` on the whole space, and individual allocations are returned to callers as `truncate`-d descriptors (with `grant`) to a new anonymous segment.

In-kernel calls: While we have presented `mmapx` as a user-space API, it can also be naturally provided within the kernel using equivalent authorization checks and handles. While in-kernel device drivers are not sandboxed in the way that user-space processes are, `mmapx` removes the burden of IOMMU configuration from drivers and allows devices themselves to be conveniently sandboxed.

Moreover, intra-kernel protection schemes such as split kernel [11] or in-kernel enclaves [28] can be naturally modeled using decoding nets, and thus become usable transparently by drivers whenever they are available.

Additional complexity: Dropping the fiction of homogeneous MMUs and a single physical address space in favor of a decoding net exposes the complexity of the memory system, and this is a two-edged sword. `mmapx` is a minimalist secure primitive which only involves two linked ASes. Clients are left to set up a potential chain of mappings (albeit safe ones).

However, this kind of problem already exists in the Linux kernel for device management, where it is addressed by system facilities like `udev`, `devfs`, and `sysfs`. For user space clients, a library operating on `info` file contents can provide a simplified end-to-end mapping interface above the full graph of address spaces, without sacrificing flexibility.

In the final part of this paper we provide some evidence that the overhead introduced by `mmapx` would be negligible.

Example: preventing QualPwn: Having compromised the WLAN firmware, the vulnerability QualPwn exploits to compromise the Linux host is a failure to distinguish between `map` and `grant` rights. The driver needs `map` rights for the DSP's address space (to add SMMU mappings), but it should

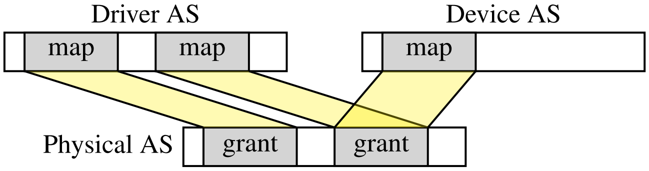

Figure 4: Memory mapping in the QualPwn scenario

not have `grant` rights to arbitrary host memory: the SMMU should *only* allow the DSP access to descriptors and packet buffers in host memory.

Defense in depth and the principle of least privilege imply that any world-facing device (e.g. a network interface) or process is vulnerable to compromise, and should thus be assumed malicious. A device (and its driver) should therefore be assigned the minimum privilege necessary to function. In QualPwn the driver is acting as a classic “confused deputy” [15], exercising the ambient authority of the kernel (to `grant` arbitrary memory) at the behest of a malicious client (the device).

While a capability-based microkernel with strict isolation would prevent a malicious driver causing damage, that is *not* the actual problem in this case: Instead, an honest driver has *mistakenly* abused its ambient authority. This mistaken abuse is prevented using `mmapx`.

Using `mmapx`, on initialization the driver is identified as manager of the device’s SMMU-implemented (virtual) AS and given a `map` descriptor for this AS. When buffers are allocated (e.g. with `dma_alloc_coherent()`), an anonymous segment is created as a `truncate`-d `grant` descriptor from host memory, either by the driver itself via a library function, or as part of a buffer pool.

Possession of the `grant` right on this memory segment together with the `map` right on the DSP’s AS allows the driver to call `mmapx` to make this memory accessible to the DSP, but as long as the driver only every manipulates SMMU mappings via the `mmapx` interface, it cannot be tricked into abusing its ambient authority and providing access to anything other than buffer memory.

4 ADDITIONAL BENEFITS

As we have shown, instead of a traditional Unix 2-level view of memory (virtual ASes spaces mapped by a page table to single physical AS), `mmapx` adopts a more faithful view of a modern computer system as a graph of address spaces connected by address translation functions. Some of these translations are fixed and trivial (such as an offset), while some are highly complex and implemented by MMUs.

It turns out that this detailed representation has benefits beyond the fine-grained device memory protection we have discussed so far. For one thing, it allows us to generalize the Unix notion of memory segments to include *partial* translations from regions of virtual memory to physical resources, in the process subsuming a number of ad-hoc mechanisms already deployed or proposed in the research literature.

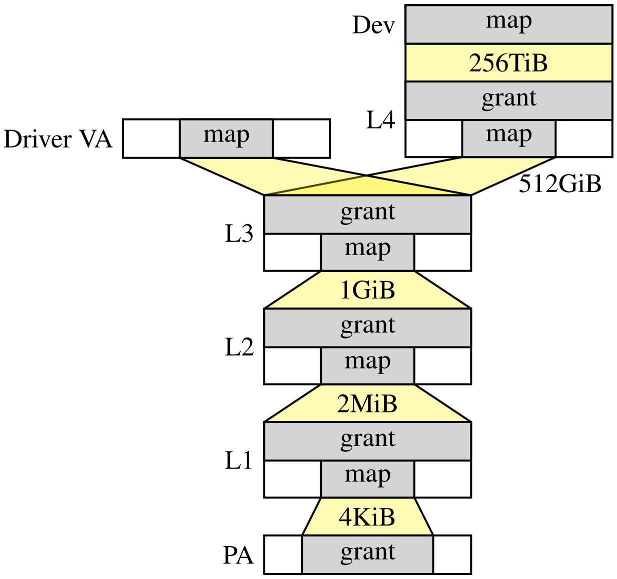

Figure 5: Example of sharing page tables between a process and a device using Intel 4-level paging

Page table structure: Rather than representing an MMU with a multi-level page table as a single translation from one AS to another, we observe that each level of the page table implements its own translation, and we can expose this in the decoding net. This allows clients to make explicit superpage mappings without have to rely on less flexible or controllable facilities like transparent superpages or `libhugetlbfs`.

However, the fact that `mmapx` can expose each layer of translation as a separate AS means that these ASes can be passed around, allowing explicitly controlled partial sharing of page mappings between processes, or even application-controlled replication of page regions between cores within a process [8], as shown in Figure 5.

This ability of a part of a page table (albeit highly abstracted) as a first-class citizen in the OS is an example of an *intermediate address space* enabled by `mmapx`. An intermediate AS has both `grant` and `map` rights, and can be passed between clients, and installed and removed within another AS using the `mmapx` call.

For example, SpaceJMP [12] allows processes to use and switch between multiple virtual ASes, although it requires the process’ stack, code and data segments to be present in the target AS. `mmapx` provides the same functionality by creating an intermediate AS and using its *grant* right to map it into the calling process with additional flexibility, since arbitrary parts of the AS can be swapped without the SpaceJMP's need to replicate most of the whole address space in each target.

Intermediate ASes can be viewed as generalizations of Multics-style segments [9], in that they do not need to be contiguous in physical memory, but can still be passed and shared between processes.

Other translation units: The flexibility of the decoding network used by in mmapx also means it can be used to securely configure other address translation units which, today, are programmed once at boot time and never touched thereafter due to a lack of safe abstractions. For example, some accelerators like the Intel Xeon Phi [16] map *physical* addresses from the accelerator's MMUs through a 32-entry lookup table, each of which translates a 16GiB region of host address space (*before* going through the host IOMMU!). mmapx would allow suitably authorized client programs to exploit the efficiency of this lookup table, reducing TLB overhead in the IOMMU and accelerator.

Supporting existing interfaces: Finally, we note that mmapx is a useful foundational primitive for supporting existing interfaces for dealing with device memory and IOMMUs. For example, Linux HMM [29] mirrors the full virtual address of a process to a device. This can conveniently be implemented over mmapx, though the more coarse-grained protection offered by HMM exposes the kernel to attacks like QualPwn.

Similarly, VFIO/DPDK [19], which exposes parts of a process AS to devices, could also be built conveniently using mmapx, and take advantage of more fine-grained protection, delegation, and also the ability to include other device's memory spaces in addition to client processes.

5 PRIOR IMPLEMENTATION

mmapx was inspired by essentially the same functionality which we had earlier implemented in the Barrelfish research OS [5], but which we had been unsuccessful in publishing [1].

Barrelfish is built around a capability system, itself based on that of seL4 [17], but adding support for distributed capability retyping and revocation as well as a richer type system for capabilities defined in a Domain Spacific Language [10]. User processes manage virtual address spaces using capability invocations that correspond roughly to mmapx system calls. The capability type system preserves all invariants and ensures that only valid page tables can be constructed, and that a principal cannot elevate privilege using a special mapping.

However, like other capability systems like KeyKOS [7], Eros [27], and CAP [30], the underlying assumption is that of virtual address mappings to a single, global, physical address space. The same is true for architectural capability units situated upstream of the MMU, such as CHERI [31]

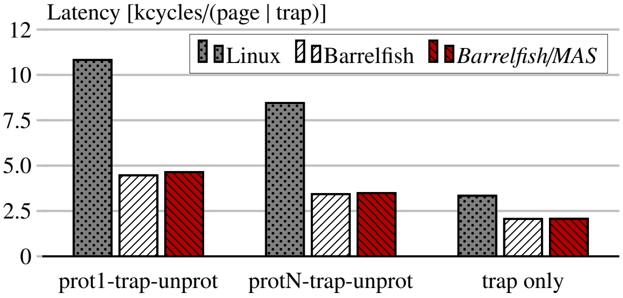

Figure 6: Appel-Li benchmarks on Barrelfish with ("MAS") and without mmapx, and Linux

We extended Barrelfish in two ways. First, we added an "address space" field to each capability, and ensured that capability operations check the validity of these values in addition to type and rights information before proceeding. Second, we extended the capability type system to represent all available page table formats (including Intel IOMMUs, AMD IOMMUs and Xeon Phi accelerator page tables), as well as the rights to program any other translation device.

mmapx's combination of /dev/as permissions and open file descriptors provides a slightly more limited, but otherwise equivalent, authorization model for mappings.

The decoding net itself was represented in Barrelfish's "System Knowledge Base" (SKB), which is a combined Prolog database and Constraint Logic Programming (CLP) solver. A simple routing policy for address mapping is provided in the SKB as a CLP query, although clients can also use their own. The info files in the /dev/as file system, combined with client library support, provide basically the same functionality in mmapx.

To provide some confidence we had got the design right, we first built an *executable specification* of mmapx's functionality in Haskell, inspired by the use of this technique in seL4 [17]. This spec then served as the basis for the C implementation in Barrelfish.

The Haskell implementation can check if a sequence of address resolution and address translation operations conforms to the specification by simulating the effects of the operations, modelling the authority each process holds as a set of access, map and grant rights. By requiring that no process ever has grant authority on addresses that are either in leaf address spaces or already mapped to addresses in another AS, the specification ensures that no "dangling pointers" can be installed and thus address resolution will always terminate at a real physical resource.

The Barrelfish implementation does provide some indication of the performance cost of the more complex view of the memory system that mmapx provides. The Appel-Li benchmark [3] exercises common virtual memory operations with three experiments:

(1) *prot1-trap-unprot.* Write-protect a randomly-chosen page, write to it, take a trap, unprotect it, and repeat.
(2) *protN-trap-unprot.* Write-protect 512 pages of memory in a single operation, write to each page of memory in turn, taking a trap and unprotecting the page.
(3) *trap only.* Write to a protected page and take the trap, then continue with next page without changing any permissions.

Figure 6 shows the results on a dual-socket Intel Xeon server for the original Barrelfish, Barrelfish with the mmapx-like extensions, and a vanilla Linux kernel 4.15 for comparison. All Specter/Meltdown mitigations in Linux are disabled in this experiment.

We make no claims here about the comparison between Barrelfish and Linux, since they are very different designs of OS, but it is clear that the additional complexity of mmapx adds negligible overhead to Barrelfish, and is likely to have similarly minimal impact in Linux, since its performance is similar. If anything, a Linux mmapx will have even less impact, both because Linux memory management is somewhat slower, and an optimized, fully in-kernel mmapx implementation should be faster than the Barrelfish version.

6 CONCLUSION

We identify the increasing complexity of the "physical" address space, the need to securely program IOMMUs/SMMUs to achieve defense in depth, and the lack of a consistent interface for this as a major vulnerability of existing OS designs.

To remedy this, we propose mmapx, a consistent, universal interface for both in-kernel and user-level construction of address spaces that faithfully captures the complexity of modern hardware while extending and reusing existing, familiar UNIX primitives, specifically: The file system as a secure naming mechanism (`/dev/as`), and memory-mapped segments (named and anonymous). We identify two distinct rights, *map* and *grant*, which are usually conflated, and demonstrate that properly and securely distinguishing them allows a least-privilege approach to SMMU configuration, addressing known vulnerabilities such as QualPwn.

We further describe exciting extended use cases for mmapx, including the safe partial sharing of virtual address space regions and the potential to subsume existing approaches including HMM and VFIO. Finally, previous implementation experience with mmapx in a different OS suggests that the runtime overhead of more faithfully representing the addressing structure of a machine is negligible.

ACKNOWLEDGEMENTS

We thank the anonymous reviews for their helpful feedback, particularly the one who described mmapx as a "Swiss Army chainsaw". We are also grateful for financial support for this work from VMware and Huawei.

REFERENCES

[1] Reto Achermann, Nora Hossle, Lukas Humbel, Daniel Schwyn, David Cock, and Timothy Roscoe. Secure Memory Management on Modern Hardware, 2020. Preprint: https://arxiv.org/abs/2009.02737.

[2] Reto Achermann, Lukas Humbel, David Cock, and Timothy Roscoe. 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*, ITP'18, pages 1–19, 2018.

[3] Andrew W. Appel and Kai Li. Virtual Memory Primitives for User Programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 96–107, New York, NY, USA, 1991. ACM.

[4] Nitay Artenstein. Broadpwn: Remotely compromising Android and iOS via a bug in Broadcom's Wi-Fi chipsets. *Black Hat USA*, 2017.

[5] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 29–44, New York, NY, USA, 2009. Association for Computing Machinery.

[6] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, October 2014. USENIX Association.

[7] Allen C. Bomberger, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, and Jonathan S. Shapiro. The keykos nanokernel architecture. In *Proceedings of the Workshop on Micro-Kernels and Other Kernel Architectures*, page 95–112, USA, 1992. USENIX Association.

[8] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 43–57, USA, 2008. USENIX Association.

[9] F. J. Corbató and V. A. Vyssotsky. Introduction and Overview of the Multics System. AFIPS '65 (Fall, part I), page 185–196, New York, NY, USA, 1965. Association for Computing Machinery.

[10] Pierre-Evariste Dagand, Andrew Baumann, and Timothy Roscoe. Filet-o-fish: Practical and dependable domain-specific languages for os development. *SIGOPS Oper. Syst. Rev.*, 43(4):35–39, January 2010.

[11] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, page 191–206, New York, NY, USA, 2015. Association for Computing Machinery.

[12] Izzat El Hajj, Alexander Merritt, Gerd Zellweger, Dejan Milojicic, Reto Achermann, Paolo Faraboschi, Wen-mei Hwu, Timothy Roscoe, and Karsten Schwan. SpaceJMP: Programming with Multiple Virtual Address Spaces. In *Proceedings of the Twenty-First International*

Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, page 353–368, New York, NY, USA, 2016. Association for Computing Machinery.

[13] Simon Gerber, Gerd Zellweger, Reto Achermann, Kornilios Kourtis, Timothy Roscoe, and Dejan Milojicic. Not Your Parents' Physical Address Space. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, page 16, USA, 2015. USENIX Association.

[14] Xiling Gong, Peter Pi, and Tencent Blade Team. Exploiting Qualcomm WLAN and Modem Over the Air. *Proceedings of the BlackHat USA 2019*, page 58, 2019.

[15] Norm Hardy. The Confused Deputy: (Or Why Capabilities Might Have Been Invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, October 1988.

[16] Intel Corporation. *Intel Xeon Phi Coprocessor System Software Developers Guide*, March 2014. SKU 328207-003EN.

[17] 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. Sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[18] Eduard Kovacs. Devices Still Vulnerable to DMA Attacks Despite Protections. https://www.securityweek.com/devices-still-vulnerable-dma-attacks-despite-protections, 2020. Accessed: 2021-02-03.

[19] Linux. Linux VFIO Readme. https://www.kernel.org/doc/Documentation/vfio.txt, 2021. Accessed: 2021-02-01.

[20] Linux Kernel Documentation. *Device Tree Source Format* , version 1.0 edition, 12 2019. Online. https://git.kernel.org/pub/scm/utils/dtc/dtc.git/plain/Documentation/dts-format.txt. Accessed 30. December 2019.

[21] Moshe Malka, Nadav Amit, and Dan Tsafrir. Efficient intra-operating system protection against harmful dmas. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, page 29–44, USA, 2015. USENIX Association.

[22] A. T. Markettos, Colin Rothwell, Brett F. Gutstein, A. Pearce, P. Neumann, S. Moore, and R. Watson. Thunderclap: Exploring Vulnerabilities in Operating System IOMMU Protection via DMA from Untrustworthy Peripherals. In *NDSS*, 2019.

[23] Alex Markuze, Adam Morrison, and Dan Tsafrir. True iommu protection from dma attacks: When copy is faster than zero copy. *SIGPLAN Not.*, 51(4):249–262, March 2016.

[24] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 1–16, Broomfield, CO, 2014. USENIX Association.

[25] Colin L. Rothwell. *Exploitation from malicious PCI Express peripherals*. PhD thesis, University of Cambridge, UK, 2019. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-934.pdf.

[26] Denis Selyanin. Researching Marvell Avastar Wi-Fi: from zero knowledge to over-the-air zero-touch RCE. *ZeroNights*, 2018.

[27] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. Eros: A fast capability system. *SIGOPS Oper. Syst. Rev.*, 33(5):170–185, December 1999.

[28] Michael M. Swift, Steven Martin, Henry M. Levy, and Susan J. Eggers. Nooks: An Architecture for Reliable Device Drivers. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, page 102–107, New York, NY, USA, 2002. Association for Computing Machinery.

[29] The Kernel Development Community. The Linux Kernel Documentation - Heterogeneous Memory Management (HMM). Online. Accessed 2021-02-3. https://www.kernel.org/doc/html/v5.0/vm/hmm.html, 2020.

[30] M. V Wilkes. *The Cambridge CAP Computer and Its Operating System (Operating and Programming Systems Series)*. North-Holland Publishing Co., NLD, 1979.

[31] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. *SIGARCH Comput. Archit. News*, 42(3):457–468, June 2014.

[32] Zhiting Zhu, Sangman Kim, Yuri Rozhanski, Yige Hu, Emmett Witchel, and Mark Silberstein. Understanding the security of discrete gpus. GPGPU-10, page 1–11, New York, NY, USA, 2017. Association for Computing Machinery.