Position Paper: Defending Direct Memory Access with CHERI Capabilities

A. Theodore Markettos
University of Cambridge, UK
theo.markettos@cl.cam.ac.uk

John Baldwin
Ararat River Consulting, USA

Ruslan Bukin
University of Cambridge, UK

Peter G. Neumann
SRI International, USA

Simon W. Moore
University of Cambridge, UK

Robert N. M. Watson
University of Cambridge, UK

ABSTRACT

We propose new solutions that can efficiently address the problem of malicious memory access from pluggable computer peripherals and microcontrollers embedded within a system-on-chip. This problem represents a serious emerging threat to total-system computer security. Previous work has shown that existing defenses are insufficient and poorly deployed, in part due to performance concerns. In this paper we explore the threat and its implications for system architecture. We propose a range of protection techniques, from lightweight to heavyweight, across different classes of systems. We consider how emerging capability architectures (and specifically the CHERI protection model) can enhance protection and provide a convenient bridge to describe interactions among software and hardware components. Finally, we describe how new schemes may be more efficient than existing defenses.

ACM Reference Format:
A. Theodore Markettos, John Baldwin, Ruslan Bukin, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. 2020. Position Paper: Defending Direct Memory Access with CHERI Capabilities. In *Proceedings of HASP'20*. ACM, New York, NY, USA, 9 pages. https://doi.org/https://doi.org/10.1145/3458903.3458910

1 INTRODUCTION

Increasingly, attacks come not just from software running on an application CPU, but from other hardware inside or external to systems. *Cross-system* attacks come about when a secondary piece of hardware is compromised and used as a springboard to attack other parts of the system. For example on an iPhone, the firmware on a Wi-Fi controller might be compromised by sending bad packets over the air [4]: arbitrary code injection is achieved on the Wi-Fi controller, and then the Wi-Fi controller is used to attack the application processor running iOS.

Extremely damaging attacks are possible when such compromised devices have direct memory access (DMA) to system memory. Without protection, attackers with access to main memory can steal all the data on a machine (e.g., passwords, encryption keys, and confidential information). They can also inject arbitrary code to be executed by the application processor. Existing defenses are heavyweight or partial, and in practice tend to see limited use [3, 18, 22]. Performance and programmability overheads are posited as blockers to wider adoption [18].

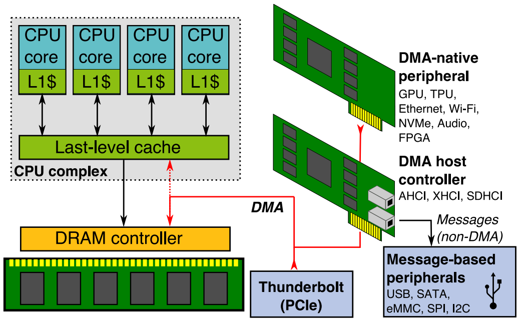

Figure 1: A typical computer system contains multiple components with rights to directly access memory – not only from internal processor cores, but also from peripheral devices. Those which do not have access to DMA typically can intermediate via a host controller that does.

This paper proposes solutions that can efficiently protect against DMA from malicious devices, be they inside a system-on-chip or external. We examine techniques used to constrain malicious software and whether, with added architectural support, they might constrain malicious hardware. We consider in particular the CHERI capability protection model [27], which shows great promise at constraining software – both in terms of applying the principle of least privilege to memory accesses, and with compartmentalization. We assess existing defenses, propose new structures, and consider how CHERI and similar mitigation technologies can be used to provide a more unified hardware/software protection model across systems.

The text is organized as follows. Section 2 outlines how computer systems are currently constructed and existing vulnerabilities to DMA attacks. Section 3 describes the threat model our work addresses and some simplifying assumptions. Section 4 decomposes the problem by systematizing DMA operations, which have often never considered security, in terms of primitives that are used for software protection. We additionally introduce the CHERI protection model. Section 5 analyzes how existing systems apply these primitives to apply some form of protection, often unintentionally as a side-effect of other system design choices.

We then divide our focus into a number of different application areas and discuss architectural primitives that may help, grouped

HASP'20, October 2020, Online

https://doi.org/https://doi.org/10.1145/3458903.3458910

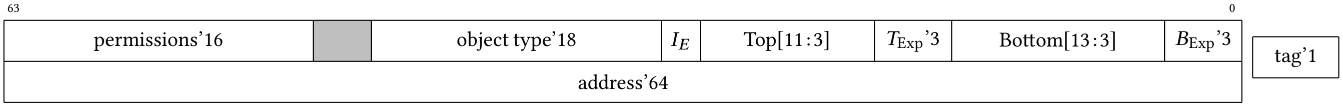

Figure 2: CHERI represents address, bounds, type and permissions in an in-memory capability plus a one-bit out-of-band tag. In the CHERI Concentrate representation [28] 64-bit pointers with top/bottom bounds are compressed into an 128-bit format using a scheme akin to floating point.

into two threat classes. First, we consider the defenses a system can employ from attaching devices with both untrustworthy hardware and untrustworthy software. This might be when a malicious network card is plugged into a Thunderbolt port [18], or a system-on-chip (SoC) designer instantiates a black-box third-party IP. Section 6 applies this to microcontroller systems, being relatively resource-constrained with simpler software stacks. Section 7 considers more complex systems that can afford translation-based protection, commonly on application processors and SoCs running full operating systems such as Linux. Section 8 examines the requirements of deeper integration between devices and applications, where performance prevents intermediation via a trusted operating system kernel.

For the second threat class, Section 9 considers a device with trustworthy hardware but untrustworthy software. For instance, an SoC might be built entirely by one vendor, but with firmware for devices vulnerable to a supply chain attack, or subject to over-the-air compromise.

Finally, Section 10 sketches a comparison of the different architectural constructs, and identifies promising ways forward. Section 11 considers next steps and Section 12 concludes.

2 BACKGROUND

Traditionally, a computer system has been viewed in the past as a 'central' processing unit (CPU) and 'peripherals'. The nature of systems has evolved substantially over the decades, and thus we need to revisit the architectural shape of a system to consider it with security in mind. Since the microprocessor era, we have been habituated to the idea that all compute happens on the CPU and the peripherals are its servants. In truth, while there remains something labeled a CPU, systems have many distributed computing elements – both towards the center and at the margins. Often, 'peripherals' have substantial compute power of their own, in which standardized interfaces are implemented by 'firmware'.

We draw a caricature of such a system in Figure 1. On this diagram we can broadly classify devices into two categories: those that have direct access to memory (DMA), and those that intermediate memory access via another device (a *host controller* that performs DMA on their behalf). Often the latter category has a message-passing paradigm (for instance, SATA or USB) in which the operating system and its device drivers are responsible for interpreting messages. In such protocols memory addresses are not passed to the remote device but remain on the host controller. DMA is typically required when performance is critical: as a hardware-only path, it does not need to intermediate via software; however, it is also less expressive.

This paper focuses on DMA, because it represents both very high risks and a very critical security challenge. Defenses from DMA attacks also apply to message-passing interfaces, which can can be used to perform DMA attacks if their host controller is compromised (although DMA defenses will not protect against bugs in driver stacks interpreting those messages).

2.1 Existing vulnerabilities

For the most part, systems have been poorly defended against DMA attacks; there is a growing literature of these emerging threats. Protocols such as Firewire, PCI, PCI Express and Thunderbolt provided unimpeded access to system memory, and attacks were straightforward for anyone who tried [25]. More recently, protections such as the Input/Output Memory Management Unit (IOMMU) have been enabled, which made attackers look more carefully for loopholes, either at boot time or through misconfiguration [12]. Recent work has indicated that in practise IOMMU protections were not being used well, and it is suspected this is due largely to performance concerns [18].

Separately, interest has grown in cross-system attacks. Firmware compromise of DSPs [16], management controllers [21, 24], network [7, 26] and Wi-Fi [4] cards has demonstrated that attackers moving from one component to another within the system is a plausible threat. Attacks on management processors [21] underscore how these interconnections can be complex and exploitable.

3 THREAT MODEL AND ASSUMPTIONS

We make some simplifying assumptions here, but only to simplify the description. We assume we are dealing with a reasonably conventional computer system. That means we have a single complex of CPUs, consisting of some number of cores joined by a cache-coherency fabric (which may be spread across multiple physical chips). The CPUs are connected to main memory, typically Dynamic RAM (DRAM), via port(s) on the caches. We assume the CPU hardware, the cache hierarchy, and the DRAM are trustworthy – i.e., they have been designed correctly and have not been manipulated by an attacker. (Thus, attacks such as Rowhammer [15] are not directly in scope, although separate mitigations may be employed.)

Further, we assume that, while applications might be untrustworthy, some level of the software running on the CPU *is* trustworthy. We refer to that as the operating system kernel – although it might be a hypervisor, microkernel, executive or other privileged component. Hardware techniques such as the CHERI instruction-set architecture and its protection model can be used to improve the trustworthiness of software, and to enforce these boundaries more clearly [6].

Away from the CPU complex, we have DMA-capable peripherals. Peripherals are largely autonomous, with their own processors running their own firmware, and they have their own ability to access DRAM. The physical form factor may vary substantially – for instance, a storage controller in an embedded SoC, a Wi-Fi chip soldered into a phone, a plugin network card in a server, or an external GPU attached to a Thunderbolt port of a laptop. Peripheral devices may also have their own internal memory. The basic problem is that the device's main memory access provides a vector for some entity controlling the device to attack the system, for instance by stealing secrets or injecting malicious code for the CPU to run. Our threat model covers the case that an attacker can attach an extra malicious device to the system, via a port such as Thunderbolt or PCI Express [18]. Additionally, it considers that an existing device may have been compromised. Once an attacker has control, they are able to make memory requests acting as that device.

Out of scope for this paper are non-traditional computing systems, for example, clusters sharing memory between servers via Infiniband. Some of our solutions may apply, but would require further study. Also out of scope are non-address-carrying protocols such as USB[1], although their *host controllers* are in scope.

Embedded systems often have a similar shape to such desktop systems, although frequently more resource constrained. Our work largely applies in that context as well – in that arena it applies to DMA-based platforms such as VME or PC/104, but not directly to non-DMA-based networks such as CAN.

We broadly classify malicious peripherals in two forms. Devices with *untrustworthy hardware and software* are fully controlled by an attacker, and may perform any operations allowed by their external interfaces. For example they may make arbitrary memory requests.

Alternatively, devices may have *trustworthy hardware, but untrustworthy software.* For instance, the hardware comes from a trustworthy vendor and has not been tampered with, but the firmware has been compromised or replaced. The scope of attacks possible from such devices depends on the restrictions imposed by their own hardware, as well as their interfaces into the memory system. At present, such restrictions are typically arbitrary and not focused on security (but rather, are whatever features the hardware designer decided to provide). However, we also consider cases in which the device hardware can impose security defenses against malicious firmware.

Our work thus covers two areas. First, we consider mechanisms to protect the host computer from malicious memory requests coming from devices, by constraining the memory a device is allowed to access. Second, we consider enforcement *within peripherals*, allowing devices to run untrustworthy software without affecting other components in the system. We assume a defense-in-depth approach that might use both techniques.

4 MEMORY PROTECTION CONCEPTS

In this section, we identify some protection concepts that are applicable to low level software such as operating systems and device drivers.

4.1 The bounded buffer

Foundational to software, and similarly to hardware, is the concept of the *bounded buffer.* This comprises a linear region of memory, which may be described by top and bottom addresses. or more commonly a base address and a length. Conceptually, such a buffer contains (the memory representation of) a software object; valid accesses to the object do not stray outside the buffer's limits. A bounded buffer may contain pointers to other bounded buffers, and thus a data structure can be represented as a graph of bounded buffers. Often, especially in non-CHERI systems, the bounds of buffers are implicit but nevertheless conceptually are well-defined.

Scatter-gather lists are lists of bounded buffers, used to transfer data to/from non-contiguous chunks of memory. Construction and manipulation of lists can remove the need to (re)pack data as segments are inserted and removed. These lists are commonly used where data might be handled by different software – for example, if the headers on a network packet were stored separately from the payload.

The *ring buffer* extends the scatter-gather list to be a common design pattern used to represent producer/consumer communication. It is a table containing addresses and lengths of bounded buffers. Head and tail pointers point to entries in the list, wrapping around as the table becomes full. For example, on the transmit path of a network card, the producer allocates bounded buffers, fills them with data, puts their pointers at the end of the list, and updates the tail pointer. The consumer uses the head pointer to find buffers to read from; once a buffer is consumed, the head pointer is updated, indicating to the producer that the buffer can be freed. Ring buffers are common in networking and storage applications, where they can imply transfer of ownership of bounded buffers between producer and consumer.

4.2 CHERI capabilities

A *capability* is an unforgeable token of authority. *CHERI capabilities* are hardware-enforced manifestations of the bounded buffer that enforce the *principle of least privilege.* Address, limits, permissions and typing are bound together to make a bounded pointer that imposes additional properties on software:

Authenticity and integrity – Capabilities are protected by an out-of-band one-bit tag which makes it impossible for software to forge or tamper with them; capabilities can be created or modified only by processor instructions that maintain the capability properties.
Permissions – Capabilities carry permissions that restrict how they can be used (for reading, writing, code execution, used to store other capabilities, etc.).
Delegation – A capability may be passed to another piece of software in the same way a pointer is passed. Passing a capability also passes rights to access the object it points to.
Monotonicity – Bounds and rights on a capability may be reduced, but can never be increased. Hence, we can pass objects with limited rights to untrustworthy software which cannot acquire extra rights.
Typing – Capabilities can be typed to enable them to be used only by software of a matching type; other software can use them as opaque tokens but not dereference them.

In practice, a capability is a datatype twice as wide as the native pointer size (128-bit with a 64-bit address, 64-bit with a 32-bit address). It comprises bounds (which can span from a single byte to the full 64-bit address space; compression imposes modest alignment requirements for larger buffer lengths), permissions and other fields as shown in Figure 2. The 1-bit integrity tag is not accessible to software and carried separately through caches and out to a separate area of DRAM that cannot be manipulated by regular instructions [14]. A memory location whose tag is set (by construction) contains a capability; one without the tag set is an integer, which cannot be used to dereference memory.

CHERI is an architecture-neutral protection model – implementations have been designed for different architectures including RISC-V,

[1] Latterly, message-based USB and DMA-based Thunderbolt have become intertwined, especially with the USB 4 and Type-C specifications. Since the security models differ, we treat them as distinct.

MIPS, x86 and, in Arm's Morello prototype [2], ARMv8. A realisation of CHERI into a particular architecture makes certain design choices and tradeoffs; in this paper we do not assume a specific architecture (indeed multiple architectures may coexist within a single system).

4.2.1 Capabilities to mitigate software vulnerabilities. Taken together, these features make capabilities compelling replacements for integer pointers; they provide stronger security guarantees for software and also enable efficient fine-grained software compartmentalization. *Bounded pointers* and *monotonicity* mitigate a large number of traditional buffer-based vulnerabilities. *Permissions* prevent confusion between code and data objects. *Authenticity and integrity* prevent pointers from being forged, removing the need for mitigation techniques such as Address-Space Layout Randomisation (ASLR). *Typing* and *delegation* can be combined in the *sealing* mechanism, which allows construction of opaque references to memory – which can be efficiently exchanged with untrustworthy code that is sandboxed (compartmentalized), without granting the sandbox rights of access and without the need for MMU memory protection. Third-party libraries are just one example where this is desirable.

Altogether, CHERI provides a toolkit for the CPU to more directly articulate a number of higher-level protection techniques, such as efficent enforcement of object bounds [6] (and especially in higher-level languages [5]), software compartmentalisation [10], and temporal safety [8].

4.2.2 Other pointer protection schemes. CHERI is a point in a design space of *pointer protection schemes*, summarized elsewhere [14]. We can decompose CHERI as providing a number of features that might exist in other schemes: 1) pointers can be distinguished from data; 2) pointers are inherently bounded; 3) bounds are embedded inline with pointers; 4) pointer and bounds manipulations are restricted by hardware. The techniques we describe in this paper may be generalizable to other protection schemes with some degree of modification. However we have focused on CHERI as it is a concrete model being evaluated as a prototype by a mainstream commercial architecture [9]. Given a CHERI-protected CPU and memory system, cross-system DMA attacks then become a way to bypass the protection model and so this becomes a pressing threat.

4.3 The translated buffer

Another paradigm modifies the bounded buffer to a *translated buffer* by externally applying operations to the address; this is frequently used when hardware generates or modifies addresses. An access to index i within a bounded buffer of base B, which would ordinarily produce an address $A=B+i$, might be modified to a new address A' in a number of ways:

Offsetting – A fixed offset T is added to the base, giving $A'=T+B+i$.

Relocation – The index i is used with a different base R, ignoring the previous base, giving $A'=R+i$.

Sequencing – Data is *streamed* with no addressing information; the address is generated by counting elements and added to a new base R. The nth element is stored at address $A'=R+n$.

Paging – Define a page size P (e.g., 4 KiB) and set $p=log_2 P$ (e.g., 12 address bits). Take the page number from the upper bits of the address $A[63:p]$ and arbitrarily map to another prefix $f(A[63:p])$, forming a new address $A'=\{f(A[63:p]),A[p-1:0]\}$. If no valid map exists, the request is denied.

A separate operation would be required to check bounds (this check is frequently missing in existing systems). In paging-based systems, bounds are sometimes approximately enforced by rounding up the bounds to an integer multiple of the page size, and inserting "guard" gaps – i.e., addresses with no translation – between objects.

If the translation mechanism is controlled by a device that is more trustworthy, we can use such methods to limit device accesses. For example, a device might be able to make random access within a relocated bounded buffer, but not random access to any part of memory. However, we would also need a bounds limit to prevent a device running off the end of a bounded buffer *within* that relocated region.

5 EXISTING PROTECTION MECHANISMS

In this section we describe existing protection mechanisms. Many of these were not conceived for protection, but we can view them as a means of applying 'trust'. For example, a more 'trustworthy' component than the device can be used to control DMA transfers. By framing DMA communication using the memory protection concepts above, we consider how existing systems use them to apply trust.

5.1 Stream-based DMA: sequencing

Early microcomputer DMA hardware (such as in the original IBM PC [13]) did not support devices generating addresses directly. A *DMA controller* on the motherboard contained an address counter, and a peripheral card indicated via DMA request/acknowledge signals to increment the counter for each word. The software which configured the DMA controller could thus constrain the access from the peripheral to a particular memory base, although bounds checking may not have been enforced.

5.2 DMA controllers: sequencing or relocation

As a more general case than the above, today many systems-on-chip provide separate DMA controller components. To simplify chip layout, the DMA controller is used to pull data from devices rather than devices being allowed to push it. A device driver programs both the device to generate the stream and the DMA controller to transfer it to memory. DMA controllers can typically drive both stream interfaces and memory-mapped I/O, performing memory-to-memory copies. Devices used by this mechanism are typically targets – they cannot initiate their own memory accesses independently. Such DMA controllers are often mini-processors executing instructions, performing basic computation such as RGB to YUV pixel conversion on the data that flows through.

5.3 Bus-mastering DMA: no protection or limited offsetting

Bus mastering refers to peripheral devices that can independently initiate arbitrary memory transactions. While early systems had a true *bus* (i.e., the same parallel data, address and control signals connecting to every device), today it can refer to any device able to generate memory traffic over any kind of I/O interconnect. In abstract terms, bus mastering is the most general case of the problem since there may be no restrictions on the transactions that can be generated. A bus-mastering device can read or write any part of system memory.

Earlier bus-mastering hardware (e.g., ISA and PCI) had hardware limits on the number of address bits a device could generate. Thus, they did not have full access to memory, but only to a limited window (for instance, 16 MiB or 4 GiB) into available memory. Either data had to be allocated within the window, or a separate base register could move the window by providing the upper bits. Software with authority over that *offsetting* can specify which limited region the device could access.

5.4 The IOMMU: paging

The IOMMU is a implementation of page-based translation, in the same way the MMU applies paging to software. As with the MMU, the IOMMU represents the translation $f(A)$ using nested *page tables*. For example Intel's MMU implementation uses four levels of page table to translate a 64-bit address. Unlike the MMU, which holds only a single translation at any time, and multiplexes *processes* by context switching, the IOMMU is used to apply a potentially different page mapping for each I/O device. Intel's IOMMU uses an additional two levels to support 2^{16} devices. Just as the MMU is typically fronted with a Translation Lookaside Buffer (TLB) for performance, so too is the IOMMU typically equipped with an IOTLB. Hits in these buffers avoid the need to consult page tables, while misses of mapped addresses must consult memory for each layer of the page table scheme. In MMU paging, we would call the input A the *virtual address* and output A' the *physical address*; for the IOMMU, the input is the *I/O virtual address* (IOVA) and the output a physical address.

PROTECTION FROM DEVICES WITH UNTRUSTWORTHY HARDWARE AND SOFTWARE

This threat class considers devices where the attacker has control of both hardware and software. This might be a pluggable Thunderbolt device, or a black-box IP core from a third-party vendor. The protection model thus interposes on the device's memory interface, accepting memory requests from such a device and decide whether or not to allow them, acting in effect as a firewall on the device's traffic.

6 MICROCONTROLLER SYSTEMS

The above existing primitives apply trust haphazardly, since they did not consider it in their design. When designing explicitly for trust, our first point in the design space considers a microcontroller with DMA peripherals. Here, microcontrollers (MCUs) are processors that do not have a Memory Management Unit: there is no address translation so applications use physical addresses directly. There may still be memory protection techniques that restrict memory visibility, perhaps as a function of the physical address being accessed, the instruction pointer, and/or the current processor ring or exception level.

For example, an MCU might have a peripheral storage controller that performs DMA. A malicious application with access to the controller could ask it to store data from a block in memory beyond the application's reach (e.g., containing secret data). The data would then be written out to storage, using the *storage controller's* authority to access memory, and the malicious application could then read it back into memory it *can* access. (This, then, is an example of the "confused deputy problem" [11].)

6.1 MPU protection for DMA

The bounded-buffer approach suggests we might wish to constrain the DMA device to accessing only specific regions. For example, the storage controller might be allowed to read/write only from buffers set up by the filesystem library.

Memory protection from malicious software may be implemented via a Memory Protection Unit (MPU). An MPU comprises a number of memory ranges, each described by a base, length and permissions. Memory accesses from software are checked against all of the MPU entries in parallel. If the access does not match any range, an exception is generated. As the MPU is relatively expensive in terms of area and power, the number of ranges is small (e.g., 8 or 16).

We could apply this structure to DMA peripherals. When a DMA request is made by a device, it would be checked against the ranges held by the MPU. The MPU is programmed in advance by the device driver to allow the device access only to those 8 or 16 ranges. This would work for peripherals that require access to only a small number of memory ranges, as the size of the MPU provides a hard limit on the number of buffers the device can access. Given the simplicity of the microcontroller software stack, it could be acceptable to limit the number of buffers being used to within the MPU capacity.

However, we are likely to have several DMA devices. Different devices may make memory accesses in parallel, and so either a separate MPU would need to be attached to each device or accesses would need to be serialized through single MPU, where the ranges to be checked are selected based on the device ID.

6.2 Capabilities in microcontroller DMA

If the microcontroller supports CHERI capabilties, we have some additional requirements to ensure DMA does not break the software protection model. Specifically, all of the following must hold, unless the device in question holds some appropriate authority:

- Devices cannot access memory outside of permitted regions.
- Devices are unable to forge new capabilities.
- Devices are unable to expand the rights of existing capabilities.
- Devices must preserve the intentionality property, preventing valid capabilties being used for incorrect purposes.

Tag clearing. With the bounded-buffer approach, we need additional safeguards to enforce these properties. A baseline approach says that devices cannot be trusted to obey them and so devices cannot store capabilities – only data. Devices can read capabilities, but the capabilities are treated as integer data because tags are not present on the I/O interconnect. Any write to a capability in memory over DMA clears the tag, and the data becomes an integer as far as the CPU is concerned. As devices are capability-unaware, device drivers provide pointers to them as integers, and devices use integers as addresses in their memory requests. To achieve this, the I/O system must have a connection into the cache/DRAM system that enforces tag clears on writes, and prevents accesses to regions of DRAM where tags are stored. Ideally I/O writes should also invalidate cached capabilities.

This design preserves the integrity of capabilities themselves, but does not preserve the integrity of data pointed to by capabilities. The device is able to generate memory transactions for arbitrary integer addresses. Therefore, we need a mechanism to constrain accesses from devices.

6.3 Capability Search Units

With a capability-enabled microcontroller, it would be a natural step to populate the MPU with capabilities, instead of ranges containing separate bases and lengths. This would enable a more natural interface for device drivers to configure the MPU – we term this design a *Capability Search Unit (CSU)*. A device driver can allocate a buffer and program the CSU directly with the returned capability. Due to the limited size of the CSU, the device driver has to manipulate the CSU entries to find a vacant spot; instead of maintaining its own table, the device driver would be able to read capabilities from the CSU and use them directly.

This provides several benefits over the MPU approach. The primary benefit concerns *intentionality*: there is no conversion between a software representation of a bounded buffer (capability) and the register format needed to set up the MPU. For instance, the driver code needed to set up the MPU registers would be much reduced. Separately, we do not need to maintain a local representation of our bounded buffers, because the CSU already contains the valid capability. Additionally, capability writes are atomic, preventing an interval in which the MPU is only partially configured. This also makes the CSU more efficient, in that a single write is needed to configure a range, as opposed to the necessary synchronization to safely manage non-atomic configuration. Due to the limited number of MPU/CSU entries, this might enable limited use of ringbuffer structures, as long as the items between head and tail pointers never exceed the maximum number of CSU entries.

With the CHERI architecture it may be feasible to delegate part or all of the CSU table to an application. The capability model implies that software cannot derive a capability to resources it is not allowed to access. Therefore, we might give applications the ability to manage their own CSU table, with the knowledge that any capabilities the application puts in there will be objects for which it already has access rights. Thus, the CSU allows software and devices to be constrained by the same capability protection model, potentially eliminating a trusted executive to handle configuring the protection. However, the primary shortcoming of the MPU remains: each memory access requires a number of parallel lookups, consuming both area and power.

7 TRANSLATION-BASED PROTECTION

7.1 IOMMU protection

Today, the IOMMU is the primary means of protecting systems from malicious devices. Any memory requests from devices are checked by looking them up in the IOMMU page table to translate the device's I/O virtual address into a physical address. A growing literature [3, 18] exploits problems with IOMMU-based protection. First, the IOMMU protects only pages rather than finer-grained objects. Second, IOMMU page-table walks are inherently expensive, and thus translations need to be cached in an IOTLB. Revoking translations in the IOTLB requires (in the worst case) a linear search of the IOTLB, which is relatively slow; for this reason, revocation is typically asynchronous, which can open up a temporal vulnerability.

More generally, page-based translation is not optimized for the I/O use case. The MMU is primarily designed for software *isolation*; programs from two users cannot access each others' memory, and shared memory between programs is relatively static. MMU mappings are often long-lived – the mapping may exist for as long as a program does, which might be seconds or days. Setup and teardown of a software process is a relatively heavyweight operation. While the MMU page table is swapped at every context switch, TLB entries maintain an address-space ID (ASID), so that the cached translations do not have to be flushed on context switches. In this way, the MMU lives mostly in a steady state that has acceptable performance.

I/O is different in that it is all about *communication*, and isolation is a secondary concern. For example, the obvious implementation would create a mapping for every network packet and delete the mapping once it is handled. This might generate millions of mappings and revocations per second, which would have serious performance degradation, especially given worst-case 6 memory operations for an IOTLB miss. A number of schemes [1, 17, 19, 20, 23] have been proposed to mitigate this performance bottleneck for particular use cases, but in the absence of a general solution. This may be why existing systems do not utilize the IOMMU to its fullest extent, providing a motivation to develop better hardware structures.

7.2 Address Lookup Tables (ALUTs)

The original motivation for the IOMMU was to delegate peripherals to virtual machines. Here we need to provide to the guest operating system a physical memory map matching its target hardware. Because the guest OS may be entirely unaware of hypervisors, the IOMMU permits construction of an arbitrary translation that mimics the guest physical address space configured by the hypervisor's Second-Level Address Translation (SLAT) page tables. With common SLAT and IOMMU page tables, the guest's device driver can configure a delegated peripheral with the correct guest physical addresses.

When protection is separate from virtualization, we do not need the full IOMMU structure. Other approaches may require support in the device driver, but result in simpler and more efficient hardware. We can also design for the observation that devices typically use only a tiny part of their 64-bit IOVA space: gigabytes, not petabytes.

We propose the Address Lookup Table (ALUT), in which we repurpose high-address bits to make a one-level non-paged translation scheme, more efficient than the IOMMU, that applies bounded-buffer semantics. The ALUT divides 64-bit I/O virtual addresses into two parts, based on a chosen split s. $I = A[63 : s]$ is the *index* and $O = A[s-1 : 0]$ is the *offset*. The index selects one of 2^{64-s} buffers of 2^s bytes long. s is chosen based on the requirements of the system – for example, chosing $s = 40$ would provide 2^{24} regions of 1 TiB each.

When a device makes a memory request, the I part of the address is looked up in its ALUT, a single-level in-memory translation table. This provides a 64-bit physical base B that is then added to the offset to produce a physical address $A' = B + O$. For efficiency, the ALUT entry containing the mapping $I \mapsto B$ may be cached using the existing last-level cache, or an additional caching unit.

We need an additional bounds check to prevent overrunning from one buffer into another. The ALUT entry also includes a bounds O_{max}, so that in parallel with the I mapping it also checks $O < O_{max}$. If there is a chance O_{max} could be near 2^s, to retain intentionality we might populate only alternate entries in the ALUT – this would prevent an overrun of buffer n from unintentionally generating valid addresses in buffer $n+1$. Permissions (read/write/etc.) would also govern accesses via the corresponding ALUT entry. Additionally, the ALUT table itself would be bounded, so a register would contain the number of slots present in the ALUT. This means the ALUT needs only as many entries as are in use, rather than taking a fixed memory allocation (substantial when small s implies many regions). The base of the ALUT would be determined from the device ID or some other means of disambiguating devices.

The ALUT offers a number of advantages over an IOMMU for I/O. First, the translation structure can be more efficient as there is no need for the ALUT to be sparse, i.e. it does not need to be multi-level. If the number of buffers in play is small, the ALUT table size is bounded and blocks of ALUT entries are likely to be cached. Second, the memory access time is bounded, in that only a single memory request is required. Each device would require a different ALUT base, but these could be provisioned from a small fast-lookup table, given that there are likely to be on the order of tens or hundreds of devices. Revocation of an entry would be a simple write of a null value to the ALUT structure, which would automatically synchronize the last-level cache.

7.3 Capability Lookup Tables (CLUT)

An ALUT represents a table of translated buffers – each ALUT entry combines physical prefix, bounds and permissions. We could build a version of an ALUT using capabilities, a *Capability Lookup Table (CLUT)*.

Each entry, consisting of base B, bounds and capability permissions, would be replaced with a capability. Instead of adding the offset O to the translated base B, index I in the CLUT would provide a capability C bearing address a and bounds b. The physical address would be computed $A' = a + O$ and checked against the bounds b and permissions.

Allowing the CLUT to hold capabilities would mean a device driver could easily maintain the translation structure. A device driver wishing to give the device a buffer to access would pick a slot n in the CLUT and write a capability there. As the device is capability unaware, the driver would need to pass to the device an IOVA comprising the offset O from the capability and the upper bits being the slot index I. This conversion is relatively efficient in software (a handful of instructions).

Such an approach takes the benefits of the ALUT while adding better interworking with capability-enabled software, and smooths intentionality – mapping/unmapping is a lightweight operation when the CPU is already using capabilities.

8 USER ADDRESS SPACES WITH I/O

Today there is increasing demand for unprivileged software to have direct access to hardware devices. For instance, a database application might wish to directly control a flash storage device without having to pay a latency penalty of indirect accesses via a filesystem stack in the kernel. Today, systems typically do this by sharing MMU and IOMMU page maps, which allow the device to share the same page map as a user process. Here we examine how a similar effect might be achieved with the CLUT.

8.1 Virtually-addressed CLUTs

One challenge that arises with the CLUT design is that, in the CHERI model, capabilities belong to a particular MMU-governed virtual address space. This means that capability addresses used to program the CLUT, by both unprivileged programs and the kernel, will be virtual, while addresses coming out of the CLUT should be physical. One possible solution would be to pre-translate capabilities in the CLUT so that they refer to physical addresses. However, this would not work for capabilities pointing to buffers larger than the page size, because the buffer might span disjoint physical pages. Alternatively, CLUT entries could be augmented with an address space identifier (ASID) that corresponds to the page table base of the process that programmed the entry. An IOMMU, distinct from the CLUT, would use the ASID's page table base to translate the virtual address produced by the CLUT into the appropriate physical address space. This design would be analogous to how application-processor capabilities work in tandem with the MMU: the CLUT provides protection, while the IOMMU provides virtualization.

Using a CLUT in this way might at first seem to be less efficient than using an IOMMU alone because it adds an additional memory lookup. However, by removing the burden of protection from the IOMMU, this design may actually offer positive performance characteristics. Specifically, IOMMU mappings in this scheme will be mostly static and can be effectively cached in an IOTLB using superpages and ASIDs. CLUT mappings will be more dynamic; they will be created and destroyed as mappings for I/O buffers are created and torn down, potentially millions of times per second for high-performance peripheral devices. Dynamic management of IOMMU mappings and IOTLB entries is known to be a significant performance bottleneck in I/O-heavy workloads. In contrast, the single-level translation structure of the CLUT simplifies the caching of translations and greatly reduces the penalty of a cache miss.

9 CAPABILITIES WITHIN DEVICES WITH TRUSTWORTHY HARDWARE

The second threat class considers devices whose hardware can be deemed trustworthy – for example, built on the same SoC from a trustworthy vendor. In this instance we wish to defend against the case where the firmware is compromised. Such devices can enforce the capability protection model on their own firmware, but we must consider how they interact with other parts of the system. We consider a spectrum of use cases.

9.1 Capabilities within descriptor rings

A typical network device interacts with its device driver through descriptor rings – ring buffers containing integer pointers and bounds corresponding to packet data stored in host memory. It would be natural to replace these with ring buffers of capabilities. The pointer to the ring buffers would also be capabilities. In this case, the head and tail pointers might not be capabilities themselves, but might be offsets within the ring capability. This would provide a common representation between device driver and ring buffer hardware.

At a basic level, devices can use the capabilities as simply a different format of descriptor entries. Internally, addresses and lengths would be used as they currently are. In this instance we would still enforce tag-clearing on writes to capabilities from devices. Becuase the device can craft arbitrary addresses on its requests, we would still need an IOMMU, CSU or CLUT to constrain those requests.

9.2 Internal use of capabilities by devices

A deeper integration would consider the use of capabilities internal to devices. For example, a Wi-Fi or Bluetooth adapter might run a software stack aimed at handling the air protocol. Like any other software stack, it may be subject to programming errors that could make it vulnerable to attacks over the air. Such designs are amenable to the CHERI protection model just as much as the application processor is.

Often, device processors are attached to private memories. In this instance, it is feasible to segregate the device processor and memory from the CPU and system DRAM. A separate capability root exists within the device. It is necessary to ensure that capabilities cannot flow from the CPU to the device processor or vice versa, which might cause confusion or aliasing. This can be achived by not allowing tags to pass over the I/O interconnect in either direction.

Such protection is independent of whether the device is trusted by the application OS, and so an IOMMU, CLUT or other protection mechanism might still be required to constrain it. If the hardware was trustworthy, the device's access can also be constrained by providing it with a limited initial set of capabilities, preventing software from gaining additional access.

9.3 Using capabilities in shared memory spaces

Devices that lack private memory may use capabilities for internal purposes – such as on-SoC devices sharing an external DRAM.

In a shared memory, we need non-interference between application-processor and device-processor capabilities. A capability used by one processor should not be able to be confused for another. This can be achieved using memory segregation – making sections of address space inaccessible to all but one client. If not achieved statically (at manufacture time), it can be enforced by IOMMU and MMUs being configured to never share pages.

Another scheme would allow capabilities to flow freely into memory, but identify them with a *color*. A color could be implemented using spare bits inside the existing 128-bit format. A processor produces capabilities with a specific color, and an exception is raised if it attempts to interact with a capability of a different color.

Both schemes achieve isolation between the application and device processors. To transfer data between them, a separate mechanism would be needed to translate data from one address space or color to another. This would be a more privileged path that would need to apply suitable scrutiny to such conversions. Care would be needed to avoid this path becoming a bottleneck, especially since much I/O traffic already crosses such domain boundaries.

10 EVALUATION

Security is notoriously hard to quantify, and this is a novel field without existing metrics. However we can sketch the tradeoffs between schemes in a number of ways (Table 1). We can represent the *expressiveness*, for example the granule of accuracy to which objects can be referred. More subjectively, we can consider the *ease of use*, for example whether the programmer can use the same representation when configuring protection hardware as is used in software. This reduces the likelihood of bugs when translating representations. Additionally we consider whether updates are atomic, which avoids some temporal vulnerabilities. We can also count the cost, here primarily in terms of the number of memory requests that need to be made and their associated latency and power costs (noting that these costs are mostly unaffected by word size). We also evaluate whether the approach would be scalable to many buffers or many devices.

	Protection methods							
	Use bydevice sw			Interpose without translation		Interpose with translation		
Properties	**Integer ptr**	**Bounded ptr**	**Capability**	**MPU**	**CSU**	**IOMMU**	**ALUT**	**CLUT**
Spatial enforcement	×	sw	hw	hw	hw	hw	hw	hw
– granularity (bytes)	×	1	1	1	1	4096	1	1
Atomic manipulation	×	×	✓	×	✓	×	×	✓
Common representation between hardware and software	✓	×	✓	×	✓	×	✓	✓
Memory request overhead (worst case)	0%	100%	0%	0%	0%	600%	100%	100%
Scalable to many buffers/devices	✓	✓	✓	×	×	✓	✓	✓

Table 1: Comparison of schemes for controlling memory access from devices.

Table 1 highlights the weaknesses of existing schemes. Looking at the schemes to protect from untrustworthy hardware, the MPU provides sufficient spatial protection, but needs privileged drivers to program it. The CSU can share the same capabilities as used by application code, providing better *intentionality* and direct manipulation by the application. Both suffer from a lack of scalability that means they are only usable in small systems.

Moving to translation-based approaches, those described are more scalable. The IOMMU has a very high worst-case overhead (600% for Intel's IOMMU) – in practise IOTLB caching will mitigate the cost to an application-dependent degree. The ALUT and CLUT impose a better bound on the worst-case cost, and are amenable to the similar caching strategies to reduce this further. They both provide byte granularity compared with the page granularity of the IOMMU, and they can be designed to use the same bounded-buffer representation for setup as that used by software. The main distinction is that the CLUT applies the capability manipulation rules to CLUT entries, which means that it is not possible to construct mappings without software having valid capabilities for those mappings. Thus the CLUT can ensure that rights transfer correctly and safely from software to hardware, without the necessity for management by privileged software.

When programming trustworthy device cores, capabilities allow enforcement of the bounded buffer model but additionally this transfer of rights – both from the application core to the device core, or between internal software components. Thus malicious software is constrained by the capability model in a way it is not using software bounded buffers.

While this analysis is purely a paper exercise, the capability-based solutions provide a good blend of transferring programmer intent from software into hardware, avoiding temporal vulnerabilities, and limiting costs compared with existing solutions.

11 FUTURE WORK

The above evaluation is obviously only a sketch of architectural costs, not a picture of a real system. However, it is valuable to apply such techniques to consider the tradeoffs inherent in different schemes, a prerequisite in planning implementation strategies.

The next steps consider implementing some of these solutions and assessing their behavior. This involves an implementation in hardware and software, across multiple parts of an SoC, including in the operating system, device drivers and device firmware. Such an implementation would test some of the properties we quantified in Table 1 - in an implementation we would discover whether the ease of programmability properties hold, or whether other issues are encountered. The costs outlined here are largely theoretical costs; an implementation would consider whether they could be mitigated by caching or other strategies, and whether certain solutions are more amenable than others. A scheme can be properly assessed only when its real-world behavior and amenity to cost mitigation have been measured within the constraints of an implementation.

Another question is whether a scheme is virtualizable, not just in terms of hypervisors but also to present a virtual interface that abstracts away hard limits in the implementation. While some of the schemes described have no hard limits, virtualizing over limited hardware resource would require further study.

Being a wide solution space, this is a signficant amount of work, however it would allow for focusing on particular points of the design space that are worthy of more substantial engineering optimization.

12 CONCLUSION

In this paper we introduce and systematize the relatively understudied problem of protecting I/O from malicious devices. We have outlined a number of approaches that may provide efficient and expressive protections. We have also considered how these protections might interwork with emerging capability hardware, and where that might improve their efficiency and intentionality.

Malicious DMA is a complex and emerging problem, in which the requirements and threat models are often disparate and cross-cutting. Since the problem has received little attention to date, many of the foundations are yet to be established. Our work should allow mapping of the space and refine the promising architectural directions that are worthy of further study.

ACKNOWLEDGMENTS

This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contracts FA8750-10-C-0237 ("CTSRD") and HR0011-18-C-0016 ("ECATS"). The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. This work was also supported by EPSRC EP/R012458/1 ("IOSEC"). We also acknowledge Arm Limited and Google Inc. for their support.

REFERENCES

[1] Nadav Amit, Muli Ben-Yehuda, and Ben-Ami Yassour. 2010. IOMMU: Strategies for Mitigating the IOTLB Bottleneck. In *6th Annual Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA).*

[2] Arm Limited. 2020. Arm Architecture Reference Manual Supplement Morello for A-profile Architecture.

[3] Muli Ben-Yehuda, Jimi Xenidis, Michal Ostrowski, Karl Rister, Alexis Bruemmer, and Leendert Van Doorn. 2007. The price of safety: Evaluating IOMMU performance. In *Ottawa Linux Symposium (OLS).* 9–20.

[4] Gal Beniamini. 2017. Over The Air: Exploiting Broadcomfis Wi-Fi Stack. (April 2017). https://googleprojectzero.blogspot.co.uk/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html

[5] David Chisnall et al. 2017. CHERI JNI: Sinking the Java Security Model into the C *(ASPLOS '17).* 569–583.

[6] Brooks Davis et al. 2019. CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment. In *ASPLOS.*

[7] Loïc Duflot, Yves-Alexis Perez, Guillaume Valadon, and Olivier Levillain. 2010. Can you still trust your network card? *CanSecWest/core10* (2010), 24–26.

[8] Nathaniel Wesley Filardo, Brett F. Gutstein, et al. 2020. Cornucopia: Temporal Safety for CHERI Heaps. In *IEEE Symposium on Security and Privacy.*

[9] Richard Grisenthwaite. 2019. A Safer Digital Future By Design. (Oct. 2019). https://www.arm.com/blogs/blueprint/digital-security-by-design

[10] Khilan Gudka et al. 2015. Clean Application Compartmentalization with SOAAP *(CCS '15).* Association for Computing Machinery, 1016–1031.

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

[12] Trammell Hudson and Larry Rudolph. 2015. Thunderstrike: EFI firmware bootkits for Apple MacBooks. In *Proceedings of the 8th ACM International Systems and Storage Conference.* ACM.

[13] International Business Machines Corporation. 1981. IBM Personal Computer Technical Reference Manual. (Aug. 1981).

[14] A. Joannou, J. Woodruff, et al. 2017. Efficient Tagged Memory. In *2017 IEEE International Conference on Computer Design (ICCD).* 641–648.

[15] Yoongu Kim, Ross Daly, et al. 2014. Flipping Bits in Memory without Accessing Them: An Experimental Study of DRAM Disturbance Errors *(ISCA '14).*

[16] Slava Makkaveev. 2020. Pwn2Own Qualcomm Compute DSP for Fun and Profit. *DEF CON Safe Mode* (Aug. 2020).

[17] Moshe Malka, Nadav Amit, and Dan Tsafrir. 2015. Efficient Intra-Operating System Protection Against Harmful DMAs. In *13th USENIX Conference on File and Storage Technologies (FAST 15).*

[18] A. Theodore Markettos, Colin Rothwell, et al. 2019. Thunderclap: Exploring Vulnerabilities in Operating System IOMMU Protection via DMA from Untrustworthy Peripherals. In *Network and Distributed Systems Symposium (NDSS).*

[19] Alex Markuze, Adam Morrison, and Dan Tsafrir. 2016. True IOMMU Protection from DMA Attacks: When Copy is Faster Than Zero Copy *(ASPLOS '16).*

[20] Alex Markuze, Igor Smolyar, et al. 2018. DAMN: Overhead-Free IOMMU Protection for Networking *(ASPLOS '18).*

[21] Nathaniel Mott. 2018. Intel Fixes Yet Another Flaw In Management Engine. (Sept. 2018). https://www.tomshardware.com/uk/news/intel-management-engine-flaw-security,37794.html

[22] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe Performance for End Host Networking. In *SIGCOMM '18.* Association for Computing Machinery.

[23] Omer Peleg, Adam Morrison, Benjamin Serebrin, and Dan Tsafrir. 2015. Utilizing the IOMMU Scalably. In *2015 USENIX Annual Technical Conference.*

[24] Tara Seals. 2019. 'USBAnywhere' Bugs Open Supermicro Servers to Remote Attackers. (Sept. 2019). https://threatpost.com/usbanywhere-bugs-supermicro-remote-attack/147899/

[25] Patrick Stewin and Iurii Bystrov. 2013. Understanding DMA Malware. In *9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'12)* (Heraklion, Crete, Greece).

[26] Arrigo Triulzi. 2008. Project Maux Mk. II, I 0wn the NIC, now I want a Shell. In *Proceedings of PacSec.*

[27] Robert N. M. Watson et al. 2020. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8).* Technical Report UCAM-CL-TR-951. University of Cambridge, Computer Laboratory. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-951.pdf

[28] J. Woodruff, A. Joannou, et al. 2019. CHERI Concentrate: Practical Compressed Capabilities. *IEEE Trans. Comput.* 68, 10 (Oct 2019), 1455–1469.