Adaptive CHERI Compartmentalization for Heterogeneous Accelerators

Jianyi Cheng
University of Edinburgh
United Kingdom
jianyi.cheng@ed.ac.uk

A. Theodore Markettos
University of Cambridge
United Kingdom
theo.markettos@cl.cam.ac.uk

Alexandre Joannou
University of Cambridge
United Kingdom
alexandre.joannou@cl.cam.ac.uk

Paul Metzger
University of Cambridge
United Kingdom
paul.metzger@cl.cam.ac.uk

Matthew Naylor
University of Cambridge
United Kingdom
matthew.naylor@cl.cam.ac.uk

Peter Rugg
University of Cambridge
United Kingdom
peter.rugg@cl.cam.ac.uk

Timothy M. Jones
University of Cambridge
United Kingdom
timothy.jones@cl.cam.ac.uk

Abstract

Hardware accelerators offer high performance and energy efficiency for specific tasks compared to general-purpose processors. However, current hardware accelerator designs focus primarily on performance, overlooking security. This poses significant security risks due to potential memory safety violations that can affect the entire hardware system. Existing methods either rely on Input-Output Memory Management Units (IOMMUs) for memory isolation between memory pages, leading to vulnerabilities in intra-page memory accesses, or modify an accelerator architecture for specialized memory protection, which requires significant effort and cannot scale across multiple diverse applications.

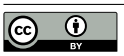
In this paper, we propose a general method for fine-grained memory protection in heterogeneous systems *without modifying accelerator architectures*. We extend the Capability Hardware Enhanced RISC Instructions (CHERI) from CPUs to an adaptive hardware interface named CapChecker. The CapChecker imports capabilities from the CPU and guards memory accesses at the pointer level from CHERI-unaware accelerators as if they were CHERI-aware natively. Over a set of benchmarks on hardware accelerators in a heterogeneous system, our approach achieves fine-grained memory protection, with a 1.4% performance overhead compared to CHERI-unaware accelerators on average.

CCS Concepts

• **Security and privacy → Hardware-based security protocols**;
• **Hardware → Integrated circuits**.

Keywords

Hardware Security, Memory Safety, Heterogeneous Systems



This work is licensed under a Creative Commons Attribution 4.0 International License.
ISCA '25, Tokyo, Japan
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1261-6/2025/06
https://doi.org/10.1145/3695053.3731062

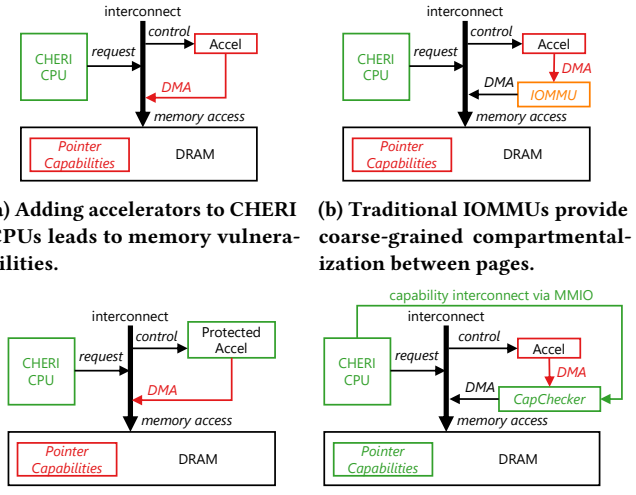
ACM Reference Format:
Jianyi Cheng, A. Theodore Markettos, Alexandre Joannou, Paul Metzger, Matthew Naylor, Peter Rugg, and Timothy M. Jones. 2025. Adaptive CHERI Compartmentalization for Heterogeneous Accelerators . In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3695053.3731062

1 Introduction

Hardware accelerators exploit the unique characteristics of specific workloads and allow a task to be executed with better performance or energy efficiency compared to general-purpose processors, such as CPUs. For example, Neural Processing Units (NPUs) are amenable to matrix operations and machine learning workloads [21, 33]; and other custom accelerators are tailored to domain-specific applications, such as Monte Carlo simulations [54, 70] and quantum simulations [10, 71]. Today's hardware accelerators are widely used in both cloud servers, such as F1 instances on Amazon AWS [2] and Brainwave on Microsoft Azure [3], and embedded systems, such as matrix accelerators in TinyML systems [36, 57] and router accelerators in networking systems [61].

A major concern regarding computing systems with hardware accelerators is security. While systems make increasing use of accelerators for high performance and energy efficiency, the design of existing accelerators often overlooks memory safety. Since accelerators are outside the view of contemporary operating systems, one could gain control of an accelerator, potentially attacking the system. Reported vulnerabilities include attacks by an accelerator task targeting both CPU tasks [42–45] and other accelerator tasks [67]. Existing software solutions often require complex implementation [30, 31, 38, 40, 64], while hardware solutions provide a simpler 'always-on' mechanism to ensure memory safety [20, 78].

Existing hardware solutions for system-level memory protection still face vulnerabilities due to *coarse granularity* and *protection heterogeneity*. First, traditional heterogeneous systems rely on Input-Output Memory Management Units (IOMMUs) for memory



(a) Adding accelerators to CHERI CPUs leads to memory vulnerabilities.

(b) Traditional IOMMUs provide coarse-grained compartmentalization between pages.

(c) Mismatches in protection mechanisms between devices face system vulnerabilities.

(d) We propose CapChecker for pointer-level compartmentalization.

Figure 1: Our work adapts existing CHERI-unaware accelerators into a CHERI-aware system without modifying them.

protection, as illustrated in Figure 1(b). IOMMUs provide memory isolation between memory pages, independent of the software objects being computed on accelerators. IOMMUs also lead to significant overhead in area and performance, often unaffordable for embedded systems and leading to a system without protection, as illustrated in Figure 1(a). Second, there has been interest in adding hardware memory protection to accelerators, such as NPUs [20], as illustrated in Figure 1(c). They tailor the memory protection scheme to a specific accelerator architecture, such as scratchpad memory and Network-on-Chip (NoC) interconnects. However, the mismatch between memory protection schemes on the CPU and accelerator could lead to memory vulnerabilities. For example, adding a new scheme may destroy the integrity of the existing scheme if they share memory.

In order to overcome these challenges, a cohesive memory safety scheme on heterogeneous systems is needed. We make the first step towards this with two restrictions. First, we focus on traditional hardware accelerators that do not have dynamic memory management. Advanced general-purpose accelerators today, such as GPUs, have complex memory management [24–26] and are out of the scope of this work. Second, our hardware memory protection focuses on spatial memory safety. For example, a task should not be able to access any data computed by another independent and concurrent task, regardless of whether either is running on a CPU or accelerator. For temporal safety, we rely on trusted software drivers to manage the accelerator memory, so errors such as use-after-free cannot be exploited by the application.

A potential hardware solution to improve system-level memory safety is Capability Hardware Enhanced RISC Instructions (CHERI) [78]. CHERI replaces pointers with the *capability*, a hardware-enforced pointer type with bounds and permissions. Capabilities restrict that a pointer can only access the resources it needs and use that access in a deliberate way, leading to fine-grained memory protection. Existing work on CHERI focuses on memory protection in CPUs and requires significant architectural modifications in accelerators to construct a CHERI-aware heterogeneous system. We seek an adaptive method between CHERI CPUs and existing hardware accelerators with minimal architectural modification, and this protection method should be general, accommodating diverse accelerator architectures.

In this paper, we propose an adaptive method that compartmentalizes accelerator tasks using CHERI without architectural modification of accelerators, forming a CHERI-aware heterogeneous system. Such a CHERI-aware system contains a component named *CAPability Checker (CapChecker)* that holds a hardware repository of capabilities from the CHERI CPU and guards memory accesses from the accelerator. The CapChecker provides protection through the memory interface of accelerators as if they used CHERI capabilities natively, as illustrated in Figure 1(d). The integration of CapChecker restricts the memory behaviors of arbitrary CHERI-unaware black-box accelerators when working with a CHERI-aware CPU. Our main contributions are as follows.

- A threat model that describes memory vulnerabilities in existing heterogeneous accelerators and a formalization of memory protection design for heterogeneous systems;
- An efficient prototype system with a lightweight CapChecker and a software driver to provide fine-grained compartmentalization, down to the pointer level; and
- Over an accelerator benchmark set, our proposed system achieves pointer-level memory protection, and our evaluation shows that the CapChecker leads to small performance overheads depending on the accelerator architecture.

The rest of the paper is organized as follows. Section 2 provides a motivating example of potential memory vulnerabilities in existing heterogeneous systems. Section 3 provides background on CHERI capabilities and existing I/O protection methods. Section 4 describes our threat model and problem formalization. Section 5 demonstrates our hardware system prototype. Section 6 evaluates the effectiveness of our work in security and scalability. Section 7 reviews related work in the field.

2 Motivating Example

Here, we present a motivating example of potential memory vulnerabilities in existing heterogeneous systems. For simplicity, we take video decoders as an example, which have been widely studied for hardware acceleration [66, 80], but the security challenges are similar to those of other accelerators. In Figure 2, a video application calls a decoder function running on the accelerator. The hardware accelerator is programmable and may run user-supplied code, such as customized decoding algorithms. The red arrow illustrates a potential memory attack that involves unauthorized memory accesses and capability forging. An attacker runs an application that launches a malicious task 'eavesdropper' on the accelerator, exploiting the following two memory weaknesses.

1) Memory Protection Granularity. The eavesdropper function may access an unauthorized memory region to steal data. For example, the eavesdropper may steal a screen-sharing session from a confidential video call. In server-class systems, existing works rely on

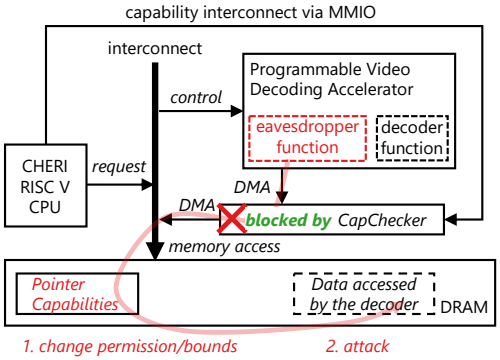


Figure 2: An accelerator task may access unauthorized data or even overwrite capabilities for CPU tasks. The latter makes existing CPU capabilities forgeable. Our CapChecker solution blocks such attacks and improves memory safety.

efficient IOMMUs and improve the management of translations due to the overhead of mapping and unmapping pages [11, 46, 49, 50, 56]. Still, the memory protection granularity remains at the memory page level and faces vulnerabilities in unauthorized access between buffers in the same memory page. For embedded systems, this leads to a more significant security challenge due to the absence of IOMMUs, where the whole memory, including the OS, is reachable by the attacker.

2) Hardware Protection Heterogeneity. Existing protection on CHERI-aware CPUs no longer holds with a CHERI-unaware accelerator in the same system. One of the essential requirements for CHERI is that pointer capabilities must be unforgeable, where the user cannot increase the protections of an object. CHERI-unaware accelerators in a CHERI-aware system must not be allowed to forge pointers. If the capabilities of pointers are reachable by the eavesdropper function, it may alter the permissions or bounds of a pointer used by a CPU task, destroying the existing CHERI system. Existing CHERI hardware systems keep CHERI-unaware accelerators entirely outside the capability model. For example, the Arm Morello system [22] prevents accelerators from writing valid capabilities.

In order to address these two problems, our protection goals are: 1) achieving fine-grained compartmentalization of accelerator tasks; and 2) matching the protection scheme on the CPU and the accelerator. Our design goals are: 1) the approach must be adaptive to diverse accelerators; and 2) no modification of existing accelerator architectures is required. In the rest of the paper, we will show how our approach extends CHERI beyond CPUs and achieves pointer-level compartmentalization among accelerator tasks.

3 Background

3.1 CHERI Capabilities

Capabilities are defined as unforgeable, delegatable tokens of authority [18, 62]. CHERI is an embodiment of the capability model for memory pointers, enforcing the *principle of least privilege* and the *principle of intentional use*. Software is built from individual objects (variables, data structures, pieces of memory, code functions)

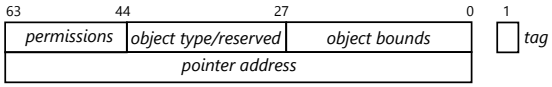


Figure 3: 128-bit CHERI capability format for 64-bit addresses [75]; this specific layout proposed by draft RISC-V standard [7].

Table 1: Comparison with traditional hardware protection methods for controlling memory accesses from devices, which motivates us to extend CHERI capabilities from CPUs to hardware accelerators.

Properties	No method	IOPMP [23]	IOMMU [11]	CHERI [47]
Spatial enforcement	✗	✓	✓	✓
– granularity (bytes)	-	1	4096	1
Common object representation	✗	✗	✗	✓
Unforgeability	✗	✗	✗	✓
Scalability	✓	✗	✓	semi
Address translation	✗	✗	✓	optional
Suitable for microcontrollers	✓	✓	✗	✓
Suitable for application processors	✓	✗	✓	✓

and CHERI capabilities allow a pointer to carry metadata about the object to which it refers. Integer addresses are replaced with a pointer type that contains metadata describing the memory region's bounds and permissible actions. The CHERI model imposes constraints on how the pointer may be dereferenced and manipulated, and it protects its integrity. Unlike conventional memory pointers, which offer unfettered access to memory, operations are restricted to enforce a model in which rights to memory can never be increased, only maintained or reduced.

A CHERI capability can be thought of as a 'fat pointer' that, in addition to pointing to a location in memory, is checked every time it is dereferenced to access memory. CHERI augments the CPU ISA to ensure that capabilities are always derived from valid manipulations of other capabilities. Corrupted capabilities cannot be dereferenced, such that the rights granted by capabilities are non-increasing. A CHERI capability contains the type of its object, the permissions determining how the pointer can be used, address bounds, and a tag bit indicating if the capability is valid [78]. On a 64-bit system, this leads to a 128-bit data structure plus the out-of-band tag bit, as illustrated in Figure 3, with bounds compressed using a scheme similar to floating point [77].

CHERI capabilities have achieved fine-grained memory protection for CPUs and verified their security both in software and hardware [8, 59]. The tag bit in Figure 3 is essential for validating the capabilities and is only preserved inside the CHERI world, i.e., the CPU side, to ensure the 'unforgeability' of CHERI systems. In this work, we extend the CHERI capability model to hardware accelerators so that existing hardware accelerators can be adapted to existing CHERI CPU systems, forming a CHERI-aware heterogeneous system. Our work focuses on the extension, and the memory protection granularity depends on the CHERI capabilities themselves.

3.2 Hardware Protection in I/O Memory

Existing Input/Output (I/O) memory protection for heterogeneous systems is limited. Table 1 compares traditional hardware protection methods with our approach in memory safety and scalability. First, a vanilla system with no protection has the simplest architecture with high performance. Microcontrollers can have a (Physical) Memory Protection Unit (MPU or PMP) which checks CPU memory requests in parallel against regions with different policies and permissions. The natural extension to devices is the RISC-V IOPMP [23]. However, the necessary associative lookup is very expensive in both area and power, and so IOPMP implementations may be limited to single-digit or teen numbers of regions [53, 76].

A more advanced approach is using IOMMUs, which have been widely used in application processors. IOMMUs [11] provide address translation that enables virtualization and protects physical memory pages. Compared to IOPMPs, IOMMUs cost more area and latency because of their complex hardware architectures and the need to fetch and cache translations from main memory. On the security side, the memory protection provided by an IOMMU is more coarse-grained, depending on the page sizes.

Our CHERI approach provides both unforgeable capabilities and fine-grained memory protection. Existing CHERI work supports capabilities with MMUs. Our work can be applied to general heterogeneous systems, ranging from microcontrollers to application accelerators. CHERI's philosophy on the CPU is to deconflate protection from translation [78], leaving translation where necessary to an MMU. No longer depending so heavily on the MMU helps to reduce dynamic MMU costs such as TLB shootdowns. Similarly, we deconflate protection from translation for accelerators. Where address translation is still required, such as for address remapping or defragmentation, some minimal IOMMU may still be required. By taking the IOMMU out of the protection path, it can potentially be substantially simplified – for example, replacing page-based translation and IOTLB caching with a (quasi-)static remapping.

4 Threat Model and Problem Formalization

4.1 Threat Model

We now present a threat model to describe the security problems of existing heterogeneous systems. A heterogeneous system typically contains three hardware components: a CPU, a hardware accelerator, and memory. Modern accelerators that natively support CHERI do not yet exist, and require significant engineering effort to build from scratch as baselines. As the first attempt at extending CHERI beyond CPUs, we rely on following assumptions to simplify our security analysis:

(1) the CPU used in the system is already protected by the CHERI capability protection model [75];
(2) the accelerator does not perform dynamic memory utilization, such as memory allocation/deallocation – these must be handled by a CPU task; and
(3) the OS kernel, driver and hardware components are all trustworthy.

Assumption 2 excludes the analysis of modern accelerators that support dynamic memory management, such as GPUs and TPUs. Even with assumption 2, the proposed systems are still realistic and representative of today's non-GPU accelerator systems. For example, Cerebras [4] has a spatial accelerator architecture that contains a large number of accelerator cores connected by reconfigurable switches. The cores are highly programmable to accelerate computation-intensive tasks by exploiting large-scale data parallelism, such as LLM training and inference. However, these cores do not support dynamic memory management and rely on the CPU to instantiate pointers for data access. The Cerebras example can be represented by the `backprop` benchmark in Section 6 as part of its training workload, but the security problem remains the same. This memory vulnerability is also observed in other variants of spatial accelerators, such as AMD Xilinx Versal devices [6] and Tenstorrent accelerators [5].

We make assumption 3 regarding the drivers due to two reasons. First, we assume our CapChecker driver implementation is correct and bug-free. We rely on this assumption to save verification effort on our prototype. Since our work does not modify accelerators, we consider existing malicious software drivers to be out of scope. Still, a secure system should ensure that the driver cannot use accelerators to do anything it could not already do.

The proposed threat model considers two actors in common use cases: *general users* who write unverified code or orchestrate third-party malicious code and run it on accelerators; and *attackers* who intentionally write accelerator code with unauthorized memory accesses to observe or modify other concurrent tasks in the system, such as the example shown in Figure 2. This means that the application code cannot be trusted, and each task running on a CPU or an accelerator could exhibit arbitrary memory behavior.

Although CHERI provides capability checks for pointers used by CPUs, the pointers used by accelerators remain vulnerable. While our work tries to address this weakness using fine-grained compartmentalization, we do not consider side-channel attacks and physical attacks, because they are implementation-specific rather than architecture-specific.

4.2 Problem Formalization

We now show how to translate the proposed threat model into a general system protection problem. We present the first problem formalization for designing a CHERI heterogeneous system. In the formalization, we focus on pointers because both memory accesses and control flow accesses are passed using pointers.

For a general heterogeneous system, tasks are mapped to various computing targets. For simplicity, here we describe the design problem for a system that contains a CPU `P` and an accelerator `A`, but our formalization is general and can scale to any number of computing targets. Let E be the set of pointers accessed by a set of concurrent computing tasks. Each pointer is represented as a three-tuple $(b, c, t) \in E$. $b \subseteq \mathbb{N}$ denotes the allocated address space based on the source. $c \subseteq \mathbb{N}$ denotes the reachable address space restricted by the capabilities of the pointer. t represents the task that accesses the pointer, where in this case $t \in \{\mathtt{P}, \mathtt{A}\} \times \mathbb{N}$. For example, $(\mathtt{A}, 1)$ and $(\mathtt{A}, 2)$ denote two independent tasks running on an accelerator.

We now show the memory weaknesses caused by capability granularity and heterogeneity. The following always holds for existing

heterogeneous systems with various approximations on c.

$$\forall (b, c, t) \in E.\, b \subseteq c \tag{1}$$

First, the IOMMU-based protection approximates c to memory pages, independent of accelerator objects b or t; and the accelerator-specific approach [20] approximates c to the memory region reachable by t. Our CHERI approach pushes c close to b, leading to pointer-level protection. In addition, CHERI also provides restrictions beyond c, such as permissions.

Second, the accelerator-specific approach [20] is tailored to a specific architecture, leading to a customized capability mapping. This mapping is different from general capabilities for CPUs, leading to a heterogeneous capability system $C(t) : \{\mathsf{P}, \mathsf{A}\} \rightarrow \{c_\mathsf{p}, c_\mathsf{a}\}$. The CPU follows a capability mapping c_p, and the accelerator follows a capability mapping c_a. The shared memory can be accessed through different protection methods, where the mismatch between the capability policies may lead to memory vulnerabilities. For example, the accelerator may forge the capability for a CPU task, enabling the CPU task to access unauthorized memory. We provide a unified capability system, i.e. $c_\mathsf{p} = c_\mathsf{a}$, to improve memory safety.

5 System Design Methodology

In order to achieve a unified capability system, we face two design choices: extending an existing CPU protection model to accelerators or extending existing accelerator protection models to CPUs. First, existing CPU protections provide complete functional support for diverse operations, including complex control flows, and can be simplified to accommodate diverse accelerator behaviors. However, the simplified protections must remain general and applicable to a wide range of accelerators. Second, existing accelerator protections, such as sNPU [20], are tailored to a specific hardware architecture. Their extension to support complex CPU instructions and other accelerator architectures faces significant engineering and verification challenges. Hence, we choose to extend CPU protection to accelerators, taking CHERI as the CPU protection scheme due to its fine-grained memory protection.

In this section, we first describe our capability model for heterogeneous systems to handle these threats. We then introduce our hardware system prototype that safely adapts CHERI-unaware accelerators. We also describe our trusted software driver implementation for the proposed system.

5.1 CHERI Capability Model

We extend the existing CHERI capability model to accelerators, treating each accelerator task as holding capabilities to its objects. Figure 4 illustrates a capability tree created by applications running on a CHERI-aware system. The bar under each object indicates the accessible memory region specified by its capability. The accessible memory region of a leaf capability is a subset of the accessible memory region of its parent capability. The capability root is created when the system is booted and is tightly controlled by the OS. Existing CHERI CPU systems create the capabilities connected with black edges, where a CPU task (variously a process, thread, or function) can instantiate another CPU task or allocate a data buffer.

The accelerator objects are illustrated in green boxes. In the proposed system, an accelerator task, i.e. the dedicated use of an accelerator functional unit for a length of time, is instantiated by a CPU task. Under assumption 2 in our threat model, the data buffers used by this accelerator task are also allocated by the CPU task. The memory accesses are authorized because the address ranges of these data buffers are subsets of the accessible memory region in the accelerator capability. We present them in green edges, including Accelerator Task 1 computing with Buffer 1 and Buffer 2 and Accelerator Task 2 computing with Buffer 3. The proposed protection model preserves the existing CHERI model and provides a flexible extension for arbitrary accelerators.

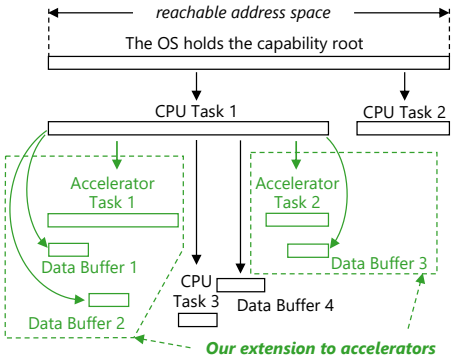


Figure 4: An example of capability tree for the proposed system. A pointer must be created by a CPU task, even if the allocated buffer is only accessed by an accelerator task.

5.2 Proposed Hardware System

We now show our hardware system for the proposed protection model. We first demonstrate our system prototype implemented on a Field Programmable Gate Array (FPGA) platform. We then describe our adaptive CapChecker design to enable hardware capability checks for accelerators and its potential design exploration for different accelerator types.

5.2.1 Prototype Hardware System. There are two possible approaches to make a heterogeneous system CHERI-aware. First, each device in the system is 'cherified' independently, where its hardware architecture needs to be modified to support CHERI natively. However, this is not scalable in terms of the significant engineering effort required for accelerator modification and the variety of accelerator architectures, as well as potential performance and area impacts. Second, a hardware protection component can be placed between a CHERI CPU and a CHERI-unaware accelerator, interposing memory requests from the accelerator and applying capability checks to capability-unaware accesses. However, the memory safety of accelerators is restricted by the visibility of their behaviors throughout the memory interface. We choose the second approach as the first step towards CHERI-aware heterogeneous systems. This enables us to investigate CHERI over a wide range of accelerators with less engineering effort.

Since our work focuses on memory safety, we keep our prototype system simple but realistic, excluding security-independent design choices. Our prototype hardware system is illustrated in Figure 2. The left of the figure is an open-source CHERI-extended softcore

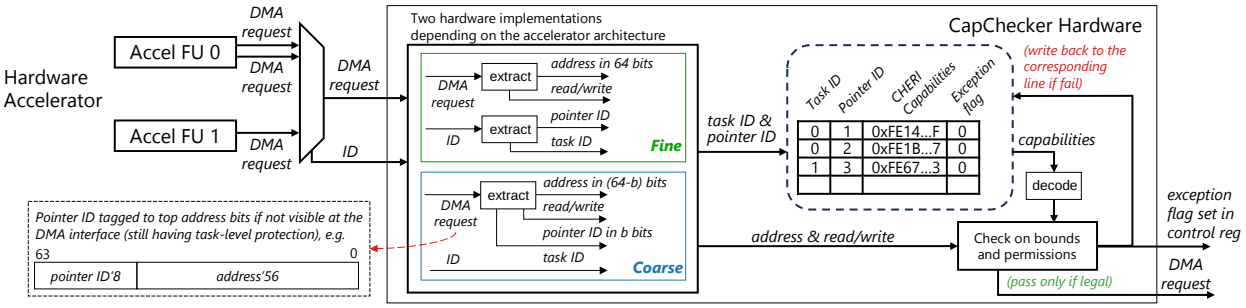


Figure 5: The proposed CapChecker hardware architecture. FU = functional unit. We developed two implementations of the CapChecker (*Coarse* and *Fine*) to adapt various memory interface organizations in accelerators.

CPU named Flute [14, 15], and the right of the figure is a hardware accelerator. The accelerator is connected to the interconnect as a slave with direct memory access (DMA) to the main memory for high performance. The system can also scale up to a large number of accelerators, which does not affect our protection model. Our hardware prototype is implemented on an FPGA but can also be synthesized into ASIC devices for better performance; however, this is security-independent under assumption 3.

In the system, the CPU sends system control to the accelerator as a memory master. All the interconnects in the prototyped system, annotated as edges, are implemented using the Advanced eXtensible Interface (AXI) [27], but our approach could be extended to other interfaces, such as Peripheral Component Interconnect Express (PCIe) [55] or Compute Express Link (CXL) [68]. The tail and head of an edge represent the master and slave ports, respectively. All the hardware interfaces are trusted and hardened in digital circuits and do not affect our security analysis.

The tag bit shown in Figure 3 is a vital component of the protection model that indicates whether a capability is valid. It is typically stored in a shadow section of memory that is off-limits to normal memory access (or, for example, in ECC bits [22]). When modifying capabilities in memory, either the agent modifying the capability must enforce the capability model, or writes to memory locations that contain a capability must also clear the tag. Since black-box hardware accelerators are CHERI-unaware, accelerator DMA does not carry capabilities.

In order to ensure memory safety through DMA, a CapChecker is placed between the accelerator functional units and the memory controller. The CapChecker communicates with the CHERI-aware CPU through its MMIOs via a separate capability interconnect, as shown at the top of Figure 2. Our proposed system protects the tag as follows. First, tags always stay in CPU-owned memory under assumption 2. The accelerator tasks cannot create new capabilities, because there is no dynamic memory management in accelerators. Second, all memory accesses from the accelerators must be guarded by the CapChecker. In addition to checking the request is permitted, it enforces that writes from accelerators will clear the tag, preventing mutation of valid capabilities into forged ones.

The CapChecker could be a single component that serializes the requests from all the accelerators, as illustrated in Figure 2. This approach provides a simplified control path but cannot scale well with a large number of accelerators or a high clock frequency. Alternatively, each accelerator could be connected to the memory controller through an exclusive CapChecker. This potentially increases the memory bandwidth but also leads to a more complex control path and a larger area. In our prototype platform, the AXI interconnect has limited bandwidth, allowing only one memory access in each clock cycle. The distribution of CapCheckers to each accelerator only increases the area and does not bring performance improvement. Instead, we use a single CapChecker in the system as an implementation.

5.2.2 CHERI Capability Checker. The CapChecker is CHERI-aware and provides run-time bound checks of memory requests from accelerators and blocks unauthorized accesses defined by the pointer capability. A key benefit of using the proposed CapChecker is its adaptability to arbitrary accelerator architectures. When initializing an accelerator task, the capabilities are sent by the CPU through the capability MMIO interconnect. The capabilities are preserved inside the CapChecker and cannot be accessed by the accelerator, making them unforgeable. Only the memory requests granted by the CapChecker are considered safe and forwarded to the memory subsystem, illustrated in the bottom right of Figure 2. This proposed architecture wraps existing CHERI-unaware accelerators inside the CHERI world with restricted behaviors while preserving the integrity of the CHERI model.

The proposed CapChecker design is illustrated in Figure 5. In the figure, two accelerator functional units send DMA requests to the CapChecker. The hardware architecture of a CapChecker involves three main components. First, the CapChecker contains a capability table to store all the pointer capabilities used by accelerators at run time. Given an accelerator task ID and buffer ID, the CapChecker fetches the indexed capability from the table. The CHERI capability is compressed in a 128-bit format for efficient memory size [77], and a capability decoder is used to recover the address bounds and permissions for memory checks.

Second, the CapChecker must keep track of the buffer identity when a memory request is received. To achieve the *principle of intentional use*, i.e. prevent unintentional access of one buffer with a pointer intended for another, this requires the memory requests to carry additional metadata to specify which object is being referred to. For example, a simple matrix multiplication accelerator may be performing $C := A \times B$ where the matrices A, B, C are objects. Each object could naturally be represented by a contiguous region

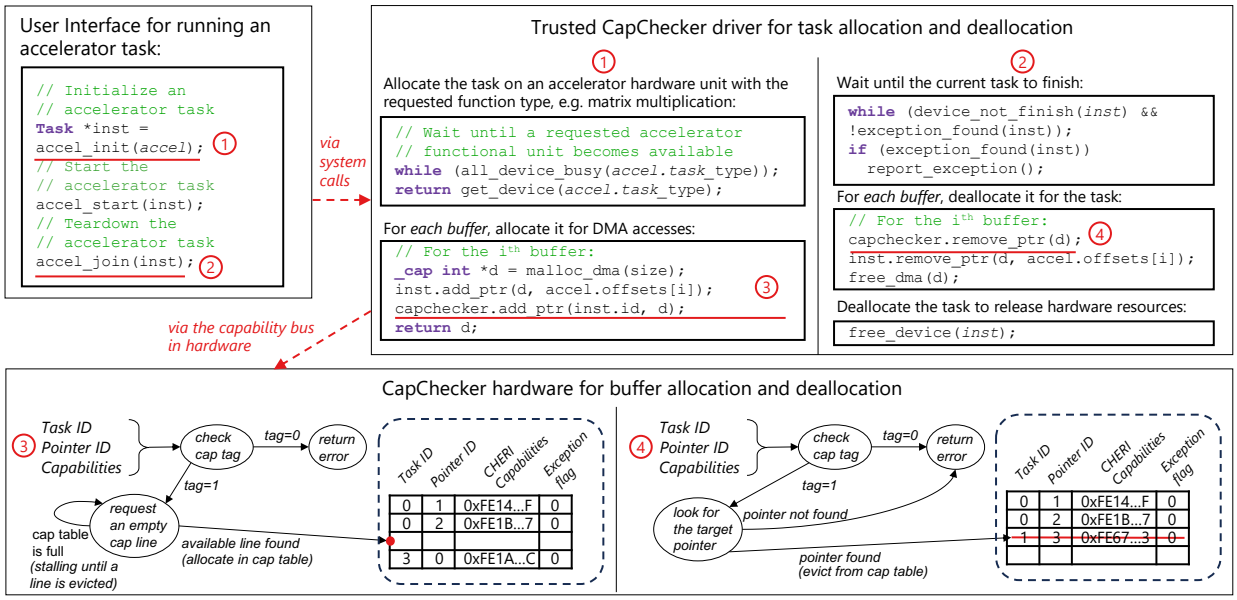


Figure 6: Software driver implementation of the CapChecker for allocating and evicting capabilities for accelerator accesses.

of memory with a defined base and length. The security model applied to an accelerator depends on how much of its object model is exposed to the host, i.e. the *provenance* of a transaction.

If each object is mapped to an independent hardware interface, such as Accel FU 0, the specific capability can be directly identified based on the hardware interface that sends the current memory request. For example, the matrix multiplier accelerator may expose $A/B/C$ as three separate memory ports in hardware. In this case, we are able to disambiguate the object to which the access refers. Even if these are subsequently multiplexed into a single port, we can maintain an object identifier as part of the metadata. This leads to an implementation in the block named *Fine*, which provides memory protection at the object level. Even if an accelerator task receives an untrusted pointer, it cannot be dereferenced to access outside the bounds of memory allocated to the capability. This leads to fine-grained compartmentalization between objects.

Finally, the capability fetched from the CapChecker capability table is used to check if the requested access from the accelerator is legal. If the requested address is legal, where the capability is tagged and has permissions for the requested access, the memory request is granted and passed to the memory controller; otherwise, an exception is raised by the CapChecker. If an exception is captured, the CapChecker sets a global flag to inform the CPU that an exception has been caught and updates the exception bit in the capability table so that the illegal memory access can be traced in software.

5.2.3 Limitations. We now discuss two limitations of the CapChecker. A key limitation is that the CapChecker has to adapt the existing accelerator interface since the accelerator architecture cannot be modified. In the worst case, an accelerator may have no provenance information, such as Accel FU 1 which shares all the memory accesses in a single memory interface. This makes disambiguating memory accesses challenging. For example, an out-of-bounds read from A generates an address that appears to be a correct read from B. In practice, we can retrofit provenance into the control information, such as addresses fed into the accelerator, as shown in the block named *Coarse*. For example, we restrict the address space for accelerators to 56 bits, illustrated at the bottom left of the figure. The top 8 bits of the address are reserved for identifying which object is being accessed. The bit width must be statically determined at the design phase of the SoC based on the accelerator workloads. These bits are only managed by the trusted driver of the CapChecker and cannot be accessed by user applications, leading to improved memory safety.

Still, this limits the scope of defenses against powerful attackers who can arbitrarily manipulate addresses inside the accelerator. For example, a matrix multiplication may overflow from one buffer into an adjacent buffer it has legitimate rights to access. A potential safeguard might add guard regions to reduce such risks; however, it still cannot prevent all attacks. For example, an accelerator which calculates array indexes based on unsanitized input data may be tricked into generating arbitrary addresses. In the worst case, the *Coarse* mode still provides compartmentalization between accelerator tasks because the task ID is identified by the source on the interconnect. In this work, we focus on the memory protection provided by *Fine*, treating *Coarse* as the worst-case scenario. We will explain our detailed security analysis in Section 6.2

Second, the number of entries in the capability table of the CapChecker depends on the accelerator workloads. If the capability table is too small, we either cannot access all the needed objects, or it requires the CPU driver to manage entries on the fly, with the potential for deadlock. Thanks to the restricted behaviors of accelerators, the minimal size of the capability table can be statically determined based on the accelerator application. For example, our

illustrative matrix multiplication accelerator requires three pointers. In our prototype, we set the CapChecker to have 256 entries, and it is sufficient for the evaluated benchmarks. Alternatively, a CapChecker could be built as a cache backing a larger in-memory table, similar to page table caching in IOMMUs/IOTLBs, but with each entry holding a capability. Since this is a microarchitectural optimization and does not affect the protection model, we consider the design of such a cache out of scope. We will compare the scalability of the CapChecker and IOMMUs in Section 6.4 with respect to the number of required entries.

5.3 Software Drivers

We now describe our trusted software driver for configuring the CapChecker, as depicted in Figure 6. The control of CapChecker is implicit in the user code but is orchestrated by the interface with accelerator drivers. The top left of the figure shows the user interface when running an accelerator task: allocation, execution, and deallocation. The execution of an accelerator task does not involve CapChecker but only interfaces with the control registers of the accelerators. Specifically, the CHERI CPU configures the accelerator functional block by setting its member variables mapping onto MMIO control registers. Since the control of each accelerator task is configured from the CPU, the existing CHERI capabilities on the CPU ensure pointer-level memory safety. If the driver alone holds capabilities to the control registers, other CPU tasks will be unable to interfere with the accelerator configuration. Or such capabilities could be delegated to the current user of the accelerator so that they can directly configure the hardware.

We now describe the allocation and deallocation processes. The data structure passed to the driver via system calls contains a set of objects, a pointer to the accelerator task, a list of address offsets for the control registers, and buffer sizes to be allocated for computation. The allocation process of an accelerator task is shown in ①, which involves two main steps. First, a given accelerator task may require specific accelerator functional units to compute. For example, there may be several matrix multiplication functional units available with different features. The driver traverses these suitable hardware units and searches for ones available to be allocated. If all suitable functional units are busy, the driver stalls until one becomes available. Second, the driver allocates buffers for the given task based on the list of buffer sizes. In our system, the CPU and the accelerator share the main memory, so the buffer can be allocated using `malloc()`. In other systems, the allocator may target accelerator-specific memory, such as HBM. The accelerator's starting address of the allocated buffer is calculated (e.g. adding an object ID in the *Coarse* scheme) and loaded to the corresponding control register on the accelerator for DMA, illustrated as `inst.add_ptr()`, In addition, its capabilities must be book-kept by the CapChecker to grant such DMA requests.

The allocation of capabilities to the CapChecker is illustrated in ③. The CapChecker contains control logic to verify the validity of a capability based on the capability tag. Given a valid capability, the CapChecker searches for an available entry in the capability table in an associative manner. If no available entry is found, the CapChecker stalls the allocation until an allocated capability by another accelerator task is evicted from the table, leaving an available slot. After the allocation, the user can initialize the data in the application on the CHERI CPU and start the accelerator task by calling the accelerator driver.

The deallocation of an accelerator task is shown in ②. Once the accelerator has completed successfully or triggered an exception error, the driver deallocates both the task and its buffers. If an exception is caught, all the buffer data is cleared, and the exception is reported back to the application at the end of the deallocation.

The deallocation of the buffer involves two steps. First, the capabilities kept in the CapChecker are evicted from the capability table so that new accelerator tasks can be allocated. Second, the control registers are cleared to avoid potential accesses from the subsequent accelerator task mapped onto the same functional unit. The buffers are deallocated using standard `free()`. Finally, the functional unit is released and is available for other accelerator tasks.

We have implemented this management flow in our bare metal testbed to be clear about the security properties. This is similar to how it would work in an embedded OS. A full application OS such as Linux or CheriBSD would require much more software engineering work, but the principles would be similar. The driver would live in the OS and arbitrate between multiple users of the accelerator (and hence the CapChecker), and be in charge of managing any address translation, depending on how that is configured in a given system.

Table 2: Data buffer sizes of benchmarks in the CapChecker. In all the benchmarks, the accelerator has eight instances and the CapChecker has 256 entries.

Benchmarks	Buffer count	Size in bytes	
		Min	Max
aes	8	128	128
backprop	56	12	10432
bfs_bulk	40	40	16384
bfs_queue	40	40	16384
fft_strided	48	4096	4096
fft_transpose	16	2048	2048
gemm_blocked	24	16384	16384
gemm_ncubed	24	16384	16384
kmp	32	4	64824
md_grid	56	256	2560
md_knn	56	1024	16384
nw	48	512	66564
sort_merge	16	8192	8192
sort_radix	32	16	8192
spmv_crs	40	1976	6664
spmv_ellpack	32	1976	19760
stencil2d	24	36	32768
stencil3d	24	8	65536
viterbi	40	256	16384

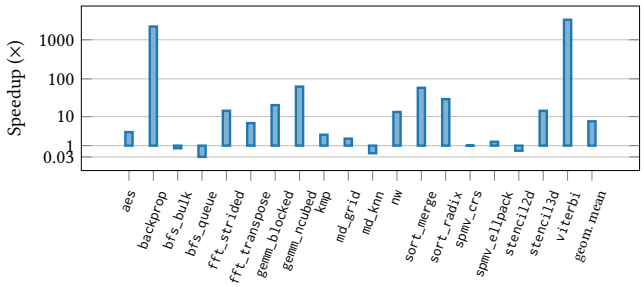


Figure 7: Accelerator speedup on the proposed system. We evaluating the CapChecker's adaptability on MachSuite [58] because it includes diverse accelerator behaviors for various applications.

Table 3: Evaluation of our work and related work on the memory safety weakness list provided Common Weakness Enumeration [51]. PG/TA/OB = protection granularity at Page/Task/Object respectively, and Object is the finest graine. NA = out of scope. *Coarse* and *Fine* are two implementations of the CapChecker in Fig. 5.

	ID	Memory Safety Weaknesses Name	No Method	IOPMP	IOMMU	sNPU [20]	*Coarse*	*Fine*
	119, 120, 122, 123, 124, 125, 126, 127, 129, 131, 466, 680, 786, 787, 788, 805, 806	(buffer overreads or overwrites)	✗	TA	PG	TA	TA	OB
ⓐ	761	Free of Pointer not at Start of Buffer	✗	✗	✗	✗	TA	OB
	822	Untrusted Pointer Dereference	✗	✗	✗	✗	TA	OB
	823	Untrusted Pointer Offset	✗	TA	PG	TA	TA	OB
ⓑ	416	Use After Free/Dangling Pointer	✗	✓	✓	✓	✓	✓
	587	Assignment of a Fixed Address to a Pointer						
	824	Access of Uninitialized Pointer						
ⓒ	244	Heap Inspection	✓	✓	✓	✓	✓	✓
	415	Double Free						
	590	Free of Memory not on the Heap						
	690	Unchecked Returned NULL Pointer Dereference						
	763	Release of Invalid Pointer or Reference						
ⓓ	121	Stack-based Buffer Overflow	NA	NA	NA	NA	NA	NA
	562	Return of Stack Variable Address						
	789	Stack Exhaustion						
ⓔ	134	Use of Externally-Controlled Format String	NA	NA	NA	NA	NA	NA
	762	Mismatched Memory Management Routines	NA	NA	NA	NA	NA	NA
ⓕ	188	Reliance on Data/Memory Layout	✗	✗	✗	✗	✗	✗
	198	Use of Incorrect Byte Ordering						
	401	Memory Leak						
	825	Expired Pointer Dereference/Dangling Pointer						

6 Experiments

Finding standard accelerator benchmarks for security analysis is a perennial problem because accelerator security is still overlooked. Here we take a standard accelerator benchmark set named MachSuite [58] built for performance analysis and evaluate the overhead of the CapChecker in performance, area, and power. The CPU we used for evaluation is an open-source RISC-V CPU core named Flute, which has previously been extended with CHERI instructions [15], and the accelerators are generated from an automated tool named AMD Xilinx Vitis high-level synthesis (HLS) [79]. The Vitis HLS tool automatically translates the high-level software specifications of a program into a custom hardware accelerator. We use the HLS tool to prototype diverse hardware behaviors of accelerators from the source provided by HLS, where each benchmark has its own accelerator architecture and hardware behavior. The hardware optimizations of accelerators are determined by the automated HLS tool, while our work focuses on the general CapChecker and does not change accelerator architectures.

In our experiments, we obtained the performance in clock cycles from hardware simulations of bare-metal execution using Verilator [65]. The maximum frequency, area, and power results were obtained from the FPGA vendor post Place & Route reports, Xilinx Vivado [1]. The FPGA we used for prototyping is the Virtex Ultrascale+ device on the VCU118 board, and the Xilinx software version is 2019.1.

6.1 Benchmark Selection

Table 2 shows a complete list of benchmarks in MachSuite [58], and their speedups achieved on the proposed system are illustrated in Figure 7. We set all the accelerator architectures to have eight instances, where each instance could be used by an independent user for acceleration. The MachSuite benchmarks present various accelerator architectures depending on the application and the algorithm used for acceleration. Significant differences in the number of buffers allocated and their sizes across benchmarks can be seen in Table 2. This enables us to investigate the adaptability of the CapChecker over a wide range of diverse accelerators.

These benchmarks also show significant differences in performance speedup due to different architectural features, leading to different performance bottlenecks. Overall, most benchmarks show better performance by offloading their tasks to accelerators. Specifically, benchmarks such as `backprop` and `viterbi` achieve more than 2000× speedup. This is because these benchmarks contain computation-intensive tasks that are sequentially executed on the CPU but are significantly parallelized on accelerators. The benchmarks including `md_knn`, `stencil2d`, `bfs_bulk` and `bfs_queue` also show worse performance when mapped to the testbed heterogeneous system, and this is because the tasks are memory-bounded. The memory bandwidth of the evaluation system is limited, leading to significant memory contention on the accelerator. The memory bottleneck could be improved by caching in accelerators which requires microarchitectural modifications of the accelerators. This is out of the scope of our work, as we focus on the memory safety and the overhead of the CapChecker on existing accelerators without modifying their architectures. Instead, we treat these accelerators as black-box hardware components in the system.

6.2 Security Analysis

Table 3 shows our security analysis of the proposed system with CapChecker over the Common Weakness Enumeration (CWE) list on memory weaknesses. The CWE test suite targets CPU systems with a rich programming environment and cannot be directly applied to accelerators, as accelerators have diverse architectures for specific classes of applications. Instead, we analyze how each weakness would apply in the accelerator context, making the two following assumptions. We define heap objects as buffers supplied as parameters to an accelerator instance, allocated by the software driver to the CapChecker; and we define stack objects as variables allocated inside an accelerator instance, never exposed to the CPU and perhaps implemented entirely in internal registers. While we refer to objects, CHERI on the CPU is able to derive capabilities to sub-objects, e.g. shrunk to individual struct members, and if passed from the CPU the CapChecker can protect those equally well.

We compare the CapChecker with four related works, three of which are listed in Table 1. We take sNPU [20] as a case of a specialized accelerator protection model for security comparison. The CapChecker has two protection methods, as illustrated in Figure 5 depending on the accelerator architecture, both evaluated in Table 3. These methods may provide certain memory protection.

We divide all the memory weakness issues into six groups based on how they are treated in the proposed systems and related work. For the memory weaknesses listed in group ⓐ, both the related

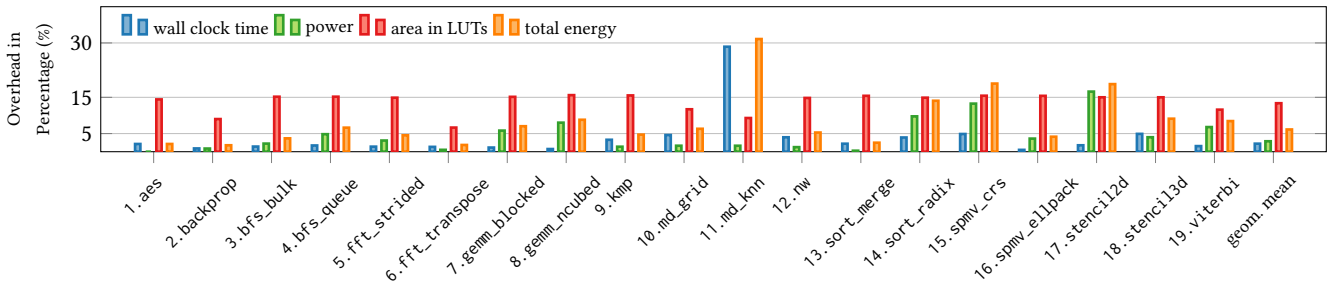


Figure 8: Overhead on overall performance, power and circuit area encountered by adding the CapChecker to the system.

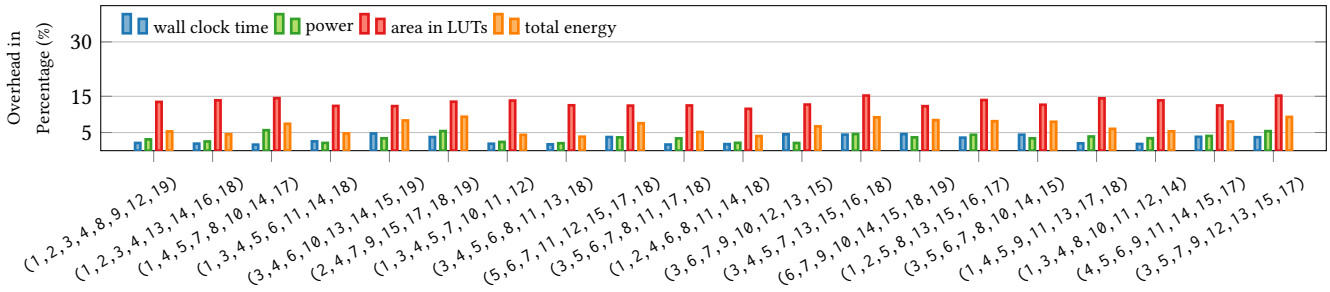


Figure 9: Overhead analysis of systems with mixed accelerators. In each experiment 8 accelerators are randomly selected from the numbered accelerator tasks in Figure 8. The overheads are similar to the geometric mean.

work and CapChecker provide memory protection but at different granularities. Here we compare the worst case of our work with the best case of related work. For example, buffer underflow (124) and buffer under-read (127) are only protected at page granularity if buffers are aligned to pages in IOMMUs. Otherwise, IOMMUs fail to protect intra-page buffer underflow. Free of pointer not at the start of the buffer (761) is a special case because it requires updating the data structure to synchronize all the uses of the target pointer. Our CHERI capability model enables us to analyze the parent capability off-the-shelf and mirror that in the CapChecker with minimal effort. Such protection requires specialized software implementation on an IOMMU, such as a shadow table to keep track of allocations on pages to ensure correctness. This additional implementation requires significant engineering effort and may introduce other memory vulnerabilities.

The other special cases in group ⓐ are untrusted pointer dereference (822) and untrusted pointer offset (823). Both require forgeability of capabilities. Although CHERI capabilities provide unforgeable pointer protection on the CPU, the accelerator remains CHERI-unaware and could dereference a pointer based on an input value. In the CapChecker, the worst case depends on the hardware interface. The main difference between *Fine* and *Coarse* is whether object IDs can be hardened in hardware interfaces. *Fine* provides the finest-grained memory protection among all methods. In *Coarse*, the object IDs reserved in upper address bits may potentially be forged from buffer overflows. The worst case of *Coarse* is that unauthorized accesses happen between pointers in the same task, leading to the same granularity as the sNPU.

The memory weaknesses listed in group ⓑ are protected by both CapChecker and related work. All these methods provide pointer capabilities to avoid unauthorized memory accesses introduced by both temporal and spatial memory safety. The memory issues listed in group ⓒ are temporal safety issues more related to software management, depending on the implementation of drivers. We rely on our drivers to ensure correct memory allocation and deallocation at the same granularity provided by the hardware. These drivers only interface with user applications through pre-defined system calls and cannot be exploited by attackers to bypass any checks under assumption 3. With our assumption of trustworthy drivers and the absence of design bugs, IOPMPs, IOMMUs, sNPU, and CapChecker can all ensure the same temporal memory safety. Slow IOMMU revocation may lead to temporal vulnerabilities [48] but is out of scope.

Group ⓓ contains memory issues related to stack memory. Accelerators often have specialized memory architectures for efficient computation. Based on our previous definition of the stack, the stack weaknesses are not applicable as we assume the accelerator cannot allocate memory by itself. Group ⓔ is also not applicable. 134 targets format strings, which are usually not computed on hardware accelerators. 762 is also out of the scope because it depends on the OS implementation.

Finally, group ⓕ contains memory weaknesses that neither the related work nor CapChecker protects. Memory layout (188) and byte ordering (198) focus on data type or format, while we focus on memory access behaviors. Memory leak (401) requires memory lifetime analysis, which is out of the scope of all compared methods. Dangling pointer (825), distinct from use-after-free, where an object

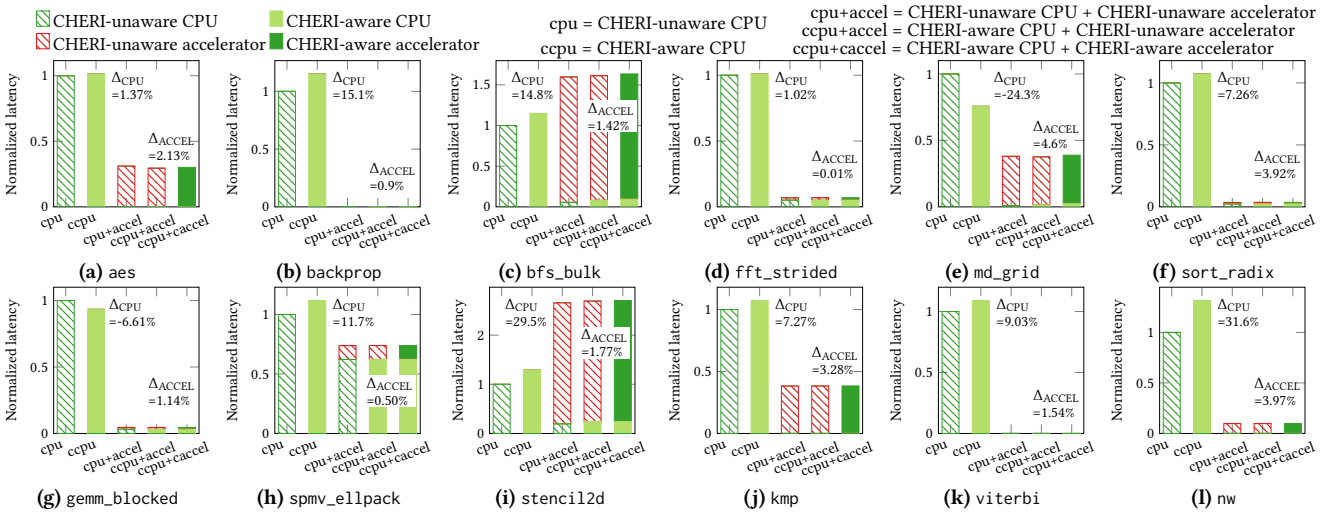


Figure 10: Wall clock time breakdown of different accelerator architectures over a set of applications. The performance overhead by adding the CapChecker is smaller than adding CHERI to the CPU for most benchmarks.

has gone out of scope and been replaced by another, depends on the accelerator driver implementation rather than the CapChecker to prevent the accelerator from using any expired object.

For the given benchmarks, all the accelerators work with capabilities because all the memory accesses for the given test data are correct. No correct memory access should be blocked by the CapChecker, so they all run under normal conditions. We observed memory issues such as buffer overflows in most accelerator benchmarks with particular test data, including `sort_radix` and `backprop`. For example, a user-defined loop bound may be larger than the size of an array accessed by the loop. The accelerator benchmark set does not contain standard test datasets for coverage analysis, but our observation on particular test data agrees with Table 3.

6.3 Overhead Analysis

In our overhead analysis, we compare our implementation, CHERI CPU systems with CHERI accelerators (*ccpu+caccel*), with four baselines: CHERI-unaware CPU-only systems (*cpu*), CHERI CPU-only systems (*ccpu*), CHERI-unaware CPU systems with CHERI-unaware accelerators (*cpu+accel*), and CHERI CPU systems with CHERI-unaware accelerators (*ccpu+accel*). We do not compare our approach with the methods in Table 1 because they all expose memory vulnerabilities described in Section 4.

Detailed performance breakdown is illustrated in Figure 10. Overall, the CapChecker shows smaller performance overhead compared to CHERI on the CPU. The overhead also varies across applications and architectures, remaining small even when the accelerator has limited performance. For example, Figure 10(c) and Figure 10(i) show that the accelerator achieves worse performance than the CPU due to the limited memory bandwidth; however, the overhead of CapChecker remains less than 2%. Figure 10(a) shows the CapChecker has more overhead than the CHERI overhead on the CPU due to the absence of the accelerator cache, but it remains small at around 2%. In Figure 10(g), the benchmark `gemm_blocked` shows better performance of *ccpu* compared to *cpu*, because the CHERI CPU ISA has a 128-bit capability copying instruction that provides more efficient memory copying compared to the 64-bit copying instruction on the standard RISC-V CPU, leading to lower latency.

Overall, the CapChecker shows small performance overheads. Figure 8 plots the overhead encountered by adding the CapChecker on performance, area, and power for all the benchmarks. Overall, the performance overhead is within 5% for most benchmarks. The benchmark `md_knn` shows large performance overhead in percentage because the benchmark has a small absolute latency (*ccpu+accel* has 3863 cycles, and *ccpu+caccel* has 5020 cycles). Other benchmarks have latencies of more than a million cycles, making such an overhead affordable. The area overhead of the CapChecker is around 15% for all benchmarks but may vary depending on the total area of the original hardware. In our experiments, the CapChecker has 256 entries for all the benchmarks and takes a constant area. The power overhead is relatively small and varies among benchmarks, depending on the FPGA synthesis tool we used to map our designs.

Apart from systems accelerating the same application, modern hardware systems may contain mixed accelerator architectures for accelerating various tasks. In order to evaluate the overhead on these realistic systems, Figure 9 shows the results of 20 systems with mixed accelerator architectures. Each system contains eight randomly selected accelerator architectures from Figure 8. The mix of various accelerator applications and workloads leads to different performances, but the security challenges remain the same as Figure 8. Overall, the overhead results of individual mixed systems are close to the geometric mean in Figure 8.

The area overhead in percentage terms depends on the total area of the CPU and accelerators. In particular, our 256-entry CapChecker prototype consists of 30k LUTs on the target FPGA. The area of the CapChecker depends on the number of entries required by the accelerator applications. However, this number of entries does not scale with the accelerator area but depends instead on the

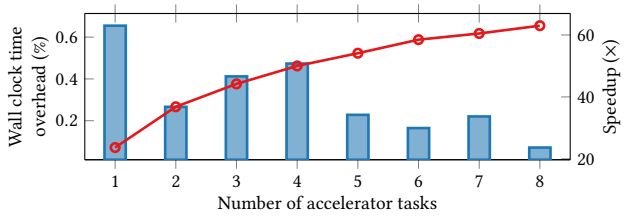


Figure 11: The wall clock time overhead of the CapChecker on `gemm_ncubed` over different degrees of parallelism.

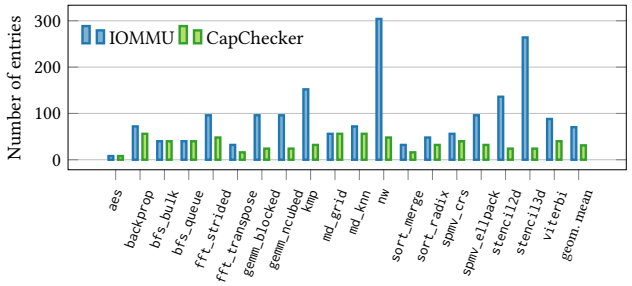


Figure 12: Number of entries required by the IOMMU and the CapChecker. IOMMU page size = 4 kB.

accelerator task complexity. For example, two matrix multiplication accelerators could occupy significantly different areas due to exploiting different degrees of parallelism and workload, but both would require only three pointers. If area were a concern, caching could be applied to the CapChecker to trade off area against latency overhead, as mentioned in Section 5.2.3. On the other hand, the CapChecker could be more lightweight than our implementation. For example, a variant of TinyML [72] embedded systems contains a microcontroller core and a small hardware accelerator, also called a custom functional unit (CFU) [57]. The CFU accelerates a specific task such as matrix multiplication as part of the machine learning application. The simple architecture of CFUs also simplifies the repository size of the CapChecker, allowing an implementation costing fewer than 100 LUTs, while the total area is around 10k LUTs.

6.4 Scalability Analysis

We now evaluate the scalability of the CapChecker by varying the number of accelerator tasks and buffers. First, Figure 11 plots the performance overhead and speedup of `gemm_ncubed` benchmark over different numbers of parallel accelerator tasks. More parallelism leads to better performance. There may be a trend that more parallel accelerator tasks may lead to a smaller performance overhead. This is because the limited memory bandwidth for shared memory accesses among more accelerator tasks leads to a longer latency on the accelerator. Still, the performance overhead of the CapChecker remains small across different degrees of parallelism.

Second, we compare the scalability of the IOMMU and the CapChecker. The hardware designs of both IOMMUs and CapCheckers depend on the application to be accelerated, and it is challenging to perform a fair comparison across diverse accelerator architectures. Instead, we compare the number of entries for both methods across various numbers and sizes of buffers in Figure 12. The page size for an IOMMU is 4 kB. To ensure fairness, we restrict that each page can hold at most one buffer to prevent intra-page attacks, leading to the same memory safety granularity as the CapChecker.

The CapChecker shows better scalability as CHERI provides pointer-level granularity. In the figure, the number of entries required by the CapChecker is smaller than the IOMMU across most benchmarks. A key difference between these two approaches is that the entry count of the IOMMU depends on both the number of buffers and their sizes, while the entry count of the CapChecker only depends on the number of buffers. Accelerators today often compute data-intensive tasks, such as large language model accelerators, which compute matrices up to tens of millions of bytes [28, 35, 41]. While this challenge may be reduced by superpages [60, 81], the IOMMU entries still scale with the buffer size.

7 Related Work

7.1 Capability Systems

Capability systems usually work under three distinct trust models: central-trust, distributed-trust, and decentralized. First, central-trust systems require a trusted authority to handle all access, derivation, transfer, and revocation processes. SeL4 [34] is a microkernel that provides an object-oriented OS interface, and all types of objects are referenced with capabilities. Capsicum [73] is a centralized capability system in FreeBSD OS [52] that replaces file descriptors with capabilities. These systems rely on the OS as the trusted authority, while our CHERI approach relies on the hardware. Second, distributed-trust systems distribute the trusted authority among multiple trusted computing elements. This approach maintains the same semantics but offers better scalability at the cost of increased complexity, as capability operations may overlap, necessitating the avoidance of race conditions. Barrelfish [63] and SemperOS [29] are distributed operating systems running on separate cores and communicating with message passing. In these systems, every core has a local capability list, and they must synchronize and keep those lists consistent when manipulating or copying capabilities. In our work, most hardware accelerators exploit data parallelism for high performance and do not support complex control or memory operations. Our approach distributes CHERI capabilities across accelerators, where these capabilities are managed by a single CPU core and treated as read-only for accelerators. This simplifies the hardware circuitry and reduces the synchronization overhead between devices. Finally, decentralized systems enable untrusted actors to manipulate and transfer capabilities, with these manipulations being verified cryptographically when the resource is accessed. Macaroons [13] uses cryptography to check the validity of a capability when used. Specifically, a capability is called a 'macaroon' that begins with an identifier and a signature, made by hashing the identifier with a secret key. They are used to protect arbitrary resources and are distributed to actors by the resource owner. Recent work combines CHERI and Macaroons [19], which protects the internal controller and the network boundary, respectively. Exploring the possibilities of decentralized CHERI capabilities in a heterogeneous system would be one of our future works.

7.2 CHERI Hardware Systems

CHERI capabilities [74] allow logical software components to access virtual memory ranges. They are stored in tagged memory and registers, manipulated with native hardware instructions, and revoked asynchronously by software using those instructions. CHERI has been adopted by industry, including Microsoft [8, 9], Codasip [16] and Arm [22, 59]. Existing CHERI work focuses on memory protection in CPU architectures, including MIPS, ARMv8-a, RISC-V and x86-64 [7, 12, 22, 75, 78]. Markettos *et al.* [47] described how to exploit the CHERI capabilities to defend against direct memory accesses but only had a paper evaluation without an implementation. Our work is the first effort to enable hardware capabilities in hardware accelerators in a working CHERI hardware system.

7.3 Memory Protection for Accelerators

Existing work on hardware memory safety for GPUs has been actively studied, but the memory safety of traditional hardware accelerators remains under-explored. For general accelerators, such as GPUs, Lee *et al.* [39] propose an approach named GPUShield. The GPUShield adds a bounds table for checking, indexed by the top 16 bits of a pointer. This provides memory safety on GPUs but still has memory vulnerabilities when computing in a heterogeneous hardware system. NVIDIA CUDA-MEMCHECK is a bounds check tool that provides bound checks at run time [37]. Ziad *et al.* [69] propose a tool named cuCatch that detects memory safety violations on GPUs. These methods cannot be directly applied to traditional hardware accelerators as they are tailored to a specific class of GPUs. sNPU [20] proposes the integration of hardware capabilities inside an NPU architecture for improved memory safety, while we focus on general capability-enabled systems without modification of accelerator architectures.

On the other hand, there have been efforts to reduce the performance overhead of IOMMUs for particular use cases [11, 46, 49, 50, 56], while our approach exploits a general hardware architecture of CapChecker with finer-grained memory protection. There are also working-in-progress RISC-V extensions for IO protection, such as CoVE-IO [17] and I/O MTT [32]. Their memory protection remains forgeable, while our proposed capability model in Figure 4 makes capabilities unforgeable across the heterogeneous platform.

8 Conclusion and Future Work

Existing heterogeneous accelerator systems expose memory vulnerabilities due to coarse granularity and capability heterogeneity in existing memory protection methods. We present the first step towards a unified and fine-grained capability system for heterogeneous systems. Our work proposes a system design method using an adaptive protection component named CapChecker and its trusted driver. The CapChecker extends CHERI capabilities from the CPU to the accelerator without architectural modification of the accelerator. This leads to a CHERI-aware heterogeneous system, preserving memory safety at the system level. We evaluate the CapChecker over a wide range of accelerator architectures with different workloads and show that our method is general and scalable with fine-grained memory protection.

Our future work will mainly involve two directions, tackling the two limitations mentioned in Section 5. First, assumption 2 of the threat model restricts the accelerator to not having dynamic memory management. We plan to lift this restriction and analyze memory safety for complex accelerator architectures such as GPUs both theoretically and practically. Second, we will investigate the architectural specialization of the CapChecker for better performance and area efficiency when working with specific accelerator architectures such as NPUs and custom accelerators. This will open up a larger optimization space in both software and hardware, including cache sizing and IOMMU co-design.

Acknowledgments

This work was supported by the UK EPSRC under the CAPcelerate Project (EP/V000381/1) and the Chrompartments Project (EP/X015963/1), both part of the Digital Security by Design (DSbD) Programme and the DSbDtech initiative.

References

[1] 2023. AMD Xilinx Vitis. https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html
[2] 2024. Amazon Web Services. https://aws.amazon.com/
[3] 2024. Microsoft Azure. https://azure.microsoft.com/
[4] 2025. Inferenceu - Cerebras. https://cerebras.ai/inference
[5] 2025. Tenstorrent. https://tenstorrent.com/en
[6] 2025. Versal Adaptive SoCs. https://www.amd.com/en/products/adaptive-socs-and-fpgas/versal.html
[7] Thomas Aird et al. 2025. RISC-V Specification for CHERI Extensions. v0.9.5, 2025-03-31. (March 2025). https://riscv.github.io/riscv-cheri/
[8] Saar Amar. 2022. An Armful of CHERIs. (Jan. 2022). https://msrc.microsoft.com/blog/2022/01/an_armful_of_cheris/
[9] Saar Amar, David Chisnall, Tony Chen, Nathaniel Wesley Filardo, Ben Laurie, Kunyan Liu, Robert Norton, Simon W Moore, Yucong Tao, Robert NM Watson, et al. 2023. CHERIoT: Complete Memory Safety for Embedded Devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 641–653.
[10] Dorian Amiet, Andreas Curiger, and Paul Zbinden. 2018. FPGA-based accelerator for post-quantum signature scheme SPHINCS-256. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2018), 18–39.
[11] Nadav Amit, Muli Ben-Yehuda, and Ben-Ami Yassour. 2010. IOMMU: Strategies for mitigating the IOTLB bottleneck. In *International Symposium on Computer Architecture*. Springer, 256–274.
[12] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E Gray, Robert M Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, et al. 2019. ISA Semantics for ARMv8-a, RISC-V, and CHERI-MIPS. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–31.
[13] Arnar Birgisson, Joe Gibbs Politz, Ulfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. 2014. Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In *Proceedings of the Network and Distributed Systems Symposium*.
[14] Bluespec, Inc. 2018. Bluespec Announces Flute. (Dec. 2018). https://bluespec.com/2018/12/13/bluespec-announces-flute/
[15] CHERI Flute. 2024. https://github.com/CTSRD-CHERI/Flute/
[16] Codasip. 2024. Codasip delivers processor security to actively prevent the most common cyberattacks. https://codasip.com/press-release/2023/10/31/codasip-delivers-processor-security-to-actively-prevent-cyberattacks/
[17] CoVE-IO v0.2.0. 2024. https://github.com/riscv-non-isa/riscv-ap-tee-io/releases/tag/v0.2.0.
[18] Jack B. Dennis and Earl C. Van Horn. 1966. Programming semantics for multiprogrammed computations. *Commun. ACM* 9, 3 (March 1966), 143–155. doi:10.1145/365230.365252
[19] Michael Dodson, Alastair R Beresford, Alexander Richardson, Jessica Clarke, and Robert NM Watson. 2020. CHERI Macaroons: Efficient, host-based access control for cyber-physical systems. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 688–693.
[20] Erhu Feng, Dahu Feng, Dong Du, Yubin Xia, and Haibo Chen. 2024. sNPU: Trusted Execution Environments on Integrated NPUs. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 708–723.
[21] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. 2018. A configurable cloud-scale DNN processor for real-time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1–14.

[22] Richard Grisenthwaite, Graeme Barnes, Robert NM Watson, Simon W Moore, Peter Sewell, and Jonathan Woodruff. 2023. The Arm Morello evaluation platform—validating CHERI-based security in a high-performance system. *IEEE Micro* 43, 3 (2023), 50–57.

[23] RISC-V IOPMP Task Group. 2024. RISC-V IOPMP Architecture Specification, Version 0.9.0. (May 2024). https://github.com/riscv-non-isa/iopmp-spec

[24] CUDA C++ Programming Guide. 2024. Dynamic Global Memory Allocation and Operations. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#dynamic-global-memory-allocation-and-operations

[25] CUDA C++ Programming Guide. 2024. Memory Protection. https://docs.nvidia.com/deploy/mps/#memory-protection

[26] CUDA C++ Programming Guide. 2024. Virtual Memory Management. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#virtual-memory-management

[27] Manoj Gupta and Ashok Kumar Nagawat. 2016. Design and implementation of high performance advanced extensible interface (AXI) based DDR3 memory controller. In *2016 International Conference on Communication and Signal Processing (ICCSP)*. IEEE, 1175–1179.

[28] Linley Gwennap. 2020. Groq rocks neural networks. *Microprocessor Report* (Jan. 2020).

[29] Matthias Hille, Nils Asmussen, Pramod Bhatotia, and Hermann Härtig. 2019. SemperOS: A Distributed Capability System. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 709–722.

[30] Weizhe Hua, Muhammad Umar, Zhiru Zhang, and G Edward Suh. 2022. MGX: Near-zero overhead memory protection for data-intensive accelerators. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 726–741.

[31] Rachel Huang, Jonathan Pedoeem, and Cuixian Chen. 2018. YOLO-LITE: a real-time object detection algorithm optimized for non-GPU computers. In *2018 IEEE International Conference on Big Data*. IEEE, 2503–2510.

[32] I/O MTT extension. 2024. https://github.com/riscv/riscv-smmtt/blob/main/chapter6.adoc.

[33] Jun-Woo Jang, Sehwan Lee, Dongyoung Kim, Hyunsun Park, Ali Shafiee Ardestani, Yeongjae Choi, Channoh Kim, Yoojin Kim, Hyeongseok Yu, Hamzah Abdel-Aziz, et al. 2021. Sparsity-aware and re-configurable NPU architecture for Samsung flagship mobile SoC. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 15–28.

[34] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*. 207–220.

[35] Anton Kos, Vukašin Ranković, and Sašo Tomažič. 2015. Sorting networks on Maxeler dataflow supercomputing systems. In *Advances in computers*. Vol. 96. Elsevier, 139–186.

[36] Adithya Krishna, Srikanth Rohit Nudurupati, DG Chandana, Pritesh Dwivedi, André van Schaik, Mahesh Mehendale, and Chetan Singh Thakur. 2024. RAMAN: A re-configurable and sparse tinyML accelerator for inference on edge. *IEEE Internet of Things Journal* (2024).

[37] Jaewon Lee, Yonghae Kim, Jiashen Cao, Euna Kim, Jaekyu Lee, and Hyesoon Kim. 2022. Securing GPU via region-based bounds checking. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA)*. 27–41.

[38] Sunho Lee, Jungwoo Kim, Seonjin Na, Jongse Park, and Jaehyuk Huh. 2022. TNPU: Supporting trusted execution with tree-less integrity protection for neural processing unit. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 229–243.

[39] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. 2014. Stealing webpages rendered on your browser by exploiting GPU vulnerabilities. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 19–33.

[40] Sunho Lee, Seonjin Na, Jungwoo Kim, Jongse Park, and Jaehyuk Huh. 2022. Tunable memory protection for secure neural processing units. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*. IEEE, 105–108.

[41] Sean Lie. 2022. Cerebras architecture deep dive: First look inside the HW/SW co-design for deep learning: Cerebras systems. In *2022 IEEE Hot Chips 34 Symposium (HCS)*. IEEE Computer Society, 1–34.

[42] M. Y. Mo. 2024. Fall of the machines: Exploiting the Qualcomm NPU (neural processing unit) kernel driver. https://securitylab.github.com/research/qualcomm_npu

[43] M. Y. Mo. 2024. GHSL-2021-1029: Use-after-free (UaF) in Qualcomm NPU driver - CVE-2021-1940. https://securitylab.github.com/advisories/GHSL-2021-1029-npu

[44] M. Y. Mo. 2024. GHSL-2021-1030: Information leak in Qualcomm NPU driver - CVE-2021-1968. https://securitylab.github.com/advisories/GHSL-2021-1030-npu

[45] M. Y. Mo. 2024. GHSL-2021-1031: Information leak in Qualcomm NPU driver - CVE-2021-1969. https://securitylab.github.com/advisories/GHSL-2021-1031-npu

[46] 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)*. 29–44.

[47] A Theodore Markettos, John Baldwin, Ruslan Bukin, Peter G Neumann, Simon W Moore, and Robert NM Watson. 2020. Position Paper: Defending Direct Memory Access with CHERI Capabilities. In *Hardware and Architectural Support for Security and Privacy (HASP)*. 1–9.

[48] A Theodore Markettos, Colin Rothwell, Brett F Gutstein, Allison Pearce, Peter G Neumann, Simon W Moore, and Robert NM Watson. 2019. Thunderclap: Exploring vulnerabilities in operating system IOMMU protection via DMA from untrustworthy peripherals. (2019).

[49] Alex Markuze, Adam Morrison, and Dan Tsafrir. 2016. True IOMMU protection from DMA attacks: When copy is faster than zero copy. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 249–262.

[50] Alex Markuze, Igor Smolyar, Adam Morrison, and Dan Tsafrir. 2018. DAMN: Overhead-free IOMMU protection for networking. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 301–315.

[51] Robert A Martin and Sean Barnum. 2008. Common Weakness Enumeration (CWE) status update. *ACM SIGAda Ada Letters* 28, 1 (2008), 88–91.

[52] Marshall Kirk McKusick, George V. Neville-Neil, and Robert N.M. Watson. 2014. *The design and implementation of the FreeBSD operating system* (second ed.). Addison-Wesley Reading, MA.

[53] Jien Hau Ng, Chee Hong Ang, and Hwa Chaw Law. 2022. A Realization of IO Physical Memory Protection for RISC-V Systems. In *2022 IEEE 15th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. IEEE, 375–380.

[54] Francisco Ortega-Zamorano, Marcelo A Montemurro, Sergio Alejandro Cannas, José M Jerez, and Leonardo Franco. 2015. FPGA Hardware Acceleration of Monte Carlo Simulations for the Ising Model. *IEEE Transactions on Parallel and Distributed Systems* 27, 9 (2015), 2618–2627.

[55] PCI-SIG. 2022. PCI Express 6.0 Specification. (Jan. 2022). https://pcisig.com/pci-express-6.0-specification

[56] Omer Peleg, Adam Morrison, Benjamin Serebrin, and Dan Tsafrir. 2015. Utilizing the IOMMU scalably. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 549–562.

[57] Shvetank Prakash, Tim Callahan, Joseph Bushagour, Colby Banbury, Alan V Green, Pete Warden, Tim Ansell, and Vijay Janapa Reddi. 2023. CFU Playground: Full-Stack Open-Source Framework for Tiny Machine Learning (TinyML) Acceleration on FPGAs. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 157–167.

[58] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. 2014. MachSuite: Benchmarks for Accelerator Design and Customized Architectures. In *Proceedings of the IEEE International Symposium on Workload Characterization*. Raleigh, North Carolina.

[59] Dan Robinson. 2022. How Arm popped CHERI architecture into Morello Program hardware. https://www.theregister.com/2022/08/26/arm_cheri_morello/. *The Register* (Aug. 2022).

[60] Theodore H Romer, Wayne H Ohlrich, Anna R Karlin, and Brian N Bershad. 1995. Reducing TLB and memory overhead using online superpage promotion. In *Proceedings of the 22nd annual international symposium on Computer architecture*. 176–187.

[61] Daniel Rozhko and Paul Chow. 2019. The Network Management Unit (NMU) Securing Network Access for Direct-Connected FPGAs. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 232–241.

[62] J.H. Saltzer and M.D. Schroeder. 1975. The protection of information in computer systems. *Proc. IEEE* 63, 9 (1975), 1278–1308. doi:10.1109/PROC.1975.9939

[63] Adrian Schüpbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs. 2008. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems*, Vol. 27. New York, NY, USA.

[64] Nivedita Shrivastava and Smruti Ranjan Sarangi. 2023. Securator: A fast and secure Neural Processing Unit. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1127–1139.

[65] Wilson Snyder. 2004. Verilator and systemperl. In *North American SystemC Users' Group, Design Automation Conference*.

[66] Zhuoran Song, Feiyang Wu, Xueyuan Liu, Jing Ke, Naifeng Jing, and Xiaoyao Liang. 2020. VR-DANN: Real-time video recognition via decoder-assisted neural network acceleration. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 698–710.

[67] Tyler Sorensen and Heidy Khlaaf. 2024. LeftoverLocals: Listening to LLM Responses Through Leaked GPU Local Memory. *arXiv preprint arXiv:2401.16603* (2024).

[68] Samuel W. Stark, A. Theodore Markettos, and Simon W. Moore. 2023. How Flexible Is CXL's Memory Protection? *Commun. ACM* 66, 12 (Nov. 2023), 46–51. doi:10.1145/3617580

[69] Mohamed Tarek Ibn Ziad, Sana Damani, Aamer Jaleel, Stephen W Keckler, and Mark Stephenson. 2023. CuCatch: A Debugging Tool for Efficiently Catching Memory Safety Violations in CUDA Applications. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 124–147.

[70] David B Thomas. 2010. Acceleration of financial Monte-Carlo simulations using FPGAs. In *2010 IEEE Workshop on High Performance Computational Finance*. IEEE, 1–6.
[71] Hasitha Muthumala Waidyasooriya and Masanori Hariyama. 2019. Highly-parallel FPGA accelerator for simulated quantum annealing. *IEEE Transactions on Emerging Topics in Computing* 9, 4 (2019).
[72] Pete Warden and Daniel Situnayake. 2019. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. O'Reilly Media.
[73] Robert NM Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. 2010. Capsicum: Practical Capabilities for UNIX. In *19th USENIX Security Symposium (USENIX Security 10)*.
[74] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. 2015. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 20–37.
[75] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Franz A. Fuchs, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. 2023. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9)*. Technical Report UCAM-CL-TR-987. University of Cambridge, Computer Laboratory. doi:10.48456/tr-987
[76] Nils Wistoff, Andreas Kuster, Michael Rogenmoser, Robert Balas, Moritz Schneider, and Luca Benini. 2023. Protego: A Low-Overhead Open-Source I/O Physical Memory Protection Unit for RISC-V. In *Proceedings of the 1st Safety and Security in Heterogeneous Open System-on-Chip Platforms Workshop (SSH-SoC 2023)*.
[77] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony Fox, Robert M Norton, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel W Filardo, A Theodore Markettos, et al. 2019. CHERI Concentrate: Practical compressed capabilities. *IEEE Trans. Comput.* 68, 10 (2019), 1455–1469.
[78] 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. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 457–468. doi:10.1109/ISCA.2014.6853201
[79] Xilinx Vitis HLS. 2023. https://www.xilinx.com/html_docs/xilinx2023_1/vitis_doc/index.html.
[80] Xiaofan Zhang, Yuan Ma, Jinjun Xiong, Wen-Mei W Hwu, Volodymyr Kindratenko, and Deming Chen. 2021. Exploring HW/SW co-design for video analysis on CPU-FPGA heterogeneous systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 6 (2021), 1606–1619.
[81] Weixi Zhu, Alan L Cox, and Scott Rixner. 2020. A comprehensive analysis of superpage management mechanisms and policies. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 829–842.

A Artifact Description

We open-source our software tools that automatically generate the proposed heterogeneous systems for evaluating CapChecker. Our artifact includes:

- A set of script to build CHERI toolchain environments;
- A script to automatically generate different design with different configurations; and
- A set of benchmarks evaluated by this work.

The artifact is available at https://doi.org/10.5281/zenodo.15100923 In order to evaluate the artifact, it is suggested to have a machine with at least 16 CPU cores and 64 GB of memory to run the required software. Additionally, a specific FPGA board, Xilinx VCU118, is required to validate the generated bitstream.

For software dependencies, both Xilinx Vitis HLS 2023.1 and Vivado 2019.1 are required. Xilinx Vitis HLS is used to generate the hardware designs in Verilog for evaluation, and Vivado 2019.1 is used to generate the bitstream for the target FPGA. We provide a Docker script to set up all the environments.

Please refer to the README files within the artifact for installation and usage instructions.