Work-in-Progress: Northcape: Embedded Real-Time Capability-Based Addressing

Eric Ackermann, Noah Mauthe, Sven Bugiel
CISPA Helmholtz Center for Information Security and Saarland University, Saarbrücken, Germany
{*eric.ackermann,noah.mauthe,sven.bugiel*}*@cispa.de*

Abstract—**Direct Memory Access (DMA) increases throughput and efficiency of transfers between I/O devices and the main memory. Therein, it raises a critical security issue: How can the computer architecture enforce that devices only read from and write to the intended I/O buffers? Within the scope of this ongoing research project, we improve existing solutions to this problem by providing a byte-granular memory protection mechanism that is enforced universally for both software and hardware. Additional design goals of the prototype are compatibility with unmodified legacy devices (with full security) and operating systems (without security advantage). We target embedded real-time devices, whose architecture is particularly vulnerable to DMA attacks.**

Northcape, our proposed system, uses a capability-based memory protection mechanism with byte granularity. In contrast to existing protection systems, access control is implemented at the bus level in the northbridge. Thereby, the protection applies to the CPU, any accelerators and DMA peripherals in the system and protects system memory and memory-mapped I/O peripherals. Our pointer tagging-based implementation ensures compatibility with legacy 64-bit addressing schemes and an unmodified AXI system bus.

1. Introduction

Transferring data efficiently between the main memory and peripheral input/output (I/O) devices such as terminals, network interface cards (NICs) and displays is crucial for modern computers. To this end, Direct Memory Access (DMA) allows a high-speed I/O device to copy data into memory independent from the CPU, allowing the CPU to perform useful work during the transfer. However, DMA raises the security issue of ensuring *access control* to memory: DMA devices are usually given the addresses of physically contiguous *DMA buffers* in memory directly or via the indirection of *DMA descriptors*. Devices are *expected to*[1] only read and write within the bounds of the DMA buffers. However, if no access control mechanism for DMA devices is in place, compromised DMA devices can be abused to read and write any address in the physical address space. It has been demonstrated that this enables stealthy rootkits or even spawning root shells [12, 15].

Two specific variants of the aforementioned DMA attacks exist: First, a software adversary with access to the DMA descriptors can abuse the DMA device for a *confused deputy* attack to accomplish, e.g., privilege escalation. Second, the DMA device can be compromised or purpose-built for performing DMA attacks: Beniamini [3] demonstrates an over-the-air exploitation of a broadcom mobile WiFi chip, turning a benign DMA device into a hardware trojan. Markettos et al. showcase an implant that poses as a benign Thunderbolt-enabled peripheral, but performs DMA attacks via Thunderbolt PCIe encapsulation [12].

A possible solution for low-cost embedded devices relies on a memory protection unit (MPU) in the CPU to segregate physical memory into several compartments. One compartment can then be utilized to confine DMA descriptors. Untrustworthy software is not given access to the compartment containing the descriptors [14]. While this approach mitigates the confused deputy attack, it can only cover software adversaries: the attacks by Beniamini and Markettos et al. are still possible.

A more advanced solution is deployment of an I/O memory management unit (IOMMU) that confines DMA devices to an individual virtual address space. Markettos et al. were able to demonstrate that even devices with IOMMUs can be vulnerable to DMA attacks: In particular, an IOMMU can only enable protection at *page granularity*, which not necessarily coincides with the size of the DMA buffer. DMA devices are still able to read and write beyond their intended buffer, accessing other structures contained on the page [12].

The attack by Markettos et al. (Thunderclap) highlights one important requirement for DMA access control: protection needs to be enforced at *byte granularity*. Otherwise, programming errors in the hardware-software-interface can enable exploitation by DMA devices.

Capability architectures are a common mechanism for byte-granular memory access control. Therein, capabilities are *unforgeable references* to a *segment* in memory, identified by starting address and length. However, the implementation of capabilities for all modern approaches lives in the CPU. Thereby, an extension of capabilities to protect DMA devices as well is non-trivial and has not been attempted before [13]. In particular, in a paging-based scheme, capabilities are restricted to a singular virtual address space and cannot easily be shared with other processes or devices.

Hence, in this ongoing research project, we are constructing a capability architecture that can universally protect memory from both hardware and software adversaries. We implement the enforcement of capabilities in the *northbridge*, which is in the unique position of controlling all accesses to system memory *and* MMIO peripherals.

1. See, for example, the documentation of the Xilinx AXI DMA: https://docs.amd.com/r/en-US/pg021_axi_dma/AXI-DMA-v7.1-LogiCORE-IP-Product-Guide

Thereby, we make memory protection a responsibility of the *front-side bus* instead of the *CPU*. By carefully designing our capability representation, we achieve *full* interoperability with 64 bit addresses.

1.1. Related Work

While MPUs and IOMMUs are clearly inadequate for protecting memory from DMA devices as laid out above, not all capability systems are approriate either. Historical capability systems such as Plessey System 250 [17] and those summarized by Levy [11] did not support DMA at all. Hence, there was no need to protect against DMA attacks. More recent capability systems such as CHERI [18] and RV-CURE [10] are focused on protecting *software* from *software adversaries*, especially, exploitation of programming errors. This is motivated by the plethora of memory-related programming errors that continues to be discovered in modern software [9]. To this end, they implement capabilities *within the CPU pipeline* (and crucially, *before or during address translation*) for efficiency, allowing DMA devices to bypass the protections. Also, capabilities are only valid *within one virtual address space*. Thereby, they are not *designed* to provide security from DMA attacks, although the authors of CHERI have recently proposed this as future work [13].

The issue of protecting memory from non-CPU devices has also been addressed from the point-of-view of rack-scale or distributed computing, again utilizing capabilities. SemperOS by Hille et al. [7] defines a distributed capability scheme in a topology built around a network-on-chip (NoC), where microkernels on compute elements explicitly manage and enforce capability-based protection. While the scheme can be employed for protecting any type of device, it offers no compatibility with legacy software or devices and relies on the NoC topology, which (as of now) is not representative of embedded devices and even most workstations and servers. CEP by Azriel et al. aims at protecting Non-Volatile Memory (NVM) with capabilities without requiring ISA changes. To this end, the authors add a CHERI-based microcontroller into the NVM memory controller, enforcing access control with byte granularity. The scheme only considers transactions between CPUs and NVM, as it relies on trusted software to relay *handles* to CHERI capabilities to userspace processes. No mechanism for protecting DRAM or non-CPU DMA-capable devices is outlined [1].

1.2. Contribution

We are designing a novel capability architecture that enforces access control for both software and hardware components on embedded realtime systems with byte granularity. In particular, Northcape satisfies the following novel functional requirements: compatibility with legacy DMA devices, MMIO peripherals, operating systems and applications; exclusive access to a segment using *locking*; a *cooperative* hardware-accelerated reference counting scheme for memory management; efficient *revocation* of capabilities *without* leaking any secrets to the OS; *device-interpreted* bits associated with each capability to support device-specific restrictions and compatibility with industry-standard buses such as AXI or PCIe and CXL.

2. Northcape: Overview

This section introduces key concepts of the envisioned Northcape system. Appendix A discusses the implementation plans in more depth.

2.1. Terminology and Research Questions

Northcape generalizes bus masters such as CPUs, NICs, accelerators etc. as *data users*, and bus slaves including memory controllers and MMIO peripherals to *data stores*. Data users run one or more *tasks* which can be implicit (e.g. sending and receiving packets on a NIC) or explicit (modules on a CPU). Data stores contain *segments* identified by a *physical address* interpreted by northbridge and slave and a byte-granular *length*. To this end, the northbridge maintains a *capability metadata table* (CMT) in DRAM. Finally, tasks hold *capabilities* which authorize them to access segments within the scope of the *permissions* specified at *capability creation time*.

2.2. Security Model

The Northcape security model generalizes confused deputy attacks as well as compromised devices in the following way. An honest data user u_h (conceptually: CPU) is running an honest task t_h (e.g., a cryptographic library) that operates on a segment s_h. Let the trusted computing base (TCB) refer to the capability-enforcing northbridge, the data store holding s_h, u_h and t_h. A probabilistic polynomial-time (PPT) adversary $\mathcal{A}$ can compromise all tasks and devices that are not part of the TCB. Physical attacks like bus snooping and speculation attacks are out of scope. We define two predicates:

1) t_h has an exclusive lock on s_h via the capability mechanism *and* $\mathcal{A}$ has a capability for s_h.
2) t_h has *no* exclusive lock on s_h *and* $\mathcal{A}$ does not have a capability for s_h.

As long as *one of* the predicates is true, Northcape needs to ensure the following properties:

1) $\mathcal{A}$ cannot distinguish any bit $b \in s_h$ from random with non-negligible probability.
2) The likelihood of $\mathcal{A}$ modifying a bit $b \in s_h$ *without* t_h *detecting this by incurring a trap on access to* s_h is negligible.

2.3. Implementation of Capabilities

In Northcape, a capability token c conceptually consists of an *identifier* n, a *MAC tag* σ and an offset o (see Section A.1 for details). n identifies an entry in the CMT, referencing a *segment* in a data store. o identifies a relative starting position in the segment for each access. σ serves the *unforgeability* of the token. It is computed over the CMT entry associated with n, which includes a nonce to prevent use-after-reallocation and replay attacks, akin to RV-CURE [10]. An additional similarity is our encoding, where n and σ are stored in the high bits of a 64-bit pointer, leaving the lower bits for o. This ensures compatibility with legacy 64-bit pointers, including pointer arithmetic *within one segment* [4, 10]. A key difference to RV-CURE is that we provide no specific instructions for converting pointers into capability tokens. All pointers in the system are *implicitly* capabilities and all accesses protected.

We differentiate between two types of capabilities, *direct* and *indirect*, shown in Figure 1. A *direct* capability *owns* the segment it is pointing to, i.e., the direct capability can be used to *revoke* all indirect capabilities referencing the same physical segment. Direct capabilities can also be *sliced* for the creation of new, non-overlapping direct capabilities and *swapped* to disk. Indirect capabilities are *derived* either from direct or other indirect capabilities, restricting access in both cases. The former is used to provide restricted sub-object capabilities, akin to CHERI and RV-CURE [10, 18]. Also, cloning and dropping operations that increase and decrease the reference count without changing permissions are provided, facilitating efficient memory management for shared capabilities. Locking of both direct and indirect capabilities is supported, ensuring exclusive access. For indirect capabilities, as *overlapping* indirect capabilities can exist, the owning direct capability is locked transparently. However, indirect capabilities cannot be used for creation of direct capabilities and revocation of indirect capabilities.

Crucially, in the Northcape system, *direct* capabilities are *exclusively used* by the *allocator*. Thereby, in the face of memory pressure or crashes, the allocator can revoke capabilities, thus preventing tasks from *stealing* memory. This invalidates the entry in the CMT and creates a new entry, allowing the physical segment to be repurposed. The design of the CMT in conjunction with the use-after-reallocation protection ensures that all further uses of indirect capabilities referencing the revoked direct capability cause a *bus error*. Revocation further overwrites the physical segment with 0-bytes to ensure no secrets are leaked to the allocator. Overwriting segments on revocation along with the *locking* mechanism we will introduce shortly allows us to *define a TCB that does not contain the allocator*. A second benefit of this strategy is that it allows the allocator to allocate *direct* capabilities for segments *larger* than what was requested. Efficient allocation algorithms for segmented memory commonly define a *minimal* segment size to limit external fragmentation, possibly allocating a larger segment than requested [21]. In Northcape, the allocator can afterwards derive an *indirect* capability with exactly the requested size, ensuring no over-reading or over-writing of the segment is possible in the application. The allocator can also safely store metadata in a segment inaccessible to the application.

For the purposes of bootstrapping the system after a power cycle, the northbridge creates a well-known *root capability* on reset. This is a direct capability that owns one segment comprising the *entire physical address space* and is encoded in a way such that naïve use of 32-bit physical addresses is *interpreted* as accessing the root capability at the corresponding offset.

2.4. Operations

Let n_w refer to the word length of the system in bits (intended to be ≥ 64). Let R refer to read, W to write, X to execute permissions. We support the following abstract operations on capabilities:

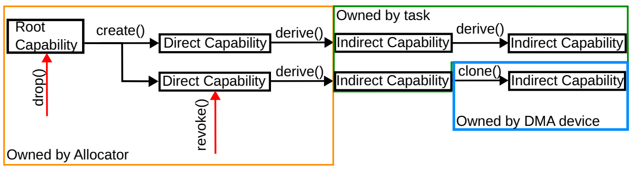

Figure 1: Types of capabilities in the Northcape system.

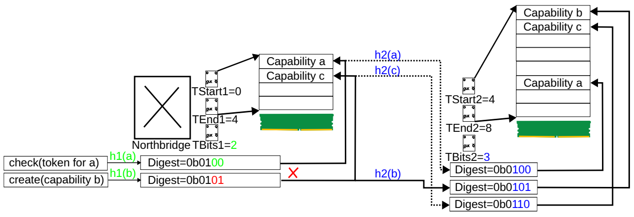

Figure 2: Capability metadata table CMT during expansion after a hash collision.

$create(c_a, l, b, p) \longrightarrow c'_a, c_b$ On input a capability c_a for a segment with offset o_a and length l_a and reference count 0, a length $l \leq l_a$, device-specific permissions $b \in \{0, 1\}^{n_w}$ and capability permissions $p \in \{R, W, X\}$, returns created capability c_b with offset o_a, length l, device-specific bits b and permissions p and reduces l_a by l. Destroys c_a if $l = l_a$.

$merge(c_a, c_b) \longrightarrow c_m$ On input capabilities c_a and c_b pointing to physically neighboring segments, creates a *merged* capability c_m with offset o_a and length $l_a + l_b$. The $create$ and $merge$ operations are intended only for the memory allocator. A comprehensive study by Wilson et al. [16] has pointed out that splitting ($create$ in our model) and merging segments are the only operations that general-purpose slab memory allocation algorithms perform in practice. Merging is particularly useful for reducing fragmentation when an application allocates and returns many small segments at once.

$derive(c_a, l, o, p) \longrightarrow c_i$ On input a capability c_a, a length $l \leq l_a$, an offset $o \geq 0$ such that $l + o < l_a$ and permissions $p \subset p_a$ creates an indirect capability with offset $o_a + o$, length l and permissions p. $derive$ is used to create indirect capabilities, which are exclusively used by all tasks and devices except the memory allocator.

$lock(c_a, tid) \longrightarrow b$ On input a capability c_a and a task identifier $tid \in \{0, 1\}^n$, attempts to *exclusively lock* the capability for the task identified by tid. Honest tasks choose tid uniformly at random and specify it for every access to a locked capability. Returns a bit $b \in \{0, 1\}$ indicating success or failure. On success, no other task including the memory allocator can access the segment identified by c_a unnoticed, *even if it has a capability that is a sub- or superset of* c_a, minimizing the TCB. A *lockable* permission prevents abusing $lock$ for DoS.

$unlock(c_a, tid) \longrightarrow b$ On input a capability c_a and a task identifier tid releases the exclusive lock of tid on c_a.

$clone(c_a) \longrightarrow b$ On input a capability c_a, increases the reference count of the capability. Returns a bit $b \in \{0, 1\}$ indicating success or failure.

$drop(c_a) \longrightarrow b$ On input a capability c_a, reduces its reference count. Returns 0 if the reference count of the capability is not zero after completion or 1 if it is zero. If c_a is an indirect capability, it is destroyed and the operation recurses on its parent. If the capability is *locked*, *drop* will not decrease the reference count below 1. *clone* and *drop* are used for the cooperative reference counting mechanism, which is useful for shared segments.

$revoke(c_a) \longrightarrow c'_a$ On input a *direct* capability c_a, destroys the CMT entry associated with the capability and creates a new direct capability c'_a for the segment identified by the capability with full permissions. *Overwrites* the segment with 0-bytes. Returns the new capability. $revoke$ is intended for the allocator to reclaim memory on process crash or memory pressure, its use might lead to bus errors when tasks continue using indirect capabilities for the segment. It only accepts *direct* capabilities to prevent tasks from *stealing* memory. This is the main difference between Northcape and Capstone's linear capabilities: By overwriting the segment on revocation, we can ensure no secrets are leaked *without* having to support uninitialized capabilities, making the implementation easier. Unfortunately, Capstone does not discuss these trade-offs or propose a hardware implementation of uninitialized capabilities [20].

$mkXonly(c_a) \longrightarrow c_s$ On input a capability c_a, creates an *execute-only capability*, i.e., an *indirect* capability with permission $p = \{X\}$ from a capability c_a.

$calls(c_s, c_r)$ This operation only applies to CPUs. On input execute-only capabilities c_s, c_r, locates the segment s identified by c_s. It then calls a function defined starting at the *second* word of s, passing the first word of s as well as c_r as arguments. c_r can be used to *return* from the call by performing a second $calls$. $mkXonly$ and $calls$ implement protected procedure calls for subsystem calls (e.g., a network stack) akin to Plessey System 250 [17].

In Northcape, the northbridge implements the operations defined above. It provides an MMIO interface which data users can utilize to invoke the aforementioned operations, conveying parameters and responses. Additionally, for *each* bus transaction, the northbridge expects to receive a capability token in the *address lanes* of the bus. Optionally (for tasks wishing to *lock* capabilities), the data user can provide a *task identifier* chosen by the task in the *User* vector of the bus. This is used by the northbridge to ensure that only the lock holder can perform accesses to a segment and is based on a similar strategy introduced by Bahmani et al. [2]. The northbridge refers to the capability metadata table (CMT) to determine whether the access is permissible under the presented token and task id.

2.5. Data Structures

The CMT is a hash table in main memory, which is maintained by the northbridge. It contains one entry for each active capability in the system. Lookup is performed based on the identifiers n contained in the token. The MAC tag σ for each entry is stored in the table to facilitate fast comparison between the actual and the presented tag. Thereby, the northbridge can ensure that the token referring to the entry is valid *at the time of the access*. In conjunction with dropping and revoking capabilities, this is the mechanism used to provide *temporal safety*. Security from replay, use-after-free and use-after-reallocation attacks requires the CMT to also include a *Nonce* for each entry. It is based on a monotonically increasing counter that gets incremented after each capability operation and is input when computing σ.

In addition to the attributes given above, entries for indirect capabilities also contain a *capability* for their *parent*, i.e., the capability they were derived from. When validating the access to an indirect capability, the northbridge *recurses* to its parent until it reaches the root *direct capability*. Thereby, when the direct capability is revoked, all uses of the derived capabilities cause *validation errors*. Hence, no explicit sweeping of capabilities to revoked segments and no lifetime tokens are needed, which is an improvement compared to existing revocation schemes such as CHERIvoke [19].

In order to allow a high number of active capabilities as well as memory efficiency, the CMT is optionally *resizable*. To this end, an adapted *extendible hashing* scheme [5] is used, utilizing the capability identifier n as key. As in the original scheme by Fagin et al., a (non-cryptographic) hash function h_1 computes a digest of the capability identifier n. We then use the t_{bits} least significant bits of the digest to map a number to a directory in our hash table. However, we *inline* an implicit 1-sized bucket into each directory entry in the hash table, leaving us with a single contiguous table. There are three reasons motivating this decision: First, all entries are padded to the same size (see Section A.2), which allows us to match the size of each hash table directory with the entries. Second, we can maintain the entire CMT in a single segment, which makes co-existence with the memory allocator in the OS significantly easier. Finally, this allows us to perform a lookup in one memory access.

Previous research by Friedman et al. [6] has pointed out that hash tables are not necessarily suitable for embedded real-time applications, as duration of retrieval and insertion operations might vary significantly. This is primarily caused by the *collision handling* algorithm: When an entry that is to be inserted into the table is mapped to a directory that contains a valid entry, the collision handling algorithm finds a way to reorganize the table such that the new entry can be inserted. Commonly used algorithms such as open addressing or chaining have variable run time depending on the internal table state, which is not ideal for real-time applications [6]. Most research involving hash tables aims at providing *amortized* constant latency for insertion and look up; the problem of *guaranteed* bounded latency for all hash table operations appears to be understudied in the literature. Thus, our proposed CMT adapts the strategy by Friedman et al. to the extendible hashing scheme: We maintain an empty *shadow CMT* that has twice as many bits as the current CMT. When an insert collision occurs for a capability c_n that is to be created, we choose a new hash function h_2 and select t'_{bits} of the shadow CMT as t_{bits+1}. We insert the new entry for c_n into the shadow table using h_2 and t'_{bits}. For each CMT access, we check both the original and the shadow CMT. As proposed by Friedman et al., we *rehash* each entry from the original CMT into the shadow CMT when it is *accessed*. This is illustrated in Figure 2. At some point, the original CMT is empty. We then update our internal metadata, promoting the shadow CMT to the active CMT, and inform the OS allocator that the CMT was freed via a status register. We then allocate a new shadow CMT and re-start the algorithm if needed. Collisions *during CMT expansion* can be handled using an *overflow buffer* [8], i.e., a small content-addressed memory in the northbridge.

3. Security Discussion

As laid out in Section 2.2, Northcape has two major security goals: First, an adversary cannot *guess* the information in a segment that it is not explicitly allowed to access. Second, an adversary cannot *change* the information in the segment without the honest task noticing (by receiving a bus error on access). In other words, Northcape protects the *confidentiality* and *integrity* of data contained in segments. The adversary is able to compromise all tasks and devices outside of the trusted computing base (TCB), which consists of the honest task owning the segment, the data user it is running on, the northbridge and the data store the segment is located in.

The locking mechanism allows us to add important distinctions: An adversary might not have been given a capability for a segment *at all*. An adversary might also *have* a capability for the segment, but the honest task holds a *lock* on the segment. Thereby, once a task gains a lock on a segment, it can be assured that the data in the segment cannot be accessed by anyone outside of the TCB.

Security in the first scenario relies on *unforgeability* of the capability tokens[2]. In order to successfully forge a token, an adversary needs to guess both the identifier n and the MAC tag σ. In the envisioned Northcape implementation, n will have 46 bits and σ will have 16. Thereby, in theory, the adversary has to guess 2^{62} bits for a forgery. However, for efficiency in the implementation, n might be predictable (e.g., derived from a counter). Thus, an adversary only needs to guess σ. Assuming a strong MAC is used, the likelihood of a successful forgery is $\frac{1}{2^{16}}$. Note that the adversary can only perform *online* guesses, and on a wrong guess, the TCB can immediately separate the compromised device from the bus. Thereby, we believe that the unforgeability of our system is adequate. Note that the adversary can trivially guess the root capability; we make the assumption that it is destroyed in the boot process before the adversary becomes active. For the sake of argument, assume that this is not done; in this case, we consider the attack to fall under the scenario where the adversary has a capability to the segment of interest.

Security in the second scenario relies on both the unforgeability of capability tokens and the secrecy of the *task identifier* chosen by the honest task. An adversary with knowledge of the task identifier and a capability for the segment of interest can *impersonate* the lock holder and gain access to the locked segment. Thus, the task identifier is chosen from an unpredictable RNG and has a large size of 2^{64} bits in the envisioned implementation, making guessing the identifier futile (and easily detectable). The identifier is stored in a write-only architectural register of the CPU that runs the honest task, which is part of the TCB, making it impossible for the adversary to read it. Remember that bus snooping and other physical attacks are out of scope, leaving the adversary no way to learn the task identifier besides guessing.

Note that again, the root capability does not give the adversary an advantage here – when the honest task locks the segment that it is working on, it effectively locks *the entire physical address space*. Everything argued earlier about the locking mechanism is still true in this case.

2. As Northcape relies on *implicit* capability delegation for backward compatibility, it cannot detect *correctly* guessed forged tokens.

One interesting scenario has not been discussed: an adversary could use the $drop$ and $revoke$ operations to gain access to the segment. Both operations require a capability for the segment of interest. Thereby, according to our security model, the honest task must hold a *lock* on the segment. The practical implication of this assumption is that tasks need to either *trust* the allocator (that is not in the TCB) not to hand capabilities for segments containing secrets to untrusted tasks or *lock* segments containing secrets. Remember that *drop* does not destroy a capability when it is locked. Hence, the adversary cannot abuse it to forcefully reclaim the memory. Also, $revoke$ *overwrites* and *invalidates* the memory. Thus, while $revoke$ technically lets the adversary modify the data in the segment, it does not learn any private task information and all attempts by the honest task to use the segment lead to a validation error, making the attack detectable.

4. Plans for Evaluation

We plan an implementation of Northcape on a Digilent Genesys 2 FPGA board, featuring a Xilinx Kintex-7 FPGA and 1 GiB of DDR-3 DRAM. Our design will comprise a RISC-V CPU and an AXI interconnect as northbridge. We will use a Xilinx DMA controller connected to an Ethernet MAC as exemplary DMA device. On the software side, we plan on using the Zephyr RTOS.

We currently plan on performing both microbenchmarks and a real-world evaluation. First, we will investigate how many active capabilities are present in a representative real-time application at any point in time. We will also measure the average and maximum depth of indirect capabilities, which directly influences the memory access latency. We will proceed to execute a set of representative benchmarks from the SPEC CPU suite. Finally, we will evaluate whether Northcape manages to ensure real-time operation of a representative application.

5. Summary and Future Work

We have presented Northcape, a proposal for a hardware capability system that provides byte-granular access control for both software and DMA devices. Therein, Northcape is fully backwards compatible with existing DMA devices, bus infrastructure and software.

Additionally, Northcape incorporates well-received features from other hardware capability systems. Northcape supports sub-object capabilities, similar to CHERI [18] and RV-CURE [10], exclusive access to capabilities similar in concept to linear capabilities in Capstone [20] (but with a simplified implementation) and revocation with lower overhead and simpler implementation than, e.g., CHERIvoke [19]. We also introduce promising new features: device-interpreted restrictions in the form of annotations to capabilities and an encoding scheme for capability tokens that supports both large identifiers and large offsets.

We are confident that we have pointed out that Northcape fulfils the envisioned security goals. However, it remains to be proven that Northcape can deliver on the performance promises. To this end, we are currently in the process of implementing Northcape on an FPGA to facilitate both microbenchmarks and real-world evaluations in realistic scenarios. Especially, we are looking to investigate the impact our system has on latency and real-time capabilities of a computing system.

Additionally, Northcape enables interesting possibilities for future research. Northcape has potential uses in rack-scale computing as *holistic* access control model for all system devices. Exploring potential extensions of Northcape that incorporate non-volatile capabilities is another interesting research direction. Finally, Northcape might also be an interesting platform for *para*-virtualization. The token-based addressing can be used to safely map devices to VMs without requiring additional hardware, and for efficient sharing of segments between VMs and the hypervisor. Calling execute-only capabilities can also serve as a type-safe alternative to hypercalls and potentially even offer latency advantages as no change of address space is incurred.

References

[1] L. Azriel, L. Humbel, R. Achermann, A. Richardson, M. Hoffmann, A. Mendelson, T. Roscoe, R. N. M. Watson, P. Faraboschi, and D. Milojicic. Memory-Side Protection With a Capability Enforcement Co-Processor. *ACM Transactions on Architecture and Code Optimization*, 2019. URL https://dl.acm.org/doi/10.1145/3302257.

[2] R. Bahmani, F. Brasser, G. Dessouky, P. Jauernig, M. Klimmek, A.-R. Sadeghi, and E. Stapf. {CURE}: A Security Architecture with {CUstomizable} and Resilient Enclaves. In *SEC*, 2021. URL https://www.usenix.org/system/files/sec21-bahmani.pdf.

[3] G. Beniamini. Project Zero: Over The Air: Exploiting Broadcom's Wi-Fi Stack (Part 1), 2017. URL https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html.

[4] D. Chisnall, C. Rothwell, R. N. Watson, J. Woodruff, M. Vadera, S. W. Moore, M. Roe, B. Davis, and P. G. Neumann. Beyond the PDP-11: Architectural Support for a Memory-Safe C Abstract Machine. *ACM SIGARCH Computer Architecture News*, 2015. URL https://dl.acm.org/doi/10.1145/2786763.2694367.

[5] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing—a fast access method for dynamic files. *ACM Transactions on Database Systems*, 1979. ISSN 0362-5915. URL https://dl.acm.org/doi/10.1145/320083.320092.

[6] S. Friedman, A. Krishnan, and N. Leidenfrost. Hash tables for embedded and real-time systems. Technical report, Washington Univ., 2003. URL https://core.ac.uk/download/pdf/233199149.pdf.

[7] M. Hille, N. Asmussen, P. Bhatotia, and H. Härtig. {SemperOS}: A Distributed Capability System. In *USENIX ATC*, 2019. URL https://www.usenix.org/system/files/atc19-hille.pdf.

[8] Z. István, G. Alonso, M. Blott, and K. Vissers. A Hash Table for Line-Rate Data Processing. *ACM Transactions on Reconfigurable Technology and Systems*, 2015.

[9] V. Katos, S. Rostani, P. Bellonias, N. Davies, A. Kleszcz, S. Faily, A. Spyros, A. Papanikolaou, C. Ilioudis, and K. Rantos. State of Vulnerabilities 2018/2019 - Analysis of Events in the life of Vulnerabilities. Technical report, ENISA, 2019. URL https://www.enisa.europa.eu/publications/technical-reports-on-cybersecurity-situation-the-state-of-cyber-security-vulnerabilities.

[10] Y. Kim, A. Kar, J. Lee, J. Lee, and H. Kim. RV-CURE: A RISC-V Capability Architecture for Full Memory Safety, 2023. URL http://arxiv.org/abs/2308.02945.

[11] H. M. Levy. *Capability-Based Computer Systems*. 2014. ISBN 978-1-4831-0106-4.

[12] A. T. Markettos, C. Rothwell, B. F. Gutstein, A. Pearce, P. G. Neumann, S. W. Moore, and R. N. Watson. Thunderclap: Exploring vulnerabilities in operating system IOMMU protection via DMA from untrustworthy peripherals. In *NDSS*, 2019. URL https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_05A-1_Markettos_paper.pdf.

[13] A. T. Markettos, J. Baldwin, R. Bukin, P. G. Neumann, S. W. Moore, and R. N. M. Watson. Position Paper:Defending Direct Memory Access with CHERI Capabilities. In *Proceedings of the 9th International Workshop on Hardware and Architectural Support for Security and Privacy*, 2021. URL https://dl.acm.org/doi/10.1145/3458903.3458910.

[14] A. Mera, Y. H. Chen, R. Sun, E. Kirda, and L. Lu. D-Box: DMA-enabled Compartmentalization for Embedded Applications. In *NDSS*, 2022. URL https://www.ndss-symposium.org/wp-content/uploads/2022-53-paper.pdf.

[15] P. Stewin and I. Bystrov. Understanding DMA Malware. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2013.

[16] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Memory Management*, 1995.

[17] S. Winkler. *Computer Communication - Impacts and Implications*. ACM, 1972. LCCN 79319088.

[18] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014. URL https://ieeexplore.ieee.org/document/6853201.

[19] H. Xia, J. Woodruff, S. Ainsworth, N. W. Filardo, M. Roe, A. Richardson, P. Rugg, P. G. Neumann, S. W. Moore, R. N. M. Watson, and T. M. Jones. CHERIvoke: Characterising Pointer Revocation using CHERI Capabilities for Temporal Memory Safety. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019. URL https://dl.acm.org/doi/10.1145/3352460.3358288.

[20] J. Z. Yu, C. Watt, A. Badole, T. E. Carlson, and P. Saxena. Capstone: A Capability-based Foundation for Trustless Secure Memory Access. In *SEC*, 2023. URL https://www.usenix.org/conference/usenixsecurity23/presentation/yu-jason.

[21] A. S. Yurchenko. Algorithm of dynamic segmented memory allocation. *Cybernetics*, 1981. URL https://doi.org/10.1007/BF01307036.

Field	Size (Bits) by Type	Remark
Type	2	Identifies sizes of capability fields
MAC tag	16	Truncated tag σ
Number	14/46/30/22	Capability number n
Offset	32/0/16/24	Offset o into segment

TABLE 1: Encoding of Northcape capabilities into 64-bit addresses, from MSB to LSB.

A. Proposed Representations

This chapter points out how we plan to represent capabilities and CMT entries in the ongoing implementation of the Northcape system. We also discuss how our scheme can be extended to systems that feature multiple northbridges.

A.1. Capability Representation

Table 1 details how we envision capabilities to be encoded into 64-bit addresses. We provide four different types of capabilities, such that we can provide both a large number of capabilities for small segments and a few capabilities for very large segments. To this end, the two most significant bits of the capability encode its type. The actual capability token contains type, MAC tag and capability number. The address can further contain an offset o into the capability, which indicates which word in the segment the task wants to access. We provide a zero-length capability where no offset is contained; this is intended for scenarios where a bus transaction access starts at the beginning of the segment and the length of the transfer is indicated by AXI burst size.

We have carefully selected the encoding scheme such that we can define a root capability in a way that makes it compatible with legacy software: For the root capability, type, tag and number fields are zero. Thereby, as long as the root capability exists, an access to an ordinary 32-bit physical address a is *interpreted* as accessing the *segment identified by the root capability* at *offset* a. By mapping the physical addresses of memory and MMIO peripherals into a 32-bit address range, legacy software will *implicitly* use the root capability to access them. Thereby, Northcape provides full backwards compatibility with legacy software. In particular, this allows us to re-use existing software such as the zero-stage loader without modification. Destroying the root capability after initial booting allows us to reach the promised security level.

A.2. Capability Metadata Table

The proposed representation of the capability metadata table (CMT) entries is shown in Table 2. In addition to the capability types introduced earlier, we have provided a paged-out capability that is especially intended for lazy loading of libraries. Access to such a capability will raise an IRQ, which allows the OS to fault the corresponding segment in before continuing execution. To this end, we have designed the paged-out capability to be convertible into a direct capability in-place. On the same note, we have added a Copy-on-Write flag for efficient implementation of fork() system calls. We have chosen to add

Field	Size (Bits) by Type	Applies to type	Remark
Type	3	all	Capability Type (direct, indirect, paged)
Base	32	all but paged	Segment start in physical memory
Pagefile number	64	paged	Identifier of the backing store of the capability
Parent	64	indirect	Capability for capability from which this one was derived
Length	32	all	Length of the segment in bytes
Ref. count	16	all but paged	Remaining references
Lock Holder	55	direct	Task id of the lock holder
User	64	direct, paged	device-specific bits assoc. with the capability
R	1	all	Read permission
W	1	all	Write permission
X	1	all	Execute permission
Locked	1	direct, paged	Lock on segment is held
Lockable	1	direct, paged	Lock on segment is allowed
CoW	1	all	Page is copy-on-write
Tag	16	all	MAC tag
Nonce	32	all	Counter to prevent use-after-reallocation etc.

TABLE 2: Encoding of Capability Metadata Table entries.

a Lockable permission to our entries to prevent tasks *abusing* the locking primitive for denial-of-service.

The memory controller of our evaluation platform (Xilinx MIG on Kintex-7) can read up to *256* Bits from DRAM per cycle. Thereby, we elected to limit the size of CMT entries to 256 bits, allowing us to read one entry in each bus clock cycle. This forced us to make a few minor adjustments to the entries:

First, the segment base address is limited to 32 bits. However, this should be sufficient for most embedded real-time systems, which is the focus of our project. An implementation of our scheme can silently increase this to 64 bits without breaking compatibility.

Also, we were only able to store 55 of the intended 64 bits of the task identifier in the lock holder field.

We believe that these adjustments do not jeopardize security of our scheme and generalizability of our results, however, as sufficiently many bits remain.

A.3. Extension to Multiple Northbridges

While the current proposal considers a single northbridge, using a partitioning scheme similar to the one proposed by Hille et al. [7], we can also support multiple northbridges in the system: Let there be n_b northbridges. For each capability identifier, we can encode the northbridge responsible for the capability into the *upper-most* $\lceil log_2(n_b)\rceil$ bits of each identifier n. Thereby, each northbridge maintains a partition of the capability space. On each capability lookup, a northbridge determines based on the upper bits of the identifier which northbridge maintains the entry for the capability and forwards the request if needed (see Figure 3). This can be implemented by adding a standard interconnect that connects each pair of northbridges in the system, possibly even at a slower clock

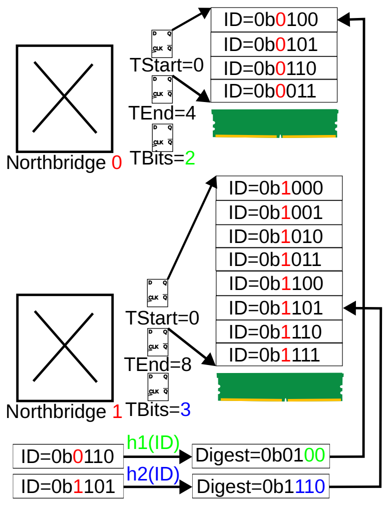

Figure 3: Implementation of CMT in a multi-northbridge system.

speed than the front-side buses. Each northbridge can then manage a CMT independently in its local memory using the same mechanisms as mentioned above.