Understanding the Security Impact of CHERI on the Operating System Kernel

Zhaofeng Li, Jerry Zhang, Joshua Tlatelpa-Agustin, Xiangdong Chen, Anton Burtsev
University of Utah

Abstract

Capability Hardware Enhanced RISC Instructions (CHERI) is a set of hardware extensions that allow enforcement of spatial and temporal safety for unsafe programming languages like C. CHERI utilizes an idea of hardware *capability* pointers to enforce bounds checks on all memory accesses and a hardware-assisted revocation scheme to enforce temporal safety. In theory, CHERI offers a surprising mix of practical adoption and strong security guarantees for traditionally unsafe environments like operating system kernels, i.e., capability extensions block a range of software safety-related vulnerabilities common to low-level systems code while requiring only a modest engineering effort.

Our work takes a deep look at the potential impact of CHERI on the security of commodity operating system kernels. We analyze a total of 439 kernel vulnerabilities in Linux and FreeBSD kernels. Our analysis shows that CHERI can block 35%-61% vulnerabilities depending on whether temporal safety is enabled in the kernel. Enabling CHERI requires a modest effort, e.g., porting the FreeBSD kernel to support pure-capability mode of execution took 7 months. Finally, we estimate that compared to Rust, CHERI blocks 70% of vulnerabilities (38% if revocation is off), a number lower than blocked by Rust, 84%, but at a much lower effort. We hope that our work improves the understanding of potential effort and benefits of capability protection in commodity kernels.

1. Introduction

Modern operating systems face hundreds of exploitable vulnerabilities every year. They are huge, complex codebases, e.g., a recent version of the Linux kernel, v6.17, is over 40 million lines of code. A typical kernel implements dozens of complex subsystems – memory management, process address spaces, scheduling, interrupt processing, network and storage stacks, page and buffer caches, file systems, etc. – each with intricate logic and cross-subsystem dependencies: a combination of manual and automated (reference counted) object lifetimes, numerous correctness and security invariants, support for dynamic extensions and generic interfaces, complex hotplug and power management protocols, intricate synchronization disciplines, and, finally, numerous optimizations that often trade modularity and code clarity for performance. Moreover, modern kernels evolve at a rapid development rate, e.g., the Linux kernel averages 75,000-90,000 commits per year.

In addition to inherent semantic complexity, modern kernels are often developed in unsafe programming languages. Right now, C remains the dominant language in commercially available kernels like Linux, FreeBSD, MacOS, and Windows. Unsafe programming languages, combined with complexity and rapid development results in a steady stream of errors and vulnerabilities. While the understanding of which software flaws should be designated as potentially exploitable vulnerabilities evolved over the years, which likely resulted in vulnerability undercounting in the past, it is safe to say that a modern full-featured kernel like Linux can identify several hundreds of vulnerabilities a year.

To address a growing number of kernel vulnerabilities, modern kernels deploy a range of state-of-the-art defenses: address space layout randomization (ASLR) [39], control-flow integrity [1], automatic stack and heap initialization [8, 35], data execution prevention [11], non-executable stack [11], stack canaries [10], allocator hardening [40, 26], and various degrees of sanitization. Nevertheless, the ever evolving understanding of attack and defenses allows attackers to bypass the most sophisticated mitigation mechanisms [34, 6, 5, 23, 19, 32]. In the last decades, attackers have significantly improved both the techniques and automation of turning a chain of seemingly benign errors into powerful exploits [20].

Facing the need to address a growing number of flaws and vulnerabilities, modern kernels are becoming increasingly open towards non-traditional approaches that can change conceptual balance of exploitation in the kernel, for example, adopting the use of safe programming languages like Rust [24]. Unfortunately, while a promising direction, adopting Rust requires significant development effort for both developers and maintainers [24, 9].

An alternative way of improving the security of the kernel is to adopt a language-agnostic approach that can enforce a range of safety properties for existing kernel code. Capability Hardware Enhanced RISC Instructions (CHERI) is a set of hardware extensions [46] which can enforce a subset of spatial and temporal properties in hardware [37]. At a high-level, CHERI replaces traditional memory references with hardware *capabilities*, fat hardware pointers that store bounds information along with the memory address [42]. The hardware transparently enforces bounds checks on all memory dereferences and implements a range of mechanisms that protect capabilities from unauthorized modifications in registers and in memory.

CHERI offers a surprising mix of practicality and security. A CHERI-aware compiler toolchain supports transparent recompilation of existing C/C++ kernel code. Minimal modifications are needed to accommodate low-level programming idioms that cannot be expressed at the level of the programming language [44]. At the same time, CHERI can stop a wide range of low-level vulnerabilities related to spatial and temporal memory safety through a combination of bounds checks and revocation schemes [47, 45, 17].

At the moment, we still have a limited understanding of both the security guarantees provided by the CHERI architecture and opportunities for attacking capability systems [43]. Prior analysis focused on the impact of CHERI on user applications, in which traditional memory safety dominates the notion of security [41]. We argue, however, that the complexity of the kernel execution environment and its pervasive use of unsafe abstractions along with the need to maintain numerous security, and semantic invariants creates a set of unique security challenges that cannot be addressed by low-level safety alone. Intuitively, while CHERI capabilities can prevent a range of low-level vulnerabilities, many classes of security attacks such as double free, resource leaks, protocol violations, race conditions, and various semantic errors might be exploitable.

Our work takes a careful look at the impact of CHERI capabilities on the security of the kernel. Specifically, we ask the following questions: What classes (and what fraction) of vulnerabilities found in the kernel can be eliminated by protecting a full-featured operating system kernel with CHERI capabilities? What effort is needed for porting the kernel to enable pure capability mode of execution? And finally, how does hardware-based safety compare to the programming language safety offered by Rust?

To answer these questions, we study a collection of 439 vulnerabilities from the FreeBSD and Linux kernels. We develop a new classification taxonomy centered around the observation that low-level manifestations like out-of-bound accesses, use-after-free, access to uninitialized memory, etc., are results of high-level flaws in the kernel code. In our analysis we track both the high-level cause and low-level manifestation for each vulnerability, which allows us to get a deeper insight into the nature of each flaw and the possibility to block exploitation at both architectural and programming-language levels.

Our analysis shows that CHERI can eliminate large classes of safety-related vulnerabilities, 61%, but only if we assume that a revocation scheme [47, 45, 17] is implemented in the kernel. CHERI blocks errors that manifest as safety symptoms, e.g., out-of-bound accesses, use-after-free, dereferences of invalid pointers, but fails to prevent double free, uninitialized memory accesses, resource leaks, explicit exceptions and panics, and other logical and semantic errors. Note, however, that CHERI blocks the most severe vulnerabilities that potentially lead to privilege escalation, i.e., typical read and write primitives via out-of-bound accesses and invalid pointer dereferences.

Temporal safety is critical for eliminating a large fraction of vulnerabilities. If revocation is not supported (as it is now in the CheriBSD kernel), the total percentage of vulnerabilities blocked by CHERI drops from 61% to 35%, which, of course, significantly alters the effectiveness of the CHERI protection scheme.

Compared to Rust, which can block 84% of kernel vulnerabilities, with revocation enabled CHERI achieves a lower fraction of 70% (38% if revocation is off). Note, to compare against Rust, we analyze the dataset used by Li et al. [25], which is limited to device driver vulnerabilities. On this dataset, CHERI mitigates 70% of vulnerabilities (compared to 61% on the full dataset). Arguably, compared to Rust, CHERI has a smaller TCB which can potentially result in stronger security guarantees. In Rust, a flow in the unsafe code can provide an attacker with a classical write primitive with access to the entire memory. Hardware capabilities significantly narrow the opportunities for exploitation through enforcement of fine-grained pointer bounds and partial control flow integrity.

An engineering effort for enabling pure capability mode in the core kernel is relatively low. Despite the fact that work on supporting capabilities in the FreeBSD kernel started in 2013, enabling the pure capability mode of execution required only 7 months. Furthermore, the effort of enabling CHERI in device drivers and other kernel extensions is very low – often only a couple of lines of code need to be changed. This is important since device drivers constitute 70% of the kernel code in modern kernels like Linux. Moreover, since UNIX kernels share conceptual similarity, one can follow the CheriBSD blueprint for enabling the pure capability mode of execution.

We hope that our work can improve understanding of the security guaranees offered by the CHERI capability architecture as well as accelerate its adoption.

2. Background

2.1. CHERI Architecture

CHERI overview CHERI is a collection of architectural CPU extensions aimed at providing support for enforcing spatial and temporal safety in native code compiled from unsafe programming languages. CHERI introduces the notion of hardware "capabilities" – unforgeable pointers that hold information about the bounds of each memory region along with the memory address itself. The CPU is extended to support a capability register file (e.g., a 129-bit wide register file on ARM Morello CHERI implementation that uses 64 bits to store the address, 64 bits to hold the capability metadata (compressed bounds information, capability type, etc.), and 1 extra bit to store the tag showing that capability is valid). All memory accesses then use capabilities instead of just memory addresses (i.e., regular load and store instructions as well as load instructions relative to the program counter which is also a capability register). On every access, the hardware validates that the access is in-bounds. The compiler relies on special capability load and store instructions and the execution environment to enforce spatial safety. For example, the memory allocator returns capability pointers that it creates (or mints) by reducing the bounds of the capability pointers received from the kernel.

An important security property of the CHERI architecture is that capabilities are "unforgeable" – the only way to create a valid capability is through the privileged `sctag` instruction. Once created, the capability is protected throughout the system – in registers and in memory. Only valid operations are allowed on capabilities, and they follow a set of specific rules. For example, capability bounds can be made smaller but an attempt to increase the bounds invalidates the capability (the "*monotonicity*" rule). This, for example, allows a user-level memory allocator to convert a capability of a large memory region returned by the kernel into smaller capabilities for individual fine-grained allocations.

To protect capability pointers from modifications, a special one-bit tag is stored along with each the capability in memory. If the capability is then tampered with, e.g., by changing its bounds or metadata, the tag is invalidated, which makes the capability invalid. The tag can be either stored in the metadata bits maintained by the memory controller or in a separate memory region [21, 46].

Hybrid and pure modes of execution CHERI supports two modes of execution: *hybrid*, and *pure*. The hybrid mode allows one to combine legacy and new capability-aware instructions in a single binary. In the pure mode, all memory accesses require capability pointers. CHERI relies on compiler and run-time extensions to enable development and execution of the pure capability code. Compiler extensions allow compilation of unsafe C/C++ code into pure capability machine code that uses explicit capability instructions for all memory accesses. The compiler treats all pointer types as capabilities (i.e., 128 bit-wide in Morello implementation) and generates capability-aware load and store instructions that operate on the capability registers for all pointer dereferences.

Control-flow integrity CHERI supports a form of control-flow integrity (CFI) via capability *sealing*. In CHERI, a capability can be sealed – meaning that its address and metadata cannot be changed until used in a corresponding *unsealing* operation. By sealing code pointers into "sentry" (sealed entry) capabilities, CHERI provides code-pointer integrity (CPI) (a subset of control-flow integrity). Each function in CHERI has a sentry capability with its value fixed to the beginning of the function and its bounds covering the entire function. When calling a function through a sentry capability, the CPU unseals the capability before setting the program counter capability – this allows the function itself to freely execute while ensuring that callers may only jump to the start of it. Similarly, return addresses are also sealed as sentries to allow the control flow to be passed back to the caller in a controlled manner.

Temporal safety A flavor of temporal safety is supported in CHERI via capability revocation schemes [47, 45, 17, 2]. At a high-level, when an object allocated on the heap is deallocated, the object enters a quarantine instead of being released into the allocator's free pool. Note that valid capabilities to an already freed object can be saved in memory, loaded in registers of process threads, and be even passed to the kernel. To invalidate capabilities saved in memory, CHERI performs an address space scan (i.e., memory of the process) invalidating capabilities that point into the quarantined objects [47, 45]. In early revocation schemes such scan required a stop-the-world freeze of all threads of the process [47, 45]. Later schemes [2, 17] minimize the freeze by combining the scan with in-line checks for whether a capability points to a quarantined region when it is loaded from memory into a register. The pages that were not yet scanned are marked read only, and if a capability load is attempted on them, it triggers a page fault [17]. CHERI still freezes all threads to inspect their register content and delays revocation until all system calls involving user capabilities exit the kernel.

At the moment, CheriBSD kernel does not implement support for revocation. In the kernel, revocation is challenging due to the high rate of allocation and deallocation (e.g., in the Linux kernel the network subsystem allocates memory for every network packet), and the high performance impact of revocation scans and page faults due to checks on capability loads. It is not yet clear if a practical kernel-level revocation scheme is possible. Yet in our work, we explore both scenarios: with and without support for temporal safety.

Note that CHERI does not support revocation for the objects allocated on the stack, as there is no easy way to implement the quarantine stage for automatic deallocation on the stack without significant performance impact. I.e., a capability to the object on the stack will remain valid if saved to memory or passed to another thread even after the function returns and the stack is reused for other purposes. Similarly, CHERI does not implement revocation of pointers to the objects allocated statically in the data and BSS sections. Modern kernels support dynamic extensions, which can be loaded and unloaded at runtime. This allows an attacker to trigger a use-after-free vulnerability into the data section of a kernel extension which was unloaded, e.g., this is similar to CVE-2023-35829 in which the kernel accesses state of an unloaded Rockchip Video Decoder driver due to the missing `cancel_delayed_work_sync()` invocation needed to cancel the work handler registered with the kernel.

3. Methodology

To understand the possible impact of CHERI capabilities on the kernel, we structure our study around three questions: **Q1.** *What classes of kernel vulnerabilities can be eliminated by CHERI?* **Q2.** *What effort is required to implement a pure capability mode in the kernel?* **Q3.** *How do capability security guarantees compare to the security achieved through programming language safety, i.e., Rust?*

To answer **Q1**, we analyze 439 CVEs from the Linux and FreeBSD kernels to understand whether CHERI prevents exploitation. CVEs from the FreeBSD kernel are a natural candidate, as CheriBSD is the first full-featured kernel that implements the pure capability model of execution. Unfortunately, the FreeBSD CVE dataset is too small: only 101 vulnerabilities over 11 years. We, therefore, extend it with 338 vulnerabilities from the Linux kernel which provides us with a much wider range of kernel errors as well as allows us to compare CHERI with Rust [25]. We pick Linux CVEs as two groups: First, we randomly selected 100 vulnerabilities discovered in 2025. This allows us to analyze a broad class of recent vulnerabilities in the Linux kernel, specifically, after the Linux kernel became a CVE assigning authority, which broadened CVE assignment to a much wider class of kernel flaws. Second, we analyzed 234 Linux vulnerabilities used by the previous study aimed to understand the impact of Rust on the security of the Linux kernel [25]. This group allows us to conduct a head-to-head comparison of the security impact of CHERI and Rust. We manually analyze and classify vulnerabilities (each vulnerability is analyzed by at least two people to ensure confidence in classification).

To answer **Q2**, we carefully analyze the changes introduced to the FreeBSD kernel from 2013 to 2025 and analyze the effort required to implement those changes.

To answer **Q3**, we utilize recent work that analyzed the potential impact of Rust on device driver vulnerabilities in the Linux kernel [25]. We take the dataset used by Li et al. [25] and analyze it to compare whether the same vulnerabilities will be blocked by CHERI.

4. Taxonomy of Kernel Vulnerabilities

Our taxonomy of kernel vulnerabilities is centered around the observation that low-level manifestations like out-of-bound accesses, use-after-free, access to uninitialized memory, etc., are results of high-level flaws in the kernel code. For example, logical errors like a mismatch between allocated and intended object size, incorrect computation of the array indexes, insufficient user-input validation, etc., can all manifest as the single symptom like an out-of-bound access. In our classification we explicitly track both the cause and manifestation for each vulnerability. This allows us to get deeper insight into the nature of each flaw and the possibility to block exploitation with both architectural level solutions like CHERI and higher-level solutions like Rust, a safe programming language with a powerful type system and ergonomic abstractions. A manifestation-centric view allows us to reason about low-level approaches like CHERI as intuitively capabilities should be able to block all CVEs that manifest as low-level safety problems. A cause-centric view, on the other hand, provides us with a higher-level view of the reasons that lead to a vulnerability and whether better programming abstractions like safe programming languages can prevent it.

4.1. Manifestations

Out-of-bound reads and writes Out-of-bound accesses manifest as loads and stores beyond the allocated memory size. Out-of-bound access is one of the most common manifestations in our dataset (21% of all errors). The root causes of out-of-bounds accesses range from logical errors like improper calculation of array and buffer indexes to calculation of allocation size, and validation of user-passed arguments. For example, CVE-2024-45288 demonstrates a buffer overflow where the developer forgets to perform a null termination check while handling a string. In contrast, CVE-2025-0373 triggers a buffer overflow due to the wrong assumption of the struct size on 32-bit vs 64-bits machine mode.

Out-of-bound accesses allow attackers to perform reads and writes to system memory, which historically was a powerful attack primitive leading to information disclosure,

```c
int exec_copyout_strings(struct image_params *imgp, uintcap_t *
     stack_base) {
  ...
  if (imgp->execpath != NULL && imgp->auxargs != NULL) {
    execpath_len = strlen(imgp->execpath) + 1;
    destp -= execpath_len;
    destp = rounddown2(destp, sizeof(void * __capability));
    imgp->execpathp = (void * __capability)
    cheri_setboundsexact(destp, execpath_len);
    error = copyout(imgp->execpath, imgp->execpathp,
         execpath_len);
    if (error != 0)
      return (error);
  }
}
```

Listing 1: An example of capability derivation in CheriBSD

privilege escalation, or kernel panic caused by an unexpected exception or subsequent denial-of-service.

Cheri ▶ **(Yes)** In most cases, CHERI can mitigate out-of-bound accesses since capability protection is enforced at the architectural level – every memory access is a subject to the capability bounds check.

Cheri ▶ **(No)** While CHERI blocks straightforward exploitation of out-of-bounds accesses, a small exploitation window remains possible. In some cases, an attacker can leverage a logical error in the computation of capability bounds, e.g., by controlling the size argument passed to capability creating primitives. Such hypothetical logical errors may occur in places similar to `exec_copyout_strings()` where the bounds of the executable path capability are set from the length of the specified string (Listing 1).

The CheriBSD allocator implements sufficient alignment and padding to ensure that heap objects' size and bounds can be represented by CHERI capabilities. In some cases, however, the kernel code allows creation of capabilities with excessive bounds. For example, the `PHYS_TO_DMAP()` macro returns a capability to the dmap region (which provides access to all physical memory) given a physical address. An even more restrictive variant, `PHYS_TO_DMAP_PAGE()`, gives access to the entire page. Logical flaws in the code may allow attacker to reach these macros and later exploit a larger capability with an out-of-bound access.

Finally, in some cases, the kernel code manually relaxes capability bounds to accommodate container and member idioms (e.g., `__containerof()` in FreeBSD). In such cases, manual annotations are added to suppress tightening and hence can provide an opportunity for out-of-bounds accesses.

Use-after-free and double free Use-after-free (UAF) and double free happen when the code dereferences or frees a previously freed pointer. Common causes for use-after-frees and double free include time-of-check to time-of-use (TOCTOU), race conditions, missing return value check, and protocol errors (e.g., implementation of tear down and de-registration protocol (CVE-2022-34495), forgetting to set the pointer to NULL after deallocation (CVE-2021-29266), etc).

Cheri ▶ **(Yes)** CHERI supports capability revocation [47, 45, 17, 2]. At the moment, the revocation scheme is not implemented in the CheriBSD kernel. The main conceptual limitation is that revocation scheme requires a system-wide freeze of all threads and can introduce prohibitive overhead due to the high frequency of object allocation in the kernel. If we assume that revocation scheme can be implemented in the kernel, most use-after-free vulnerabilities will be blocked by CHERI. Note, CHERI does not support revocation for capabilities to objects allocated on the stack and in the data sections.

Cheri ► **(No)** CHERI provides no protection against double-free. To guard against double free, a memory allocator has to implement a hardening scheme [40, 26]

Uninitialized memory access Uninitialized memory accesses occur due to failure to initialize memory properly, i.e., zero-out previously freed memory upon allocation from the free pool, partial or incomplete initialization due to alignment and padding gaps between the fields of the data structure, or logical failure to follow the initialization protocol before using the data structure. Uninitialized accesses result in attacks ranging from information disclosure to powerful write primitives and control flow violations (due to re-use of previously allocated pointers). Information disclosure is possible due to both access to stale memory values (e.g., CVE-2024-26638) as well as more nuanced padding errors when compiler fails to initialize the space between naturally aligned fields (CVE-2016-4482). To maximize performance, both Linux and FreeBSD memory allocators do not zero-out memory on the stack and heap upon allocation, which, in some cases, allows attackers to access these stale values. Access to stale function pointers can result in control flow violations. Finally, access to stale capabilities may provide attacker with powerful read and write primitives.

Cheri ► **(Yes)** If revocation scheme is implemented, CHERI invalidates stale pointers to data structures allocated on the heap, hence making it impossible to re-use these pointers if they are leaked to the attacker due to failure to zero-out the data structure. Further, CHERI prevents null pointer dereferences, as zeroed-out null pointer is not a valid capability (does not have a valid tag).

Cheri ► **(No)** CHERI fails to block information disclosure attacks due to the lack of initialization of previously used memory or padding. Since revocation schemes are limited to pointers to the data structures allocated on the heap, CHERI fails to block re-use of pointers if the stale pointer is pointing to the data structure allocated on the stack or in the data section.

Resource leak Memory and other resource leaks happen when unsafe code violates resource deallocation protocol leaving resource in an allocated but unused state. A typical kernel like Linux supports a combination of manual and reference counted lifetimes. Resource leaks are violations of lifetime protocols, i.e., developer forgets to decrement a reference counter or manually free an object. Resource leaks allow an attacker to exhaust system memory and other resources and cause a denial of service.

Cheri ► **(No)** CHERI does not prevent resource leaks as it does not automatically detect or reclaim memory and other resources that are no longer used.

Invalid pointer dereference Invalid pointer dereference can lead to an undefined behavior [3]. In the kernel, invalid

```
static int brcm_nvram_parse(struct brcm_nvram *priv)
  len = le32_to_cpu(header.len);
  data = kzalloc(len, GFP_KERNEL);
+ if (!data)
+  return -ENOMEM;
  memcpy_fromio(data, priv->base, len);
  data[len - 1] = '\0';
```

Listing 2: Null pointer derefence as a result of a missing return value check (CVE-2023-3359)

pointer dereferences often result in page faults due to accessing an unmapped memory access, such as 0x0, the most common invalid pointer instance, but can also provide access to sensitive kernel data structures. For example, on some architectures the 0x0 address is mapped as an exception vector table, thus dereferencing the null pointer might result in a vector handler override. Listing 2 shows a classic instance of a null pointer dereference due to a missing return value check.

Cheri ► **(Yes)** CHERI blocks invalid pointer dereferences, as invalid pointers are not valid capabilities and trigger a fault when dereferenced.

Exception or panic A range of high-level logical errors manifest as low-level exceptions or explicit kernel panics and kernel assertion macros like `BUG_ON()`. Both exceptions and panics are manifestations of an inconsistent kernel state ,from which recovery is impossible (i.e., there is no good way to recover from reaching a division by zero). While the kernel can survive some exceptions, in most cases, some functionality is unrecoverable, which results in a denial of service.

Cheri ► **(No)** CHERI does not protect against exceptions due to logical errors in the code.

Failure to release CPU Some vulnerabilities cause the kernel to monopolize the CPU, leading to system hangs. For example, CVE-2018-6918 results in an infinite loop in the FreeBSD's IPsec option parsing code, which has a logical flaw and causes the length to be zero in some cases. Since the length value is used as the loop iterator, the loop never progresses, and the thread fails to yield the CPU. A similar issue occurs in Linux, CVE-2023-5158. In CVE-2023-2269, failure to release the CPU stems from a recursive locking scheme in the device mapper subsystem: a write lock is acquired and reentered as a read lock through an indirect call path, leading to a deadlock that indefinitely stalls the thread. In CVE-2023-31084, the code invokes interruptible sleep while holding a semaphore, which may delay lock release, temporarily monopolizing CPU resources.

Cheri ► **(No)** This class of vulnerabilities arises from semantic errors in the design or implementation of the code. Since failure to release the CPU does not violate spatial or temporal memory safety, CHERI does not mitigate this class of vulnerabilities.

Control flow violation Control flow violation is a powerful attack primitive that in many cases can lead to privilege escalation. Control flow violation occurs when critical execution data gets overridden such as return addresses or function pointers, but also can be a result of a data only attack that breaks the expected execution protocol, resulting in early exit, incorrect resource cleanup, deadlocks etc.

```
1  static int
2  epair_clone_create(struct if_clone *ifc, char *name, size_t len,
         caddr_t params)
3  {
4  ...
5  - if (params) {
6  -   scb = (struct epair_softc *)params;
7  -   ifp = scb->ifp;
8  -   /* Copy epairNa etheraddr and change the last byte. */
9  -   memcpy(eaddr, scb->oifp->if_hw_addr, ETHER_ADDR_LEN);
10 -   eaddr[5] = 0x0b;
11 -   ether_ifattach(ifp, eaddr);
12 -   /* Correctly set the name for the cloner list. */
13 -   strlcpy(name, ifp->if_xname, len);
14 -   return (0);
15 - }
16 ...
17 }
```

Listing 3: Control flow violation(CVE-2020-7452)

Listing 3 shows an example of function pointer corruption in the FreeBSD `epair` (virtual ethernet interface pair) device driver. `epair_clone_create` is a driver implementation of the network clone interface that is responsible for dynamically creating virtual network interfaces. The implementation casts user provided argument `param` as a `struct epair_softc*` (line 6). The driver then fully trusts user provided `struct epair_softc`, including its member `ifp` (line 7, 11). Since `ifp` is a pointer to `struct ifnet` that contains function pointers used by kernel as network interface routine, an attacker can trick the kernel into executing arbitrary payload by fabricating `struct ifnet`.

Cheri ▶ **(Yes)** CHERI blocks majority of control-flow vulnerabilities that target function and return pointers. CHERI achieves code-pointer integrity through sealed capabilities for both function pointers and return addresses. The only way to affect the control flow is to replace one sealed capability with another valid sealed capability.

Cheri ▶ **(No)** CHERI does not enforce the backward edge control flow. While CHERI seals return capabilities on the stack, an attacker can still alter the control flow if the return capability is overwritten with a valid sealed capability. While trivial stack smashing attacks are impossible, an attacker can replace one return capability with another via one of the capability-preserving memory copy functions. CHERI also cannot enforce control flow integrity when the user input is used directly as function pointers like mentioned in example listing CVE-2020-7452.

High-level specification violation High-level specification violations can be more generally defined as violation of expected behavior (or specification if one exists), e.g., CVE-2022-47522. Some specification violations result in denial of service by exhausting system resource when kernel fails to set limits for resource allocation such as size parameter to `malloc` (CVE-2024-39281), or number of nodes allocated in a linked list(CVE-2019-5599). Specification violations are hard to detect, as they do not manifest as hardware fault conditions.

Cheri ▶ **(No)** CHERI cannot affect outcomes of errors that manifest as deviations from expected system behavior due to their high-level nature.

Access control violation Access control violations are a subset of high-level specification violations, in which access control policy in the kernel is violated. Access control

```
static bool index_hdr_check(const struct INDEX_HDR *hdr, u32
      bytes)
...
  if (!IS_ALIGNED(off, 8) || tot > bytes || end > tot ||
- off + sizeof(struct NTFS_DE) > end) {
+ size_add(off, sizeof(struct NTFS_DE)) > end) {
      /* incorrect index buffer. */
      return false;
  }
...
```

Listing 4: Integer overflow (CVE-2025-22081)

violations are most often results of programming errors due to the lack of complete understanding of all possible invocation contexts for a given function and complexity of access control policies. For example, CVE-2020-25584 in the FreeBSD kernel is lacking a permission check, and therefore, a sudo process inside a jail with `allow.mount` permission can change root directory outside of the jail. In CVE-2021-27363, the `CAP_SYS_ADMIN` check is missing for the iSCSI driver, which allows a normal user to end arbitrary iSCSI sessions.

Cheri ▶ **(No)** Like other high-level specification errors, CHERI does not block access control violations due to their high-level nature.

4.2. Vulnerability Causes

Vulnerability causes, i.e., high-level reasons that lead to low-level manifestations, provide us with an alternative view of what leads to a specific exploitable vulnerability. In contrast to manifestations, which provide an insight for whether exploitation of a CVE can be blocked, we use vulnerability causes, to gain a deeper understanding of whether a specific class of software flaws can be avoided through the use of programming language abstractions.

4.2.1. Programming Language Limitations

Limitations of low-level languages like C lead to a broad set of flaws ranging from integer overflows and type errors to missing object initialization and lifetime violations.

Integer overflow, underflow Integer overflow or underflow happens when an arithmetic operation results in a number that cannot be represented with a given number of bits (a limitation of a programming language that makes a practical choice to sacrifice infinite numbers for efficiency). Overflows can lead to a diverse group of manifestations such as out-of-bound access and violations of expected system behavior.

Listing 4 shows an integer overflow vulnerability in the NTFS3 filesystem driver in the Linux kernel. On 32-bit systems, an addition involving an offset and the size of a struct can wrap around, leading to incorrect bounds checks. The function `index_hdr_check()` attempts to validate whether an offset plus the size of a directory entry (`struct NTFS_DE`) stays within bounds. However, the expression `off + sizeof(struct NTFS_DE)` may overflow on 32-bit architectures, bypassing the intended bounds check. The fix replaces the raw addition with `size_add()`, which performs checked addition and avoids overflow by clipping on `SIZE_MAX`.

Polymorphism Due to the lack of support in C, kernel developers resolve to ad hoc implementations of polymorphic

```
1  static int
2  ext2_vptofh(struct vop_vptofh_args *ap)
3  {
4    struct inode *ip;
5    struct ufid *ufhp;
6  + _Static_assert(sizeof(struct ufid) <= sizeof(struct fid),
7  +  "struct ufid cannot be larger than struct fid");
8    ip = VTOI(ap->a_vp);
9    ufhp = (struct ufid *)ap->a_fhp;
10   ufhp->ufid_len = sizeof(struct ufid);
11   ufhp->ufid_ino = ip->i_number;
12   ufhp->ufid_gen = ip->i_gen;
13   return (0);
14 }
```

Listing 5: Polymorphism (CVE-2025-0373)

```
    host->reg_va09 = regulator_get(hba->dev, "va09");
- if (!host->reg_va09)
+ if (IS_ERR(host->reg_va09))
      dev_info(hba->dev, "failed to get va09");
    else
      host->caps |= UFS_MTK_CAP_VA09_PWR_CTRL;
```

Listing 6: Confused pointer error (CVE-2023-23001)

behavior, e.g., the use of `void *` to represent pointers to arbitrary types, allocating a memory area to hold objects of different types, encoding error conditions inside pointer types instead of using an option type, etc. Unsafe polymorphism relies on unsafe type casts and frequently leads to errors.

For example, Listing 5 shows a buffer overflow caused by an incorrect type cast. `ext2_vptofh()` is an ext2fs implementation of a vnode operation, `vop_vptofh`, which is used by the NFS server to create an opaque file handle. `ext2_vptofh()` casts the user argument `ap` (which user treats as type `struct fid`) as a pointer to `struct ufid` (line 9). Then it writes to `ap->a_fhp` as if it is a `struct ufid` (line 5). However, the size of `struct fid` is smaller than the size of `struct ufid` on 64 bit architectures. This mismatch causes an out-of-bound write (line 12).

Another example of low-level polymorphism common in the Linux kernel is encoding of errors as "negative" pointers. In order to correctly resolve the type (error or a valid pointer) the caller has to use the `IS_ERR` macro. Developers often forget this unsafe idiom (e.g., CVE-2023-23006), which fails to correctly check an error. For example, Listing 6 shows an incorrect return value check in Linux. The original vulnerable code can lead to setting the `UFS_MTK_CAP_VA09_PWR_CTRL` bit under wrong conditions and then dereference of the invalid `reg_va09` pointer. In this example, CHERI blocks dereference of the invalid pointer, however will still allow setting the bit which might be exploitable 2.

Container of Another popular kernel idiom, the `container_of` macro, which provides access to the parent data structure from a given field of the member structure, compensates for the lack of language support for safe object composition. This creates an assumption about the container type which may not be correct in all execution contexts. In CVE-2023-1076, the `tap` driver calls `sock_init_data()` to initialize the low-level `struct sock` type given a `struct socket`. However, the function assumes that the `struct socket` passed is part of a `struct socket_alloc`. This type confusion manifests as a high level access control violation, as the `sk_uid` field

```
static void thrustmaster_interrupts(struct hid_device *hdev) {
...
  /* Are the expected endpoints present? */
- u8 ep_addr[1] = {b_ep};
+ u8 ep_addr[2] = {b_ep, 0};

  if (!usb_check_int_endpoints(usbif, ep_addr)) {
    hid_err(hdev, "Unexpected non-int endpoint\n");
  }
```

Listing 7: Stack out-of-bounds read caused by sentinel array (CVE-2025-21794)

can be assigned to a misconfigured value. Thus, CHERI cannot mitigate this specific bug. In a more generic case, CHERI would still fail to mitigate this particular cause, as the capability pointing to `struct socket` would be configured to allow the access to the parent object, hence allowing attacker to read kernel heap memory.

Sentinel arrays (null byte termination) Due to lacking support for a generic vector type (array with bounds), kernel represents arrays of variable size as NULL-terminated arrays of variable length (using NULL as a sentinel). Sentinel arrays are most commonly used to represent strings, but can be used to represent array of other types, such as USB endpoint numbers in Linux (`ep_addr` in Listing 7). A common error is a failure to properly handle the size of the array (i.e., check for NULL), which, in most cases, results in an out-of-bound access. In our example, `usb_check_int_endpoints()` is called to ensure that the USB endpoint is present and is of the correct type. The function expects a null-terminated array of USB endpoints as an argument, but the stack array `ep_addr` is not, leading to out-of-bounds read and eventually to the kernel panic.

Improper memory initialization Improper memory initialization can happen in a variety of cases, e.g., programmer fails to pass the zero flag to `malloc()` (CVE-2024-8178), forgets to use the struct initializer to initialize a struct (CVE-2024-26638), or fails to clear registers after the syscall return (CVE-2019-5595). Most frequent manifestation of improper memory initialization is uninitialized memory access. Occasionally, it can also manifest as an invalid pointer dereference.

Object lifetimes C allows developers to access any heap memory as long as they hold a pointer to it. Naturally, violations of object lifetimes are frequent in the complex kernel code (16% of our dataset). Violation of object lifetimes results in use-after-free, double free, and memory and resource leaks.

4.2.2. Protocol violations

Protocol violations are a set of programming errors that break a protocol of expected interaction between kernel subsystems. The errors can stem from missing, racing, and reordered invocations of interface functions that provide registration and tear down functionality, processing of requests, control lifetimes of objects shared between subsystems, etc.

Missing or re-ordered protocol steps Kernel subsystems interact according to a set of poorly-documented protocols, i.e., sequences of steps that allow subsystems to register with the kernel, process requests, react to power-down events, request and release kernel resources, etc. Violation of the protocol can leave the system in an inconsistent, often exploitable state. Protocol violation can lead to a range of manifestations such as out-of-bound access, use after free, null pointer dereference, etc. (Table 4). For example, CVE-2023-3090 fails to clear the socket control buffer `skb->cb` that is meant to be temporarily used by each network layer. The uncleared `skb->cb` buffer later gets reused as the `skb->cb->opt` pointer to store data.

Sleeping in atomic contexts An important kernel invariant is that certain execution contexts prohibit sleep and wait functions that yield the CPU (e.g., while holding a spinlock, executing an interrupt handler, etc.). Violating this invariant may block the CPU for a long time or result in a deadlock or a system hang. On a multicore system, sleeping in an atomic context does not always result in a system hang or crash, thus making it hard to statically detect such conditions. A common mistake is to invoke one of the kernel functions that can internally yield the CPU if resource is unavailable. For example, invocation of the kernel memory allocator from inside an atomic context requires passing the `GFP_ATOMIC` flag to prevent the allocator from yielding the CPU if memory is not available.

Failure to yield Depending on the configuration, the Linux kernel may or may not support preemption when a thread executes inside the kernel. Long-running kernel activities are expected to yield the CPU in a cooperative manner via explicit invocation of the scheduler. A failure to yield results in a denial of service, freezes of individual cores, and in the worst case, complete system freeze. The major causes of failure to yield vulnerabilities include infinite loops (CVE-2019-3900) and semantic errors failing to invoke a yielding function. For example, in CVE-2015-5364, the driver fails to invoke the `cond_resched()` function.

4.2.3. Races

Data races are a specific class of protocol violations that break atomicity of accesses in the face of concurrent contexts of execution: parallel threads of execution, asynchronous interrupts, and even asynchronous hardware updates. Execution environment of the kernel is inherently concurrent. As a result, data race errors are frequent (15% of CVEs in our data set, Table 4). Data races manifest as a range of subtle bugs such as use-after-free, failure to release CPU, null-pointer dereferences, deadlocks, logical inconsistencies, etc.

Improper use of synchronization primitives A kernel is inherently concurrent. The execution contexts of the kernel include preemptable kernel threads executing in parallel on multiple CPUs, asynchronous interrupt handlers, and non-maskable interrupts. Moreover, the hardware that has asynchronous access to memory via the DMA and MMIO interfaces can update the kernel state shared with hardware devices concurrently. In a modern kernel nearly any piece of code has to use some type of a synchronization mechanism such as read-copy update (RCU) primitives, spinlocks, wait queues, condition variables, semaphores, etc. Each primitive adheres to a well-defined usage discipline and provides synchronization under a specific set of assumptions. Failure to implement proper synchronization protocol results in a variety of errors.

One of most heavily used synchronization primitives in the kernel are spinlocks. Improper use of locks can manifest as failure to release the CPU, use after free, etc. A deadlock can happen when the programmer acquires the same lock twice (CVE-2025-22098), acquires locks in different order (CVE-2025-21807), or fails to release the lock on an error path (CVE-2025-21672). Beyond locks, other synchronization primitives can also lead to deadlocks, e.g., CVE-2023-31084, which performs a blocking operation during `wait_event_interruptible()`.

Use after free can occur when locks are missing. In CVE-2020-7457 the developer fails to acquire a write lock upon entering a method that can access a kernel object that can be freed by another thread. Another interesting example is CVE-2024-23848 where the lifetime of the lock itself is mismanaged. A lock variable is a member of a data structure that is a subject to a race condition and deallocation by another thread.

Time-of-check to time-of-use Time-of-check to time-of-use (TOCTOU) is a subset of race conditions that allow attacker to leverage weaknesses in synchronization between checking or validating a set of correctness invariants about a data structure and the use of the data structure if it's changed concurrently by an attacker. A concurrent change of the data structure can result in the semantic inconsistency and lead to a wide variety of low-level manifestations: out-of-bound accesses, use-after free, and so forth (Table 4).

For example, in CVE-2022-23085, a buggy `nmreq_copyin()` first computes the size of a user buffer by traversing a linked list. The code can then use the computed size to copy the user data into the kernel heap. Just right after the calculation of the buffer size and before copy operation, an attacker can attempt to swap the actual user buffer. If the buffer being swapped is smaller, the kernel performs an out-of-bound read.

4.2.4. Semantic violations

Logic error Logic errors are failures to implement the high-level behavior of the system, i.e., violating specific algorithms or breaking a high-level property of a data structure like a proper tree balance. For example, in CVE-2024-26648, a null pointer dereference occurs due to a logic error in pointer usage: the pointer `link` is dereferenced before being checked for NULL. While this might typically be considered an input validation issue, in this case, the null check was present but mistakenly placed after the dereference. Another example is CVE-2019-5608 where the code fails to handle the case of fragmented buffers, i.e., when the buffer is spread across multiple buffers. This can cause an out-of-bounds access since the size of the buffer will be larger than each fragmented piece. In this example, the implementation deviates from the intended specification of the network protocol, thus a logic error.

Improper input validation A large set of vulnerabilities can be attributed to incorrect validation of the untrusted input

```
static void
cdceem_handle_cmd(struct usb_xfer *xfer, uint16_t hdr, int *offp
      ) {
...
+ if (pktlen > m->m_len) {
+   CDCEEM_WARN(sc, "buffer too small %d vs %d bytes",
+   pktlen, m->m_len);
+   if_inc_counter(ifp, IFCOUNTER_IQDROPS, 1);
+   m_freem(m);
+   return;
+ }
    usbd_copy_out(pc, off, mtod(m, uint8_t *), pktlen);
```

Listing 8: Improper input validation (CVE-2020-7459)

coming from the user space programs, network packets, and even misconfigured or malicious devices. Such vulnerabilities can result in a number of low-level manifestations ranging from simple out-of-bound accesses to protocol and semantic errors that allow bypassing security policies (Table 4). CVE-2020-7459 in FreeBSD allows malicious USB ethernet devices to cause out-of-bound writes beyond the allocated packet buffers (Listing 8).

Specification error Specification errors occur when a driver or subsystem faithfully implements the specification (intended behavior), but the specification itself is flawed. This is distinct from protocol violations and logical errors that violate specification. Specification errors stem from the flawed understanding of what the system is supposed to do, not its implementation. Specification errors can manifest in a variety of ways and lead to different types of vulnerabilities (Table 4). Whether CHERI can mitigate such issues depends on if the flawed behavior violates memory safety. For example, in CVE-2024-26681, when an acquisition of the mutex fails, the driver's specification was to retry continuously without delay. Endless re-tries resulted in hogging of the CPU. The specification was later changed to introduce a delay between retries.

Another example is CVE-2021-26931 where an error path was defined to call `BUG_ON()`, which triggered a kernel exception. Although this matched the intended behavior, it created a potential denial-of-service vulnerability. The specification was updated to handle the error more gracefully without crashing the system.

Security and permissions The kernel is responsible for enforcing a range of system-wide security policies ranging from classical UNIX-like file permissions to mandatory access control frameworks (MAC) like SELinux. In Linux, capabilities are used to grant fine-grained permissions to perform operations traditionally reserved for processes running as root, such as mounting filesystems. Processes running as root may also drop capabilities or be launched without them. In CVE-2020-25284 (Listing 9), a missing check for the `CAP_SYS_ADMIN` capability in the Ceph block device driver allowed root processes that have dropped all capabilities to add or remove arbitrary Reliable Autonomic Distributed Object Store (RADOS) block devices. Missing or improper security checks can manifest in a range of symptoms, including access control violations and violations of high-level specifications (Table 4).

Loop termination In many cases loops in the kernel code use user and sometimes device-provided values as the termination criteria. An absence of range checks can potentially

```
static ssize_t do_rbd_remove(struct bus_type *bus, ...) {
...

+ if (!capable(CAP_SYS_ADMIN))
+   return -EPERM;
+
```

Listing 9: Missing capability check in a RADOS block device (CVE-2020-25284)

lead to infinite loops and thus denial-of-service and potentially other manifestations.

One example is a faulty goto logic that forms an infinite loop (CVE-2023-4010). Another involves iterators that rely on resources that are cleaned up during device unregistration (CVE-2025-21681), which breaks the iterator and prevents the loop from terminating. Loop termination failures stem from semantic errors in how loop conditions and iterator logic are defined. As such, CHERI cannot mitigate them unless they also involve spatial or temporal memory safety violations.

Missing return value check A missing return value check refers to a semantic error where the return value of a function, particularly one that can indicate a failure or exception condition, is not checked before subsequent operations proceed. Many functions, especially those involved in memory allocation, file handling, field initialization, or hardware interaction, return a status code or pointer that must be validated to ensure safe progress of the code. Omitting such checks can cause the program to operate under incorrect assumptions, potentially leading to crashes, security vulnerabilities, or incorrect system behavior. This flavor of errors is common in C where error handling is manual hence allowing silent propagation of unchecked values and conditions through the program. Failure to check the return value can manifest in a variety of ways. For example, if the check is missing from a memory allocation function, it may result in a null pointer dereference, since allocators commonly return a null pointer on failure.

5. Analysis

We utilize the taxonomy introduced in Section 4 to answer the questions about the impact of the CHERI capabilities on the security of the kernel and its effectiveness compared to Rust. Specifically, we analyze the dataset of 439 CVEs from the Linux and FreeBSD kernels and assign a pair of cause and manifestation to each vulnerability. We consider two possibilities: the capability revocation scheme is either implemented or not. We then analyze commit history of the CheriBSD kernel to understand the development effort required to extend a full-featured kernel with capability protection.

5.1. CHERI Impact on the Kernel

Manifestation-centric view of the dataset confirms that CHERI blocks vulnerabilities that manifest as safety problems (Table 1, also see Table 4 for a detailed mapping between manifestations and causes). Specifically, if we assume that revocation scheme is implemented in the kernel, CHERI can block 264 vulnerabilities (60% of the dataset). If revocation is not supported (as it is now in the CheriBSD

TABLE 1: Manifestation-centric view of CVEs in Linux and FreeBSD kernels

	Linux		FreeBSD		Total	
Manifestation	Yes	No	Yes	No	Yes	No
OOB access	67	0	27	0	94	0
Use after free	96 | 0	0 | 96	17 | 0	0 | 17	113 | 0	0 | 113
Double free	0	14	0	0	0	14
Uninitialized memory access	0	15	0	16	0	31
Resource leak	0	24	0	0	0	24
Invalid pointer dereference	60	0	0	0	60	0
Explicit exception/panic	0	11	0	3	0	14
Control flow violation	0	0	0	1	0	1
Failure to release CPU	0	23	0	1	0	24
High level spec violation	0	18	0	10	0	28
Access control violation	0	10	0	26	0	36
Total	223 | 127	115 | 211	44 | 27	57 | 74	267 | 154	172 | 285
Effectiveness	66% | 38%	34% | 62%	44% | 27%	56% | 73%	61% | 35%	39% | 65%

TABLE 2: Cause-centric view of CVEs in Linux and FreeBSD kernels

	Linux		FreeBSD		Total	
Cause	Yes	No	Yes	No	Yes	No
Language	56 | 18	45 | 83	15 | 10	15 | 20	71 | 28	60 | 103
Integer overflow	6	2	5	1	11	3
Polymorphism	8	0	3	0	11	0
Container of	0	1	0	0	0	1
Sentinel arrays	2	0	2	0	4	0
Improp. mem. init.	2	12	0	14	2	26
Lifetime violation	38 | 0	30 | 68	5 | 0	0 | 5	43 | 0	30 | 73
Protocol	4 | 3	6 | 7	0	1	4 | 3	7 | 8
Missing prot. steps	4 | 3	5 | 6	0	1	4 | 3	6 | 7
Sleeping in atomic	0	1	0	0	0	1
Race condition	48 | 6	12 | 54	7 | 2	1 | 6	55 | 8	13 | 60
Improp. use Of sync.	26 | 4	9 | 31	2 | 0	0 | 2	28 | 4	9 | 33
TOCTOU	22 | 2	3 | 23	5 | 2	1 | 4	27 | 4	4 | 27
Semantic	115 | 100	52 | 67	22 | 15	40 | 47	137 | 115	92 | 114
Logic error	26 | 20	15 | 21	9 | 4	13 | 18	35 | 24	28 | 39
Improper input val.	56	8	13 | 11	6 | 8	69 | 67	14 | 16
Spec error	10 | 3	16 | 23	0	7	10 | 3	23 | 30
Security & perm.	0	9	0	11	0	20
Loop termination	0	3	0	1	0	4
Missing ret. val.	23 | 21	1 | 3	0	2	23 | 21	3 | 5
Total	223 | 127	115 | 211	44 | 27	57 | 74	267 | 154	172 | 285
Effectiveness	66% | 38%	34% | 62%	44% | 27%	56% | 73%	61% | 35%	39% | 65%

kernel), the total percentage of vulnerabilities blocked by CHERI drops to 34%, making it way less effective. CHERI blocks all invalid pointer dereferences (60 CVEs) unconditionally, and all out-of-bound accesses (94 CVEs) and use-after-free (113 CVEs), if we assume that revocation is implemented in the kernel. Naturally, CHERI fails to block 172 vulnerabilities that do not manifest as safety violations: double-free (14 CVEs), uninitialized memory accesses (31 CVEs), resource leaks (24 CVEs), panics (14 CVEs), failure to release the CPU (24 CVEs), high-level specification violations (28 CVEs), control flow violation (1 CVE) and access control violations (36 CVEs). CHERI fails to block one instance of the control flow violation from our dataset, CVE-2020-7452, which we discussed in Section 4.1. Note, it is important to emphasize that even if CHERI fails to block double-free and uninitialized memory access vulnerabilities it significantly narrows the exploitation window for these CVEs.

To get a better insight into which classes of software flaws benefit from capability protection, we utilize the cause-centric view of the dataset (Table 2).

Language limitations CHERI prevents exploitation of 54% of language-related errors when revocation is implemented (but only 21% without it). Pure capability mode blocks exploitation of most integer overflow errors that in most cases result in out-of-bound access (11 out of 14 language-related causes, Table 4). However, in some cases, the overflown value has complex semantics and flows deep inside the kernel, like the packet's payload length for reassembled ipv6 packet in FreeBSD in CVE-2023-3107, thus potentially resulting in a high-level logical error or other manifestations. We label such cases (3 CVEs in our dataset) as 'No'.

A large fraction of improper memory initialization flaws (25 out of 28 CVEs) result in uninitialized memory access via partially initialized data structures or non-zeroed memory and a subsequent information disclosure. Such vulnerabilities cannot be mitigated by CHERI. In two special cases of improper memory initialization (CVE-2024-26867 and CVE-2022-38096), developers fail to initialize pointer fields, which results in invalid pointer dereferences.

All 11 CVEs that are caused by limitations related to polymorphism are mitigated by CHERI (they result in out-of-bound accesses and invalid pointer dereferences).

Even with revocation enabled, CHERI fails to block 30 lifetime violations that manifest as memory leaks, double free, and failures to release the CPU. CHERI fails to mitigate a bug caused by the "container of" idiom (CVE-2023-1076 discussed in Section 4.2). Four CVEs caused by the use of sentinel arrays lead to out-of-bounds accesses that are blocked by CHERI.

Protocol violations CHERI can mitigate 36% of CVEs caused by protocol violations. Out of 10 CVEs caused by missing protocol steps, 4 CVEs that are mitigated by CHERI result in invalid pointer dereference and use-after free. The other 6 CVEs that cannot be mitigated by CHERI result in high-level specification violations (2 CVEs), double free (2 CVEs), explicit exception (1 CVE), and failure to release the CPU (1 CVE). Our dataset has only one example of a CVE triggering sleep in the atomic context – the CVE results in failure to release the CPU, which CHERI cannot mitigate. Revocation changes the exploitation outcome for only one vulnerability, CVE-2025-21968, where the programmer forgot to follow the expected protocol to cancel the delayed work when destroying a workqueue, leading to a classic use-after-free.

Race conditions CHERI prevents exploitation of 81% of race conditions if revocation is implemented (but only 19% if it's not). Depending on whether revocation is implemented, CHERI does or does not mitigate use-after-free vulnerabilities (47 CVEs). CHERI can mitigate out-of-bound access and invalid pointer dereference through enforcement of valid capabilities. However, it cannot mitigate failure to release the CPU, which commonly arises in race conditions due to deadlocks or hangs. High level spec violations are also not addressed by CHERI, as they typically involve semantic errors that operate within the bounds of valid capabilities.

Semantic vulnerabilities Among semantic vulnerabilities, 60% can be blocked by CHERI (50% if revocation is not implemented). All 4 loop termination cases in our dataset result in failure to release the CPU, which cannot be mitigated by CHERI. Out of 20 security and permissions cases, 18 lead to access control violations and 2 to high level spec violations, all of which cannot be mitigated by CHERI. Our dataset has 26 missing return value check cases, out of which 20 result in invalid pointer dereference, 2 in use after free,

TABLE 3: Cheri vs Rust (cause-centric view)

Cause	CHERI		Rust		Vulns. blocked(%)	
	Yes	No	Yes	No	Cheri	Rust
Language	43 | 13	30 | 60	69	4	58% | 18%	94%
Integer overflow	3	1	4	0	75%	100%
Polymorphism	8	0	8	0	100%	100%
Container of	0	1	1	0	0%	100%
Sentinel arrays	1	0	1	0	100%	100%
Improp. mem. init.	1	7	5	3	12%	62%
Lifetime violation	30 | 0	21 | 51	50	1	59% | 0%	98%
Protocol	3	4	3	4	43%	43%
Missing prot. steps	3	3	3	3	50%	50%
Sleeping in atomic	0	1	0	1	0%	0%
Race condition	36 | 3	7 | 40	40	3	84% | 7%	93%
Improp. use Of sync.	19 | 2	4 | 21	21	2	83% | 9%	91%
TOCTOU	17 | 1	3 | 19	19	1	85% | 5%	95%
Semantic	84 | 70	27 | 41	87	24	76% | 63%	78%
Logic error	16 | 10	9 | 15	17	8	64% | 40%	68%
Improper input val.	39	5	39	5	89%	89%
Spec error	9 | 3	4 | 10	10	3	69% | 23%	77%
Security & perm.	0	7	0	7	0%	0%
Loop termination	0	2	0	2	0%	0%
Missing ret. val.	20 | 18	0 | 2	20	0	100% | 90%	100%
Total	166 | 89	68 | 145	198	36	70% | 38%	84%

and 1 each in an out-of-bound access, uninitialized memory access, high level spec violation, and explicit exception or panic. Three CVEs cannot be mitigated by CHERI, as they result in from uninitialized memory access, a high-level specification violation, and an explicit exception or panic.

5.2. CHERI vs Rust

To compare the effectiveness of protection offered by CHERI against the language-based safety provided by Rust, we analyze a subset of CVEs from the recent study of Rust and its impact on the Linux kernel [25]. If we assume that revocation is implemented in the kernel, we observe that CHERI is able to mitigate a lower but significant portion of vulnerabilities (166 or 70% compared to 198 or 84% for Rust) despite the lack of language-level safety guarantees (Table 3, also see Table 5 for the manifestation-centric view). Without revocation, CHERI can block only 89 CVEs or 38%.

Integer overflow Rust compiler supports a compiler flag that inserts runtime checks for integer overflows, hence allowing Rust to mitigate integer overflow vulnerabilities (note, the flag is often disabled in release builds). CHERI, however, can only mitigate an integer overflow if the CVE leads to an out-of-bounds access (e.g., an overflown value is used as a memory address). If the overflown value is used as data or participates in the control logic of the program, the overflow remains undetected.

Polymorphism Rust provides native support for tagged unions via the `enum` type hence eliminating vulnerabilities related to many flavors of polymorphism typical in the kernel. The language manages the type tag and enforces type safety for each variant. Rust enforces explicit *pattern matching* to access the tagged values hence preventing unsafe accesses. With the `enum` types, Rust provides safety for NULL-able types like (`Option<T>`) and return values, e.g., `Result<T, E>`.

Improper memory initialization Rust requires all fields to be initialized when creating an instance of a data structure. This requirement mitigates a large fraction of improper memory initialization errors. CHERI can mitigate improper memory initialization flaws if they result in invalid pointer accesses, like in CVE-2024-26867, in which a field `spriv->io` was left uninitialized, and later accessed. Rust, however, may fail to mitigate improper memory initialization bugs on the boundary with unsafe APIs, e.g., the developer may fail to use correct memory allocation flags to zero-initialize allocated memory.

Race conditions Rust protects developers from improper use of synchronization primitives through its ownership discipline. Specifically, Rust forces developers to encapsulate objects in the synchronization primitives like `Mutex<T>`. The programmers have to acquire the lock before accessing or freeing the object. For example, in CVE-2024-23848, one thread frees the object without checking if any other thread is holding the lock. This error will be caught by the Rust compiler as it violates the Rust lifetime rules. Similarly, Rust prevents time-of-check-time-of-use errors that are also introduced by improper use of synchronization primitives. CHERI, however, provides no support for mitigating race conditions unless they manifest as safety violations.

Improper input validation Rust does not directly mitigate improper input validation errors, but can reduce their impact. For example, CVE-2023-31085 and CVE-2021-20292 are two CVEs caused by improperly checked inputs that lead to integer overflow and double free respectively. These would be caught by Rust with Rust's integer overflow/underflow checks and lifetime rules. Such vulnerabilities remain unaffected by CHERI.

5.3. Development Effort

Finally, to understand the development effort required for protecting a typical full-featured kernel with CHERI capabilities, we analyze the commit history of the CheriBSD operating system [36]. The process of introducing CHERI capabilities to the FreeBSD kernel largely followed three phases: 1) extending the kernel with support for pure-capability user-level processes, 2) enabling pure-capability mode of execution in the kernel, and 3) incrementally porting device drivers and kernel extensions.

Pure-capability userspace The first commit that added kernel support for CHERI userspace to the MIPS port of CheriBSD was made in late 2013 [13]. Subsequent commits added additional userspace infrastructure for CHERI: a library called `libcheri` which implements CHERI-based sandboxing [14], adjustments to stack alignment in the libc startup code [15], and drivers for the graphics compositor on the CHERI MIPS hardware [16]. To support pure-capability userspace, the kernel was compiled in hybrid mode, in which C pointers remained plain pointers by default unless annotated with the `__capability` attribute. Efforts to annotate the necessary pointers as capabilities were initially concentrated around the kernel-userspace boundary as the kernel internally had limited interactions with capabilities.

Pure-capability kernel The work on supporting pure-capability mode in the kernel appears to have started in late

2020 and continued through mid-2021. As of March 2025, there are 267 commits labeled `purecap-kernel:`, plus some additional commits related to pure capability mode of execution. In total, commits related to enabling pure-capability mode of execution in the kernel have resulted in 16,850 lines of changes (14,016 additions, 2,834 deletions), of which 16,511 lines modified the kernel code. Additionally, 10,283 lines of changes were involved in implementing userspace capability revocation in CheriBSD, of which 8,417 were in the kernel.

Main changes include implementing infrastructure for representing virtual addresses as capabilities throughout the kernel [30], support for relocation of capability enabled executables [27], changing kernel allocator to return capabilities [29], and capability support in exception handlers [28].

Security and use of root capabilities The CheriBSD kernel maintains several powerful "root" capabilities from which other smaller capabilities are derived. Root capabilities are initialized during early boot. They allow access to the entirety of the kernel and userspace portions of address spaces and enable the creation of sealed entry capabilities for enforcing control flow integrity (Section 2.1). Therefore, it is important that kernel code responsible for deriving fine-grained capabilities from the root capabilities does so correctly. The kernel carefully limits the use of powerful root capabilities, by setting their bounds to specific kernel objects with functions like `cheri_setboundsexact()`.

Some kernel subsystems still have access to powerful capabilities. For example, the FreeBSD kernel has access to all physical memory through a region of virtual memory called `dmap` ("direct map"). The `PHYS_TO_DMAP` macro is used to convert a physical address to its corresponding location in the `dmap` region. Drivers typically then cast the result to a typed pointer to a C `struct` (or an array of them) that correspond to hardware register layouts. Since this macro does not allow drivers to specify upfront the length of the hardware registers they need to access, this is problematic with regards to spatial safety: In CheriBSD, the macro returns a capability with bounds covering the entire dmap region. To mitigate this issue, CheriBSD introduces the `PHYS_TO_DMAP_PAGE` macro which constrains the returned capability to only cover a single page. There are 401 invocations of `PHYS_TO_DMAP` in CheriBSD, of which 107 have been converted to `PHYS_TO_DMAP_PAGE`. Another example, is the Bhyve virtual machine monitor which uses its own root capabilities with wide access to virtual address space.

Device drivers and filesystems For most device drivers, we observe that only limited changes are required to add support for CHERI. A typical device driver relies on a collection of kernel functions to access hardware interfaces, e.g., perform direct memory access (DMA). Those functions were changed to support capability interfaces, e.g., a collection of `bus_dma` functions to implement safe access to DMA interfaces.

One common class of changes is improving type precision regarding integers and pointers. The CheriBSD team maintains a fork of OpenZFS with CHERI-specific adaptations which total around 500 lines of changes [12]. As an example, ZFS stores its data in `nvlist` containers which are made up of name-value pairs. It has the `nvl_priv` field which stores a pointer to private data but it has an integer type. The CHERI-enabled fork updates it to the pointer-sized type, `uint64ptr_t`.

6. Related work

Kernel and device-driver vulnerabilities Device drivers and kernel extensions have long been considered one of the main sources of vulnerabilities in the kernel [7, 33]. In 2001, an empirical study of faults in the Linux kernel by Chou et al. found that device drivers contained 7 times more faults compared to other subsystems [7]. This observation was confirmed in 2011 despite rapid evolution of the Linux kernel, improved testing, and static analysis [33]. Chen et al. provided a comprehensive overview of how the kernel gets exploited, i.e., analysis of common vulnerabilities in the Linux kernel [4]. Li et al. performed an analysis of the CVE database from 2014 to 2023 which shows that that device drivers account for 16-59% of all vulnerabilities in the Linux kernel [25]. Our work extends the dataset provided by Li et al. with CVEs in the FreeBSD kernel as well as 100 new CVEs in the Linux kernel, and refines the taxonomy of vulnerabilities to clearly separate the causes and effects.

Security impact of CHERI Joly et al. conducted the first security analysis of the CHERI ISA [22]. They highlighted main possibilities for exploitation of pure capability systems. Some of their techniques were subsequently fixed in CheriBSD, e.g., overly permissive capability bounds, exact capability bounds in the memory allocator, etc.

Impact of Rust on low-level vulnerabilities Early reports estimated that nearly 70% of security issues in the low-level code that are typically assigned a CVE are related to memory safety and hence could be eliminated with Rust [31, 18]. A similar study reports that Rust can eliminate 53 out of 95 known security flaws in cURL, a data transfer utility written in C [38].

7. Conclusions

CHERI architecture is a powerful security mechanism that can potentially alter complexity of kernel exploitation. Our analysis shows that CHERI can block a significant fraction of kernel vulnerabilities – while not as large as a safe programming language CHERI can prevent exploitation at a much lower level of development effort. We hope that our work improves understanding of potential security benefits offered by the CHERI capability architecture, and, hopefully, result in wider adoption of this practical security mechanism.

Acknowledgements

We would like to thank ACSAC '25 reviewers and the artifact evaluation committee for multiple insights that helped us to improve this work. This research is supported in part by the National Science Foundation under the Grant Number 2239615.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (CCS), 2005.

[2] Saar Amar, Tony Chen, David Chisnall, Nathaniel Wesley Filardo, Ben Laurie, Hugo Lefeuvre, Kunyan Liu, Simon W. Moore, Robert Norton-Wright, Margo Seltzer, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. CHERIoT RTOS: An OS for fine-grained memory-safe compartments on low-cost embedded devices. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles* (SOSP), 2025.

[3] C coding standard, 2004. URL: https://users.ece.cmu.edu/~eno/coding/CCodingStandard.html.

[4] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems* (APSys'11), 2011.

[5] Dustin Childs. Pwn2Own Berlin 2025: Day one results, Zero Day Initiative, May 15, 2025. URL: https://www.zerodayinitiative.com/blog/2025/5/15/pwn2own-berlin-2025-day-one-results.

[6] Dustin Childs. Pwn2Own Berlin 2025: Day three results, Zero Day Initiative, May 17, 2025. URL: https://www.zerodayinitiative.com/blog/2025/5/17/pwn2own-berlin-2025-day-three-results.

[7] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating system errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (SOSP), October 2001.

[8] Kees Cook. Security: Implement Clang's stack initialization, April 19, 2019. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=709a972efb01efaeb97cad1adc87fe400119c8ab.

[9] Jonathan Corbet. Resistance to Rust abstractions for DMA mapping, LWN.net, January 30, 2025. URL: https://lwn.net/Articles/1006805/.

[10] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7* (SSYM'98), 1998.

[11] Data execution prevention, Microsoft, May 1, 2023. URL: https://learn.microsoft.com/en-us/windows/win32/memory/data-execution-prevention.

[12] Brooks Davis. CHERI memory safety with OpenZFS, 2022. URL: https://www.youtube.com/watch?v=dLeZlz52d-k.

[13] Brooks Davis. Commit 519ca7bf21222d3deb03202f0205a7cb6dd177bb: "Merge kernel support for CHERI capabilities." October 24, 2013. URL: https://github.com/CTSRD-CHERI/cheribsd/commit/519ca7bf21222d3deb03202f0205a7cb6dd177bb.

[14] Brooks Davis. Commit c9179b86fbc78ddcdf9f522740ecdd310f18c4e1: "Merge libcheri and cheritest." October 24, 2013. URL: https://github.com/CTSRD-CHERI/cheribsd/commit/c9179b86fbc78ddcdf9f522740ecdd310f18c4e1.

[15] Brooks Davis. Commit e73baa995a080e494d517745bf3544e180b08e32: "Align the stack to a 32-byte boundary on program initialisation in csu", October 25, 2013. URL: https://github.com/CTSRD-CHERI/cheribsd/commit/e73baa995a080e494d517745bf3544e180b08e32.

[16] Brooks Davis. Commit fb6852dcfcc696eb57ad5163a4773cef942ff2d4: "Merge userspace support for Philip Withnall's compositor." October 25, 2013. URL: https://github.com/CTSRD-CHERI/cheribsd/commit/fb6852dcfcc696eb57ad5163a4773cef942ff2d4.

[17] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Jessica Clarke, Peter Rugg, Brooks Davis, Mark Johnston, Robert Norton, David Chisnall, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. Cornucopia Reloaded: Load barriers for CHERI heap temporal safety. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (ASPLOS), 2024.

[18] Implications of rewriting a browser component in Rust. URL: https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust/.

[19] Microsoft Threat Intelligence. Exploitation of CLFS zero-day leads to ransomware activity, Microsoft Security, April 8, 2025. URL: https://www.microsoft.com/en-us/security/blog/2025/04/08/exploitation-of-clfs-zero-day-leads-to-ransomware-activity/.

[20] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block oriented programming: Automating data-only attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (CCS), 2018.

[21] Alexandre Joannou, Jonathan Woodruff, Robert Kovacsics, Simon W. Moore, Alex Bradbury, Hongyan Xia, Robert N.M. Watson, David Chisnall, Michael Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alfredo Mazzinghi, Alex Richardson, Stacey Son, and A. Theodore Markettos. Efficient tagged memory. In *2017 IEEE International Conference on Computer Design (ICCD)*, 2017.

[22] Nicolas Joly, Saif ElSherei, and Saar Amar. Security Analysis of CHERI ISA. Technical report, Microsoft Security Response Center (MSRC), October 2020.

[23] Mateusz Jurczyk and Google Project Zero. The Windows Registry adventure #7: Attack surface analysis, Project Zero, May 23, 2025. URL: https://

googleprojectzero . blogspot . com / 2025 / 05 / the - windows-registry-adventure-7-attack-surface.html.

[24] The kernel development community. The Linux kernel documentation - Rust. URL: https://docs.kernel.org/rust/index.html.

[25] Zhaofeng Li, Vikram Narayanan, Xiangdong Chen, Jerry Zhang, and Anton Burtsev. Rust for Linux: Understanding the security impact of Rust on the Linux kernel. In *Proceedings of the 40th Annual Computer Security Applications Conference* (ACSAC), 2024.

[26] llvm.org. Scudo hardened allocator, 2024. URL: https://llvm.org/docs/ScudoHardenedAllocator.html.

[27] Alfredo Mazzinghi. Commit 4330adc7aaf546712603412c57b8c1b35e12ef71: "purecap-kernel: Implement R_MORELLO_RELATIVE self-relocions at boot." April 8, 2021. URL: https://github.com/CTSRD-CHERI/cheribsd/commit/4330adc7aaf546712603412c57b8c1b35e12ef71.

[28] Alfredo Mazzinghi. Commit 46322b14516a03efe41521bb66bd4253f3d577a2: "purecap-kernel: Enter exception handlers in C64 mode." April 11, 2021. URL: https : / / github . com / CTSRD - CHERI / cheribsd / commit / 46322b14516a03efe41521bb66bd4253f3d577a2.

[29] Alfredo Mazzinghi. Commit 856f48789a63954c91c83a816c65ed33316c291aa: "purecap-kernel: Add support to allocate capabilities from vmem arenas", February 25, 2021. URL: https://github.com/CTSRD-CHERI/cheribsd/commit/856f48789a63954c91c83a816c65ed33316c291aa.

[30] Alfredo Mazzinghi. Commit d89c5ad14b60c01c64c9f1215ecf6d6131276853: "purecap-kernel: Introduce the vm_pointer_t type", November 27, 2020. URL: https : / / github . com / CTSRD - CHERI / cheribsd / commit / d89c5ad14b60c01c64c9f1215ecf6d6131276853.

[31] Microsoft. We need a safer systems programming language, July 18, 2019. URL: https://msrc.microsoft.com / blog / 2019 / 07 / we - need - a - safer - systems - programming-language/.

[32] notselwyn. Flipping pages: An analysis of a new Linux vulnerability in nf_tables and hardened exploitation techniques, May 26, 2024. URL: https://pwning.tech/nftables/.

[33] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in Linux: Ten years later. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), 2011.

[34] Valentina Palmiotti. Racing round and round: The little bug that could, 2024. URL: https://www.ibm.com/think/x-force/little-bug-that-coulds.

[35] Alexander Potapenko and Linus Torvalds. mm: Security: Introduce init_on_alloc=1 and init_on_free=1 boot options, July 12, 2019. URL: https : / / git . kernel . org / pub / scm / linux / kernel / git / torvalds / linux . git / commit / ?id = 6471384af2a6530696fc0203bafe4de41a23c9ef.

[36] Trustworthy Secure Research and Development. CheriBSD, 2025. URL: https://github.com/CTSRD-CHERI/cheribsd.

[37] Alexander Richardson. *Complete spatial safety for C and C++ using CHERI capabilities*. PhD thesis, October 2019.

[38] Rust for curl. URL: https://timmmm.github.io/curl-vulnerabilities-rust/.

[39] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security* (CCS), 2004.

[40] System hardening in Android 11, 2020. URL: https://source.android.com/docs/security/test/scudo.

[41] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, 2013.

[42] Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Peter G. Neumann. An Introduction to CHERI. Technical report UCAM-CL-TR-941, University of Cambridge, Computer Laboratory, September 2019.

[43] 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. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9). Technical report UCAM-CL-TR-987, University of Cambridge, Computer Laboratory, September 2023.

[44] Robert N. M. Watson, Alexander Richardson, Brooks Davis, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Filardo, Simon W. Moore, Edward Napierala, Peter Sewell, and Peter G. Neumann. CHERI C/C++ Programming Guide. Technical report UCAM-CL-TR-947, University of Cambridge, Computer Laboratory, June 2020.

[45] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. Cornucopia: Temporal safety for CHERI heaps. In *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[46] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson,

Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The CHERI Capability Model: Revisiting RISC in an age of risk. *ACM SIGARCH Computer Architecture News*, (3), June 14, 2014.

[47] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G Neumann, Simon W Moore, Robert NM Watson, et al. CHERIvoke: Characterising pointer revocation using CHERI capabilities for temporal memory safety. In *52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

Appendix

1. Causes and Manifestations

To reason about the impact of both low-level safety mechanisms like CHERI and high-level approaches like safe programming languages, it is important to understand the relationship between high-level vulnerability causes and their low-level manifestations. Table 4 provides a detailed breakdown of mapping between manifestations and causes. In our taxonomy we classify vulnerabilities into four broad classes: language defects (131 vulnerabilities), protocol violations (11 vulnerabilities), data races (68 vulnerabilities), and semantic errors (229 vulnerabilities).

Language limitations Language defects constitute 30% of our dataset. Roughly 17% of total CVEs are caused by lifetime violations. Lifetime violations (73 CVEs) are the largest vulnerability cause along with improper input validation (83 CVEs) and logic errors (63 CVEs). Lifetime violations manifest as use after free (43 CVEs), resource and memory leaks (19 CVEs), double free (9 CVEs), and failures to release the CPU (2 CVEs). Most of improper memory initialization errors (28 CVEs) manifest as uninitialized memory access, however, 2 manifest as invalid pointer dereference and 1 manifests as high-level specification violation. Most integer overflows manifest as out-of-bounds accesses (12 CVEs).

CHERI will be able to mitigate UAF, OOB accesses, and invalid pointer dereferences through capability enforcement. However, it will not mitigate explicit exceptions or panics, as these typically follow valid code paths to the deliberate exception. Likewise, CHERI does not prevent failure to release the CPU, such as deadlocks or hangs, nor does it address high level spec violations, since these do not stem from memory safety violations but rather from semantic logic errors.

Protocol violations Protocol violations constitute is a small group of vulnerabitlies (2.5%) in our dataset. Missing protocol steps (10 CVEs) restult in a variety of manifestations: invalid pointer dereferences (3 CVEs), double free (2 CVEs), high-level specification violations (2 CVEs), use-after-free, explicit exceptions, and failure to release the CPU.

Race conditions Race conditions constitute 15% of CVEs in our dataset and are split between 37 improper use of synchronization primitives and 31 time-of-check-time-of-use manifestations. Among 37 cases of improper usage of synchronization primitives, 24 lead to use after free, 8 lead to failure to release the CPU, 3 to invalid pointer dereference, 1 to out-of-bound access, and 1 to a high-level specification violation. Out of 31 TOCTOU, 23 lead to use after free, 2 to OOB access, 2 to invalid pointer dereference, 2 to explicit exception or panic, 1 to failure to release CPU, and 1 to a high level spec violation.

Semantic vulnerabitlies and logic errors Finally, semantic errors correspond to 52% of all kernel vulnerabilities. Out of 63 logic error cases, 16 lead to OOB bounds access, 11 to access control violations, 4 to uninitialized memory access, 11 to use after free, 2 to resource leaks, 5 to high level spec violations, 3 to explicit exception or panic, 3 to failure to release CPU, 8 to invalid pointer dereference. Among the 83 improper input validation cases, 52 result in OOB access, 15 in invalid pointer dereference, 5 in explicit exception or panic, 3 in access control violation, 2 in use after free, 3 in high level spec violation, and one double free, CFI violation, and failure to release CPU. Finally, out of the 33 spec error cases, 8 result in high level spec violation, 7 in use after free, 4 in failure to release CPU, 1 in uninitialized memory access, 3 each in out of bounds access, access control violation, and resource leak, and 2 each in double free and explicit exception or panic.

2. CHERI vs Rust: Manifestation-Centric View

Table 5 extends the cause-centric view of the effectiveness of Rust and CHERI which we discussed in Table 3 with the view centered around low-level manifestations. Like CHERI, Rust implements both spatial and temporal safety which allow it to block low-level safety manifestations. Rust however, extends low-level safety with its ownership discipline and high-level programming language abstractions that guard developers from a range of errors: double free, resource leaks, uninitialized memory accesses, and even some fraction of panics, failures to release the CPU, high-level specification violations, and access control violations (Table 5).

Compared to CHERI Rust fails to mitigate several safety vulnerabilities: 4 out-of-bound access and 2 use-after free CVEs. Those CVEs stem from semantic and specification-level errors such as improper input validation and logic mistakes that would not be resolved by porting the code to Rust. For example, CVE-2021-38204 illustrates a specification-level error in the MAX-3421 driver that manifests as use-after-free. The driver cached toggle state in memory to reduce the SPI traffic, but continued to reference the cached data even after the corresponding device was removed. This stale reference caused writes to freed memory. Fixing the issue required changing the driver's specification to always read and write the toggle state on each transfer. Although Rust can generally eliminate lifetime and use-after-free bugs, it cannot correct flawed design assumptions in low-level I/O drivers. Such specification-level mistakes can persist in Rust because they arise in unsafe hardware interaction paths where the compiler cannot infer lifetimes. In contrast, CHERI would invalidate the cached stale pointer, preventing

TABLE 4: Causes and manifestations

Cause vs. Manifestation	OOB access	Use after free	Double free	Uninit. mem. access	Resource leak	Invalid pointer dereference	Explicit exception/panic	Failure to release CPU	Control flow violation	High level spec violation	Access control violation	Total
Language												
Integer overflow	11	0	0	0	0	0	0	0	0	3	0	14
Polymorphism	4	0	0	0	0	7	0	0	0	0	0	11
Container of	0	0	0	0	0	0	0	0	0	0	1	1
Sentinel arrays	4	0	0	0	0	0	0	0	0	0	0	4
Improp. mem. init.	0	0	0	25	0	2	0	0	0	1	0	28
Lifetime violation	0	43	9	0	19	0	0	2	0	0	0	73
Protocol												
Missing prot. steps	0	1	2	0	0	3	1	1	0	2	0	10
Sleeping in atomic	0	0	0	0	0	0	0	1	0	0	0	1
Race condition												
Improp. use Of sync.	1	24	0	0	0	3	0	8	0	1	0	37
TOUTOC	2	23	0	0	0	2	2	1	0	1	0	31
Semantic												
Logic error	16	11	0	4	2	8	3	2	0	6	11	63
Improper input val.	52	2	1	0	0	15	5	1	1	3	3	83
Spec error	3	7	2	1	3	0	2	4	0	8	3	33
Security & perm.	0	0	0	0	0	0	0	0	0	2	18	20
Loop termination	0	0	0	0	0	0	0	4	0	0	0	4
Missing ret. val.	1	2	0	1	0	20	1	0	0	1	0	26
Total	94	113	14	31	24	60	14	24	1	28	36	439

TABLE 5: Cheri vs. Rust (manifestation-centric view)

Manifestation	CHERI		Rust		Vulns. blocked(%)	
	Yes	No	Yes	No	CHERI	Rust
OOB access	49	0	45	4	100%	91%
Use after free	77 | 0	0 | 77	75	2	100% | 0%	97%
Double free	0	13	12	1	0%	92%
Uninitialized memory access	0	11	5	6	0%	45%
Resource leak	0	14	14	0	0%	100%
Invalid pointer dereference	40	0	40	0	100%	100%
Explicit exception/panic	0	8	2	6	0%	25%
Failure to release CPU	0	10	3	7	0%	30%
High level spec violation	0	4	1	3	0%	25%
Access control violation	0	8	1	7	0%	12%
Total	166 | 89	68 | 145	198	36	70% | 38%	84%

the UAF even under the flawed specification.

Compared to CHERI, however, Rust is able to mitigate 12 double frees, 5 uninitialized memory accesses, 14 resource leaks, 2 explicit exceptions/panics, 3 failures to release the CPU, 1 high-level specification violation, and 1 access control violation. Those CVEs are mitigated via Rust's ownership and lifetime enforcement which require developers to resolve such bugs during the porting process [25]. For example, all resource leaks in our dataset are eliminated because they resulted from lifetime violations, which Rust prevents through its ownership discipline.

Similarly, lifetime violations accounted for most double-free cases, which Rust can mitigate. The sole case that was not mitigated (CVE-2021-20292) stemmed from missing protocol steps, which is not something Rust can fix. Vulnerabilities involving information leaks from uninitialized memory, such as CVE-2024-26638 and CVE-2020-11494, are prevented through a combination of Rust's type system and data flow analysis that enforces that all variables are initialized before use. Note, Rust cannot fully mitigate cases where initialization is not the sole issue. For instance, CVE-2021-26930 illustrates how correctly initialized values can still lead to unsafe behavior when logic errors in control flow produce unexpected states at runtime. Since proper mitigation of such CVEs depends heavily on their underlying cause, only 45% of uninitialized memory errors are properly mitigated by Rust. Other manifestations are also case-specific, as Rust does not guarantee protection through a single underlying mechanism. Explicit exception/panic cases, for example, may result from input validation errors, missing protocol steps, race conditions, etc., which Rust wouldn't handle. Only 2 explicit exception/panic CVEs are mitigated by Rust: CVE-2023-31085, a divide-by-zero error that Rust's runtime checks mitigate by causing a deterministic panic rather than undefined behavior in languages like C, and CVE-2020-15437, which causes a null pointer dereference due to lifetime violation.

3. Artifacts and Reproducibility

To ensure the availability and reproducibility of our research, we make all datasets and scripts used in this paper available as a publicly hosted GitHub repository (https://github.com/mars-research/cheri-impact-artifact). The datasets contain all CVEs analyzed in our study along with the explanation for how the labeling and classification was done.

The dataset can be accessed as a publicly accessible Google Spreadsheet and as a Python script that generates the queries required for constructing tables in this paper. We used Google Sheets for rapid prototyping and interactive analysis of the dataset, but then cross-verified final results with a Python script. In a spreadsheet environment, it is hard to reason about the flow of computation which can often lead to small but critical mistakes, A Python script, on the other hand, provides a readable and reproducible artifact.

We accompany the dataset with detailed instructions for reproducing the vulnerability statistics reported in this paper (see `README.md` in the artifact GitHub repository).

We tested the scripts for generating the data for all tables in the paper with Python 3.9.6 and Pandas 2.3.2 on macOS 15.6.1.