Sacha RUCHLEJMER

SEOC

2024

Ericsson

Torshamnsgatan 21, 164 40 Kista, Sweden

Secure Rewind and Discard on Arm Morello

from 21/02/2024 to 07/07/2024

Under the supervision of:

- Company supervisor: Merve, GÜLMEZ, merve.gulmez@ericsson.com
- Phelma Tutor: Cyrille, CHAVET, cyrille.chavet@grenoble-inp.fr

Confidentiality: ☐ yes ☒ no

Ecole nationale
supérieure de physique,
électronique, matériaux

Phelma
Bât. Grenoble INP - Minatec
3 Parvis Louis Néel - CS 50257
F-38016 Grenoble Cedex 01

Tél +33 (0)4 56 52 91 00
Fax +33 (0)4 56 52 91 03

http://phelma.grenoble-inp.fr

Contents

List of Figures

Listings

List of Tables

Glossary

API Application Programming Interface. 20–25, 33

Assembly Assembly language is a low-level programming language that uses mnemonic codes and symbols to represent instructions that can be directly understood by a computer's CPU (Central Processing Unit). Unlike high-level languages, assembly language corresponds closely to the machine code instructions of a specific computer architecture, making it powerful for tasks requiring precise control over hardware resources and performance optimization. 24, 25

C C is a general-purpose, procedural programming language. It provides low-level access to memory and hardware, making it suitable for system programming, such as operating systems and embedded systems. 11, 12, 17, 20, 23, 38

C++ C++ is an extension of the C programming language. It includes object-oriented features such as classes and inheritance, making it suitable for large-scale software development. 11, 12

CHERI Capability Hardware Enhanced RISC Instructions. 9, 10, 12–15, 17–23, 25, 26, 28, 29, 31, 33–38

ISA Instruction-Set Architecture. 9, 12, 14

Kernel The kernel is the core component of an operating system, responsible for managing system resources and facilitating communication between hardware and software. It handles critical tasks such as process management, memory management, device management, and system calls, ensuring efficient and secure operation of the computer system.. 15, 16, 34

LIFO Last In, First Out. 25

MMU Memory Management Unit. 12

MPK Memory Protection Key. 15, 26, 36, 38

OS Operating System. 12, 15, 16, 26, 27

RISC Reduced Instruction Set Computer. 9

SDRaD Secure Domain Rewind and Discard. 9, 10, 15, 16, 19–23, 25–29, 33–36, 38

TLSF Two-Level Segregated Fit. 28–32, 34–36, 38

UDI User Domain Index. 21–23, 25

Acknowledgments

I would like to express my sincere gratitude to Merve Gülmez, my thesis supervisor, security researcher at Ericsson, for her invaluable guidance, patience, and encouragement throughout this research. Her expertise and insightful feedback were instrumental in shaping this work. She knew how to give me all the keys I needed to accomplish this work.

I am also thankful to Cyrille Chavet, my school supervisor, for his support, guidance, and for always being attentive to any questions I had.

I am also grateful to Thomas Nyman, a security expert at Ericsson, for his help and support during crucial moments, and for always offering valuable advice.

I would also like to extend my gratitude to Christoph Baumann, a researcher, for his invaluable support, assistance in acquiring the resources I needed, and offering valuable advice.

Special thanks to the other Master's Thesis students and colleagues Sönke and Panagiotis for their stimulating discussions and unwavering support.

This project is done at the network platform and telecommunication company Ericsson.

1 Introduction

Numerous applications are developed using memory-unsafe languages, rendering them susceptible to runtime attacks such as control-flow attacks and data-oriented attacks. These vulnerabilities provide attackers with avenues to gain unauthorized access to programs, by exploiting weaknesses to manipulate and corrupt their behavior. According to a U.S National Security Agency report "The Case for Memory Safe Roadmaps" [4], two-thirds of reported vulnerabilities in memory-unsafe programming languages still relate to memory issues. Nowadays, there are well-known solutions to mitigate these vulnerabilities like stack canaries [5], but at the end such mitigations terminate the process to prevent attacks exploiting such vulnerabilities from being successful, already corrupted memory will, under normal circumstances prevent the normal operation of the applications. This is especially problematic for service-oriented applications such as web-servers, which must maintain consistent service for all clients even in presence of malicious clients.

State-of-the-art approaches address two related challenges, 1) how to improve the resilience of applications, and 2) how to prevent programs from being exploited by memory-related attacks. **Secure Domain Rewind and Discard (SDRaD)** [2] is prior work addressing the first challenge by allowing parts of processes (i.e, sub-processes or routines) to be isolated by creating new, logical protection domains within a conventional process, each with its own stack and heap distinct from the main application stack and heap, and those of other domains. Thanks to this in-process isolation it is possible to discard any domains which memory is corrupted by run-time attacks and going back to a safe anchor point allowing the application to continue running even if the part is corrupted.

Capability Hardware Enhanced RISC Instructions (CHERI) [6] is prior work addressing the second challenge by extending conventional hardware Instruction-Set Architectures (ISAs) with new architectural features to enable fine-grained memory protection and highly scalable software compartmentalization. It uses the concept of capability-based addressing to store metadata, such as bounds information about pointers that is used to prevent buffer overflows from happening.

The objective of this thesis is to integrate SDRaD with CHERI to introduce resilience into this novel architecture and leverage its security advantages to make it more lightweight and address previous limitations by leveraging the compartmentalization capabilities in CHERI.

In this thesis, Section 2 explains memory-related attacks and explores the concept of CHERI, including an introduction to the secure rewind and discard concept. Section 3 discusses the current limitations of CHERI and stack canaries, highlighting areas where improvements are needed. Section 4 introduces the high level adaptation for secure rewind and discard to the CHERI architecture (see in Section 4.1), explains the implementation of the CHERI-SDRaD library (see in Section 4.2), examines use cases and discusses the performance overhead (see in Section 5). Lastly, the result of this thesis is discussed in Section 6.

Secure Rewind and Discard on Arm Morello Master's Thesis artifact will be available at `https://github.com/secure-rewind-and-discard/`.

2 Background

2.1 Memory safety

In programming, memory safety is a critical concern that should be addressed to prevent undesirable behaviors. Many memory issues arise from developer errors rather than from the language itself. Today, memory-safe languages have demonstrated potential in mitigating a broad range of threats. However, they often come with trade-offs, such as increased resource requirements for executing code and limitations on developers' ability to manage low-level memory. As a result, memory-unsafe languages such as C and C++ remain widely utilized due to their unmatched performance control and suitability for tasks that demand direct memory manipulation [7].

Memory violations Memory violations occur when a program accesses memory in an unintended or unauthorized way, leading to unpredictable behavior or crashes.

Runtime attacks Runtime attacks exploit vulnerabilities during program execution, often targeting memory-related weaknesses to inject malicious code or alter the program's intended flow.

Control-flow attacks Control-flow attack is one class of runtime attacks. It involves manipulating the sequence of instructions executed by a program, typically through exploiting vulnerabilities in control-flow mechanisms like function pointers or return addresses. Control-flow attacks can be categorized into two main types: code-injection and code-reuse attacks. Code-injection attacks utilize methods like buffer overflows to manipulate the return address of functions, redirecting the program execution to previously introduced malicious code. On the other hand, code-reuse attacks aim to alter the application's behavior by modifying return addresses to unintended existing functions.

Buffer overflows Buffer overflows happen when a program writes data beyond the allocated buffer space, potentially overwriting adjacent memory, which can be exploited by attackers to execute arbitrary code or manipulate program behavior. It can be used to overwrite a previous return address with a new one, redirecting execution to a malicious program injected by an adversary. This vulnerability can manifest in various forms; two significant

examples include during memory copy operations between variables of different sizes and when an application accepts inputs without verifying if they exceed the allocated buffer size.

Data-oriented attacks Data-oriented attack is another class of runtime attacks. It focuses on exploiting vulnerabilities related to how data is processed and accessed within a program, aiming to gain unauthorized access to sensitive information or modify program state. Unlike control-flow attacks, which manipulate the program's execution flow, data-oriented attacks often target weaknesses in how data are handled, such as insecure data storage, inadequate input validation, or insufficient data sanitization. These vulnerabilities can be exploited to steal sensitive information or manipulate program behavior to the attacker's advantage.

2.2 Capability Hardware Enhanced RISC Instructions

CHERI (Capability Hardware Enhanced RISC Instructions) [6] represents a collaborative research between SRI International and the University of Cambridge, aimed at reevaluating fundamental design principles in both hardware and software to significantly enhance system security.

CHERI augments traditional hardware ISAs with novel architectural elements, enabling fine-grained memory protection and scalable software compartmentalization. These enhancements in memory protection empower memory unsafe programming languages like C and C++ to offer robust, compatible, and efficient defenses against many memory-related attacks such as buffer overflow. Additionally, the scalable compartmentalization capabilities of CHERI enable fine-grained segmentation of Operating System (OS) and application code, thereby mitigating the impact of security vulnerabilities in ways not previously feasible with existing architectures.

Notably, CHERI adopts a hybrid capability architecture, integrating architectural capabilities with conventional Memory Management Unit (MMU)-based architectures, microarchitectures, and established software stacks built on virtual memory and C/C++. This approach allows for gradual and easier integration into existing ecosystems [1].

Capability-Based addressing

CHERI employs a capability-based addressing scheme. The principle is to replace classical pointers with protected objects called capabilities, which will store, in addition to the pointer address, additional information like permissions or the pointer's intended bounds.

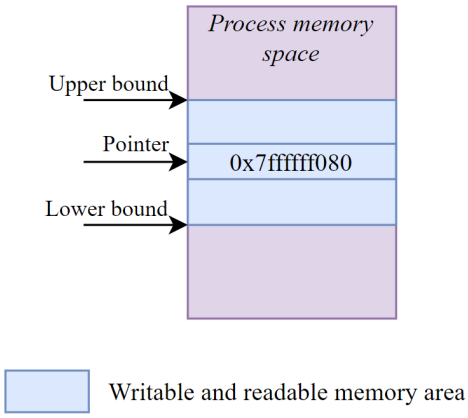


Figure 1: Capability representation.

As illustrated in Figure 1, capability-based addressing not only reveals the precise memory location occupied by a capability, but also enables the detection of overflows or overreads before they can occur by explicitly defining the bounds within the pointer's own definition.

CHERI capability layout

The in-memory layout of a CHERI capability is shown in Figure 2 where it is depicted as consisting of two layers. The lower layer represents the conventional pointer familiar in traditional programming, while the upper layer includes all the additional information that transforms the pointer into a capability, thus rendering it a protected object. One notable observation is that while a conventional pointer typically occupies 64 bits of memory, the introduction of capabilities necessitates doubling this space to 128 bits.

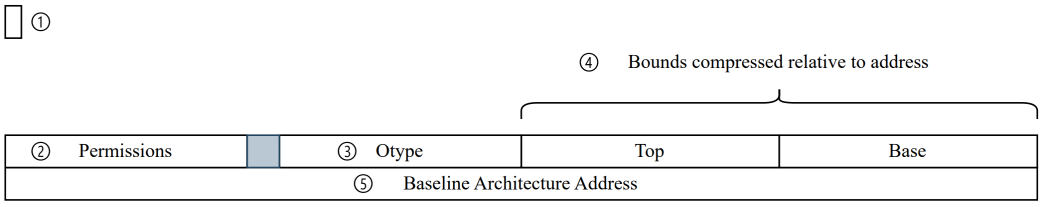


Figure 2: CHERI capability layout, adapted from [1].

This architecture introduces several enhancements to improve data security and integrity:

1. **Validity Tag** (①): Each capability is associated with a 1-bit validity tag, which is maintained automatically in registers and memory. The

tag tracks the validity of a capability and ensures that special capability write instructions are needed to create valid capabilities. Regular memory writes, even partial ones, to the memory area of the capability clear the validity tag and invalidate the capability, preventing corrupt capabilities from being dereferenced.

2. **Permissions** (②): The permissions mask controls how the capability can be used, such as by restricting loading (read) and storing (write) of data and/or capabilities, or by prohibiting instruction fetch (execute). Every load, store, or instruction fetch in a CHERI-enabled microprocessor architecture must be authorized by an architectural capability with the corresponding permissions.

3. **Object Type (Otype)** (③): Object types allow multiple capabilities to be associated with each other, facilitating software compartmentalization where only a specific set of capabilities can be used within a logically isolated compartment.

4. **Bounds** (④): The lower and upper bounds describe the portion of the address space to which the capability authorizes loads, stores, and/or instruction fetches, depending on the permissions the capability grants.

5. **Baseline Architecture Address** (⑤): A conventional pointer in the underlying ISA native format.

CHERI implements two modes of operation. The first, called hybrid mode, allows conventional pointers and CHERI capabilities to coexist and be used independently. In most cases, this enables programs that were not originally developed for a CHERI architecture to continue functioning on CHERI-enabled hardware. The second mode, known as pure-capability (often referred to as purecap), is the most secure mode of operation. In this mode, pointers are completely replaced by capabilities, resulting in the most secure programs utilizing CHERI technology.

Arm Morello Board

The Arm Morello development board (referred to as subsequently as simply “the Morello board") [8] is an industrial demonstrator of a capability architecture developed by Arm, featuring a prototype System-on-Chip (SoC). This board incorporates a CHERI-extended ARMv8-A processor, GPU, peripherals, and a memory subsystem. The Morello board allows for hardware and software to be tested in real-world conditions, enabling the evaluation

of CHERI's viability and performance impact. Its primary objectives are to facilitate industrial evaluation of CHERI hardware and software concepts, gather evidence for potential adoption, and support ongoing research and development. By integrating CHERI into a widely deployed, real-world architecture with a high-end, mature processor design and a robust software ecosystem, the Morello board aims to advance the practical application of these technologies.

The Morello board runs an adapted version of the FreeBSD OS called CHERIBSD. The experimental work described in this thesis has been conducted using a Morello board on loan to Ericsson to obtain firsthand insight into the performance metrics, rather than relying solely on simulated data, which could be influenced by the underlying hardware support.

2.3 Secure Domain Rewind and Discard

SDRaD (Secure Domain Rewind and Discard)[2] project aims to improve the resilience of applications against run-time attacks. Indeed, today's mitigation techniques against run-time attacks terminate the application when they detect an attack. However, terminating the application in response to an attack is disruptive to service-oriented applications that service many independent clients simultaneously. If the service process is terminated all clients being serviced will lose their connections because of one attack. Motivated by this, alternative approaches are being explored, with SDRaD being one of them. The main idea behind SDRaD is to isolate different parts of a program from each other in separate domains using in-process isolation. This domain mechanism allows the code inside it to be executed in a different part of memory than the process's own. When a new domain is created, it initializes with its own stack and heap. Additionally, to prevent access to unwanted memory spaces, such as those of other domains, an isolation mechanism, such as Memory Protection Keys (MPKs) is used to check if the rights to use this memory space are present. This isolation prevents domains from accessing each other's memory, which stops any domain affected by a run-time attack from corrupting memory belonging to another domain.

The initial SDRaD prototype implementation targeted the 64-bit x86 architecture and utilized MPK, a technology developed by Intel and introduced in the "Skylake" microarchitecture [9]. MPK is a hardware feature that provides memory protection at the page granularity. The main purpose of MPK is to allow software developers to define memory regions with specific protection keys. This enables fine-grained control over memory access without the need to rely on kernel-enforced access control for memory pages, which is required for regular processes. Such reliance adds a costly context switch every

time the execution context changes from one protection domain to another. In practice, MPK works by associating a protection key with each memory page. The permissions for the keys are stored in a special register called the protection key register, and the keys are added to the page table. The page table is a data structure used by the OS to map virtual addresses to physical addresses in memory. A program can specify a protection key when accessing a memory page, and the processor checks if the provided key matches the key associated with that page without involving the kernel. If the keys match, memory access is allowed; otherwise, an exception is triggered.

In summary, SDRaD provides an in-process-based solution that allows compartmentalizing the application into distinct domains, where each domain operates independently and can be discarded if its memory has been corrupted, and guarantees that memory belonging to other domains are unaffected.

3 Problem Statement: Limitation with CHERI and other defenses

The limitations of current mitigation techniques and the defense provided by CHERI is demonstrated in Listing 1, 3 and 4 using a small program that prompts a user to enter characters into the program buffer.

```
1  void get_request(){
2      char buff[5];
3      printf("Waiting for the request:\n");
4      scanf("%s",buff);
5      printf("Handling the request\n");
6  }
7
8  int main(void){
9      int i = 0;
10     for(i = 0; i < 5; i++){
11         get_request();
12     }
13
14     return 0;
15 }
```

Listing 1: Example of unsafe code.

Listing 1 shows an example of a program that uses an unsafe C function, `scanf()`, to read user input. The `scanf()` function is implemented in the correct way, but it's unsafe because it has no robust input validation. In classic C, entering a string larger than the buffer size does not generate an error. Although there are existing solutions such as stack canaries [5] that can detect such attacks, as shown in Figure 3.

```
eccraus@E-5CG2302YJ7:~/cheri_experiments/src$ ./test
Waiting for the request:
dsdsdsdsdsdsdsdsdsdsdsdsds
Handling the request:
*** stack smashing detected ***: terminated
Aborted
```

Figure 3: Handling buffer overflows with canaries.

As illustrated in Figure 3, the stack canaries are able to detect when a user attempts to input too many characters into the buffer. A stack canary is a random value placed after local variables on the stack. Before and after potentially unsafe operations (e.g. updating local variables), the program checks the integrity of the canary value. If the program detects that the canary was overwritten, it indicates that the input exceeds the size allocated for the variable, resulting in a buffer overflow. As a result, because already corrupt memory cannot be recovered, the application must be terminated completely upon detecting the buffer overflow.

However, with the use of CHERI capabilities and their metadata describing the bounds of the object a pointer refers to, such as the buffer buff in Listing 1, it becomes possible to detect at the hardware level if the string intended for the buffer exceeds the allocated size. As illustrated in Figure 4, CHERI raises an exception if the string exceeds the specified bounds of the capability.

```
eccraus@cheribsd:~/cheri_experiments/bin$ ./jmp_purecap
Waiting for the request:
1234
Handling the request:
Waiting for the request:
12345
In-address space security exception (core dumped)
```

Figure 4: Handling buffer overflows with CHERI.

CHERI is also able to detect attempts to perform a buffer overflow. In this case the detection occurs *before* any memory corruption takes place. Therefore, using CHERI provides better defense because the memory remains intact. However, it does not address the issue of resilience, as the program is immediately stopped upon detecting a memory violation.

The Problem Statement

How can the memory-safety guarantees provided by the CHERI architecture be combined with Secure Rewind and Discard to improve software resilience against run-time attacks?

4 Secure Domain Rewind and Discard on Arm Morello Board

The goal of this thesis is to adapt the Secure Domain Rewind and Discard to the CHERI architecture, allowing vulnerable applications to be recovered upon detecting an attack. CHERI is capable of detecting buffer overflows before memory corruption occurs. This method eliminates the need to compartmentalize the stack when creating new domains, thereby allowing for a lighter-weight SDRaD design. The design requires:

1. Creating return points that splits the application into distinct crash-resistant domains.
2. Leveraging CHERI's hardware capabilities to detect attacks.
3. Heap compartmentalization to keep track of domain heap allocation.

4.1 High-level idea

The objective of adapting SDRaD to the CHERI architecture is to provide a solution that allows for crash-resistance and resilience against runtime attacks that exploit memory vulnerabilities but also to make it lighter and with better performance compared to the prototype SDRaD implementation targeting 64-bit x86.

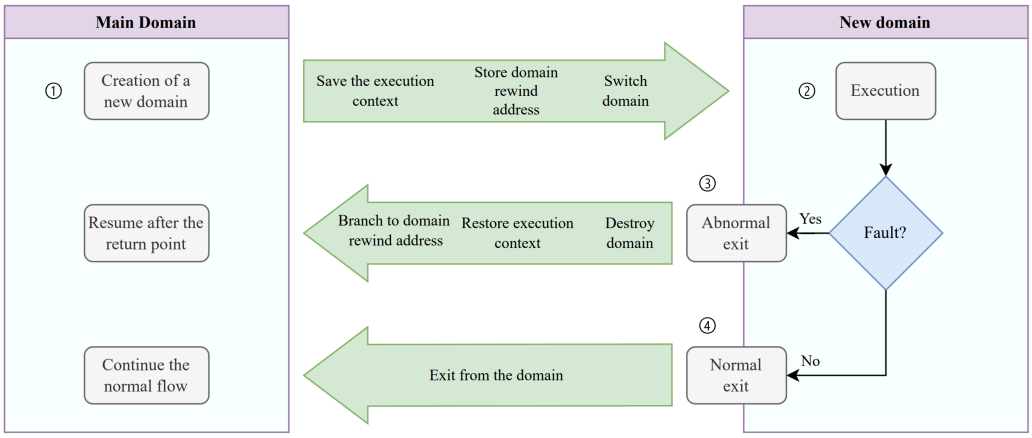


Figure 5: High-level idea of SDRaD for CHERI, adapted from [2].

Figure 5 illustrates the high-level idea of SDRaD. If a particular section of code in a program is at risk and needs to be isolated with the possibility of being rewound, it can be placed into a new domain. Initially, the application

runs within the main domain. Before executing the risky code, a new domain is created ① by saving the execution context (i.e., the program's status) and the current memory address to establish a checkpoint for potential rollback. The program then enters the newly created domain. ② Within this domain, two scenarios are possible. If memory corruption is detected, the domain exits prematurely with an abnormal exit ③, destroys the faulty domain, restores the saved execution context, and returns to the main domain, indicating the previously created domain exited abnormally. The caller is expected to handle abnormal exits, similar to code that might throw exceptions. If no memory corruption is detected, the domain exits normally ④, and the program continues its usual flow. In that case, this domain still exists and can be used again later. This approach ensures that risky code can be isolated and rewound from if necessary, without affecting the main domain, thus enhancing the program's resilience.

4.2 CHERI-SDRaD

The CHERI-SDRaD C library was developed to adapt SDRaD for the CHERI architecture using approximately 1.2k LoC of C code and 20 lines of Arm assembly code. This library provides an Application Programming Interface (API) that allows developers to integrate secure rewind and discard mechanisms into their applications. The following section will describe the design and implementation of the CHERI-SDRaD library.

Domain Manager

To manage the domains created within the application, a global manager was introduced, as illustrated in Listing 2. The Domain Manager is defined as a global variable so that it can be accessed from anywhere in the code. It has two members: the first one, `active_domain` ①, indicates the current domain, and the second, `domain_info` ②, stores the informations about currently existing domains. One limitation of the Domain Manager is that domain information is stored in an array with a static size (here 16). The maximum number of domains supported by the CHERI-SDRaD library must be defined at compilation time. This limitation could be addressed in future work by dynamically defining the domains, such as using a linked list. However, one advantage of the array-backed domain information storage is that information of any domain can be accessed in constant-time.

```
#define NUMBER_MAX_DOMAIN 15
enum State{UNINIT, INIT};

typedef struct _return_reg_type_s {
    void *c29;
    void *c30;
}return_reg_type_s;


typedef struct _domain_info_s {
    jmp_buf env;
    return_reg_type_s return_address;
    tlsf_t tlsf;
    uint32_t parent_udi; ③
    enum State domain_init;
    enum State heap_init;
} domain_info_s;


typedef struct _global_manager_s{
    uint32_t active_domain; ①
    domain_info_s domain_info[NUMBER_MAX_DOMAIN+1]; ②
} global_manager_s;
```

Listing 2: CHERI-SDRaD manager.

Furthermore, the CHERI-SDRaD manager includes a `parent_udi` ③ in the `domain_info` section. This means that domain nesting, i.e., having one domain inside another, is possible recursively within this implementation.

CHERI-SDRaD API

Developers can use the API specified in Table 1 to improve resilience inside their application.

API function name	arguments	description
① cheri_domain_setup()	udi	Create a new domain with the specified udi
② cheri_domain_enter()	udi	Enter inside an already created domain with its udi
③ cheri_domain_exit()	-	Exit from the current domain

Table 1: List of API functions.

Domains are initialized by invoking `cheri_domain_setup` ①. This function creates a new domain with a unique User Domain Index (UDI) and prepares it for use by saving the execution context and the return address to the Domain Manager, and completes the initialization process. Upon initialization, this call provides a return value: a positive value indicates successful initialization or that the domain with the specified UDI is already initialized, while a negative value signifies an out-of-bounds UDI. Once initialized,

a domain can be entered using `cheri_domain_enter` ②. This operation returns an error code if the specified UDI is out of bounds or not associated with any existing domain. Conversely, it provides a success code if the UDI is correct, enabling entry into the specified domain. When a domain is no longer needed, it can be exited by calling `cheri_domain_exit` ③, returning to its parent domain.

Listing 3 illustrates how the previous unsafe code can be encapsulated into a domain to ensure resilience in case of any corruption.

```
int main(void){
    int i = 0;
    int err;
    int uid = 1;
    for(i = 0; i < 5; i++){
        ⑤ err = cheri_domain_setup(uid); ①
        if(err == SUCCESSFUL_INITIALIZE || err == ALREADY_INITIALIZE){
            cheri_domain_enter(uid); ②
            get_request(); ④
            cheri_domain_exit(); ③
        } else {
            printf("Bad input!\n");
        }
    }

    return 0;
}
```

Listing 3: Unsafe code encapsulated into a new domain.

By using the API, the `get_request` ④ function now operates in a different domain from the main application. The `err` variable ⑥ captures the return value sent by `cheri_domain_setup`. If this value indicates successful initialization, `err` will have a positive value, signifying that the domain is correctly initialized. In this case, the *if* condition is met, allowing entry into the domain to start computing the unsafe function.If no bad behavior is detected during the execution of the unsafe function, the domain is exited by calling `cheri_domain_exit`, and the for loop continues from the next iteration. However, if the user provides input that would overflow the destination buffer, the invalid buffer operation is detected by CHERI, but the signal indicating that an invalid memory access was about to happen is captured by a signal handler in CHERI-SDRaD, which causes a abnormal domain exit. This is indicated to the caller via a false return value stored in `err`. Consequently, the *if* condition is no longer satisfied, causing the program to enter the *else* block, which warns the user about the invalid input.

Figure 6 shows, using the example application from Listing 3, how using CHERI-SDRaD handles buffer overflows in a domain. When a user submits an input that is too large, the routine is interrupted, causing the domain to exit without completing the execution of the code inside. As a result, when encountering a bad input, the request is not processed further. However,

a too large input does not stop the application; it continues running until completion. Therefore, the resilience of this application has been improved by using CHERI-SDRaD.

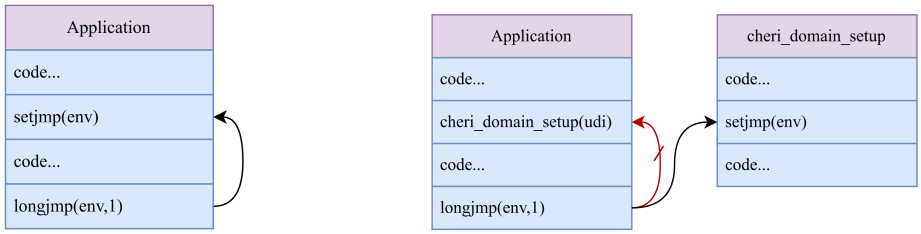
```
eccraus@cheribsd:~/cheri_experiments/morello_handler/bin$ ./cheri_sdrad_overflow
Waiting for the request:
1234
Handling the request
Waiting for the request:
12345
SIGPROT detected
Bad input!
Waiting for the request:
ssssssssssssssssssss
SIGPROT detected
Bad input!
Waiting for the request:
s
Handling the request
Waiting for the request:
sssssssssssssss
SIGPROT detected
Bad input!
```

Figure 6: Handling buffer overflows with CHERI-SDRaD.

Saving the execution context

To efficiently handle abnormal domain exits and safely rewind to the main domain. When establishing a new domain, three pieces of information must be preserved: the new UDI, which identifies the newly created domain; the domain rewind address, which instructs the system on where is the entry point to which execution is rewound if the domain needs to be abnormally exited; and the execution context, encompassing all the elements that are required to allow a program to resume its operation at the specific point and state before the domain was created. This includes the stack pointer, the program counter, the link register, and all general-purpose registers. The UDI and domain rewind address are straightforward to retrieve and store as they can be directly accessed within the code. However, saving the execution context and being able to return to it requires using a special C standard library call, **setjmp**. The setjmp API library provides two functions: `setjmp()`, which saves the execution context, and `longjmp()`, which allows to return to a previously saved execution context, effectively restoring the state to where `setjmp()` was called. The `setjmp()` function takes one argument of type `jmp_buf`, which is a special type provided by the API. When called, `setjmp()` stores the execution context in the `jmp_buf` variable given as an argument. The `longjmp()` function takes two arguments: the first one is a `jmp_buf` variable, which represents the execution context to be restored, and the second one is a strictly positive value which specified the return code code for `setjmp()`.

Normally, `setjmp()` would be used to save the execution context, but with the API adding an intermediate function layer, the execution context cannot be save as usual. The problem is shown in Figure 7.



(a) Normal setjmp. (b) Setjmp inside API.

Figure 7: Problem with setjmp in the API.

Invoking `longjmp()` with a stored context created in a function that has already returned results in undefined behavior. The solution is to devise a trampoline directly in assembly code. This trampoline ensures the context stored in the `jmp_buf` when `cheri_domain_setup` is invoked matches the context of the caller. By using an assembly function and directly manipulating memory, this trampoline tricks the system into believing that the call to `setjmp()` occurred within the same function that initiated the API call. After the context has been saved, the trampoline invokes the code that creates the necessary run-time structures for the new domain, as shown in Listing 4.

```
1  cheri_domain_setup:
2      stp      c29, c30, [csp, #-32]! ①     // store the rewind address on the
       stack
3      str      c0, [csp, #-16]! ②           // store the udi on the stack
4      sub      csp, csp, #512               // store some space for the env
       variable on the stack
5
6      mov      c0, csp ③                    // give the environnement variable
       to setjmp
7      bl       setjmp@PLT
8      cbnz     w0,.ljmp                     // if non 0 go to .ljmp
9      mov      c0, csp ④
10     bl       cheri_domain_init@PLT
11
12     add      csp, csp, #528               // restore the stack pointer
13     ldp      c29, c30, [csp], #32   ⑤      // restore the return address and
        the stack pointer
14     ret      c30   ⑥
15
16
17 .ljmp:
18     bl       cheri_domain_destroy@PLT
19     ldp      c29, c30, [c0]               // load the return address
20     add      csp, csp, #0x230             // restore the stack pointer
21     ret      c30   ⑦
```

Listing 4: Assembly function that stores initialization metadata.

The assembly function, `cheri_domain_setup()`, illustrated in Listing 4 is the one used by the API to initiate a new domain. As mentioned earlier, to create a new domain, three pieces of information(UDI, execution context and return address) need to be saved by employing a trick within the application.

In assembly, a limited number of registers are available for data manipulation. One way to overcome this limitation is by saving register values onto the stack, which is a memory area functioning as a Last In, First Out (LIFO) structure. The stack pointer register (here `csp`) points to the last empty location in this memory area.

When a function is called, the address of the call is automatically stored in the `c30` register. The first step of this trick is to decrement the stack pointer to allocate space for storing the domain rewind address, the UDI of the new domain and the execution context ①. The second line of the assembly code stores the value of `c0` on the stack because when an argument is passed to a function, it is stored in the `c0` register for later use ②.

Note that, Arm Morello architecture [10] defines that if branch with a link instruction generates a sealed capability in c30, to unseal c30 later it must be used in a valid unseal, operation, such as ret in ⑥, ⑦. The sealed c30 and frame pointer in c29 need to be saved and restored using the store capability pair (STP) ① and load capability pair (LDP) ⑤ instructions to avoid invalidating their tag bits.

The next step is to store the current stack pointer into `c0`. Recall that the stack pointer holds the address of the newly allocated `jmp_buf` allocated on the stack. By storing that address into `c0` the address is passed as argument to `setjmp()` that will save the execution context into the allocated space ③. The `setjmp()` function returns 0 if the execution context was saved successfully. A non-zero return value for `setjmp()` indicates it was executed as a result of a `longjmp()` call that restored the previously saved execution context. The latter case corresponds to an abnormal domain exit, in which case the trampoline code destroys the saved information for the faulting domain by calling the internal `cheri_domain_destroy()` functions and returns from the trampoline by restoring the saved return address from `jmp_buf`.

The final part of the trampoline ④ is to store the stack pointer into `c0` so that `cheri_domain_init()` can access it as an argument and store these variables. Listing 5 illustrates this function. The `cheri_domain_init()` function will store the execution context, the UDI, and the rewind address from the stack into variables ①. These variables will eventually be stored in the CHERI-SDRaD Manager if the initialisation is successful. The order of the fields in the `cheri_init_stack_s` struct is important because, since the stack operates on a LIFO basis, the last argument stored in the assembly code will be the first one stored in the `cheri_init_stack_s` and vice-versa.

The next steps are to check if this domain can be created ②, and if so, to verify if it already exists ③. If initialization of the domain is successful and it does not already exist, then the domain is initialized ④. The parent domain that invoked the creation of this new domain is saved ⑤, along with the rewind address ⑥ and the execution context ⑦, stored in the `domain_info` field of the CHERI-SDRaD Manager.

```
struct cheri_init_stack_s{
    jmp_buf env;
    uint64_t udi;
    return_reg_type_s return_address;
};

int cheri_domain_init(void *base_address){
    struct cheri_init_stack_s *cis_ptr;
    cis_ptr = (struct cheri_init_stack_s *)base_address; ①
    long udi = cis_ptr->udi;
    global_manager_s *gm_ptr = &cheri_sdrad_manager;


    if(udi > (NUMBER_MAX_DOMAIN) || udi < 1){ ②

        printf("invalid udi, you should choose one between 1 and %d\n",
    NUMBER_MAX_DOMAIN);
        return UDI_OUT_OF_BOUNDS;
    }

    if(gm_ptr->domain_info[udi].domain_init == INIT){ ③
        printf("This domain is already initialised\n");
        return ALREADY_INITIALIZE;
    }
    gm_ptr->domain_info[udi].domain_init = INIT; ④

    gm_ptr->domain_info[udi].parent_udi = manager.active_domain; ⑤

    gm_ptr->domain_info[udi]. return_address = cis_ptr->return_address; ⑥
    memcpy(gm_ptr->domain_info[udi].env, cis_ptr->env, sizeof(jmp_buf)); ⑦

    return SUCCESSFUL_INITIALIZE;
}
```

Listing 5: cheri_domain_init function.

CHERI protection violation handler

The second step in implementing resilience is to modify the application's behavior to prevent it from crashing. An application crash occurs when a program encounters an error or a set of conditions that it cannot handle, leading to an abrupt termination of its operation. SDRaD works on both Intel and AMD architectures by detecting the SIGSEGV signal that can be attributed to MPK-related access faults, such domain violation, or SIGABRT sent as a result of a failed stack canary check. However, on the Morello board, a CHERI capability fault is reported by the OS to the application via a "protection violation fault" (SIGPROT) signal. Therefore, the CHERI-

SDRaD signal handler had to be adjusted accordingly, as illustrated in Listing 6.

```
__attribute__((constructor)) ①
void cheri_setup_signal_handler()
{
    struct sigaction sa;
    sa.sa_flags =  SA_SIGINFO;
    sa.sa_handler = cheri_signal_handler;
    sigemptyset(&sa.sa_mask);
    if (sigaction(SIGPROT, &sa, NULL) == -1) {
        printf("sigaction");
    }
}

void cheri_signal_handler(int signum)
{
    global_manager_s *gm_ptr = &cheri_sdrad_manager;
    domain_info_s *di_ptr = &(gm_ptr->domain_info[gm_ptr->active_domain]);
    int udi = gm_ptr->active_domain;


    if(signum == 34){
        if(udi != 0){
            printf("SIGPROT detected\n");

            longjmp(di_ptr->env,14);
        }
        else{
            exit();
        }
    }else{
        printf("signum: %d\n",signum);
    }
}
```

Listing 6: Signal handler code.

The **constructor** attribute[1] is a Clang attribute that allows a function to run before the main execution start. This attribute is associated with the`cheri_setup_signal_handler` function ①, ensuring that this function is executed before the main function. The `cheri_setup_signal_handler` will change the behavior of the OS signal delivery delivering mechanism to use a custom handler when delivering `SIGPROT`. The new `SIGPROT` handler, `cheri_signal_handler()`,verifies if a `SIGPROT` signal has indeed been detected. If so, it also checks whether the active domain is not the main one, closing the application if it is. Finally, if all conditions are met, it uses the `longjmp()` function to restore the execution context saved prior to creating this domain, allowing the program to resume from that point.

[1] `https://clang.llvm.org/docs/AttributeReference.html#constructor`

Heap management

Implementing heap management for CHERI-SDRaD allows to maintain separate heaps for different domains and ensures that allocations occur in the appropriate heap. This is important to make sure that allocations that occur in domains that exit abnormally can be freed without leaking memory.

Heap allocator Creating an isolated domain requires generating an isolated heap for each domain. An attempt was made initially to adapt a compartmentalizing allocator from the Cambridge CHERIs project [11] to CHERI-SDRaD. The compartmentalizing allocator was designed for software operating in hybrid-capability mode to compartmentalize its heap into smaller, isolated areas. However, it was found during the adaptation of the allocator for purecap mode that the proof-of-concept compartmentalizing allocator did not support freeing individual allocations, only full compartments. This posed an issue for integration into CHERI-SDRaD since an allocator was needed that could be used as a drop-in replacement for POSIX `malloc()` and `free()` on code paths exiting domains normally. The original SDRaD prototype employs Two-Level Segregated Fit (TLSF) allocator that allows allocations to be directed to distinct "memory pools". For a meaningful comparison between CHERI-SDRaD and the 46-bit x86 SDRaD prototype, it would be advantageous to adapt TLSF as well. However, that meant porting the TLSF allocator to CHERI.

TLSF

The Two-Level Segregated Fit (TLSF) memory allocator is designed for efficiency by reducing fragmentation and optimizing memory utilization. TLSF divides memory into segregated blocks based on their size, establishing separate pools to cater to different ranges of block sizes. This segmentation ensures that memory blocks are allocated with minimal wastage and fragmentation, as blocks of similar sizes are grouped together. TLSF employs a two-level structure, consisting of broad and fine levels of segregation. At the broad level, memory is divided into larger chunks, while at the fine level, these chunks are further subdivided into smaller blocks. This hierarchical organization facilitates efficient searching and allocation of memory blocks, significantly reducing overhead and improving allocation speed. TLSF can split or merge memory blocks as needed to accommodate varying allocation sizes, further minimizing fragmentation and improving memory utilization. Moreover, TLSF is designed with low overhead in mind, both in terms of memory usage and processing time. This makes it particularly suitable for

deployment in resource-constrained environments such as embedded systems and real-time operating systems, where efficient memory utilization is critical for optimal performance.

Implementation SDRaD TLSF is based on an open-source project developed by Matt Conte [3]. The TLSF implementation was ported to CHERI by accommodating the increased size of the in-memory representation of CHERI capabilities compared to the baseline architecture pointers where necessary. For example, the `ALIGN_SIZE_LOG2` variable was changed from 3 to 4 to ensure that the addresses of memory allocations made by TLSF are aligned to 16 bytes instead of 8 bytes. Moreover, the values of `block_header_overhead` and `block_start_offset` need to be updated by doubling `sizeof(size_t)` expressions because it is now using 16-byte capabilities. This is because the CHERI architecture invalidates an assumption made by the original TLSF developer that `size_t` represents the size of a pointer (in memory).

```
1 static block_header_t* offset_to_block(const void* ptr, size_t size) {
2     return tlsf_cast(block_header_t*, tlsf_cast(tlsfptr_t, ptr) + size);
3 }
```

Listing 7: An Example Function from TLSF library: offset_to_block function. Extract from [3].

```
1 static block_header_t* offset_to_block(const void* ptr, size_t size) {
2     return tlsf_cast(block_header_t*, cheri_address_set(ptr, tlsf_cast(
      tlsfptr_t, ptr)+ size));
3 }
```

Listing 8: CHERI Ported offset_to_block function. Adapted from [3].

As illustrated in Listing 7 and 8, to adapt TLSF to CHERI, it is necessary to utilize CHERI functions to modify capabilities. Indeed, CHERI use a single-provenance semantics, i.e, every capability needs to be derived from another one. Within this function, acquiring a new capability from an existing one requires the use of `cheri_address_set` to copy the permissions and bounds of `ptr` to the new capability `(ptr + size)`.

The porting of TLSF to CHERI required the modification of 7 lines out of a total of 840 lines within TLSF, amounting to only 0.83% of the total codebase.

Allocator functions

The next step is to modify the behavior of all the allocation process to manage each memory allocation, and associate a TLSF-pool for each domain to obtain a different heap for each one of them. As illustrated in Listing 2, each

domain possesses its own TLSF structure, representing the heap for that particular domain. The initial step involves defining a function to create these separate heaps, as depicted in Listing 9.

```
1  void cheri_heap_init(){
2
3      size_t  app_heap_size;
4      uintptr_t  app_heap_address;
5      global_manager_s *gm_ptr = &cheri_sdrad_manager;
6      domain_info_s *di_ptr = &(gm_ptr->domain_info[gm_ptr->active_domain]);
7
8      char *pTmp;
9      pTmp = getenv( "APP_HEAP_SIZE");
10
11     if(pTmp != NULL){
12         app_heap_size = atoi(pTmp);
13     }else{
14         app_heap_size = APP_DEFAULT_HEAP_SIZE;
15     }
16
17     app_heap_address = (uintptr_t)mmap(NULL, APP_DEFAULT_HEAP_SIZE,
       PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
18
19     if(app_heap_size <= TLSF_MAX_POOL_SIZE){
20         di_ptr->tlsf = tlsf_create_with_pool((void *)app_heap_address,
       app_heap_size);
21     }else{
22         di_ptr->tlsf = tlsf_create_with_pool((void *)app_heap_address,
       TLSF_MAX_POOL_SIZE);
23         app_heap_size = app_heap_size - TLSF_MAX_POOL_SIZE;
24         app_heap_address =  app_heap_address + TLSF_MAX_POOL_SIZE;
25         while (app_heap_size  > TLSF_MAX_POOL_SIZE)
26         {
27             tlsf_add_pool(di_ptr->tlsf, (void *)app_heap_address,
       TLSF_MAX_POOL_SIZE);
28             app_heap_size = app_heap_size - TLSF_MAX_POOL_SIZE;
29             app_heap_address = app_heap_address + TLSF_MAX_POOL_SIZE;
30         }
31         tlsf_add_pool(di_ptr->tlsf,(void *)app_heap_address, app_heap_size);
32     }
33 }
```

Listing 9: Heap creation function.

The function illustrated in Listing 9 allows the user to create a dedicated heap for the active domain in the application. First, the heap area is allocated with `mmap()` on line 18. After that, to use the TLSF memory allocator, this space needs to be associated with a pool. However, a single call to `tlsf_create_with_pool()` or `tlsf_add_pool()` cannot use more than `TLSF_MAX_POOL_SIZE` for the size of the heap. To address this limitation, a while loop is introduced to add pools until the total size of the heap is reached.

The `cheri_heap_init()` function is not intended to be called directly. Instead, it is invoked by the `malloc()`-family of allocation functions. This approach optimizes memory allocation by creating a dedicated heap only when necessary.

To achieve this, the classic `malloc()`-family of allocation functions were overridden, as illustrated in Listing 10 with the new version of the `malloc()` function using the TLSF memory allocator.

```
void *malloc(size_t size){
    global_manager_s *gm_ptr = &cheri_sdrad_manager;
    domain_info_s *di_ptr = &(gm_ptr->domain_info[gm_ptr->active_domain]);


    if (di_ptr->heap_init != INIT) {
        cheri_heap_init();
        di_ptr->heap_init = INIT;
    }

    void *ptr;
    size_t rounded_len = __builtin_cheri_round_representable_length(size);


    TLSF_MUTEX_LOCK();
    ptr = tlsf_malloc(di_ptr->tlsf, rounded_len);
    ptr = __builtin_cheri_bounds_set(ptr, rounded_len);
    TLSF_MUTEX_UNLOCK();

    return ptr;
}
```

Listing 10: Adapted malloc function.

Listing 10 shows that in this new version of `malloc()`, the first step is to check if the heap of the active domain is initialized. If it is not, the `cheri_heap_init()` function will be called and the function continues. If it is already initialized, the function just continues. The next step is to round the size of the capability to ensure that it is a multiple of 16. Afterward, the TLSF malloc allocator is called instead of the classical `malloc()` allocator. Finally, the bounds are set to correspond to the size of the capability. With the same idea, all the allocation functions were overridden to check if the heap is initialized, to use the TLSF version of the allocation function, and to correctly set the bounds according to the CHERI capabilities.

Not only the allocation function needed to be modified, the `free()` function needed also to be modified. These modifications are illustrated in Listing 11.

```
void free(void *ptr)
{
    global_manager_s *gm_ptr = &cheri_sdrad_manager;
    domain_info_s *di_ptr = &(gm_ptr->domain_info[gm_ptr->active_domain]);

    TLSF_MUTEX_LOCK();
    ptr = __builtin_cheri_address_set(di_ptr->tlsf,
    __builtin_cheri_address_get(ptr));
    tlsf_free(di_ptr->tlsf, ptr);
    TLSF_MUTEX_UNLOCK();
}
```

Listing 11: Modified version of free.

All the capabilities allocated with TLSF are associated with a header. This header precedes the capability in memory, specifically before the lower bounds. The approach illustrated in Listing 11 involves creating a new capability derived from the entire heap but referencing the address of the capability intended for deallocation. This allows access to the capability's metadata in its header. To achieve this, `cheri_address_get()` is called to obtain the bounds and permissions of the entire heap using tlsf control structure. Subsequently, `cheri_address_set()` is invoked to generate a new capability using the heap's bounds and permissions, along with the address of the capability planned for deallocation.

5 Evaluation

One of the application domains that benefits from increased resilience is service-oriented applications. Servers must have the capability to operate within isolated domains, allowing for the independent shutdown and restart of each domain. Additionally, maintaining a high-level of performance impact is critical for servers. Therefore, the performance of SDRaD for CHERI was evaluated on server software to determine if it is a viable approach to enhance security and resilience.

5.1 Case study: Nginx

Nginx [12] is a versatile, open-source software that can be used as a web server, reverse proxy, load balancer, and HTTP cache. Originally developed by Igor Sysoev, Nginx is knowed for high performance, stability, and low resource consumption, making it a popular choice for a wide range of web server and proxy server needs. It is a multiprocessing application featuring a master process and one or more worker processes. The master process oversees the worker processes, which manage client HTTP requests across multiple connections simultaneously. In case of a malicious client request causing memory corruption, a worker process might crash. But fear not! The master process promptly restarts it! Nevertheless, this does mean that any ongoing connections handled by that specific worker are lost in the process. Moreover, Nginx has recently been ported to CHERI, making it an ideal fit as a study case for evaluation. The original SDRaD project [2] also used Nginx as a case study.

5.2 Performance evaluation

Given its complexity and exposure to untrusted inputs, the HTTP parser stands out as a potential vulnerability within Nginx. The solution involves parsing each client HTTP request within a nested domain. To achieve this, the `ngx_http_parse_header_line` and `ngx_http_parse_request_line` functions are wrapped using the API, as depicted in Listing 12, specifically focusing on the `ngx_http_parse_request_line` function. These functions will be executed instead of the normal ones to proceed with the encapsulation.

```
ngx_int_t __real_ngx_http_parse_request_line(ngx_http_request_t *r,
    ngx_buf_t *b);
ngx_int_t __wrap_ngx_http_parse_request_line(ngx_http_request_t *r,
    ngx_buf_t *b)
{
    ngx_int_t rc;

    cheri_domain_enter(NGX_NESTED_DOMAIN);
    rc = __real_ngx_http_parse_request_line(r, b);
    cheri_domain_exit();

    return rc;
}
```

Listing 12: The wrapped ngx_http_parse_request_line.

Consequently, in the event of detecting memory corruption within the parser, an abnormal domain exit is triggered. This allows us to securely discard the content associated with the nested domain and revert back to the main domain without necessitating a restart of the worker process. Although the connection to the malicious client is closed, all other connections remain unaffected by this operation.

The performance evaluation of Nginx was conducted in four different configurations: baseline (unmodified) Nginx, Nginx ported to CHERI in purecap mode, purecap Nginx with the TLSF allocator and finally purecap Nginx compartmentalized using CHERI-SDRaD as described in section 5.1. Throughput measurements were obtained using the WRK [13] benchmarking tool. WRK enables us to send a high volume of requests with various configurations to assess the server's performance. This tool allows us to specify the number of simultaneous requests, the number of local machine cores to utilize, and for how long the test should run. It was evaluated with 32 cores, 128 simultaneous requests for 1 minute. The benchmark uses WRK to request files of different sizes (0kB, 1kB, 4kB, 16kB), and to obtain the average result, each benchmark for a specific file size is conducted 10 times to obtain the average result.

The experiment employed two distinct machines: for server deployment, the Arm Morello Board with 4 cores at 2.5GHz and 16GB of RAM running CheriBSD with a FreeBSD kernel version 14.0-CURRENT was utilized. Meanwhile, the benchmark was executed on a separate machine equipped with a 32-core Intel(R) Xeon(R) CPU E5-2658 0 clocked at 2.1GHz, along with 66GB of RAM, operating on Ubuntu 22.04.4 with Linux kernel 5.15.0. Nginx 1.24.0 release and 1.24.0-with-cheri-fixes release are used, and compiled with -O2 optimization, also to enable Nginx for purecap mode is compiled with `-target aarch64-unknown-freebsd -march=morello -mabi=purecap -Xclang -morello-vararg=new`.

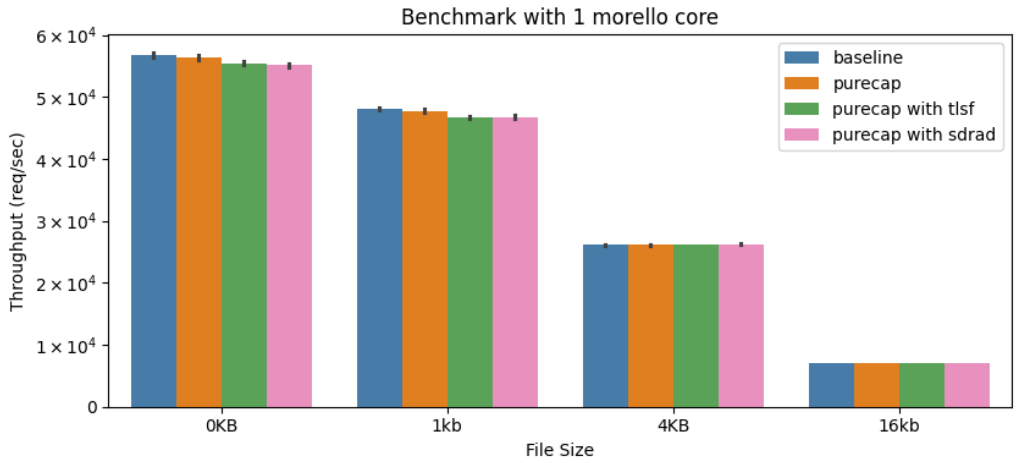


Figure 8: Nginx Benchmark with 1 core.

Figure 8 summarizes the results of experiments conducted on the Nginx server across four distinct configurations. Initially, the Nginx server was tested without any additional modifications (baseline), establishing a reference point to observe the impact of different layers of security. The second configuration involved Nginx with the CHERI modifications running in purecap mode (purecap), serving as the baseline to assess the overhead introduced by the solution. Since CHERI-SDRaD requires a specialized allocator, the performance impact of the TLSF allocator in CHERI purecap mode was evaluated without the CHERI-SDRaD library (purecap with tlsf). Finally, a version of Nginx in CHERI purecap mode, compartmentalized with CHERI-SDRaD using the TLSF allocator (purecap with sdrad), was evaluated. The result of the benchmark using the CHERI-port of Nginx using 0kB files, designed to ask the server to send a file of 0kB, showed a 0,78% degradation of throughput in purecap mode, indicating the performance degradation of CHERI being negligible compared to unmodified software on the Morello board. Introducing the TLSF allocator to the CHERI-port of Nginx resulted in a slightly higher throughput degradation of 1.73%, reflecting the increased computational demands. Integrating CHERI-SDRaD into CHERI Nginx led to a 2.22% throughput degradation, balancing security enhancements with computational efficiency. These results demonstrate the viability of CHERI integration and highlight the trade-offs between security and performance in server configurations.

Figure 8 illustrates a benchmark conducted using a single core of the Morello Board CPU. Additionally, an attempt was made to utilize all four cores of the CPU, as depicted in Figure 9. However, the results were not

reliable due to the inability to fully saturate the cores of the Morello Board.

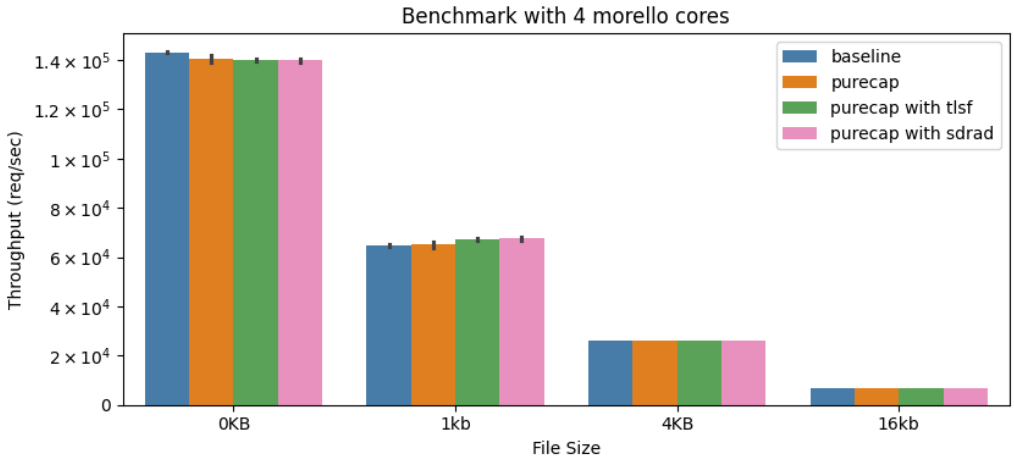


Figure 9: Nginx Benchmark with 4 cores.

5.3 Comparison to SDRaD on 64-bit x86

By adapting SDRaD for the Arm Morello Board, several improvements were noticed:

1. During the benchmarking of Nginx with the Intel-based version of SDRaD, the measured overhead was approximately 5.70% for 1 worker and 0kb case. Although a direct comparison with the 64-bit x86 version of SDRaD is not possible, a smaller throughput degradation (2.22%) was measured for CHERI-SDRaD on the Morello board. Additionally, 0.78% of the throughput degradation was attributed to CHERI purecap mode, and approximately 0.95% to TLSF, suggesting that the relative impact of CHERI-SDRaD on the Morello board is smaller than that of SDRaD evaluated on an Intel-based architecture. Our conclusion is thus that adapting SDRaD to a CHERI architecture resulted in a more lightweight version with less performance impact compared to the baseline 64-bit x64 version architecture.

2. In the 64-bit x64 version SDRaD a notable drawback of employing Memory Protection Keys (MPK) lies in its reliance on tagging memory pages with protection keys using the last 4 unused bits. Consequently, this method is restricted to 16 distinct keys for tagging memory pages. However, when basing compartmentalization on the run-time memory

protection capabilities of CHERI, there is no longer a hardware limitation on the number of domains that can be supported. Although our software implementation limits the number of domains to the number of preallocated domain information storage slots (see section 4.2) this software limitation could be lifted by employing an alternate design that dynamically scales the number of domain information storage slots as needed.

6 Conclusion

This thesis describes the design and implementation CHERI-SDRaD prototype adaption of secure rewind and discard of isolated in-process domains that leverages the memory-safety properties inherent to the CHERI. CHERI-SDRaD results in a design with reduced performance degradation (2.2% in Nginx benchmarks) compared to earlier results obtained with the original SDRaD prototype on an Intel-based architecture. The adaption to CHERI additionally allows limitations inherent to the earlier MPK-based approach to be resolved.

The CHERI-SDRaD C library along with the CHERI-ported version of TLSF memory allocator will be made open source at `https://github.com/secure-rewind-and-discard`.

The library developed in this work could also be evaluated for other use-cases, such as Memcached or lighttpd. Also, this work can be extended to provide some level of automation similar to SDRaD-FFI [14].

7 References

[1] R. N. M. Watson, S. W. Moore, P. Sewell, and P. G. Neumann, "An introduction to cheri," Computer Laboratory, Tech. Rep. UCAM-CL-TR-941, September 2019. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-941.pdf

[2] M. Gülmez, T. Nyman, C. Baumann, and J. T. Mühlberg, "Rewind & discard: Improving software resilience using isolated domains," in *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2023, pp. 402–416.

[3] M. Conte, "tlsf." [Online]. Available: https://github.com/mattconte/tlsf

[4] CISA, NSA, FBI, A. ACSC, CCCS, NCSC-UK, NCSC-NZ, and CERT-NZ, "The case for memory safe roadmaps," 2023. [Online]. Available: https://media.defense.gov/2023/Dec/06/2003352724/-1/-1/0/THE-CASE-FOR-MEMORY-SAFE-ROADMAPS-TLP-CLEAR.PDF

[5] B. Bierbaumer, J. Kirsch, T. Kittel, A. Francillon, and A. Zarras, "Smashing the stack protector for fun and profit," in *ICT Systems Security and Privacy Protection*, L. J. Janczewski and M. Kutyłowski, Eds. Cham: Springer International Publishing, 2018, pp. 293–306.

[6] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI capability model: Revisiting RISC in an age of risk," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, Jun. 2014, pp. 457–468. [Online]. Available: https://ieeexplore.ieee.org/document/6853201

[7] T. Nyman, "Toward Hardware-assisted Run-time Protection," Doctoral thesis, School of Science, 2020. [Online]. Available: http://urn.fi/URN:ISBN:978-952-64-0065-5

[8] R. N. M. Watson, G. Barnes, J. Clarke, R. Grisenthwaite, P. Sewell, S. W. Moore, and J. Woodruff, "Arm Morello Programme: Architectural security goals and known limitations," Computer Laboratory, University of Cambridge, 15 JJ Thomson Avenue Cambridge CB3 0FD United Kingdom, Technical Report UCAM-CL-TR-982, Jul. 2023. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-982.pdf

[9] I. Corporation, "Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide," https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[10] Arm Ltd, *Arm® Architecture Reference Manual Supplement Morello for A-profile Architecture*, Arm Ltd, Cambridge, UK, 2022, pROTO_REL 04. [Online]. Available: https://developer.arm.com/documentation/ddi0606/latest/

[11] CapableVMs, "Compartmentalising allocator." [Online]. Available: https://github.com/capablevms/cheri-examples/tree/master/example_allocators/compartment_alloc

[12] I. Sysoev, "Nginx." [Online]. Available: https://nginx.org/en/

[13] W. Glozer, "wrk - a http benchmarking tool." [Online]. Available: https://github.com/wg/wrk

[14] M. Gulmez, T. Nyman, C. Baumann, and J. Muhlberg, "Friend or foe inside? exploring in-process isolation to maintain memory safety for unsafe rust," in *2023 IEEE Secure Development Conference (SecDev)*. Los Alamitos, CA, USA: IEEE Computer Society, oct 2023, pp. 54–66. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SecDev56634.2023.00020

1 Appendices

1.1 Appendix A: Project Timeline.

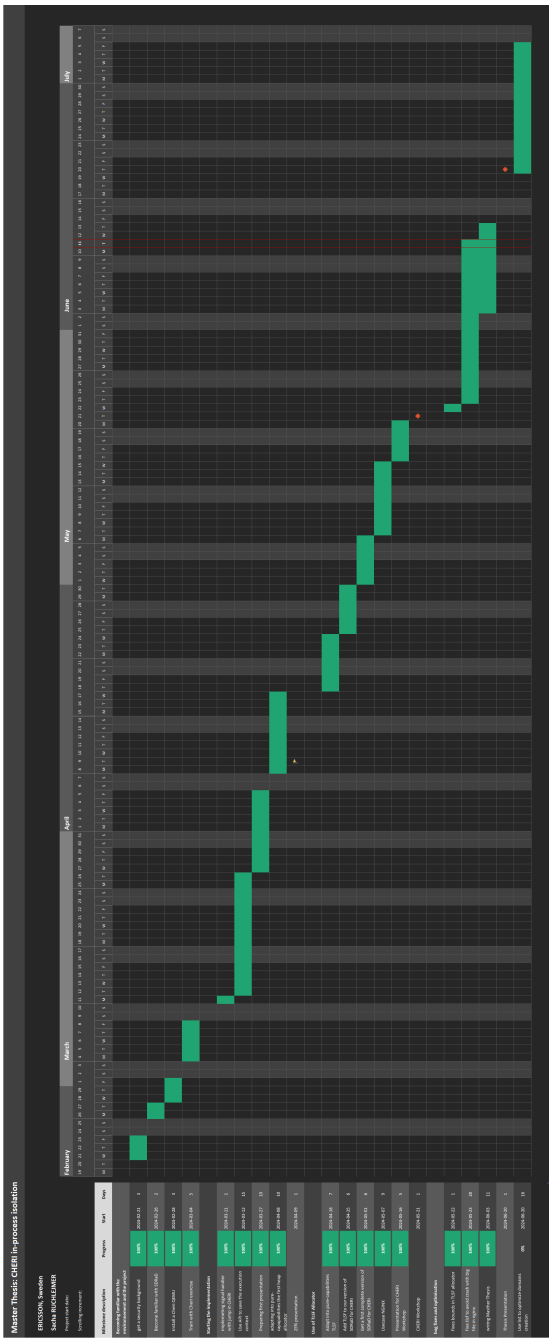


Figure 10: Project Timeline.

Abstract

Memory-unsafe programming languages such as C and C++ are the preferred languages for systems programming, embedded systems, and performance-critical applications. The widespread use of these languages makes the risk of memory-related attacks very high. There are well-known detection mechanisms, but they do not address software resilience. An earlier approach proposes the *Secure Domain Rewind and Discard (SDRaD)* of isolated domains as a method to enhance the resilience of software targeted by runtime attacks on x86 architecture, based on hardware-enforced *Memory Protection Key (MPK)*. In this work, SDRaD has been adapted to work with the *Capability Hardware Enhanced RISC Instructions (CHERI)* architecture to be more lightweight and performant. The results obtained in this thesis show that CHERI-SDRaD, the prototype adaption that leverages the memory-safety properties inherent to the CHERI architecture, results in a solution with less performance degradation (2.2% in Nginx benchmarks) compared to earlier results obtained with the original SDRaD prototype on an Intel-based architecture. The adaption to CHERI additionally allowed limitations inherent to the MPK-based approach to be resolved.

Résumé

Les langages de programmation non sécurisés en mémoire, tels que C et C++, sont les langages privilégiés pour la programmation système, les systèmes embarqués et les applications nécessitant de hautes performances. L'utilisation répandue de ces langages rend le risque d'attaques liées à la mémoire très élevé. Il existe des mécanismes de détection bien connus, mais ils n'abordent pas la résilience logicielle. Une approche antérieure propose *Secure Domain Rewind and Discard (SDRaD)* des domaines isolés comme méthode pour améliorer la résilience des logiciels ciblés par des attaques à l'exécution sur l'architecture x86, basée sur la technologie matérielle *Memory Protection Key (MPK)*. Dans ce travail, SDRaD a été adapté pour fonctionner avec l'architecture *Capability Hardware Enhanced RISC Instructions (CHERI)* afin d'être plus léger et performant. Les résultats obtenus dans cette thèse montrent que CHERI-SDRaD, le prototype d'adaptation qui exploite les propriétés de sécurité mémoire inhérentes à l'architecture CHERI, offre une solution avec moins de dégradation des performances (2,2% dans les benchmarks Nginx) par rapport aux résultats antérieurs obtenus avec le prototype original SDRaD sur une architecture basée sur Intel. L'adaptation à CHERI a également permis de résoudre les limitations inhérentes à l'approche basée sur MPK.