KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Shining Light on Critical Gaps in Memory-Safety:

From Programming Language to Hardware

Merve Gülmez

Supervisors:
Prof. dr. ir. W. Joosen
Prof. dr. J.T. Mühlberg
(Université Libre de Bruxelles)
Dr. ir. C. Baumann
(Ericsson AB)

Dissertation presented in partial fulfillment
of the requirements for the degree of
Doctor of Engineering Science (PhD):
Computer Science

December 2025

SHINING LIGHT ON CRITICAL GAPS IN MEMORY-SAFETY:

FROM PROGRAMMING LANGUAGE TO HARDWARE

Merve GÜLMEZ

Supervisors:
Prof. dr. ir. W. Joosen
Prof. dr. J.T. Mühlberg
(Université Libre de Bruxelles)
Dr. ir. C. Baumann
(Ericsson AB)

Members of the
Examination Committee:
Prof. dr. ir. J. De Schutter, chair
Dr. E. Truyen
Prof. dr. ir. S. Volckaert
Prof. dr. P. Valcke
Prof. dr. ir. F. Piessens
(KU Leuven)
Prof. dr. A. Paolillo
(Vrije Universiteit Brussel)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

December 2025

Uitgegeven in eigen beheer, Merve Gülmez, Leuven (Belgium)

Preface

This thesis has been a challenging journey for me, as I started from scratch in computer security. That is why I am grateful to those around me for patiently teaching me: Christoph and JT, for providing a security background and for kick-starting my PhD; Thomas, for introducing me to memory-safety hardening techniques and helping me structure the thesis idea; Håkan and Hossam, for our joint work on hardware security; and Hans from SSG and Adriaan, for contributing to my compiler knowledge.

I would also like to thank: Gianluca, for following my progress on this PhD and for your friendship; the CHERI team, especially Hesham, John, Peter, Robert, and Jessica, for all kinds of technical discussions; Alex, for enabling me to explore a new area; Asokan, for being an example with the effort you put in; Jialun, for nice collaboration and building on top of work this thesis started; Anjo, for your valuable feedback; and my interns Sacha, Hugo, and Ruben for making this journey more enjoyable.

To my thesis committee members, Peggy, Eddy, Antonio, Frank, Wouter, and Stijn: thank you for taking the time to read my thesis and for your constructive feedback.

Thomas, I have really enjoyed pursuing all the papers and ideas together, and I am happy that we still have a lot to do. Thank you for believing in me from the beginning and for being with me throughout this thesis. Your encouragement has made this thesis possible and has made me more confident.

Christoph, you spent so much time being the best company during the lockdown and patiently helping me improve my language skills. Thank you for all your time, for our many discussions, and for supporting me.

JT, thank you for managing my cat photos, for our enjoyable discussions, and for coining a term for my language and understanding it. You always managed to support me, no matter what difficulties I have, and you made this journey more fun.

To the 5GhOSTS team: Eva, Wouter, Eddy, Christoph, JT, and Peggy; thank you for making this project possible. I am also grateful to the European Union for financially supporting this thesis. To my 5GhOSTS buddies, Gerald, Mykta, and Gianluca, thank

for the time we spent together.

To my Ericsson colleagues: Ulf, thank you for giving me every opportunity I needed. Mohsin, Vlasios, Erik, Eva, Karl, Göran, Francesca, John, Patrik, Niklas, Kim, Christian, Santeri, İlhan, Sini, and Peter, thank you for always being supportive and for all the fika we had together.

To the Distrinet TEE group: thank you for hosting me during my brief visit. Even though, due to the circumstances, I did not have as much time to spend with you as I would have liked, I am proud of this team and happy to read all your exciting papers.

To my former colleagues from NETAS, especially R&D manager Ersin Öztürk: thank you for believing in me at the beginning of my career and for supporting me throughout my master's degree, which led me to pursue my PhD.

To my bachelor's degree teacher and friends: thanks a lot for supporting us during difficult times at school and in periods of political turmoil that burdened us.

To my friend Prajwol: thank you for supporting me through many challenging times and for standing by me throughout this thesis. You, Vani, and Sarita mean a lot to me; Ahmet, thanks for encouraging me to go abroad and welcoming me to Stockholm five years ago; I hope you are proud that I am now graduating. Fatih, your friendship has always been special to me; thank you for still listening my ideas. Rabia, I have no words to express how much your/your family's support, mean to me. I hope that one day, hand in hand with your brother, we will look at the sky and celebrate justice for all people who are suffering now. Emine, Gülsüm, Daniyal, Aydın, Selin, Zeynep, Tuğçe, Süveyda, Betül, Hasan, Nurefsan, and many others who have touched my life: thank you for all your support, the endless talks, and our board game nights!

Anne, senin dualarının hep yanımda olduğunu hissettim. Çok teşekkür ederim. To my little sister, I am happy for every moment we spend together and for the chance to witness your growth. Thank you for all the figures you drew for PhD journey. To my family-in-law, thank you for your endless support and care. Your kindness and warmth have made me feel truly welcomed and loved.

To my cat, Nisan: you are the best company I can imagine; your connection melts my heart. Thank you for all the cuddles and love you give me every day.

To those who have given me unconditional support: you stayed with me through the most difficult moments of this thesis and celebrated the happiest ones, thanks for taking such good care of me. I am grateful beyond words for having you in my life.

To all those oppressed around the world who have lost their freedom or their lives, who have fled their countries, or who continue to struggle to survive: I wrote this thesis with a heavy heart, thinking of you. This thesis is dedicated to all of you.

Merve

Popularized Abstract

A significant challenge for the development of software that closely interacts with computer hardware is ensuring that every access to computer memory is valid, and that all stored data in memory remains correct. Software that correctly operates on memory is called "memory safe". However, developers might make mistakes during programming that cause software to use memory incorrectly. For example, a program might read or write data outside the memory area that was reserved for it. Such mistakes, called "memory-safety violations", can cause the software application to crash or, in some cases, let hackers break into the computer system. This thesis addresses critical gaps in existing state-of-the-art solutions for memory safety.

One common practice is to detect memory-safety violations after they occur, for example, with guard variables that check whether memory adjacent to the reserved area has been modified. However, detecting memory-safety violations just makes them less useful for hackers. Only detecting memory-safety violations does not allow the application to continue its job since data might have been (accidentally or intentionally) overwritten and lost, or the wrong data may have made its way throughout the application and corrupted its integrity. To overcome the challenge of detection-based approaches lacking resilience, this thesis proposes *secure rewind and discard*, an approach that allows software to recover from memory-safety violations and continue its job without crashing. The rewind and discard approach requires dividing the program into smaller parts, called *domains*, which limit the effects of memory errors and can be discarded without crashing the application if individual domains are corrupted.

Another approach for preventing memory-safety errors is the use of memory-safe programming languages. These languages make programmers follow stricter rules regarding how memory must be managed, manage memory on their behalf, and automatically add run-time checks to detect memory errors. This thesis examines Rust, a modern language with memory safety. However, Rust code can still call into older, less safe code. We adapt secure rewind and discard to Rust, making it easier for developers to safeguard their Rust applications against errors in such older code.

Finally, memory safety can be enforced by computer hardware that validates every memory reference against the memory area that is reserved for that reference. This thesis extends such a hardware architecture, CHERI, which prevents many memory errors but cannot avoid *uninitialized memory* issues, cases where memory content is undefined because the memory is read before being written. We extend CHERI to allow it to track which memory addresses have been written at a particular memory reference, before it allows memory to be read from there.

Through these contributions, this thesis furthers the pursuit of comprehensive memory safety solutions by shining light on previously under-represented challenges: *improving software resilience and availability*, and *preventing uninitialized memory access*.

Gepopulariseerde Samenvatting

Een belangrijke uitdaging bij de ontwikkeling van software die nauw samenwerkt met computerhardware, is ervoor te zorgen dat elke toegang tot het computergeheugen geldig is en dat alle gegevens die in het geheugen zijn opgeslagen correct blijven. Software die correct werkt met het geheugen wordt “geheugenveilig” genoemd. Ontwikkelaars kunnen echter tijdens het programmeren fouten maken waardoor software het geheugen onjuist gebruikt. Een programma kan bijvoorbeeld gegevens lezen of schrijven buiten het geheugengebied dat ervoor was gereserveerd. Dergelijke fouten, “geheugenveiligheidsinbreuken” genoemd, kunnen ervoor zorgen dat de softwaretoepassing crasht of, in sommige gevallen, dat hackers in het computersysteem kunnen inbreken. Dit proefschrift behandelt gaten in bestaande state-of-the-art oplossingen voor geheugenveiligheid.

Een veelgebruikte methode is om schendingen van de geheugenveiligheid te detecteren nadat ze zich hebben voorgedaan, bijvoorbeeld met beveiligingsvariabelen die controleren of het geheugen naast het gereserveerde gebied is gewijzigd. Het detecteren van schendingen van de geheugenveiligheid maakt ze echter alleen maar minder bruikbaar voor hackers. Alleen het detecteren van schendingen van de geheugenveiligheid maakt het niet mogelijk dat de applicatie zijn werk voortzet, aangezien gegevens (per ongeluk of opzettelijk) overschreven en verloren kunnen zijn gegaan, of omdat verkeerde gegevens zich door de applicatie kunnen hebben verspreid en de integriteit ervan kunnen hebben aangetast. Om het probleem van detectiegebaseerde benaderingen die niet resiliënt genoeg zijn te overwinnen, stelt dit proefschrift *secure rewind and discard* voor, een benadering waarmee software kan herstellen van schendingen van de geheugenveiligheid en zijn werk kan voortzetten zonder te crashen. De rewind and discard-benadering vereist dat het programma wordt opgedeeld in kleinere delen, *domeinen* genaamd, die de effecten van geheugenfouten beperken en kunnen worden verwijderd zonder dat de toepassing crasht als afzonderlijke domeinen beschadigd zijn.

Een andere manier om geheugenveiligheidsfouten te voorkomen, is het gebruik van geheugenveilige programmeertalen. Deze talen zorgen ervoor dat programmeurs strengere regels moeten volgen voor het beheer van het geheugen, beheren het geheugen voor hen en voegen automatisch runtime-controles toe om geheugenfouten op te sporen. Deze scriptie onderzoekt Rust, een moderne taal met geheugenveiligheid. Rust staat ontwikkelaars echter nog steeds toe om oudere onveilige code aan te roepen. We passen secure rewind and discard toe op Rust, waardoor het voor ontwikkelaars gemakkelijker wordt om hun Rust-toepassingen te beveiligen tegen fouten in dergelijke oudere onveilige code.

Ten slotte kan geheugenveiligheid worden afgedwongen door computerhardware die elke geheugenverwijzing valideert ten opzichte van het geheugengebied dat voor die verwijzing is gereserveerd. Dit proefschrift breidt een dergelijke hardwarearchitectuur, CHERI, uit, die veel geheugenfouten voorkomt. CHERI kan echter niet voorkomen dat er gevallen zijn waarin de inhoud van het geheugen ongedefinieerd is omdat het geheugen wordt gelezen voordat het wordt geschreven, ook wel bekend als *niet-geïnitialiseerd geheugen* problemen. We breiden CHERI uit zodat het bijhoudt naar welke geheugenlocaties is geschreven en daarna alleen lezen vanaf die locaties toestaat.

Door middel van deze bijdragen bevordert dit proefschrift het streven naar uitgebreide oplossingen voor geheugenveiligheid door aandacht te besteden aan uitdagingen die tot nu toe onderbelicht zijn gebleven: *het verbeteren van de resiliëntie en beschikbaarheid van software* en *het voorkomen van niet-geïnitialiseerde geheugentoegang*.

Abstract

Memory safety refers to a program's property of ensuring that memory is accessed only in valid and intended ways. Memory-safety guarantees can be reinforced through programming languages with built-in safety features, such as garbage collection, compile- and run-time checks, or through hardware-based solutions like capability architectures. This thesis focuses on critical gaps in the current state of the art: the lack of fault tolerance of software-based mitigations for C and C++, limits of the memory-safety properties in Rust, and initialization-time safety in hardware capability architectures, such as CHERI.

C and C++ are still the preferred languages for system programming, embedded systems, and various critical applications due to their performance. However, these languages lack built-in memory-safety properties. While several well-known defense techniques can mitigate common faults and memory safety vulnerabilities in software, many do not address the challenge of software resilience and availability—specifically, whether a system can continue to function and remain responsive under attack or when subjected to malicious inputs. As a solution, we propose *secure rewind and discard of isolated domains* as an efficient and secure method of improving the resilience of software that is targeted by run-time attacks. We show the practicability of our methodology by realizing a software library for Secure Domain Rewind and Discard (SDRaD) and demonstrate how SDRaD can be applied to real-world software.

Rust has performance characteristics close to traditional system programming languages such as C and C++ but, unlike these languages, Rust has memory safety guarantees enforced by compile-time analysis. However, in order to interact with hardware or call into non-Rust libraries, Rust provides *unsafe* language features that shift responsibility for ensuring memory safety to the developer. Failing to do so may lead to memory-safety violations in Rust code, which can violate the safety of the entire application. To shield safe program sections from safety violations that may happen through unsafe language features, we adapt SDRaD to protect Rust code. To be practical, security features such as SDRaD must be easy for developers to adopt. We design a Rust-native application programming interface for SDRaD that

leverages Rust's powerful metaprogramming features to enable easy sandboxing of unsafe interfaces.

Up to 10% of memory-safety vulnerabilities in languages like C and C++ stem from uninitialized variables. Capability-based addressing, such as CHERI, mitigates many memory defects, including spatial and temporal safety violations at an architectural level. CHERI, however, does not handle undefined behavior from uninitialized variables. We extend the CHERI capability model to include "*conditional capabilities*", enabling memory-access policies based on prior operations. This allows enforcement of policies that satisfy memory-safety objectives such as "*no reads to memory without at least one prior write*".

Through these contributions, this thesis furthers the pursuit of comprehensive memory safety solutions by shining light on previously under-represented challenges: *improving software resilience* and *availability* and *preventing uninitialized memory access*.

As complementary contributions, this thesis presents an efficient and comprehensive system call interposition mechanism, and provides compiler-assisted automated compartmentalization for Rust. In addition, it evaluates different memory-safety-defense techniques, such as stack canaries and shadow stacks, in terms of their effectiveness and performance. Orthogonal to these works, it proposes an extension to CHERI for enforcing data oblivious computation to harden software against timing side channels. Finally, it discusses environmental sustainability considerations related to SDRaD.

Beknopte samenvatting

Geheugenveiligheid verwijst naar de eigenschap van een programma dat het alleen op een geldige en beoogde manier omgaat met geheugen. Geheugenveiligheid kan worden versterkt door programmeertalen met ingebouwde veiligheidsfuncties, zoals *garbage collection*, toegangscontroles tijdens het compileren en uitvoeren, of door hardwaregebaseerde oplossingen zoals capability-architecturen. Dit proefschrift richt zich op de hiaten in de huidige stand van zaken: het gebrek aan fouttolerantie van softwaregebaseerde mitigaties voor C en C++, de beperkingen van de geheugenveiligheidseigenschappen in Rust en de initialisatiezekerheid in hardwarecapaciteitsarchitecturen, zoals CHERI.

C en C++ zijn vanwege hun prestaties nog steeds de voorkeurstalen voor systeemprogrammering, *embedded* systemen en diverse kritieke toepassingen. Deze talen missen echter ingebouwde geheugenveiligheidseigenschappen. Hoewel verschillende bekende verdedigingstechnieken veelvoorkomende fouten en kwetsbaarheden in de geheugenveiligheid van software kunnen beperken, bieden veel daarvan geen oplossing voor de uitdaging van softwareresiliëntie en -beschikbaarheid, met name de vraag of een systeem kan blijven functioneren en responsief kan blijven bij aanvallen of wanneer het wordt blootgesteld aan kwaadaardige invoer. Als oplossing stellen we *veilig terugspoelen en verwijderen van geïsoleerde domeinen* voor als een efficiënte en veilige methode om de resiliëntie van software die het doelwit is van runtime-aanvallen te verbeteren. We tonen de bruikbaarheid van onze methodologie aan door een softwarebibliotheek voor Secure Domain Rewind and Discard (SDRaD) te construeren en we laten zien hoe SDRaD kan worden toegepast op echte software.

Rust heeft prestatiekenmerken die dicht bij traditionele systeemprogrammeertalen zoals C en C++ liggen, maar in tegenstelling tot deze talen biedt Rust garanties voor geheugenveiligheid die worden afgedwongen door analyse tijdens het compileren. Om echter te kunnen communiceren met hardware of om niet-Rust-bibliotheken aan te roepen, biedt Rust *unsafe* taalfuncties die de verantwoordelijkheid voor het waarborgen van geheugenveiligheid verschuiven naar de ontwikkelaar. Als dit niet gebeurt, kan dit leiden tot schendingen van de geheugenveiligheid in Rust code, wat de

veiligheid van de hele applicatie in gevaar kan brengen. Om veilige programmasecties te beschermen tegen veiligheidsschendingen die kunnen optreden door onveilige taalfuncties, passen we SDRaD aan om Rust code te beschermen. Om praktisch te zijn, moeten beveiligingsfuncties zoals SDRaD gemakkelijk door ontwikkelaars kunnen worden toegepast. We ontwerpen een *Rust-native* applicatie-programmeerinterface voor SDRaD die gebruikmaakt van de krachtige meta-programmeerfuncties van Rust om eenvoudige *sandboxing* van onveilige interfaces mogelijk te maken.

Tot 10% van de kwetsbaarheden op het gebied van geheugenveiligheid in talen als C en C++ zijn het gevolg van niet-geïnitialiseerde variabelen. Geheugenadressering op basis van *capabilities*, zoals CHERI, vermindert veel geheugendefecten, waaronder ruimtelijke en temporele veiligheidsschendingen op architecturaal niveau. CHERI kan echter geen ongedefinieerd gedrag van niet-geïnitialiseerde variabelen afhandelen. We breiden het CHERI *capability*-model uit met "Conditionele *Capabilities*", waardoor geheugentoegangsbeleid op basis van eerdere operaties mogelijk wordt. Hierdoor kan beleid worden afgedwongen dat voldoet aan doelstellingen op het gebied van geheugenveiligheid, zoals "geen leesbewerkingen op deze geheugenlocatie zonder ten minste één voorgaande schrijfbewerking".

Via deze bijdragen bevordert dit proefschrift het streven naar uitgebreide oplossingen voor geheugenveiligheid door licht te werpen op uitdagingen die voorheen onderbelicht bleven: *het verbeteren van de resiliëntie en beschikbaarheid van software* en *het voorkomen van niet-geïnitialiseerde geheugentoegang*.

Als aanvullende bijdragen presenteert dit proefschrift een efficiënt en uitgebreid mechanisme voor het onderscheppen van systeemaanroepen en beschrijft het geautomatiseerde, compiler-ondersteunde compartimentalisatie voor Rust programma's. Daarnaast worden verschillende technieken voor geheugenveiligheid, zoals *stack canaries* en *shadow stacks*, geëvalueerd op hun effectiviteit en prestaties. Orthogonaal aan deze werken stelt dit proefschrift een uitbreiding op CHERI voor om *data-oblivious computation* af te dwingen zodat software versterkt wordt tegen timing *side channels*. Ten slotte worden overwegingen met betrekking tot de duurzaamheid van SDRaD besproken.

List of Abbreviations

%ebp x86 base pointer register. 5

%rbp x86-64 base pointer register. 5, 7

%rsp x86 and x86-64 stack pointer register. 5

ALU arithmetic logic unit. 22, 93, 94

API application programming interface. 11, 15–17, 19, 20, 36, 42, 44, 47, 49, 50, 57, 59, 62, 63, 65–69, 74, 75, 98, 101, 113, 116

ASLR address-space layout randomization. 6, 26, 41, 43, 50, 52, 82, 106

BSS block starting symbol. 3

BTB branch-target buffer. 22

CFI control-flow integrity. 6, 26, 50

CHERI Capability Hardware Enhanced RISC Instructions. viii, xxi, 2, 12–14, 16–18, 21, 22, 50, 81–86, 88, 89, 91, 93–96, 98, 99, 101, 102, 104, 107–111, 114

CISA Cybersecurity & Infrastructure Security Agency. 2, 12

CoTS commercial off-the-shelf. 28, 52, 53

CP conditional permission. 17, 83, 84, 86, 87, 89–94, 96, 98, 99, 101, 102, 107–110, 134, 135

CVE Common Vulnerability Enumeration. 14, 84

DoS denial-of-service. 1, 2, 7, 14, 26

DPC++ Data Parallel C++. 105

FFI foreign function interface. 2, 10, 51, 56, 57, 62, 63, 65, 79

FPGA field-programmable gate array. 18, 82, 99, 102, 108

g.m. geometric mean. 103

GEP `GetElementPtr`. 97, 98

IP intellectual property. 83, 92, 93, 99

IPC inter-process communication. 60, 66, 115, 116

IR intermediate representation. 96, 97

ISA instruction-set architecture. 5, 12, 82, 85, 89, 92, 94, 99, 107

JIT just-in-time. 90

LAM Linear Address Masking. 94

LB lower bound. 87

LUT lookup table. 99

MMU memory management unit. 3, 52, 85

MPK Memory Protection Keys. 15, 16, 19, 20, 29, 55, 57, 115

MSRC Microsoft Security Response Center. xxi, 13, 14, 82, 84, 86

MSVC Microsoft Visual C++. 104–106

NIST National Institute of Standards and Technology. 20, 83, 99

NSA National Security Agency. 2

OB operation bound. 87, 89, 90, 92–94, 96, 109

ONCD Office of the National Cyber Director. 12

OS operating system. 3, 26, 28, 29, 61, 94

PCC program counter capability. 91

PKRU protection key rights register. 15, 19, 20, 40–42, 48, 52, 61, 69, 75

PKU Protection-Keys for Userspace. 27, 29, 36, 40, 49, 52, 63, 68, 74

RSB return stack buffer. 22

SFI sofware-fault isolation. 8, 28, 29, 40, 42, 52, 61

SGX Software Guard Extensions. 34, 50, 51

SLOC source lines of code. 10, 45, 48

SSA static single-assignment. 96, 97

STL store to load. 22

SUD syscall user dispatch. 19

TBI Top Byte Ignore. 94

TEE trusted execution enviroment. 51

TLSF Two-Level Segregated Fit. 42, 46, 48, 62, 67, 92, 103

UAI Upper Address Ignore. 94

UB upper bound. 87

x86-64 64-bit x86. xxi, 3, 5, 15, 20, 21, 27, 115

XOM execute-only-memory. 90

List of Symbols

a address. 95

E exponent. 95

I_E internal exponent. 95

MW mantissa width. 95

b base. 93, 95

o operation top. 93, 95, 109, 110

t top. 95, 109

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern computers are rooted in the *Von Neumann architecture* that introduced the design of a stored-program computing machine where programmable memory holds *both program code*—instructions the computer executes—*and data*—the information the program processes [228]. While this design revolutionized computing, it also created the need to separate executable code and non-executable data to prevent unauthorized memory access. Historically, the risks of improper memory access were first acknowledged in 1972 in studies commissioned by the US Air Force [16]. These concerns became increasingly significant with the rise of digital communication, which enables billions of network interactions. Connectivity allows unprecedented collaboration but creates opportunities for attackers to exploit software vulnerabilities remotely, potentially escalating to arbitrary code execution. The destructive potential of memory-related bugs for networked computers was demonstrated by the 1988 Morris Worm incident [170], which exploited a buffer overflow to spread from one computer to another across the early Internet, ultimately causing widespread denial-of-service (DoS) attacks affecting roughly 10% of all connected computers at the time.

These risks remain as critical today as memory-related vulnerabilities continue to be exploited in cyberattacks. Google Project Zero's "0day In the Wild" dataset reveals that over 70% of zero-day vulnerabilities discovered between July 2014 and June 2022 were attributed to memory-safety issues [80]. At the same time, the potential consequences of a Morris Worm-scale incident are much higher. The 2024 CrowdStrike incident disabled roughly 1% of Windows computers, causing billions of US dollars in damages and disruptions across various sectors critical for society. The incident was caused by an accidentally introduced memory-safety error in a Windows operating system driver [50].

At the same time, leading international cybersecurity organizations such as the US

Cybersecurity & Infrastructure Security Agency (CISA) and the National Security Agency (NSA) are advocating for the adoption of memory-safety principles across the software industry [39, 166]. CISA initially focused on promoting the benefits of memory-safe programming languages with built-in safety features, such as garbage collection and compile-time or run-time checks. However, given the vast multibillion-line C and C++ codebase that is difficult to fully replace at scale, other approaches to memory safety have gained emphasis. These include compiler technologies that mitigate memory-safety issues in C and C++, as well as hardware-based solutions such as capability architectures.

This thesis and contributions. This thesis explores various mechanisms for enhancing memory safety, highlighting previously overlooked gaps in current approaches. It examines programming languages such as C and C++, memory-safe languages like Rust [105], and advancements in memory-safe hardware like Capability Hardware Enhanced RISC Instructions (CHERI) [234]. This thesis emphasizes the importance of ensuring software resilience as well as detecting and mitigating memory-safety vulnerabilities.

First, we provide background information on memory-related defense techniques and highlight the existing gaps with the problem statement and research questions ($\mathcal{RQ}s$) in Section 1.1. Then, we outline the scope and contributions ($\mathbb{C}$) of the thesis in Section 1.2 and complementary contributions ($\mathbb{CC}$) in Section 1.3.

In summary, the main contributions are:

- $\mathbb{C}1$: Secure Domain Rewind and Discard (SDRaD), an approach for in-process software compartmentalization that improves an application's resilience against DoS conditions caused by memory vulnerabilities.
- $\mathbb{C}2$: SDRaD-foreign function interface (FFI), an adaptation of the SDRaD solution for Rust to protect Rust applications from memory-safety vulnerabilities in foreign function interfaces, allowing protected calls into unsafe C libraries.
- $\mathbb{C}3$: Mon CHÉRI, an extension to the memory-safe hardware architecture CHERI that extends its memory-safety properties to additionally prevent uninitialized memory access, a source of memory issues elaborated on in Section 1.1

While the focus of this thesis is on bridging previously overlooked gaps in memory-safety defenses, it has additionally yielded the following complementary contributions, which further orthogonal aspects to the central research questions considered in this thesis. These complementary contributions are:

- $\mathbb{CC}1$: lazypoline, an efficient and comprehensive syscall interposition mechanism.

- ℂℂ2: SandCell, a compiler-assisted automated compartmentalization for Rust beyond unsafe code as a followup work for ℂ1 and ℂ2.
- ℂℂ3: An evaluation of different memory-safety-defense techniques, such as stack canaries and shadow stacks, in terms of their effectiveness and performance.
- ℂℂ4: `BLACKOUT`, an extension to memory-safe hardware for enforcing data oblivious computation to harden software against timing side channels.
- ℂℂ5: Environmental sustainability considerations related to ℂ1 and ℂ2.

1.1 Memory Safety and Why It Matters

Memory is a critical resource in computer systems for storing and processing data, including inputs, intermediate values derived during computation, and outputs generated by the program.

In the early days of computing, programmers managed memory directly through punch cards or assembly code, writing low-level instructions tailored to the specific hardware architecture. As computer systems grew more complex, programming languages evolved to provide higher levels of abstraction with compiler support. This innovation simplified code development, reduced the need for direct hardware manipulation, and improved portability.

The evolution of operating systems (OSs) closely parallels these changes in programming paradigms. Early systems offered little to no abstraction for memory management, relying on programmers to allocate and manage memory manually. With the advent of multitasking and shared-resource systems, operating systems introduced virtual memory to isolate processes and provide a unified abstraction for memory usage through hardware-based technologies such as the memory management unit (MMU). This innovation enabled programs to access memory without considering the underlying physical hardware, improving security and portability.

In modern operating systems, such as Linux on the 64-bit x86 (x86-64) architecture, each process is assigned a contiguous virtual memory region, typically within a 48-bit address space for user-level programs, i.e., *userspace*. This memory typically includes a program's stack, heap, data, block starting symbol (BSS), and text segment. The text segment contains the program's instructions. Static variables are allocated at load time in the data or BSS segment and dynamic variables are allocated either automatically in the program's stack or explicitly at run time in the program's heap.

Correct and efficient memory management is fundamental to the reliable function of software and hardware systems. Secure memory management implies that a program is memory safe. Memory safety consists of properties such as*:

- *spatial safety*–memory accesses remain within the bounds of the intended memory object,
- *temporal safety*–memory is accessed only while it remains valid, and
- *initialization safety*–memory is always explicitly initialized before it is used.

These properties form the foundation of secure memory management across all computing environments, regardless of the programming language or system architecture.

One way of categorising programming languages is by how they manage process memory. *Managed memory-safe languages*, such as Java and Python, include automatic memory management features like garbage collection. These languages typically run on a language-level virtual machine (e.g., JVM for Java, CPython for Python), which intercepts and validates memory accesses on behalf of the program. While these features reduce the likelihood of memory-related bugs, they may introduce additional run-time or compile-time overhead.

Memory-unsafe languages, such as C and C++, give developers direct control over memory management within a process. While this provides flexibility and efficiency, it also increases the risk of memory-related errors, such as buffer overflows and dangling pointers.

Meanwhile, other memory-safe languages, such as Rust [105] and Safe C++ [20], rely on compile-time memory safety analysis and checks inserted directly into the compiled program. These approaches ensure memory safety without relying on garbage collection, minimizing run-time overhead while maintaining performance and security.

The flexibility of manual memory management of memory-unsafe languages has historically resulted in vulnerabilities, particularly when memory access occurs outside designated boundaries. One of the earliest well-documented memory-safety bugs, buffer overflows, saw a significant rise in both discovered and exploited cases during the mid-1990s [119].

In-Memory Layout of a Linux Process. To illustrate how memory is structured at run-time, consider the example in Figure 1.1. The call stack at this moment includes the

*There is no universal definition of memory safety, different programming languages have their own notion of safety and soundness definitions. The properties listed above are common for all systems discussed in this thesis.

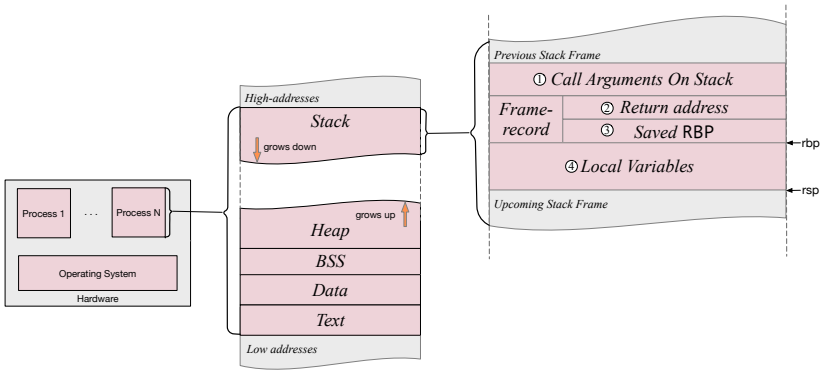


Figure 1.1: The virtual memory layout of a process and call stack structure with virtual memory segments (text, data, BSS, heap, and stack) and the organization of stack frames in the x86-64 architecture.

stack frame function $\mathcal{F}$. Before function $\mathcal{F}$ is invoked, its arguments are either passed in registers or, if the arguments outnumber the available registers, on the call stack (①) followed by the functions frame record including its return address (②, typically pushed onto the stack), the saved frame pointer (③), and finally allocations for local variables (④).

In the x86 instruction-set architecture (ISA), including its contemporary x86-64 variant [22], the base pointer register `%rbp` (`%rbp` in x86-64 and `%ebp` on x86) records the beginning of local variables in a function's stack frame throughout the execution of a function. In contrast, the `%rsp` records the end of the stack frame. The saved `%rbp` holds the value of `%rbp` for the caller, i.e., the beginning address of the previous function's stack area, while the return address holds the address succeeding the `call` instruction in the caller. When a function exits via the `ret` instruction, the return address is retrieved from the stack, and control is transferred back to the calling function at that address. By corrupting the saved `%rbp` or return address, an attacker can either override the location of the previous stack frame or the address the function returns to. Modern compilers (with optimizations like `-O2` enabled) often omit the use of `%rbp` as a frame pointer. They use `%rsp` exclusively, calculating the offsets to local variables directly. In this optimized layout, the concepts of a "saved `%rbp`" and a chain of frame pointers disappear, but the return address is still stored on the stack and remains a critical target.

Memory-Safety Issues as Vulnerabilities. Early exploitation techniques such as "*stack smashing*" [119] target buffer overflows in stack-allocated arrays since these are particularly easy to exploit. As shown in Figure 1.1, stack-allocated buffers (④) are placed just below the frame record containing the saved frame pointer (③) and return

address (②). Consequently, writing beyond the bounds of a local variable can overwrite these values, leading to control-flow hijacking.

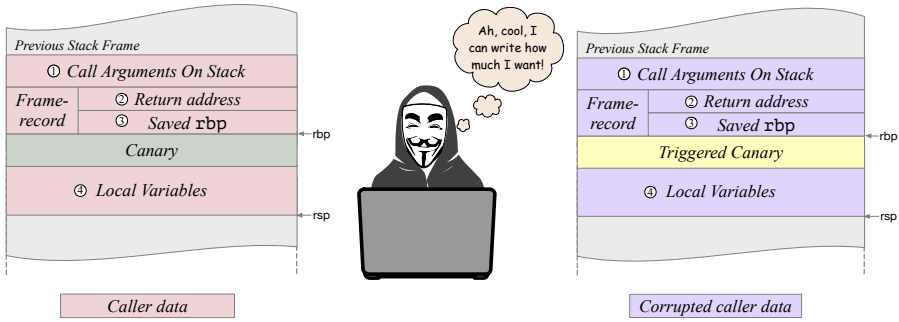
To mitigate early stack smashing attacks, Alexander Peslyak (using his online alias "Solar Designer") introduced the non-executable stack patch to the Linux kernel, which makes the stack portion of a userspace process's virtual address space non-executable. This means the code injected into the program stack by attackers cannot be executed. However, at the same time, Peslyak also published the first known return-to-libc attack, which replaces a return address on the call stack with the address of an existing function in the process's executable memory, thus bypassing the non-executable stack defense [63]. Over time, code-reuse attacks have become more sophisticated, evolving into techniques such as *Return-Oriented Programming* (ROP) [182]. In ROP, attackers craft a malicious "program" by stitching together short instruction sequences called *gadgets* which are already present in legitimate code segments. By chaining these gadgets, an attacker can execute arbitrary actions without injecting new code, effectively circumventing non-executable memory protections.

Beyond these control-flow hijacking methods, attackers have also developed *data-oriented attacks* that manipulate non-control data without violating control-flow integrity [35]. Instead of injecting or redirecting execution, data-oriented attacks manipulate existing data structures to achieve malicious outcomes while staying within the program's legitimate execution path. Techniques such as *data-oriented programming* (DOP) enable attackers to orchestrate complex, unintended operations by chaining corrupted data fields [92].

To counter these evolving attack techniques, researchers and practitioners have developed a range of mitigation strategies. The following section provides an overview of these techniques, especially focusing on how to mitigate control-flow hijacking.

1.1.1 Problem Statement, Research Objectives, Methodology

Techniques for mitigating memory-safety issues have evolved significantly over the decades in response to increasingly sophisticated attacks. Early techniques included data execution prevention (DEP), which prevents code execution in non-executable memory regions like the stack and heap; stack canaries, which detect stack buffer overflows by placing a sentinel value next to the return address [231]; address-space layout randomization (ASLR) randomizes an application's memory layout to make it harder for attackers to predict the location of executable memory regions and gadgets [140]. As attacks are becoming more advanced, modern systems are introducing techniques like control-flow integrity (CFI), which ensures that control flow transfers, such as function calls, occur to target only legitimate destinations [8].



(a) Stack canaries are instrumented stack layout of Figure 1.1.

(b) Application memory can be irrevocably corrupted by attacker when the attack is detected.

Figure 1.2: Stack layout with stack canaries before (a), after memory corruption (b).

However, these mechanisms can still result in DoS conditions, as state-of-the-art mitigations typically stop the attack by terminating the victim application.

To illustrate why the application is in an unrecoverable state after the attack is detected, consider the operation of stack canaries, a commonly deployed mitigation technique. Figure 1.2a shows an instrumented stack layout, where stack canaries (①) are sentry values placed on the stack between a function's local variables (②), and the stored `%rbp` (③).

Before returning from the callee function, a compiler-inserted check verifies that the canary value has not been altered; if it has, an error-handling routine (`__stack_chk_fail()`) is invoked to terminate the program rather than returning using a corrupted return address.

Assume an attacker employs a linear buffer overflow (as illustrated in Figure 1.2b). In this attack, the overflow first corrupts the stack canary before modifying the stored `%rbp` (③) and return address (④). Since the stack canary check detects the tampering of the stack frame only when the function exits, the overflow can alter the contents of the used portion of the stack freely before the function exits. By that time, the application's memory has been irrevocably corrupted. Thus the program's correct execution cannot continue.

A similar issue arises with shadow stacks [29], a mitigation mechanism in which a copy of the return address is stored in a separate stack to detect potential tampering. If a mismatch between the return address on the call stack and return address on the shadow stack is detected, the shadow stack mitigation prevents the victim function from returning to a potentially corrupted address. Similar to stack canaries, the application must be terminated, as other parts of memory are not recoverable from the contents of

the shadow stack. We elaborate on the effectiveness of modern shadow stack defenses in thwarting stack smashing attacks in ℂC3.

We identified a gap in current defenses: while they can detect attacks, they are forced to terminate the application, as the contents of the application's memory that are corrupted by a memory error are irrevocably lost.

Where is the gap?

Current defenses can detect attacks but are forced to terminate the application, as the contents of the application's memory corrupted by a memory error is irrevocably lost, preventing application execution from progressing.

Existing solutions that prioritize system availability typically rely on *replication* or *checkpoint-and-restore* mechanisms, which regularly snapshot the application state. In the event of a fault, the system can restore from the most recent snapshot. However, these approaches incur significant performance overhead [90], making them unsuitable for applications that require low-latency operations.

Another well-known technique for ensuring software-fault isolation is compartmentalizing the application, which can be achieved through *process-based* or *in-process* isolation.

Process-based isolation protects the system's integrity and resilience by providing the following key properties:

(i) integrity: each process runs in its own virtual memory space that prevents a malicious process from accessing the memory of another process, and

(ii) resilience: each process has its own failure boundary so one process' failure does not affect others.

While process-based isolation provides strong resilience guarantees against memory-safety vulnerabilities, it often requires code refactoring and incurs significant overhead due to costly inter-process communication [111, 178].

In contrast, in-process isolation is based on the notion of creating compartmentalized security domains within the memory space of a single application process. The main benefit of in-process isolation is that since transitions from one domain to another stays within the same process, in-process isolation can significantly reduce the runtime cost of context switching compared to traditional process isolation. In-process isolation is enabled through sofware-fault isolation (SFI) [214], a technique for using program transformations to establish logical protection domains via compiler-based

program transformation [31, 132, 144, 232], binary rewriting [68], or hardware-assisted enforcement [33,89,98,104,109,116,148,181,190,207,224,230,233]. While in-process isolation techniques are lightweight and efficient, they lack the software resilience guarantees provided by process-based isolation. Independently, this limitation is also emphasized by Lefeuvre et al. [117] in a recent Systematization of Knowledge (SoK) of software compartmentalization.

As our first research question $\mathcal{RQ}1$, we investigate how to recover application state and maintain availability without sacrificing performance after an attack is detected.

$\mathcal{RQ}1$

How to provide an efficient solution to recover application state and maintain availability after a memory-corruption attack is detected?

1.1.2 Programming language perspective: Rust

Rust is a systems programming language designed to strongly emphasize memory safety [105]. To ensure spatial safety, the Rust standard library provides smart pointers with built-in boundary checking and associated metadata. If a memory access goes out of bounds, the program explicitly panics at run-time to prevent undefined behavior. To enforce temporal safety, Rust relies primarily on three foundational compile-time concepts: *ownership*, *borrowing*, and *aliasing* $\oplus$ *mutability*.

Ownership Every piece of data in Rust has a single owner at any point in time. When the owner's lifetime ends, Rust automatically frees the memory associated with that data. This provides temporal safety guarantees that prevent use-after-free errors, common issues in manual memory management.

Borrowing Borrowing allows one or more parts of a Rust program to temporarily access data without taking ownership. This is done through references that behave similarly to pointers in C and C++ but are subject to the additional restrictions explained below.

Aliasing $\oplus$ mutability Rust enforces a rule which ensures one mutable reference or any number of immutable references can exist to the same data, but not both. Aliasing refers to multiple references to the same memory location, whereas mutability allows the modification of the memory value. The principle of aliasing $\oplus$ mutability ensures

that if (mutable) data can be changed, it cannot be accessed simultaneously from different parts of the program, thus preventing data races.

Rust achieves this safety property through a compile-time feature called the *borrow checker*. The borrow checker analyzes the lifetimes and scopes of references to ensure that the aforementioned safety rules are adhered to.

Unsafe Rust Rust has a language superset called Unsafe Rust where the programmer is responsible for upholding the memory-safety invariants that are checked and enforced by Rust's borrow checker for "Safe Rust" code [186, 216]. Unlike Safe Rust, where the compiler enforces safety rules, unsafe code requires the developer to manually perform these checks and ensure that specific safety conditions or "invariants" are met. Unsafe Rust code blocks in otherwise Safe Rust programs can thus be viewed as informal developer contracts where the developer takes on a heightened level of responsibility and trust which is implicit in languages such as C and C++. The explicit difference between Safe and Unsafe Rust is emphasized in the Rust community, e.g. through Unsafe Rust having a distinct mascot (Corro the Unsafe Rust Urchin) which contrast with the hardened nature of Rust's Ferris the Crab mascot (see Figure 1.3). In Unsafe Rust, developers can perform operations such as raw pointer dereferencing, manipulating smart pointer metadata, or calling C functions through FFI. However, improper use of these features can lead to soundness bugs, which occur when Rust's type system or safety guarantees are violated in a way that compiles successfully but results in unintended behavior at run time. Listing 1.1 demonstrates an unsafe operation where a buffer overflow can occur due to out-of-bounds memory access. The method `.as_mut_ptr()` ❷ provides a raw pointer to the N array declared in ❶, which is then used with `.add (i)` ❸ to write values at specific locations. However, if $M > N$, the unsafe code can lead to memory corruption and undefined behavior. According to Li et al. [121] more than 72% of packages on the official Rust package registry (`crates.io`) depend on at least one unsafe FFI-bindings package, highlighting the widespread use of Unsafe Rust in real-world applications.

Open-source projects such as Mozilla Firefox and the Linux Kernel have, since 2009 and since 2020 respectively, gradually incorporated Rust into their codebases. For example, as of September 2025, Rust accounts for 12.3% of the source lines of code (SLOC) in Mozilla's Gecko development project. However, C and C++ code still accounts for 40.1% of the codebase [76]. This demonstrates that transitioning entirely to memory-safe languages will take significant time, and mixed-language applications will continue to persist in practice. We identified the gap that using Rust alongside memory-unsafe languages can undermine Rust's memory safety and software-resilience guarantees.

Listing 1.1: Stack-based overflow using unsafe block in Rust

```
fn main() {
    let mut buffer: [u8; N] = [0; N]; ❶

    unsafe {
        let ptr = buffer.as_mut_ptr(); ❷ // Unsafe API

        for i in 0..M {
            *ptr.add(i) = i as u8; ❸ // Bounds violation if M > N
        }
    }
}
```

Where is the gap?

Mixed-language applications undermine memory-safety guarantees of safe Rust.

Early approaches to alleviate this problem focus on isolating safe and unsafe code using compiler-driven compartmentalization [19, 104]. However, in practice, these automated techniques are limited to boundaries derived from language-level constructs. They can only distinguish between safe and unsafe regions, but lack the flexibility to isolate different unsafe components from each other. This limitation leads to problematic edge cases, where most or all memory allocations end up being delegated to the unsafe domain, undermining the effectiveness of the isolation. Another approach, Sandcrust [114], provides an easy-to-use application programming interface (API) that allows developers to annotate functions for process-based isolation, which can introduce significant overhead. Additionally, aside from Sandcrust, existing methods have overlooked the resilience of Rust applications. As our second research question $\mathcal{RQ}2$, we consider how we can adapt the approaches that address $\mathcal{RQ}1$ to protect Rust code in multi-language applications.

$\mathcal{RQ}2$

How can approaches that address $\mathcal{RQ}1$ be adapted to protect Rust code in multi-language applications?

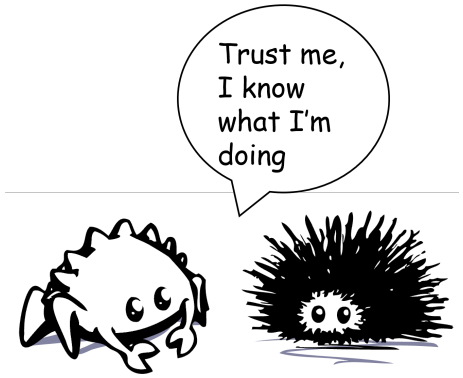


Figure 1.3: In Rust we trust!

1.1.3 Memory-safety at hardware level: CHERI

Capability Hardware Enhanced RISC Instructions (CHERI) is an ISA extension developed by the University of Cambridge and SRI International [234]. The project began as a research initiative in 2010 and has since evolved into a comprehensive effort, including CHERI-enabled software stack for a fork of the FreeBSD operating system (renamed to CheriBSD), compiler support in LLVM, proof-of-concept support for the Linux Kernel, and hardware adaptions for RISC-V, MIPS, and industrial demonstrators for the 64-bit Arm architecture through the Arm Morello platform. CHERI has been highlighted by reports from CISA [39] and US White House Office of the National Cyber Director (ONCD) [167] as a viable solution for achieving memory safety at scale, particularly beneficial when working with memory-unsafe languages like C and C++.

CHERI incorporates *capability-based addressing* that replaces conventional pointers with integrity-protected objects called capabilities. These capabilities contain permission and object-bounds metadata (see Figure 1.4), that allow a CHERI-enabled processor to enforce spatial and temporal safety, along with fine-grained permission policies. The metadata is stored within the capability, making each CHERI capability twice the width of the native integer pointer type of the baseline architecture. Every capability has an associated *tag bit* which is stored separately to maintain integrity, preventing unintended or malicious modifications. The in-memory representation of a capability includes, in addition to the baseline architecture address, the following data:

- **Bounds Information:** Includes upper (UB) and lower bounds (LB) relative to the baseline address, which defines the portion of the address space the capability can access. The bounds are stored in a compressed format to reduce the memory

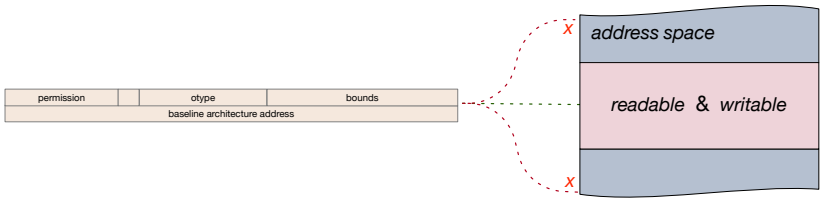


Figure 1.4: Each CHERI capability is double the width of a regular pointer and consists of the address (identifying a location in memory the capability points to), permission information, object type, and bounds information.

footprint of compressed capabilities.

- **Permission Information:** Defines fine-grained access policies such as read, write, and execute permissions for the memory the capability grants access to.
- **Object Type:** Enables fine-grained software compartmentalization by sealing capabilities with unique type, so that objects are only accessible through well-defined CHERI call gates.

To guarantee spatial safety, CHERI requires that each capability is equipped with bounds information derived from the in-memory object it points to. The CHERI software stack derives bounds for stack allocations through compiler instrumentation. Similarly, heap allocations are bounded by using a CHERI-aware heap allocator, which returns capabilities strictly bounded to the size of each allocated object.

To provide temporal-safety guarantees, CHERI requires the ability to revoke capabilities that no longer point to valid memory allocations, i.e., when a memory allocation accessed by a capability is freed, any capabilities for the same allocation should be revoked to avoid the CHERI-equivalent of dangling pointers. Cornucopia [72] is a lightweight capability revocation system explicitly designed for CHERI that provides deterministic temporal safety. Cornucopia tracks capabilities throughout memory, allowing them to be revoked when the corresponding memory allocations are freed.

As a result, CHERI can mitigate large classes of memory-safety vulnerabilities. According to the Microsoft Security Response Center (MSRC) in vulnerabilities reported to MSRC, between 2017 and 2019, CHERI would have prevented 31% of spatial memory safety errors in its default mode and 24% of temporal safety errors with Cornucopia, as seen in Figure 1.5.

However, CHERI intentionally excludes specific memory-safety properties from its scope. A notable omission is undefined behavior associated with *uninitialized variables* that are declared but not assigned a value, potentially containing junk data or previously

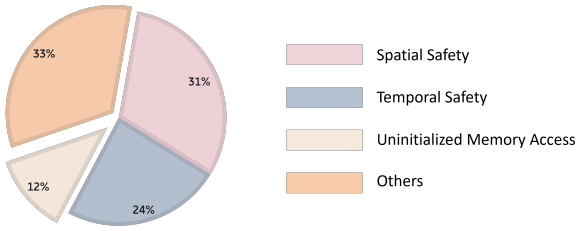


Figure 1.5: Breakdown of vulnerabilities reported to MSRC in 2019 by vulnerability type, indicating the fraction of vulnerabilities which could have been mitigated by CHERI. The "Others" category covers type confusion, bounds violations missed due to compressed bounds in CHERI, and DoS issues requiring separate fixes.

deallocated potentially sensitive data. MSRC reports that uninitialized memory vulnerabilities accounted for approximately 5–10% of all Common Vulnerability Enumerations (CVEs) issued by Microsoft [24, 99]. The proportion of uninitialized memory-related vulnerabilities reported by MSRC is consistent with statistics reported for the broader software industry by the CVE program where uninitialized memory accounts for around $\approx 9\%$ of all memory-safety related vulnerabilities recorded between 2015 and 2022 [209]. CHERI fails to prevent such uninitialized memory access [99].

Where is the gap?

CHERI cannot express policies that prevent uninitialized memory access.

Software-based mitigations for uninitialized memory issues are well established. For instance, mainstream compilers such as GCC and Clang/LLVM can instrument stack-based variables to be automatically zeroed before their first use [24], and researchers have proposed various security enhancements to hardened memory allocators to zero allocations either before use, or when memory is freed [38]. However, these methods incur performance overhead and may interfere with debugging tools and other development-time practices aimed at detecting and resolving memory safety issues in the software development pipeline [168]. As our third research question, we consider to what extent an architectural solution can prevent uninitialized memory access.

*RQ*3

How can an architectural solution be designed to prevent uninitialized memory access?

1.2 Thesis Contributions

This thesis addresses the research questions identified in the previous sections by bridging previously overlooked gaps in memory-safety defenses through three main contributions. These contributions are summarized below and detailed in the following chapters.

ℂ1: **Rewind & Discard: Improving Software Resilience using Isolated Domains** To address *RQ*1, we propose *secure rewind and discard of isolated domains* as an efficient and secure method of improving the resilience of software that is targeted by run-time attacks. A prerequisite for this approach is that the application must be compartmentalized into distinct isolation domains. We assume pre-existing run-time defense mechanisms have been integrated to the system. When an attack is detected in one domain, our solution ensures that exploited memory vulnerabilities cannot affect any other domains, allowing the system to safely return to a pre-established *rewind point* while discarding the affected domain and its (potentially) corrupted state. We implement the concept of secure domain rewind and discard as *SDRaD*–a C-language Linux library for the x86-64 architecture using Memory Protection Keys (MPK), a hardware mechanism that control access-control permission for userspace memory pages (see Section 2.2.3). SDRaD provides APIs to control the life cycle of domains to allow the integration of the rewind and discard mechanism into existing applications.

SDRaD benefits service-oriented applications, which are particularly susceptible to disruptions caused by memory-safety issues because even a temporary failure in a critical component can affect a large number of clients. For example, consider Memcached—a general-purpose distributed memory-caching system that lacks persistent storage. Restarting Memcached due to an irrecoverable memory error after a run-time attack requires that all clients resend their data and keys to restore its previous state. However, when SDRaD is applied, our result shows that the secure-rewind mechanism recovers Memcached to a normal operating state in a safe manner several orders of magnitude faster than a full restart of application process.

Using MPK alone for compartmentalization does not provide a complete solution—attackers can arbitrarily modify the protection key rights register (PKRU) which controls the memory which is accessible by a given domain at run-time or invoke

system calls to change memory mappings that assign specific regions of the domain's memory. To harden MPK-based in-process isolation, established methods such as binary scanning and system call filtering are necessary. However, we assume such measures are in place for this work. As a follow-up step to enhance SDRaD's in-process isolation, we have integrated it with system call interposition, and binary scanning methods as explained in ℂℂ2.

Additionally, we leverage stack canaries and domain violation detection to enable secure rewind and discard. In a subsequent effort, S. Ruchlejmer ported the SDRaD approach to the CHERI architecture under my supervision [184]. The CHERI architecture shifts attack detection to an earlier stage, allowing attacks to be identified before they can alter the contents of memory. This early detection further enhances the ability to rewind and discard the compromised domain upon detecting an attack.

I designed an early SDRaD prototype with T. Nyman, while C. Baumann cooperated with us to improve the SDRaD API design which is presented in the published work. I was responsible for the complete implementation and evaluation of SDRaD, as well as its integration to the case studies presented in the publication. The code and benchmark evaluation have been released as open-source at `https://github.com/secure-rewind-and-discard`. All co-authors contributed to writing the manuscript. This work was presented as a poster at Ericsson Research Day in 2022.

This work was published as:

> ***M. Gülmez**, T. Nyman, C. Baumann and J. T. Mühlberg, "Rewind & Discard: Improving Software Resilience using Isolated Domains" in 53rd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Porto, Portugal, 2023, pp. 402-416, `https://doi.org/10.1109/DSN58367.2023.00046`.*

ℂ2: Friend or Foe Inside? Exploring In-Process Isolation to Maintain Memory Safety for Unsafe Rust To address $\mathcal{RQ}2$, we propose Secure Rewind and Discard of Isolated Domains for Foreign Function Interface (SDRaD-FFI), an approach that leverages in-process isolation using MPK to protect programs written in safe Rust from memory-safety violations originating in unsafe Rust. We build upon our first contribution, SDRaD, and adapt our C-language library to operate within the Rust runtime environment. SDRaD-FFI offers a Rust-native API that leverages Rust's powerful metaprogramming features to facilitate sandboxing of unsafe interfaces.

SDRaD-FFI provides an easy-to-use API for compartmentalization, enabling data argument passing through a serialization method and facilitating communication across different domains. This allows for more fine-grained compartmentalization compared

to other state-of-the-art approaches [12, 19, 104] and reduced developer effort compared to ℂ1 by requiring only minimal annotations from developers.

By building on SDRaD, we advance the state of the art in Rust in-process isolation by allowing memory errors in foreign function interfaces with C to be contained and recovered from. SDRaD-FFI is the first work to explore crash-resistant, in-process mitigations to vulnerabilities at the cross-language threat surface to enhance the resilience of Rust applications.

During the development of SDRaD-FFI, we found that a major contributor to the performance impact in APIs from application compartmentalization is the overhead associated with serializing and deserializing data across the isolation domain boundary. In this work, we additionally conduct an in-depth evaluation of serialization approaches applicable to in-process isolation, allowing us to significantly reduce serialization overhead compared to prior state-of-the-art solutions.

The source code and evaluation artifacts for SDRaD-FFI are available under an open-source license at `https://github.com/secure-rewind-and-discard`. In addition to the initial publication at IEEE SecDev'23, this work was presented at the Rust Devroom at FOSDEM'24 [151].

I designed the concept for SDRaD-FFI together with T. Nyman. I was responsible for the complete implementation and evaluation of SDRaD-FFI. All co-authors contributed to the manuscript.

This work was published as:

> ***M. Gülmez**, T. Nyman, C. Baumann and J. T. Mühlberg, "Friend or Foe Inside? Exploring In-Process Isolation to Maintain Memory Safety for Unsafe Rust" in 2023 IEEE Secure Development Conference (SecDev), Atlanta, GA, USA, 2023, pp. 54-66, `https://doi.org/10.1109/SecDev56634.2023.00020`.*

ℂ3: Mon CHÉRI: Mitigating Uninitialized Memory Access with Conditional Capabilities To address ℛ𝒬3, we extend the CHERI capability model to express memory-access policies that eliminate undefined behavior associated with uninitialized memory. Specifically, we introduce the novel concept of conditional permissions (CPs) to capability-based addressing. CPs can express memory-access policies that take previous operations on memory into account. This enables *conditional capabilities* with fine-grained policies that can satisfy different memory-safety objectives at varying granularities, such as *"no reads of memory which has not been the subject of at least one write"* (`Write-before-Read`) or *"this memory can be written to only once"* (`Write-Once`).

Tracking initialization for a single variable would in principle require just a single bit indicating whether an operation has occurred. However, tracking initialization for structs or arrays requires additional metadata—specifically operation bounds—to mark initialized regions. We encode these bounds within the baseline architecture address part of the CHERI capability representation.

The most challenging part of Mon CHÉRI was to realize compiler support for conditional capabilities. Whenever a capability is subject to an operation that requires updating the operational bounds, synchronization must be maintained across capabilities that reference the same memory area. To address this challenge, we provide a *store linearization pass* that ensures the compiler generates code such that a single, unique capability remains alive during the lifetime of the capability.

We evaluate this approach on the QEMU full-system emulator and a modified CHERI-RISC-V softcore realized on a field-programmable gate array (FPGA). Our evaluation shows conditional capabilities are practical, with high detection accuracy while adding a small overhead which is comparable to the cost of baseline CHERI capabilities.

I conceived the idea for conditional capabilities and designed them together with T. Nyman. I implemented the necessary compiler analysis and instrumentation along with our initial QEMU-based prototype on my own. Also, I implemented the hardware support for conditional capabilities for a pre-existing CHERI-RISC-V softcore together with H. Englund. All co-authors contributed to the manuscript. This work was presented as a poster at the Digital Security by Design Showcase (DSbD) event held in the UK in February 2025.

This work was published as:

> ***M. Gülmez**, H. Englund, J. T. Mühlberg and T. Nyman, "Mon CHÉRI: Mitigating Uninitialized Memory Access with Conditional Capabilities" in 46th IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2025, pp. 829-847,* `https://doi.org/10.1109/SP61157.2025.00133`

1.3 Complementary Contributions

While the focus of this thesis is on bridging previously overlooked gaps in memory-safety defenses through three main contributions that address the research questions identified in the previous sections, it has also yielded complementary contributions that explore orthogonal aspects of the central research questions. These complementary contributions are listed as follows:

ℂℂ1: System Call Interposition Without Compromise System call (abbreviated as `syscall`) interposition is vital for tools that monitor or modify application behavior. While mainstream operating systems provide APIs for syscall interposition, such as ptrace [126], seccomp [127], and syscall user dispatch (SUD) [217], these typically incur significant overhead, especially in syscall-intensive applications. One of the recent approaches, zpoline [248], reduces overhead by rewriting `syscall` instructions with `call rax` at application load time, allowing syscalls to invoke interposers directly without requiring kernel interaction.

However, these methods may fail to comprehensively intercept all `syscall` instructions, such as dynamically generated code. In this work, we introduce *lazypoline* [97], an interposition technique that initially employs slower kernel-based interfaces, SUD, to identify syscall instructions exhaustively and then lazily rewrites these instructions for efficient direct interposition upon subsequent executions. Our evaluations demonstrate that *lazypoline* is non-intrusive, fully exhaustive, and achieves performance comparable to pure rewriting methods, even on datacenter-scale workloads.

As we explained in ℂ1, especially in-process isolation based on MPK requires a system call interposition technique to prevent arbitrary reconfiguration of PKRU. However, the existing techniques for system call interposition require either kernel modification or incur significant overhead [224, 230]. This work gives a way forward for enabling fast and secure interposers with full compatibility without sacrificing performance for different use cases.

A. Jacobs was responsible for the implementation of lazypoline. I evaluated the performance overhead of lazypoline in syscall-intensive workloads such as web servers, and further demonstrated its exhaustiveness by showing that it successfully interposes all system calls in dynamically generated code. A. Andries handled the compatibility evaluation of lazypoline. The code and benchmark evaluations have been released as open-source at `https://github.com/lazypoline`. As part of the DSN 2024 artifact evaluation process, our code was peer-reviewed and awarded the "Available", "Reviewed", and "Reproducible" badges. All co-authors contributed to the manuscript.

This work was published as:

> *A. Jacobs, **M. Gülmez**, A. Andries, S. Volckaert and A. Voulimeneas, "System Call Interposition Without Compromise" in 54th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Brisbane, Australia, 2024, pp. 183-194,* `https://doi.org/10.1109/DSN58291.2024.00030`

ℂℂ2: SandCell: Sandboxing Rust Beyond Unsafe Code In ℂ1 and ℂ2, SDRaD provides APIs for compartmentalization, while SDRaD-FFI offers function-level annotations which have to be specified by the developer. However both approaches lack full automation, putting some of the burden of integrating compartmentalization on the developer. In this work, we present SandCell, an extension to the `rustc` compiler, that facilitates flexible and lightweight isolation by applying SDRaD automatically to existing syntactic boundaries of Rust. SandCell allows programmers to specify which components should be isolated, such as crates or function, with minimal annotation effort. The system also includes techniques for reducing the overhead of data passing between sandboxes and a mechanism for system call filtering to prevent privilege escalation. Our evaluation demonstrates SandCell's effectiveness in preventing vulnerabilities across various Rust applications while maintaining reasonable performance overheads.

J. Zhang conceived the idea of SandCell and was responsible for the compiler analysis and evaluation. I contributed to the paper by extending the SDRaD C library, provided in ℂ1, with a lightweight system call interposition technique, zpoline [248], to harden MPK-based isolation and enable in-process syscall filtering; adding support for the mimalloc allocator to improve compatibility with Rust; and incorporating an existing binary scanning method to detect and prevent misuse of the PKRU configuration. This work is currently under submission with a pre-print available as:

> *J. Zhang,* ***M. Gülmez****, T. Nyman, and G. Tan. "SandCell: Sandboxing Rust Beyond Unsafe Code"* `arXiv:2509.24032 [cs.SE]` *(2025)* `https://doi.org/10.48550/arXiv.2509.24032`

ℂℂ3: Do we still need canaries in the coal mine? Measuring shadow stack effectiveness in countering stack smashing As described in Section 1.1, stack canaries and shadow stacks provide similar levels of protection against sequential stack-based overflows in theory. In this work, we systematically evaluate whether x86-64 systems benefit from enabling stack canaries in addition to x86-64 shadow stack enforcement, using the US National Institute of Standards and Technology (NIST) Juliet Test Suite [163], which contains thousands of C and C++ test cases that demonstrate common programming defects leading to memory vulnerabilities.

We observe differences in overflow detection rates between the GCC and Clang compilers as well as across optimization levels, which we attribute to variations in the stack layouts generated by these compilers. Stack layout differs not only between compilers, but also across optimization levels and variants of stack-canary instrumentation. Our results further show that x86-64 shadow stack implementations

are more effective and outperform stack canaries when combined with a stack-protector-like stack layout.

To this end, we propose new Clang compiler options for x86-64 shadow stack instrumentation that emulate stack-protector layouts while relying solely on shadow stack checks. Our evaluation demonstrates that these options improve detection accuracy, allow stack canary checks to be omitted, and incur only a small performance overhead, which is lower than that of the corresponding stack canaries.

Source code artifacts for our evaluation are available at `https://github.com/ReSP-Lab/2025-ares-coal-mine`

H. Depuydt conducted the evaluation. I co-supervised the work with T. Nyman and J.T Mülberg. All co-authors contributed to the manuscript.

This work was published as:

> *H. Depuydt, **M. Gülmez**, T. Nyman, J. T. Mühlberg, "Do We Still Need Canaries in the Coal Mine? Measuring Shadow Stack Effectiveness in Countering Stack Smashing" in Availability, Reliability and Security (ARES 2025) Lecture Notes in Computer Science, vol 15993. Springer, Cham. `https://doi.org/10.1007/978-3-032-00627-1_10`*

ℂℂ4: **BLACKOUT: Data-Oblivious Computation with Blinded Capability.** In addition to memory safety, another prominent challenge for secure implementation for software, such as cryptographic libraries, is side-channel leakage that poses a significant threat to confidentiality and is particularly problematic for software handling secret data. Memory-safe languages such as Rust (see Section 1.1.2), promise to reduce memory safety bugs, but makes implementing side-channel leakage more difficult as they employ more higher level abstractions compared to C or C++ [21, 105]. As explained in Section 1.1.3, memory-safe hardware, CHERI effectively addresses spatial and temporal safety issues; however, it does not protect against side-channel attacks. In principle, CHERI-capable hardware could be complemented with existing functionality for data-oblivious computation, such as OISA [251] or BliMe [66], which aim to protect against side channels. However, these approaches introduce considerable performance overhead on their own, because they require the addition of hardware-managed tag bits to memory.

Orthogonally to the work presented in this thesis, we propose an extension to CHERI that integrates a hardware taint-tracking policy to guarantee the confidentiality of secret data against side channels. We introduce *blinded capabilities* that add a permission bit to CHERI capabilities that marks whether data referenced by the capability is classified as secret. Registers are extended with a *blindedness* bit, which is set when the

register is loaded with secret data. arithmetic logic unit (ALU) operations propagate the blindedness bits from the operands to the destination registers; therefore, any data derived from secret data is also marked as blinded. Crucially, any attempt to misuse secret data—such as affecting control flow or as an address in memory operations—results in a fault, effectively preventing side-channel leakage.

We implement our realization of blinded capabilities, BLACKOUT, on a FPGA softcore based on the speculative out-of-order CHERI-Toooba processor and extend the CHERI-enabled Clang/LLVM compiler and the CheriBSD operating system with support for blinded capabilities. Most importantly, BLACKOUT allows developers to write constant-time code with minimal additional annotations, benefit from compile-time diagnostics, and turn previously silent constant-time bugs into explicit errors reported through CHERI's exception mechanism.

We show that BLACKOUT ensures memory operated on through blinded capabilities is securely allocated, used, and reclaimed and demonstrate that, in benchmarks comparable to those used by previous work, BLACKOUT imposes only a small performance degradation compared to the baseline CHERI-Toooba processor. As shown by a test suite in [74], CHERI-Toooba is vulnerable to Spectre branch-target buffer (BTB) [106], return stack buffer (RSB) [110, 143] and store to load (STL) [91]. We blinded the secret value in all test cases and showed that BLACKOUT prevents all Spectre attacks from [74] catching side-channel violations even during speculation, but suppresses faults until speculation is confirmed (and ignores them otherwise).

The idea for BLACKOUT was jointly conceived by all co-authors. H. ElAtali and I shared responsibility for the implementation. H. ElAtali independently implemented the hardware taint tracking mechanism on the CHERI-Toooba core. I implemented the software stack for blinded capabilities, support for CheriBSD, and corresponding compiler modifications. H. ElAtali and I jointly evaluated BLACKOUT on FPGA. All co-authors contributed to the manuscript. Source code artifacts for the BLACKOUT hardware and software stack are available at `https://github.com/blindedcapabilities`. As part of the CCS 2025 artifact evaluation process, our code was peer-reviewed and awarded the "Available", "Reviewed", and "Reproducible" badges. This work was presented at the CHERITech'25 conference held in the UK in November, 2025 and published as:

> *H. ElAtali*, **M. Gülmez***, T. Nyman, and N. Asokan. 2025. "BLACKOUT: Data-Oblivious Computation with Blinded Capabilities" In Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan. ACM, New York, NY, USA, 15 pages.* `https://doi.org/10.1145/3719027.3765169`

*Both authors contributed equally to this work.

ℂ5: Exploring the Environmental Benefits of In-Process Isolation for Software Resilience As orthogonal work to this thesis, we highlight how improving software resilience against memory attacks can contribute to environmental sustainability while also enhancing security. By reducing the need for redundancy in dependable systems software, resilient designs can lower hardware demands. For instance, techniques such as replication or software diversification can decrease the likelihood of memory-related attacks and increase software longevity. However, these approaches often lead to over-provisioning of hardware resources, which is not environmentally sustainable. In contrast, SDRaD (see in ℂ1) and SDRaD-FFI (see in ℂ2) enables fast recovery from memory errors without relying on replication or diversification and imposes only minimal run-time overhead. In practice, SDRaD and SDRaD-FFI significantly reduce the time required to recover from a fault, supporting both secure and sustainable software systems.

I was inspired to write this paper by a talk given by D. Gruss at a summer school at Graz University, which motivated me to explore and evaluate this thesis's contributions through the lens of sustainability. Although this work builds on those early discussions, we hope that the concept of security for sustainability will gain broader recognition and be emphasized across all fields. T. Nyman, C. Baumann, and J. T. Mühlberg contributed to refining these ideas in relation to our joint work. All co-authors contributed to the manuscript.

This work was published as:

> ***M. Gülmez**, T. Nyman, C. Baumann and J. T. Mühlberg, "Exploring the Environmental Benefits of In-Process Isolation for Software Resilience" in 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S), Porto, Portugal, 2023, pp. 203-205,* `https://doi.org/10.1109/DSN-S58398.2023.00056.`

Chapter 2

Rewind & Discard: Improving Software Resilience using Isolated Domains

This chapter consists of $\mathbb{C}1$ and was previously published as:

> ***M. Gülmez**, T. Nyman, C. Baumann and J. T. Mühlberg, "Rewind & Discard: Improving Software Resilience using Isolated Domains" in 53rd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Porto, Portugal, 2023, pp. 402-416,* `https://doi.org/10.1109/DSN58367.2023.00046`.

This chapter addresses $\mathcal{RQ}1$: "How to provide an efficient solution to recover application state and maintain availability after a memory-corruption attack is detected?"

Abstract: Well-known defenses exist to detect and mitigate common faults and memory safety vulnerabilities in software. Yet, many of these mitigations do not address the challenge of software *resilience* and *availability*, i.e., whether a system can continue to carry out its function and remain responsive, while being under attack and subjected to malicious inputs. In this chapter we propose *secure rewind and discard of isolated domains* as an efficient and secure method of improving the resilience of software that is targeted by run-time attacks. In difference to established approaches, we rely on compartmentalization instead of replication and checkpointing. We show the practicability of our methodology by realizing a software library for Secure Domain

Rewind and Discard (SDRaD) and demonstrate how SDRaD can be applied to real-world software.

2.1 Introduction

A high level of availability for software systems is hard to achieve. The problem becomes even harder when dependability guarantees for distributed systems are required and software-level attacks are in scope: software written in unsafe languages can suffer from various memory-related vulnerabilities [69] that allow run-time attacks, such as control-flow attacks and non-control-data attacks [32], to compromise program behavior and let adversaries gain access to vulnerable systems.

According to the Google "0day In the Wild" dataset over 70% of the zero-day vulnerabilities between July 2014 and June 2022 can be attributed to memory-safety issues [80]. Research into run-time attacks has, during the past 30 years, led to an ongoing arms race between increasingly sophisticated attacks and run-time defenses to mitigate such attacks [213]. Today, major operating systems (OSs) provide such mitigations by default. This includes non-executable stack and heap areas, address-space layout randomization (ASLR) [140], toolchain hardening options such as stack canaries [231], and hardware-enforced control-flow integrity (CFI) [58]. However, virtually all currently known defenses mitigate detected attacks by terminating the victim application [9, 28, 31, 48, 68, 115, 122, 123, 144, 164, 165, 172, 192, 195, 213, 226, 232]. Thus, even though applications are hardened against run-time attacks, the response can still be leveraged by attackers to create availability issues and denial-of-service (DoS) conditions while the application is restarted, or to bypass security controls by resetting volatile system state, e.g., counts of failed login attempts.

Mitigations that focus on system availability typically involve application replication or checkpoint-and-restore mechanisms [90], which have a non-negligible performance impact themselves. Service-oriented applications are at particular risk as even a temporary failure in a critical component can affect a large number of clients. An example for such an application is *Memcached*, a general-purpose distributed memory-caching system, which is commonly used to speed up database-driven applications by caching database content. In this case, for example, restarting the application after a fault can easily take several minutes, while checkpointing or replication will have to deal with the large run-time state of the application.

Contributions. To address limitations of current defenses and to improve the resilience of software that is being targeted by run-time attacks, we propose *secure rewind and discard of isolated domains*: a mechanism that allows the state of a victim application under attack to be efficiently restored to an earlier state that is known to

be unaffected by the attack. This is possible by leveraging hardware-assisted software fault isolation to compartmentalize the application into distinct *domains* that limit the effects of run-time attacks to isolated memory compartments. An application can be instrumented to isolate, e.g., "high-risk" code that operates with untrusted input in a secure in-process sandbox and rewind the application state if an attack is detected against sandboxed code. Domains can be nested to allow for efficient and secure rewinding in different software architectures and use cases. In particular, service-oriented applications can be augmented with resilience against run-time attacks to limit the impact to concurrent clients.

We show the practicability of our methodology by realizing a software library for *secure domain rewind and discard* (SDRaD) for commodity 64-bit x86 processors with *Protection-Keys for Userspace (PKU)* [3, 6] and demonstrate how SDRaD can be applied to real-world software in case studies on Memcached, a popular distributed memory-cache system (Section 2.5.1), the NGINX web server (Section 2.5.2), and OpenSSL (Section 2.5.3). In summary, the contributions of this chapter are:

- *Secure Rewind and Discard of Isolated Domains* is a novel scheme to improve software resilience against run-time attacks by rewinding the state of a victim application (Section 2.3).
- We explore different *design patterns for compartmentalization and rewinding* and discuss their applicability to retrofit software with secure rewind and discard (Sections 2.3.4 to 2.3.6).
- We provide SDRaD, a realization of secure rewinding for commodity 64-bit x86 (x86-64) processors with PKU (Section 2.4, [83]).
- We show that SDRaD can be used with limited refactoring of application code only, as we apply it in three case studies (Section 2.5). Benchmarks on Memcached and NGINX exhibit a worst case performance overhead <7.2%, negligible overhead (2%–4%) in realistic multi-processing scenarios, and negligible memory overhead (0.4%–3%).
- We assess the security, applicability, and limitations of our approach in Section 2.6 and compare it to related work in Section 2.7.

2.2 Background

Rollback recovery techniques [67] have been studied in the context of a wide variety of applications ranging from programming language constructs [177] to recovery protocols for distributed system [67]. At a high-level, such techniques described in prior work

can be divided into *checkpoint-based* and *log-based* approaches. Checkpoint-based techniques, such as *checkpoint & restore* (Section 2.2.1) rely on recording transient system state so that it can later be recovered from a previously prepared *checkpoint*. Checkpoints in prior work are predominantly based on reproducing the process's memory image in a manner that can later be restored [90, 130, 133, 155, 250, 254]. Log-based approaches [118] combine checkpointing with recording the necessary information of nondeterministic events to replay each event during the recovery process.

In this work, we avoid the pitfalls of checkpoints that reproduce process memory by leveraging hardware-assisted fault isolation to partition an application process into distinct, isolated domains. This compartmentalization facilitates secure rewinding of application state by isolating the effects of memory errors. This enables rewinding to application states that precede the point of failure in the application's call graph and are unaffected by a caught and contained error.

Pre-existing work that bases rollback on compartmentalization properties is aimed at improving OS reliability to driver failures [210, 211] or targets embedded systems [11]. In contrast, secure rewind and discard of isolated domains targets application software in commercial off-the-shelf (CoTS) processors and does not require OS or hardware changes.

2.2.1 Checkpoint & Restore

Application checkpoint & restore [90] is a technique for increasing system resilience against failure. By saving the state of a running process periodically or before a critical operation, a failed process can later be restarted from the checkpoint. The cost of this depends on the amount of data needed to capture the system's state and the checkpointing interval.

Several studies have focused on optimizing system-level and application-level checkpointing [133, 250, 254]. Secure checkpointing schemes [155] generally leverage cryptography to protect the integrity and confidentiality of checkpoint data at rest. However, they generally do not consider attacks that tamper with checkpointing code at the application-level. Furthermore, bulk encryption of checkpoint data is too costly for latency-sensitive applications, e.g., network traffic processing or distributed caches, unless load balancing and redundancy is present.

2.2.2 Software Fault Isolation

Sofware-fault isolation (SFI) [214] is a technique for establishing logical protection domains within a process through program transformations. SFI instruments the

program to intermediate memory accesses, ensuring that they do not violate domain boundaries. Since transitioning from one domain to another stays within the same process, SFI solutions can offer better run-time efficiency compared to traditional process isolation, especially in use cases where domain transitions are frequent. SFI has been successfully deployed for sandboxing plug-ins in the Chrome browser [249], isolating OS kernel [183] and modules [31, 68, 144], as well as code accessed through foreign function interfaces in managed language runtimes [206].

SFI enforcement can be realized in different ways. The principal method to realize SFI for native code binaries is the use of an inline reference monitor through binary [68] or compiler-based rewriting [31, 132, 144, 232] of the application binary. Recent SFI approaches leverage hardware-assistance (Section 2.2.3) to further improve enforcement efficiency [33, 89, 98, 104, 109, 116, 148, 181, 190, 207, 224, 230, 233]. Existing approaches to SFI share the drawback of memory vulnerability countermeasures as they respond to detected domain violations by terminating the offending process.

2.2.3 Memory Protection Keys

Memory Protection Keys (MPK) provide an access control mechanism that augments page-based memory permissions. MPK allows memory access permissions to be controlled without the overhead of kernel-level modification of page table entries (PTEs), giving it a significant performance advantage compared to completely OS-controlled memory protection facilities, e.g., `mprotect()` [172]. On 64-bit x86 processors, PKU are supported in Intel's [3] and AMD's [6] microarchitectures. Similar hardware mechanisms are also available in ARMv8-A [5], IBM Power [7], HP PA-RISC [1] and Itanium [2] processor architectures.

Each memory page is associated with a 4-bit protection key stored in the page's PTE on 64-bit x86 processors. The access rights to memory associated with each protection key are kept in a *protection key rights register* (PKRU) that allows write-disable and access-disable policies to be configured for protection keys. These policies are enforced by hardware on each memory access.

2.3 Secure Domain Rewind and Discard

We propose *secure rewind and discard of isolated domains*, a novel approach for improving the resilience of userspace software against run-time attacks that augments existing, widely deployed run-time defenses. First, we present our threat model, system

requirements (Section 2.3.1), and high-level idea (Section 2.3.2). Sections 2.3.3 to 2.3.6 delve into specific aspects of the design.

2.3.1 Threat Model and Requirements

Assumptions. In this work, we assume that the attacker has arbitrary access to process memory, but is restricted by the following assumptions about the system:

A1 A W⊕X policy [213] restricts the adversary from modifying code pages and performing code-injection.

A2 The application is hardened against run-time attacks and can detect an attack in progress, but not necessarily prevent the attacker from corrupting process memory.

We limit the scope of **A2** to software-level attacks. Transient execution [246] and hardware-level attacks, e.g., fault injection [198], rowhammer [154] etc., which are generally mitigated at hardware, firmware, or kernel level, are out of scope.

Requirements. Our goal is to improve the resilience of an application against active run-time attacks that may compromise the integrity of the application's memory. We introduce a mechanism for *secure rewind* with requirements as follows:

R1 The mechanism must allow the application to continue operation after system defenses (**A2**) detect an attack.

R2 The mechanism must ensure that the integrity of memory after recovering from the detected attack is maintained.

To facilitate **R1** and **R2** the application is compartmentalized into isolated domains with the following requirements:

R3 Run-time attacks that affect one domain must not affect the integrity of memory in other domains

R4 The attacker must not be able to tamper with components responsible for isolation or transitions between domains, or data used as part of the rewinding process.

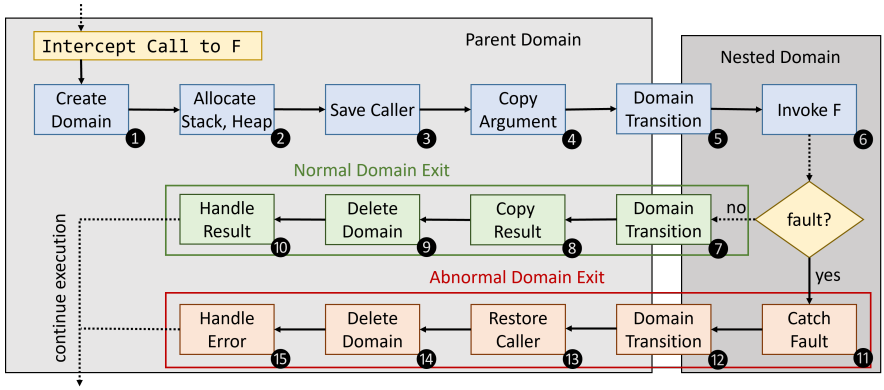


Figure 2.1: Domain life cycle for calling an internal or library function F in a nested domain from a parent domain. Dotted arrows represent execution of user space instructions.

2.3.2 High-level Idea

The objective of the secure rewind mechanism is to recover the application's execution state, after a memory defect has triggered, to a prior state before the application's memory has been corrupted (**R2**). Thus, the application can resume its execution and continue to provide its services without interruption (**R1**). To facilitate this, the application is compartmentalized into separate domains that each execute in isolated memory compartments allocated from the process's memory space. Should the execution of code inside a domain fail due to a memory defect, the memory that belongs to the domain must be considered corrupted by the attacker. However, since the effects of the memory defect are isolated to the memory belonging to the failing domain (**R3**, **R4**) the application's execution can now be recovered by: 1) discarding any affected memory compartments, 2) rewinding the application's stack to a state prior to when the offending domain began its execution, and 3) performing an application-specific error handling procedure that avoids triggering the same defect again, e.g., by discarding the potentially malicious input that caused it.

2.3.3 Domain Life Cycle

The overall life cycle of a domain is depicted in Figure 3.1, showing the steps taken to isolate execution in a *nested domain* from its parent. We consider a call to a function or library F that takes one in-memory argument and returns a value. The call is instrumented by code for Secure Domain Rewind and Discard, which first creates a new domain ❶ and then allocates separate stack and heap memory ❷. Then the

caller's *execution context*, containing information such as register values, including the stack and instruction pointers, and signal mask, is saved for later use by the rewind mechanism ❸ (in a manner similar to C `setjmp()` [128]). Next, the input argument is copied onto the new heap ❹ and the domain transition step ❺:

- updates the hardware-enforced memory access policy: grant access to the new domain's memory areas and protect all other memory, global data is made read-only;
- switches execution to the stack of the new domain.

After entering the newly spawned domain, all subsequent code will run in that domain until the next transition. Here, we just call *F* ❻ which may result in one of two outcomes: A *normal domain exit* occurs when *F* runs to completion and returns back to the call site without any errors. Execution resumes in the parent domain ❼, the result is stored in the return variable ❽, and the nested domain is deleted ❾. At last, control transfers to developer-provided handler code for normal exits ❿.

An *abnormal domain exit* occurs if *F*'s code tries to access memory past the confines of the domain's memory area, or a possible run-time attack is detected by defense mechanisms ⓫. The execution of the domain is halted and privileges of the parent domain are restored ⓬. Application execution is resumed by restoring the calling environment from the information stored prior to invoking the offending domain ⓭, effectively rolling back application state to the point before the domain started executing. The failing domain is deleted and its memory is discarded ⓮. Control transfers to a custom error handler ⓯.

In general, after an abnormal domain exit, the application is expected to take an alternate action to avoid the conditions that led to the previous abnormal exit before retrying the operation. For example, a service-oriented application can close the connection to a potentially malicious client.

Rewinding the application state is limited to the state of application memory. Operations that have side-effects on an application's environment, e.g., reading from a socket, are still visible to the application after rewinding. Generally, different software architectures may require different design patterns for secure rewinding. We highlight some of these patterns below.

2.3.4 Domain Types and Patterns

As part of process initialization, all application memory, including stack, heap, and global data, are assigned to the *root domain*, which forms the initial isolated domain where an application executes. Application subroutines are compartmentalized into

nested domains from which rewinding can be performed at any point during execution. As the names suggests, domains can be nested in the sense that several domains can be entered subsequently starting from the root domain, each with a dedicated rewind procedure (cf. Section 2.3.5). While read-only access from any nested domain to data in the parent domain may be allowed, writable access is forbidden in order to contain any memory safety violations to the nested domain. By default, nested domains have read access to the root domain to enable reading global variables. We envision two flavors of isolation with rewinding for application subroutines:

- **Protecting the application from a subroutine:** Code that may have unknown memory vulnerabilities, e.g., third-party software libraries, can be executed in a nested domain by instrumenting calls to the functionality so that they execute in their own domain and may be rewound in case memory-safety violations are detected.
- **Protecting a subroutine from its caller:** application code operating on sensitive data such as cryptographic keys can be isolated from vulnerabilities in its callers, preventing the leak and loss of such data. For example, encryption, decryption, or key derivation functions from the OpenSSL library can be isolated in their own nested domain to protect the application's cryptographic keys if a fault occurs in a calling domain. Listing 2.2 in Section 2.4.1 shows a concrete example of isolating OpenSSL.

To support the two application scenarios described above, we identify two design patterns for nested domains. The type of a domain determines how it continues its life cycle, in particular what happens to it upon normal domain exit.

Persistent Domains A persistent domain retains all its memory areas after the application's execution flow returns to the parent domain. Another code path may enter the persistent domain again, at which point access to memory areas that belong to the persistent domain is again allowed. Practically, in Figure 3.1, creation step ❶ is only needed for the first invocation and deleting the domain in step ❾ is omitted.

Modules that maintain state information across invocations should be isolated in a persistent domain so that their state is not lost after normal domain exits, e.g., some software libraries may encapsulate such state by creating "context" objects that persist across calls. Distinct contexts can be compartmentalized by ensuring each object is allocated in different persistent domains. An example of this pattern is OpenSSL which can be instantiated multiple times within an application using different contexts and associated key material. Cryptographic keys of one context remain isolated in memory from others if each concurrent context is assigned separate persistent domains.

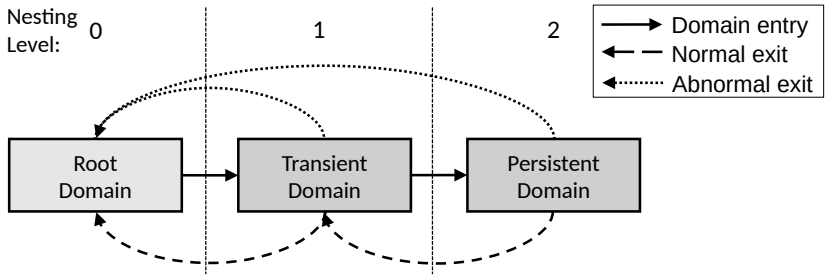


Figure 2.2: Deeply nested domains. Normal exits occur in reverse domain entering order. Abnormal exits may deviate from that: both persistent and transient domain rewind to root domain.

On abnormal exits from any domain, the rewind mechanism is triggered and all state of the domain is discarded. This may have serious repercussions for an application, if the program state depends on data isolated in a persistent domain. When, for example, the abnormal exit leads to the loss of session keys for a TLS connection, the application may only be able to recover by re-initializing the affected cryptographic context and close all connections that were using the now lost keys.

The parent domain may or may not be given access to a persistent nested domain's memory. For instance, in the case of cryptographic libraries, access to the persistent domain's memory from any other domain should be blocked to protect sensitive data stored by the library. In such cases data cannot be directly passed between the caller and callee in distinct domains and shared data, e.g., call arguments and results need to be copied between domains via a designated shared memory area. This is similar to how data is passed between different protection domains in, e.g., Intel Software Guard Extensions (SGX) enclaves [4].

Transient Domains Memory areas assigned to transient nested domains persist until the application's execution flow returns from such a domain to the parent domain at which point the stack as well as *unused* heap memory areas assigned to the transient nested domain are discarded. For this the rewind mechanism needs to keep track of memory allocations in a nested domain. Any allocated memory in a transient nested domain's heap area can be merged back to the parent domain's heap area or discarded upon a normal domain exit, depending on the specific application scenario. Generally, transient domains are most useful for functionality that is called only once during a typical program invocation.

2.3.5 Domain Nesting and Rewinding

Domain nesting enables creating new isolated domains within others. Each nested domain has exactly one parent domain, which is responsible for creating the nested domain. All domains may have zero or more nested child domains, i.e., nested domains may be created by already nested domains.

Transient and persistent-style domains can be nested with each other. One example of a domain nesting configuration is illustrated in Figure 2.2. Here, the first level of domains is transient and its subsequent nested domain can be persistent. This setup allows developers to simplify error handling by directing a rewind from the more deeply nested persistent domain to also return to the recovery point established for the transient domain. On the other hand, abnormal exits from the transient domain do not affect the nested persistent domain and leave it to developer-provided error handling routines to decide if the persistent domain needs to be destroyed as well.

At an abnormal domain exit, the rewind can occur from any nesting level to a lower one as required by an application, but an abnormal root domain exit terminates the program.

2.3.6 Multithreading

The secure rewind mechanism supports POSIX threads. In most cases, threads need to communicate with each other using shared memory, hence they need to access root domain memory. Consequently, it would not be possible to isolate two threads completely from each other or from the main process.

Nevertheless, it is still possible to isolate partial code paths within the thread to separate domains, i.e., each thread can still create nested domains with stacks and heaps that are isolated from the root domain and other nested domains. Hence, each thread may recover via rewinding from errors in such nested domains. If one of the threads suffers an abnormal exit from the root domain, the rewind mechanism cannot recover other threads and the application must be terminated.

Threads have shared access to global and root domain heap memory, to per-thread stack areas and thread-local storage. A shared root domain allows a higher number of parallel threads to be supported, if the available domains provided by the underlying memory protection mechanism is limited, as only threads that instantiate nested domains consume domain slots. However, one could allow the developer to configure stricter, non-uniform access privileges between threads.

Table 2.1: SDRaD API. udi: user domain index.

API Name	Arguments	Description
① `sdrad_init()`	udi, options	Initialize Domain *udi*
② `sdrad_malloc()`	udi, size	Allocate *size* memory in domain *udi*
③ `sdrad_free()`	udi, adr	Free memory at *adr* in domain *udi*
④ `sdrad_dprotect()`	udi, tddi, PROT	Set domain *udi*'s access permissions to *PROT* on target data domain *tddi*
⑤ `sdrad_enter()`	udi	Enter Domain *udi*
⑥ `sdrad_exit()`	—	Exit Domain *udi*
⑦ `sdrad_destroy()`	udi, options	Destroy Domain *udi*
⑧ `sdrad_deinit()`	udi	Delete return context of Domain *udi*

2.4 Prototype Implementation

We implement the concept of secure domain rewind and discard as *SDRaD* [83] – a C-language Linux library for the 64-bit x86 architecture using PKU as the underlying isolation primitive. The library provides application programming interfaces (APIs) to control the life cycle of domains. *SDRaD* does not require Linux kernel patches beyond those potentially needed by run-time defense mechanisms. The Linux kernel supports PKU from version 4.9. Further details on the implementation are provided as a technical report [88].

2.4.1 SDRaD API

Developers use the SDRaD API calls shown in Table 2.1 to flexibly enhance their application with a secure rewind mechanism, accounting for the design patterns described above.

Domains are initialized by `sdrad_init()`① where the developer chooses a unique index to reference the domain in future API calls. *Execution* and *data* domains may be created, where the latter may hold shareable data pages but cannot execute code. For execution domains we further distinguish domains that are *accessible* or *inaccessible* to their parent, and whether an abnormal domain exit should be handled in the parent or grandparent domain. A domain can only be initialized once per thread (unless it is deinitialized or destroyed before by the programmer) and the point of initialization for execution domains marks the execution context to which control flow returns in case of an abnormal domain exit.

The API call's return value fulfills two roles. When the domain is first initialized, it returns *OK* on success or an error message, e.g., if the domain was already initialized

in the current thread. On abnormal domain exit, control flow returns another time from the init function and the return value signifies the index of the nested domain that failed and was configured to return to this point. This means that error handling for abnormal domain exits needs to be defined in a case split on the return value of the `sdrad_init()`① function.

After initialization, memory in an execution or data domain can be managed using `sdrad_malloc()`② and `sdrad_free()`③, e.g., to be able to pass arguments into the domain. Note that this is only allowed for child domains of the current domain that are accessible. For inaccessible domains, a shared data domain needs to be used to exchange data. Using `sdrad_dprotect()`④, access permissions to a data domain can be configured for child domains.

An execution domain initialized in the current domain can be entered and exited using `sdrad_enter()`⑤ and `sdrad_exit()`⑥. This switches the stack and heap to the selected domain and back, and changes the memory access permissions accordingly. Currently, the SDRaD does not copy local variables on such domain transitions. Such variables need to be passed via registers or heap memory.

Supporting the transient domain design pattern, child domains can be deleted using `sdrad_destroy()`⑦, with the option to either discard the domain's heap memory or, if accessible, merge it to the current domain. The persistent domain pattern is then implemented by simply not destroying the domain after exiting it, so that it can be entered again.

An important requirement is that a nested execution domain needs to be destroyed before the function which initialized that domain returns. Otherwise, the stored execution context to which to return to would become invalid as it would point to a stack frame that no longer exists. To provide more flexibility, `sdrad_deinit()`⑧ allows to just discard a child domain's execution context but leave its memory intact. Before entering the domain again, it needs to be re-initialized, setting a new return context for abnormal exits.

Usage Example We implement the domain life cycle shown in Figure 3.1 for a function *F* that receives pointer `arg` to an object of size `size` as input and returns an integer-sized value. In the example, we first initialize a new accessible execution domain `udi_F` for function *F*①. If an abnormal exit occurs in that domain, control returns here, so we save the error code in `err`. If initialization succeeded, we allocate a local variable `r` in a register to retrieve the return value later ②. We also allocate memory for the input argument at `adr` in the new domain ③. Since that domain is accessible, we can copy the argument directly from the parent domain ④. Afterwards, we enter the nested domain ⑤ and invoke *F* on the copy of the argument, saving the return value ⑥. After exiting, we are back in the parent domain ⑦. We can copy the return value to the

```
int err = sdrad_init(udi_F, EXECUTION_DOMAIN |
                     ACCESSIBLE | RETURN_HERE);   ①
if (err == OK ) {
   // prepare passing return value and argument
   register int r asm ("r12");  ②
   register void *adr asm ("r13");
   adr = sdrad_malloc(udi_F, size);  ③
   if (!adr && size>0) { return MALLOC_FAILED; }
   if (size>0) { memcpy(adr, arg, size); } ④
   sdrad_enter(udi_F);  ⑤
   // invoke F on copy of argument and save return value
   r = F(adr); ⑥
   sdrad_exit(); ⑦
   if (ret) { *ret = r; } ⑧
   sdrad_free(udi_F, adr); ⑨
   sdrad_destroy(udi_F, NO_HEAP_MERGE); ⑩
}
return err;
```

Listing 2.1: Using API calls (orange) to call `F(arg,size)` in its own domain (pseudocode for Figure 3.1, error handling omitted).

desired location (which is inaccessible to the nested domain) ⑧. We free all temporary memory ⑨ and destroy the nested domain, freeing its remaining memory ⑩. As a side note, calling `sdrad_free()` is actually redundant here; `sdrad_destroy()` would free this memory as well with the `NO_HEAP_MERGE` option. Finally, we return `OK` in case of normal domain exit, or the error code otherwise. Users of the shown isolated version of *F* can then define their own error handling depending on this return value.

OpenSSL Listing 2.2 shows an `EVP_EncryptUpdate()` wrapper for OpenSSL that implements the persistent domain pattern in Section 2.3.4. Here, OpenSSL allocates its data, such as the context (`ctx`) ❶ in a domain which is inaccessible from its parent. The caller can hold a pointer to `ctx`, but the object itself is inaccessible to the parent domain. Arguments are copied in via a data domain ❷. The wrapped function must read buffered input and write its output to its parent domain. There are three possible design choices for passing data between the respective domains: 1) the OpenSSL domain has read-only access to the parent, i.e., it's called from the root domain; input can be read directly, but output must be copied through the data domain used for argument passing ❸, ❺ 2) the parent domain is inaccessible to the OpenSSL and it must copy both input and output via the data domain used for argument passing ❸, ❹, ❺ 3) the parent domain is responsible for setting up a shared data domain between the respective domains and the wrapper can access the shared area directly via the argument pointers.

This persistent domain can be combined with a transient domain as shown in Figure 2.2 to 1) encapsulate the pointer to `ctx` within an outer domain, 2) protect the root domain from errors in the caller, e.g., an `out` buffer of insufficient size, and 3) simplify error handling for the OpenSSL domain.

```
1  int __wrap_EVP_EncryptUpdate(EVP_CIPHER_CTX *ctx❶, unsigned char *out, int *outl, const unsigned char *in, int inl) {
2      register evp_encrypt_update_args_t *args asm ("r12"); // holds copied function arguments and return value, is kept in
3                                                            // a callee-saved register (r12) to remain accessible after
4      ...                                                   // sdrad_enter(OPENSSL_UDI) changes the domain stack
5
6      args = sdrad_malloc(OPENSSL_DATA_UDI, sizeof(evp_encrypt_update_args_t));
7  ❷   args->ctx = ctx;                          // copy ctx from current domain to shared data domain
8      args->inl = inl;                           // copy inl from current domain to shared data domain
9
10
11     if (out != NULL && inl >= 0) {             // inl + cipher_block_size is upper bound for yet unknown output size
12         args->out = sdrad_malloc(OPENSSL_DATA_UDI, inl + cipher_block_size);
13     } else { args->out = NULL; }
14                                                → ❸ Set up output buffer. Replaced with args->out = out if out points to shared data domain.
15
16     if (in != NULL && inl >= 0) {
17         args->in = sdrad_malloc(OPENSSL_DATA_UDI, (size_t)inl);
18         memcpy(args->in, in, inl);             // copy in from current domain to shared data domain
19     } else { args->in = NULL; }
20                                                → ❹ Set up input buffer. Replaced with args->in = in if in points to a readable domain.
21     sdrad_enter(OPENSSL_UDI);                  // execute real EVP_EncryptUpdate in inaccessible domain
22     args->ret = __real_EVP_EncryptUpdate(args->ctx, args->out, &(args->outl), args->in, args->inl);
23     sdrad_exit();
24
25     *outl = args->outl;                        // copy out outl value from shared data domain
26     if (out != NULL) {                         // copy out encrypted data from shared data domain
27         memcpy(out, args->out, (size_t)*outl);
28     }
29                                                → ❺ Copy out. Omitted if out points to a shared data domain.
30     ...
31 }
```

Listing 2.2: Wrapper function for `EVP_EncryptUpdate()` that executes OpenSSL in a persistent nested domain (excerpt). Data is passed between parent and nested domain via shared data domain `OPENSSL_DATA_UDI`. Error handling is omitted for brevity.

2.4.2 Implementation Overview

The SDRaD implementation [83] has four components: 1) a hardware mechanism for enforcing in-process memory protection, 2) an isolated *monitor data domain* that houses control data for managing execution and data domains, 3) initialization code that is run on startup of an application that is linked to SDRaD, setting up the monitor domain and memory protection, and 4) a trusted reference monitor that realizes the SDRaD API calls with exclusive access to the monitor data domain.

Memory Protection SDRaD uses PKU (cf. Section 2.2.3) as a hardware-assisted SFI mechanism to create different isolated domains within an application governed by different memory access policies. When a domain is created, a unique protection key is assigned for it. At each domain transition, the protection key rights register (PKRU) is updated to grant access to memory areas as permitted for the newly entered domain, and to prevent access to other memory areas. Our evaluation platform supports Intel PKU, hence it allows us to manage up to 15 isolated domains at a time for each process. Software abstractions for MPK, like libmpk [172], increase the number of available domains at the cost of falling back to the much slower `mprotect` system call.

SDRaD Control Data SDRaD stores global control data in the monitor data domain for keeping track of, e.g., the registered domain identifiers and the protection key usage. It also stores per-thread information for domains such as stack size, heap size, parent domain, and memory access permissions. To support abnormal domain exits, the currently executing domain and the saved execution contexts are stored, too.

Initialization An application is compiled with the SDRaD library to use the rewind mechanism. Then, a constructor function is executed before `main()` to assign all application memory to the initial isolated domain as a root domain associated with one of the PKU keys. It also initializes SDRaD global control data where the default stack and heap size for domains is configurable through environment variables. Furthermore, it sets the root domain as active domain, to be updated at domain transitions by the reference monitor, and finally initializes a signal handler. For the multithreading scenario, SDRaD has a thread constructor function as well, that is executed before the thread start routine function to assign a thread memory area to a domain and associate it with one of the protection keys.

Reference Monitor The reference monitor records domain information in SDRaD control data. It also performs domain initialization, domain memory management, and secure domain transitions, including updating the memory access policy, and saving

and restoring the execution state of the calling domain. Only the reference monitor may update the PKRU register to gain access to the monitor data domain. The monitor code is executed using the stack of the nested domain that invoked it.

Error Detection Memory access violations are generally reported to userspace software either via 1) a segmentation fault (`SEGFAULT`) signal, e.g., when a domain tries to write past the confines of its memory area, or 2) calls to runtime functions inserted by instrumentation, e.g., GCC's stack protector calls `__stack_chk_fail()` if a stack guard check fails. Standard glibc versions of this function terminate the application, but we replaced it with our own implementation, allowing SDRaD to respond to stack guard violations. Moreover, during process initialization, SDRaD sets up its own signal handler for the `SEGFAULT` signal, where the cause for a segmentation fault is given by a signal code (`si_code`) provided by the runtime to the signal handler [129], e.g., PKU access rule violations are reported by `SEGV_PKUERR` signal code. In Linux, the `SEGFAULT` signal is always delivered to the thread that generated it. If the SDRaD signal handler is triggered by a violation in a nested domain, it causes an abnormal domain exit. For faults occurring in the root domain or being attributed to a cause the SDRaD signal handler cannot handle, the process is still terminated. SDRaD can be extended to incorporate other run-time error detection mechanisms, such as Clang CFI [40] or heap-based overflow protections (e.g., heap red zones [195]), improving the recovery capabilities. Probabilistic and passive protections such as ASLR do not detect but hinder the exploitation of memory safety violations. Our rewind mechanism is compatible with ASLR, as domains are created at run-time.

Rewinding Secure rewinding from a domain is achieved by the reference monitor saving the execution context of the parent domain into SDRaD control data when that domain is initialized. SDRaD uses a `setjmp()`-like functionality to store the stack pointer, the instruction pointer, the values of other registers, and the signal mask for the context to which the call to `sdrad_init()` returns. Note that we cannot simply call `setjmp()` within `sdrad_init()` because that execution context would become invalid as soon as the initialization routine returns. On an abnormal domain exit, the saved parent execution state is used to restore the application's state to the initialization point prior to entering the nested domain by using `longjmp()`. This rewind lets the application continue from that last secure point of execution that is now redirected to the developer-specified error handling code. To simplify programming under these non-local goto semantics [128], we only allow to set the return point once per domain and thread. Moreover, the convention that a domain needs to be destroyed or deinitialized before the function that initialized it returns, ensures that the saved execution context is always valid.

2.4.3 Memory Management and Isolation

Our recovery mechanism is enabled by creating different domains within an application and ensuring that a memory defect within a domain only affects that domain's memory, not the memory of others. Using the underlying SFI mechanism based on PKU, domains are mutually isolated.

Global Variables We modify the linker script to ensure that global variables are allocated in a page-aligned memory region that can be protected by PKU. At application initialization, all global variables are assigned to the root domain, but if that one was completely isolated, globals would not be accessible to nested domains. As a compromise, we make the root domain by default read-only for all nested domains. Write access to global data may then be achieved by allocating it on the heap of a shared data domain, referenced by a global pointer. Note that this approach breaks the confidentiality of the root domain towards nested domains. As our main goal is integrity, and confidential data can still be stored and processed in separate domains, we find it a reasonable trade-off for SDRaD.

Stack Management SDRaD creates a disjoint stack for each execution domain to ensure that the code running in a nested domain cannot affect the stacks of other domains. The stack area is allocated when first initializing a domain and protected using the protection key assigned to that domain. As an optimization, we never unmap the stack area, even when the domain is destroyed, but keep it for reuse, i.e., when a new domain is initialized. At each domain entry, we change the stack pointer to the nested domain stack pointer and push the return address of the `sdrad_enter()` call, so that the API call returns to the call site using the new stack. Then we update the PKRU register according to memory access policy for that domain. A similar maneuver is performed when switching back to the parent domain's stack via `sdrad_exit()`.

Heap Management Because heap isolation in SDRaD requires memory management with strict guarantees that allocations within a domain are satisfied only from memory reserved for that domain, we opted to use an allocator that natively supports fully disjoint heap areas instead of the default glibc GNU Allocator [242]. For our implementation, we chose the Two-Level Segregated Fit (TLSF) allocator [43, 147]. TLSF is a "good-fit", constant-time allocator that allocates memory blocks from one or more pools of memory. Each SDRaD domain is assigned its own TLSF control

structure and memory pool that correspond to the domain's subheap. The initial size of these pools is configurable via an environmental variable.

We interpose functions from the `malloc()` family with wrappers and place SDRaD before libc in library load order. Upon first call to memory management within a domain, its heap is initialized and the memory pool is associated with the domain's protection key. Having independent subheaps allows the developer to either discard or merge a domain's subheap with the parent's when the former domain is destroyed. Note however, that subheaps are never merged back after abnormal exits, as the data must be considered corrupted. We extended TLSF with a straight-forward implementation of subheap merging but omit a detailed description for brevity.

2.5 Case Studies

We evaluate the performance of SDRaD with three different real-world case studies: Memcached, NGINX, and OpenSSL. Two aspects are evaluated: 1) rewinding latency on an abnormal exit, 2) performance impact of the isolation mechanism. We run our experiments on Dell PowerEdge R540 machines with 24-core MPK-enabled Intel(R) Xeon(R) Silver 4116 CPU (2.10GHz) having 128 GB RAM and using Ubuntu 18.04, Linux Kernel 4.15.0. We compiled Memcached and NGINX with `-O2` optimizations, `-pie` (for ASLR), `-fstack-protector-strong`, and `-fcf-protection`.

2.5.1 Memcached

Memcached [149] is a general-purpose distributed memory caching system, which is used to speed up database-driven applications by caching database content. To do so efficiently, Memcached stores its state in non-persistent memory; after termination and restart, clients must start over and resend a large amount of requests to return to the situation prior to the restart. Even in real-world deployments with built-in redundancy and automatic remediation, small outages can take up to a few minutes to re-route requests to an unaffected cluster [161]. Several studies propose to use low latency persistent storage for Memcached [145, 256] but these solutions come with a non-negligible performance overhead. As availability and resilience of Memcached to unforeseen failures is of high importance, it is a worthwhile target for hardening with SDRaD. The main thread in Memcached accepts connections and dispatches them among worker threads to handle related requests. Memcached uses a hash table to map keys to an index and slab allocation to manage the in-memory database. Memcached has an event-driven architecture, handling each client request as an event. The clients can send get, set, and update commands with key and value arguments. To handle a request, command parser subroutines in Memcached classify the client request, then

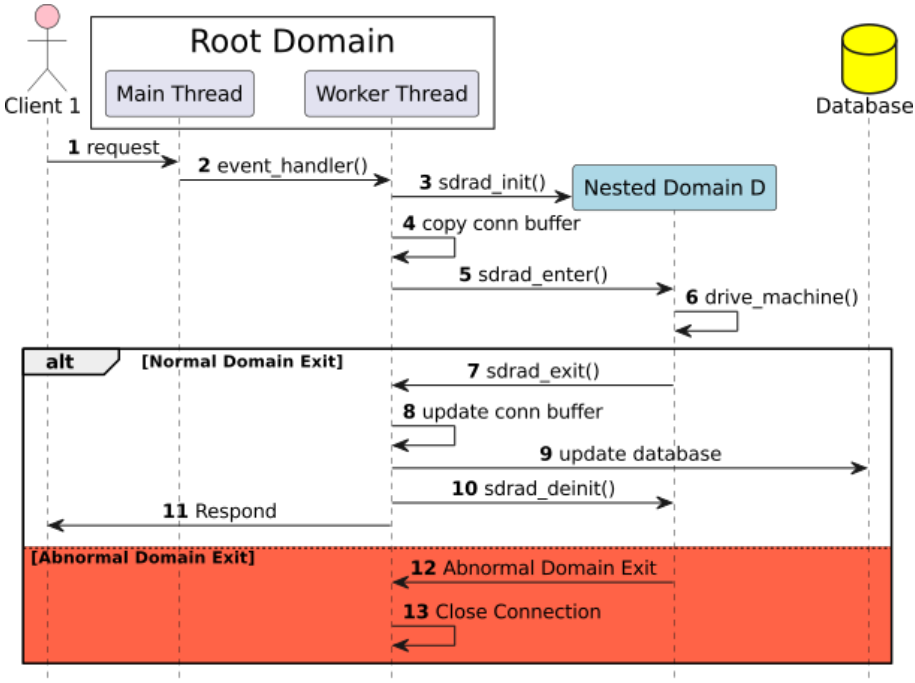


Figure 2.3: Sequence diagram of client request for Memcached with SDRaD. Function `drive_machine()` is executing in nested domain *D* until normal or abnormal domain exit.

the key-value pairs are fetched from, inserted in, or updated in the database. If a client event contains a malicious request leading to memory corruption, the database and hash table, as well as the complete application memory area, are corrupted and Memcached must be restarted. As a result, one malicious request affects the availability of the service to all clients.

Memcached with SDRaD We propose that each client event should be handled in a nested domain. In case of memory corruption, the abnormal domain exit occurs in the nested domain, we discard the related nested domain contents and come back to the root domain securely. Memcached closes the related connection, and it can continue its execution, handling another client request without restarting. Figure 2.3 shows a sequence diagram of Memcached with SDRaD. We configure SDRaD with a shared root domain (see in Section 2.3.6), because the main thread needs to communicate with the worker threads. Each event is handled using the `drive_machine()` (**6**) function with a corresponding connection buffer. We isolate this function using the SDRaD API. Recall from Section 2.3.4 that nested domains may only have read access to data that belongs

to the parent domain. Nevertheless, certain subroutines, such as `drive_machine()`, need to update shared state residing in a parent domain, e.g., the connection buffer. As a solution, the event handler that calls `drive_machine()` initializes an accessible, nested domain *D* (**3**) and makes a *deep copy of the connection buffer* that is made available to `drive_machine()` (**4**). It then enters *D* (**5**) and calls `drive_machine()` (**6**) to handle the client request, working on a copy of the connection buffer. After successfully handling the request, it exits from *D* (**7**), the original connection buffer in the parent domain is updated with any changes present in the shared copy (**8**). On abnormal domain exit, the copied connection buffer is discarded. Since the event handler returns after handling the request, we need to invalidate the saved execution context of *D*. As `drive_machine()` does not allocate persistent state in *D*, we could use the transient domain pattern and destroy *D*. However, for efficiency, we reuse the copied connection buffer used by the domain, hence `sdrad_deinit()` (**10**) is used.

The `drive_machine()` function also needs to read and write the hash table and database to perform look-ups, insertions, and updates. We allocate it in a dedicated data domain, accessible by the nested domain of each thread.

To allow inserts and updates, we wrap the `slabs_alloc()` function, that normally returns a pointer to a memory area in the database, to return a copy of that area to insert the key-value pair. Similarly, we wrap `store_item()` which stores new data and updates the hash table. Each event handler first performs its operation on a copy of the corresponding item. On normal domain exits from *D*, we insert the key-value pair to the database, and update the hash table (**9**). On an abnormal domain exit (**12-13**), the corrupt key-value pair is discarded along with all other domain memory. Note that this solution delays updates to the database. However, due to the atomic nature of the Memcached requests, consistency is not affected.

Our changes were limited to two source files in Memcached and 484 new lines of wrapper code. In total, the changes amounted to ~550 LoC of the 29K source lines of code (SLOC) code base (~2%).

Memcached uses a shared mutex to synchronize worker threads. Here, our copying mechanism for shared data does not work, because it would hide concurrent accesses to the mutex and break the synchronization. We opted to create a separate data domain for the mutex that every worker can access. See Section 2.6 for a security discussion of this scheme.

Rewinding Latency We reproduced CVE-2011-4971 [54] to verify the SDRaD rewind mechanism and compiled Memcached v1.4.5 with SDRaD. This CVE crashes Memcached via a large body length value in a packet, creating a heap overflow but SDRaD ensures that this overflow is limited to current execution domain, hence it triggers the domain violation and an abnormal domain exit occurs. We measured the

mean latency of abnormal domain exit starting with catching `SEGFAULT` until after we close the corresponding connection to 3.5μs (σ=0.9μs). For comparison, in our experiments the restart and loading time for 10GiB of data into Memcached was about 2 minutes. Thus, an attacker who successfully launches repeated attacks could knock out the Memcached service without rewinding (DoS). While this is clearly dominated by the loading time, even applications without such volatile state, but ultra-reliable low-latency requirements, can benefit from rewinding. For reference, we measured the mean latency to restart the Memcached container automatically at about 0.4s (400000μs, σ=19000μs).

Performance Impact We used the Yahoo! Cloud Service Benchmark (YCSB) [44] to test the impact of SDRaD on Memcached performance. YCSB has two phases: a loading phase that populates the database with key-value pairs, and a running phase which performs read and update operations on this data. We used workloads with sizes of 1KiB, with a read/write distributions of 95/5. For our measurements, we stored 1×10^7 key-value pairs (1KiB each) and performed 1×10^8 operations on those pairs. Operations were performed with a Zipfian distribution over the keys. We compiled Memcached v1.6.13 and evaluated the performance with the TLSF allocator and with SDRaD as described in Section 2.5.1. We compare the results against YCSB on unmodified Memcached. Figure 2.4 shows the load and running phase throughput (operations/second) of the three versions for 1, 2, 4, and 8 workers over 5 benchmark runs. Each thread was pinned to separate CPU cores. We used 32 YCSB clients with 16 threads pinned to separate cores for each test. We fully saturated Memcached cores for 1, 2, and 4 threads but were unable to reach saturation for 8 threads. We concluded that TLSF has negligible impact on throughput in all our tests (< 1%). For Memcached augmented with SDRaD the load and running phase overhead is 2.9% / 4.1%, respectively, for 4 threads, and 4.5% / 5.5% for 2 threads. SDRaD introduced a worst-case overhead of 7.0% / 7.1% for a single thread. We measured a performance degradation of < 4.1% for 8 threads but lack confidence in the soundness of that result as the CPU was not saturated. We measured the memory overhead of SDRaD by comparing the maximum resident set size (RSS) after the YCSB load phase and of Memcached with SDRaD to the baseline. The mean RSS increase is 0.4% (σ=171KiB).

2.5.2 NGINX

NGINX [158] is an open-source web server implemented as a multiprocessing application with a master process and one or more worker processes. The master process is responsible for maintaining the worker processes that handle client HTTP requests for several connections at a time. If a malicious client request leads to memory corruption, the worker process may crash and the master process restarts it, however all active connections of that worker are lost. Due to its complexity and exposure

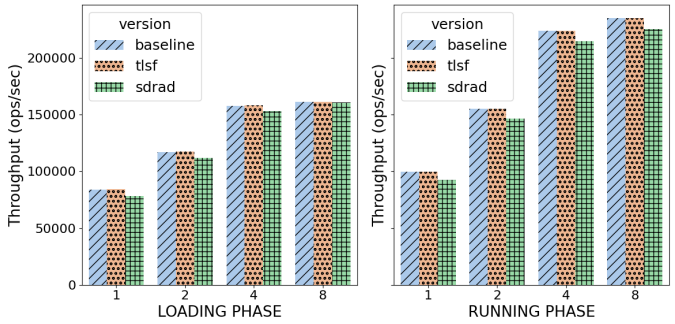


Figure 2.4: Throughput of different Memcached instrumentations for different numbers of threads.

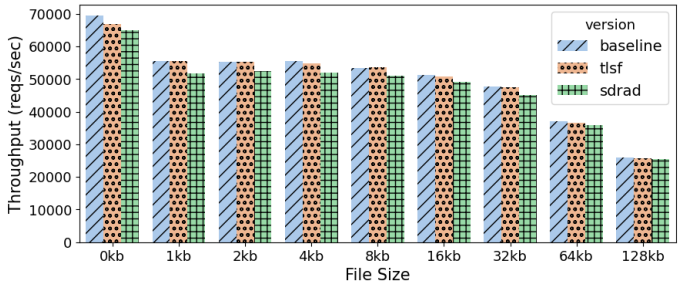


Figure 2.5: Throughput of different NGINX instrumentations with one worker for different file sizes.

to untrusted inputs, the HTTP parser is a vulnerable component of NGINX. Similar to [247], we propose that each client HTTP request is parsed in a nested domain. Thus, if a memory corruption is detected in the parser, an abnormal domain exit occurs and we discard the related nested domain content to come back to the root domain securely without restarting the worker process. The related connection is closed, but all other connections are unaffected.

We sandboxed the HTTP parser (`ngx_http_parser.c`) to execute in an accessible permanent nested domain, by instrumenting all NGINX parser functions using the SDRaD API. NGINX creates a temporary memory pool for each client request to hold a *request buffer*. We direct the allocation of these pools to a separate data domain that is accessible by the nested domain. The request buffer data structure links back to header data and URI data in the connection buffers. To protect this root domain data and make it accessible to the parser, it is copied into the nested domain and the results are copied back on domain exit. The NGINX Parser executes multiple phases, e.g., parsing request lines or headers. Thus, domain transitions occur repeatedly in

one request. On an abnormal domain exit, it is not important which parser phase has corrupted the memory: we always close the corresponding connection. Hence we save as execution context the first entry point of the NGINX parser to come back to at an abnormal domain exit. Our changes were limited to one file in NGINX and 195 new lines of wrapper code. In total, the changes amount to ~220 LoC of the 150K SLOC code base (0.15%).

Rewinding Latency We reproduced CVE-2009-2629 [53] to verify the rewind mechanism and compiled NGINX v.0.6.39 with SDRaD. The CVE causes a buffer underflow in the linked connection buffer data. By having the parser operate on copies of that data in the nested domain, the underflow triggers a domain violation and thus an abnormal domain exit. We measured the latency of the abnormal domain exit starting from catching SEGFAULT to accepting a new connection. The mean latency is 3.4μs (σ=0.67μs). We compared it with restarting the worker process by the master process for reference. The mean latency is 996μs (σ=44μs). It should be noted however, that avoiding service disruption for other clients by preserving all connections handled by the worker process is arguably the more important benefit of SDRaD here.

Performance Impact We measured the SDRaD overhead to connect to NGINX remotely over keep-alive HTTP connections using ApacheBench tool. Each test has 75 concurrent connections and all clients request the same file size ranging from 0KiB to 128KiB. Figure 2.4 shows mean throughput (requests/second) of the three versions of NGINX with one worker process for different file sizes over 5 benchmark runs. We compiled NGINX v.1.23.1 and compared it to NGINX with TLSF allocator and SDRaD. The latter introduced overheads between 1.6% (128KiB) and 6.5% (1KiB). We scaled the number of workers for NGINX with SDRaD and observed that the overhead is independent of that number, as expected. We measured the memory overhead of SDRaD from the maximum RSS after benchmarking the 128-KiB file size with four worker processes, and comparing the RSS of NGINX with SDRaD to the baseline. The mean RSS increase is 3.06% (σ=50KiB). Profiling domain switching, we observed that 30% – 50% of the cost comes from writing the PKRU register, which flushes the processor pipeline [172, 224].

2.5.3 OpenSSL

SDRaD allows for isolating a library without changing it, enabling later integration into applications. As discussed in Section 2.3.4, we may either protect the application from the library or the library from the application. To demonstrate the first case, we reproduced CVE-2022-3786 in OpenSSL 3.0.6 [162]. When OpenSSL processes X.509 client certificates, the CVE causes a buffer overflow that puts an arbitrary number

on the stack and may cause denial of service by crashing the application. It can be detected by stack canaries, so we isolated the vulnerable X.509 certification verification API of OpenSSL and compiled NGINX with that library and SDRaD. We verified that the CVE triggers a rewind and NGINX closes the related connection and reinitializes the OpenSSL domain before continuing execution. We added ~14 LoC in NGINX code base, ~18 LoC in OpenSSL code, as well as ~140 new lines of wrapper code.

To demonstrate protecting OpenSSL from the rest of the program, we instrumented OpenSSL 1.1.0 according to all three design choices explained in Section 2.4.1. We evaluated the performance impact by adapting the built-in OpenSSL speed benchmark and running the *aes-256-gcm* cipher via the `EVP_EncryptUpdate` function (cf. Listing 2.2) for 3s, measuring the number of encryptions. As expected, `memcpy` operations cause notable performance overhead and the third option, a parent-managed shared domain, performed best. Even without copy operations, SDRaD substantially degraded the performance of cryptographic operations for small input sizes (4% to 80%). For more realistic input sizes ≥32KiB we did not measure any statistically significant overhead (< 2%).

2.6 Discussion

Security Evaluation Our primary security requirement is

R1 The mechanism must allow the application to continue operation after system defenses (**A2**) detect an attack.

Our proposal satisfies this requirement by compartmentalizing applications into isolated domains where an attack against a child domain can be detected, which leads to the termination of that domain, while the parent domain is informed and can continue operation. With respect to attack detection, we assume that an attack or fault will exhibit an illegal memory access that triggers a `SEGFAULT` signal. We then use signal handlers to detect and handle the failure, leading to a termination of the crashed child domain and a rewinding of the parent domain to a well-defined state. Compartmentalization is achieved by implementing the following two requirements:

R2 The mechanism must ensure that the integrity of memory after recovering from the detected attack is maintained.

R3 Run-time attacks that affect one domain must not affect the integrity of memory in other domains.

In our implementation of SDRaD, we use PKU as a mechanism to enforce in-process isolation while facilitating efficient domain switches. The security of this mechanism

critically relies on protecting potential gadgets in the SDRaD implementation that allow an attacker to manipulate the PKRU register.

To guarantee the security of SDRaD, the following orthogonal defenses need to be in place: 1) PKU crucially relies on untrusted domains to not contain unsafe WRPKRU or XRSTOR instructions that manipulate the PKRU register [42]. This can be guaranteed through W⊕X and binary inspection [224]. Alternatively, hardware designs for PKU-like security features restrict access to userspace configuration registers [190]. 2) Since the SDRaD necessarily contains WRPKRU instructions, we must employ a CFI mechanism to protect the API implementation of Reference Monitor and statically ensure that its API does not contain abusable WRPKRU or XRSTOR gadgets. 3) An alternative way to modify PKRU is by utilizing `sigreturn` is described in [42], which can be mitigated by ASLR [46] and requires kernel-level authentication of `sigframe` data [122]. 4) Existing PKU sandboxes do not sufficiently safeguard the syscall interface [42, 189]. Several researchers propose efficient syscall filtering to prevent untrusted domains from invoking unsafe syscalls [189, 230]. With the above security mechanisms in place, SDRaD satisfies our fourth requirement:

R4 The attacker must not be able to tamper with components responsible for isolation or transitions between domains, or data used as part of the rewinding process.

It is possible to implement secure rewind and discard of isolated domains on top of other isolation mechanisms e.g., within Intel SGX to equip enclaves with rewinding or by using capability-based enforcement of isolation, e.g., Capability Hardware Enhanced RISC Instructions (CHERI). Such uses will incur different low-level security requirements and exhibit different performance characteristics. Furthermore, SDRaD is not limited to rely on SEGFAULT handling but could employ different attack oracles that, e.g., trigger when a domain invokes an unexpected system call.

Applicability As highlighted earlier, secure rewind and discard of isolated domains is particularly suited for service-oriented applications that need strong availability guarantees and may hold volatile state like client sessions, TLS connections, or object caches. Redundancy and load balancing can minimize the impact of DoS attacks, but loss of volatile state can still degrade service quality for clients, which our approach mitigates.

As demonstrated by our case study on Memcached (Section 2.5.1), applications such as caching (proxy) servers and web accelerators which exhibit the *cold start problem*, i.e., the system does not reach its normal operation capacity until some time after initial start or restart, can benefit greatly from SDRaD.

A prime target for our mechanism are subroutines and libraries that handle sequences of external, untrusted, unsanitized data, e.g., functions that perform input validation,

JavaScript engines in web browsers, database front-ends, or video, image, and document renderers, due to a heightened degree of exposure towards potential attacks. Isolating such components in their own domain allows recovering from potential memory corruptions. Furthermore, rewindings in long-running services may be reported as incidents to a Security Information and Event Management system, serving as early warning signals of an attack campaign. Blocking malicious clients via firewall rules as a response may then shield the overall system from repeated attacks.

In practice, the specific setup of domains and protections as well as the acceptable trade-off between the performance impact and provided benefit in resiliency will depend highly on application architecture and use case. As such, our design incorporates different options for compartmentalization, aiding adoption of the rewind mechanism in new and existing developments. We see retrofitting applications written in unsafe programming languages as a compelling use case in-lieu of a complete re-write in a memory-safe language.

Applications written in memory-safe languages are inherently protected against the run-time attacks we consider in this work. Nevertheless, even such applications may depend on unsafe libraries called via foreign function interface (FFI). Adapting SDRaD for use with FFI from other programming languages is, however, outside the scope of this work.

Limitations We presented several examples where our approach substantially improved the dependability of relevant applications and where the refactoring required one to three person weeks of a developer not previously familiar with the code base. The effort is comparable to retrofitting applications to make use of trusted execution enviroments (TEEs) such as Intel SGX and the process may be supported by automated tools, e.g., [124]. However, it is clear that not all applications can be easily compartmentalized and refactored to make use of SDRaD. For example, applications that rely on global mutexes may suffer from availability issues when a child domain holding a lock crashes and the lock is not released prior to continuation of the parent domain. Options for resolving this are, e.g., an SDRaD-aware locking mechanism as part of our library, or to employ local locks with well-defined scope instead of global locks. This also aids serializing access, e.g., when domains operate on copies of protected objects. Generally, isolating routines that work on shared state is tricky but possible using the deferred update method demonstrated for Memcached, as long as the isolated routine updates the shared state atomically and at most once.

Another potential issue comes with complex data structures used by target applications. Similar to other strong isolation mechanisms such as Intel SGX, data needs to be copied into the address space of the protection domain [47], which is done by entry wrappers. Both, manual as well as automated generation of these wrappers can be error prone and may hamper security [225]. Generally, domain transitions and domain termination

bear subtle risks. Currently, confidentiality of child domain data is not guaranteed after destroying it and we leave it to the developer to realize such requirements, e.g., scrub sensitive allocations from memory before leaving the domain.

Using SDRaD to increase the availability of long-running services may open up a side-channel attack surface as observable effects of a rewind (e.g., delayed execution) can give an attacker insights into an application's execution. Coupled with the absence of re-randomization of the application's memory layout, an attacker could potentially use this to break probabilistic defenses such as ASLR. A potential protection against such attacks is to force an application restart after a configurable number of rewindings, similarly to probabilistic defenses for pre-forking applications [123].

Moreover, the defense mechanisms used in our approach are not perfect, i.e., memory or control flow may still be compromised by an undetected attack. While our approach is orthogonal to the mechanisms used, the compartmentalization of a program into domains may still help discover subsequent exploitation of such compromise more quickly as malicious memory accesses may potentially cause domain violations.

Ultimately, the security of SDRaD depends on the correctness of our library implementation and further exploration of the attack surface as well as potentially formal verification of our code are envisaged to harden our approach.

2.7 Related Work

Hardware-assisted compartmentalization The idea of using MPK for compartmentalizing applications is not new; PKU in 64-bit x86 is used to augment SFI approaches that generally suffer from high enforcement overheads [194, 232]. Such work falls into two categories: 1) in-process isolation [33, 89, 98, 104, 109, 181, 190, 224, 230, 233], and 2) isolation for unikernels and library OSs [116, 148, 207]. Lack of PKRU access control has been scrutinized for leaving PKU-based schemes vulnerable to bypass of established isolation domains [42, 189, 230]. Proposed countermeasures include code rewriting [109], binary inspection [224], system call filtering [189, 230] and variations on the PKU hardware design [59, 73, 190]. Secure multi-threading has also been considered [34, 104, 215]. However, in-process SFI does not consider how to recover from attacks. This work addresses this gap by introducing capabilities for secure rewinding. Light-Weight Contexts [130] introduce memory management unit (MMU)-based in-process isolation with a notion of rewinding. In difference to our work, [130] requires OS extensions and has unclear performance characteristics. Capability schemes, such as CHERI [234, 244] also enables compartmentalized fault isolation. CompartOS [11] provides recovery capabilities, but unlike SDRaD, it is geared toward safety-critical embedded systems, not CoTS processors.

Checkpoint & restore Existing approaches to checkpoint & restore, such as CRIU [101] provide support for process snapshots that can enable rollback-like functionality. However, checkpoint & restore approaches generally suffer from high overheads due to relying on reproducing process memory, do not consider in-memory attacks in their threat models [133, 155, 239, 250, 254], or target special computing paradigms, such as functions-as-a-service [13, 199]. SDRaD avoids these drawbacks by utilizing in-process isolation to limit the scope of attacks and to ensure the integrity of memory after rewinding.

N-variant Execution N-variant Execution [229] provides resilience by introducing redundancy through running multiple, artificially diversified variants of the same application in tandem and terminating instances that show divergent behavior. While SDRaD shares the goal of improving software resilience, our work targets use cases for which the high cost of replicating computations and I/O across each instance is impractical.

SDRaD In comparison with related work, SDRaD aims at improving dependability of applications on CoTS processors, e.g., in cloud settings, allowing recovery from attacks that compromise heap or stack memory. SDRaD requires no OS changes, incurs comparatively small runtime overheads and enables very fast application recovery, at the expense of a non-trivial but feasible engineering effort to adapt the application. This combination of features makes our approach unique in the area of dependable software.

2.8 Conclusion & Future Work

We presented the novel concept of *secure rewind and discard of isolated domains*, which complements protection mechanisms against memory safety vulnerabilities. It provides a hardening mechanism to recover from detected violations, thus improving the availability and resilience of software applications. At its core, secure rewind uses hardware-assisted in-process memory isolation to sandbox exposed functionality in separate domains: a compromise in one domain cannot spread to other parts of a program's memory. When a compromise is detected by selected defense mechanisms, a rewinding to a previously defined consistent state of the application occurs, enabling error handling and efficiently resuming the application.

We introduced *SDRaD*, our prototype library implementation of secure domain rewind and discard. We demonstrated its applicability to real software by adding it to the multi-threaded Memcached system, multiprocessing NGINX web server, and the OpenSSL library, exhibiting marginal performance overhead. Applications profit not only from

faster recovery times due to the rewind mechanism, but also from avoiding possible service disruption for clients after an application crash and subsequent connection loss. In practice, no compartmentalization strategy will likely be able to recover from each and every fault or attack. Still, providing an amenable, secure, and efficient implementation of the secure rewind mechanism will fill an important gap in the current software security architecture.

A technical report with extended insights into the workings of SDRaD, our prototypic implementation, and further elaboration on the cases studies is available at [88]. Our implementation artifacts are available under a BSD license on GitHub [83]: `https://secure-rewind-and-discard.github.io/`

Chapter 3

Friend or Foe Inside? Exploring In-Process Isolation to Maintain Memory Safety for Unsafe Rust

This chapter consists of ℂ2 and was originally published as:

> ***M. Gülmez**, T. Nyman, C. Baumann and J. T. Mühlberg, "Friend or Foe Inside? Exploring In-Process Isolation to Maintain Memory Safety for Unsafe Rust" in 2023 IEEE Secure Development Conference (SecDev), Atlanta, GA, USA, 2023, pp. 54-66,* https://doi.org/10.1109/SecDev56634.2023.00020.

This chapter addresses $\mathcal{RQ}2$: "How can approaches that address $\mathcal{RQ}1$ be adapted to protect Rust code in multi-language applications?"

Abstract Rust is a popular memory-safe systems programming language. In order to interact with hardware or call into non-Rust libraries, Rust provides *unsafe* language features that shift responsibility for ensuring memory safety to the developer. Failing to do so, may lead to memory-safety violations in unsafe code which can violate safety of the entire application. In this work we explore in-process isolation with Memory Protection Keys (MPK) as a mechanism to shield safe program sections from safety violations that may happen in unsafe sections. Our approach is easy to use and

comprehensive as it prevents heap and stack-based violations. We further compare process-based and in-process isolation mechanisms and the necessary requirements for data serialization, communication, and context switching. Our results show that in-process isolation can be effective and efficient, permits for a high degree of automation, and also enables a notion of application rewinding where the safe program section may detect and safely handle violations in unsafe code.

3.1 Introduction

Rust is an emerging system programming language with memory safety guarantees and performance characteristics close to traditional system programming languages such as C and C++ [173]. Rust is designed to make programs difficult to exploit by attackers by providing compile-time type-checking and memory management based on ownership rules. These static analyses in Rust are conservative: While the compiler will never accept an unsafe program, it may reject safe programs. To work around this incompleteness of compile-time analyses but also to accommodate the need to interact with inherently unsafe low-level operating system interfaces or hardware, Rust provides *unsafe Rust* [105]. In unsafe Rust it is possible to, e.g., dereference a raw pointer or to modify a mutable static variable. Rust then relies on the developer for the correctness of unsafe parts of a program, and failure to ensure correctness will lead to memory errors that may compromise safe parts of the program.

A common use of unsafe Rust is to call C library functions. While the Rust ecosystem is growing, it is not feasible to reimplement common library functionality in Rust right away and the Rust community embraced a foreign function interface (FFI) [105] to conveniently call non-Rust code. Foreign functions are always assumed to be unsafe by the compiler and it obliges the developer to integrate C libraries safely. Common causes of unsafety regarding the FFI are library interfaces that are not thread-safe, pointer arguments that are not completely sanity checked, and the use of raw pointers. The use of FFI is pervasive: According to Li et al. [121] more than 72% of packages on the official Rust package registry (`crates.io`) depend on at least one unsafe FFI-bindings package. With many high-profile open-source projects such as the Mozilla Gecko browser engine (≈ 10% Rust*) adopting Rust, ongoing efforts to add support for the language to the Linux kernel†, and Microsoft announcing the uptake of Rust in the Windows OS,‡ the language is gaining a strong foothold across industry sectors and developing easy-to-use and efficient ways to safely integrate legacy libraries through FFI is highly relevant [150].

*Mozilla Gecko repository: `https://github.com/mozilla/gecko-dev`

†Rust for Linux project: `https://rust-for-linux.com/`

‡"You will actually have Windows booting with Rust in the kernel in probably the next several weeks or months, which is really cool." – David Weston, director of OS security for Windows, at BlueHat IL 2023.

Earlier approaches exist to make invoking unsafe code safer, e.g., by executing unsafe code in a separate process [114] or by utilizing MPK to provide in-process heap isolation [104, 180], based either on developer knowledge or on automated inference in a compilation framework [19, 131]. Separating safe and unsafe program sections into multiple processes has the advantage of combining comprehensive heap and stack isolation with the potential of recovering execution of the safe application part after a crash of the unsafe process, albeit incurring substantial runtime overheads. In-process isolation, in comparison, has mostly been used to isolate safe from unsafe heaps only and with exceptions leading to program termination, yet with much smaller overheads.

This chapter and contributions In this work we study Secure Rewind and Discard of Isolated Domains [87] to protect Rust applications that make use of unsafe language features. As a mechanism that enables compartmentalized in-process isolation of safe and unsafe program sections, our approach relies on MPK to provide stack and heap protection, and allows the safe compartment to recover from violations caused in unsafe code, albeit without relying on a custom compiler and with much smaller overheads than earlier related work that achieves similar properties by means of process-based isolation. Our study also engages with the question of how to fairly compare different approaches to compartmentalization and isolation as they require different approaches to memory management, data serialization, and communication between compartments, which incur a majority of the overheads. Specifically, we make the following contributions:

- We present Secure Rewind and Discard of Isolated Domains for FFI to protect the integrity of a Rust application from memory-safety violations in unsafe program parts or C libraries, utilizing in-process isolation. This approach further increases the Rust application's availability through a secure rewinding mechanism.
- We implement stack and heap protection for Rust-FFI as a Rust crate by leveraging in-process isolation and the Secure Domain Rewind and Discard (SDRaD) C library. We provide an easy-to-use application programming interface (API) with a high degree of automation for sandboxing programmer-selected functions.
- We compare the costs of context switching and serialization for process-based and in-process isolation using libpng and snappy libraries as case studies. Specifically, we evaluate different serialization approaches in Rust.

We open-source our prototype and experimental evaluation data under a BSD license on GitHub:

`https://secure-rewind-and-discard.github.io/` .

We discuss the Rust FFI and approaches to isolating unsafe code in Sect. 3.2 and explain the objectives of our study in Sect. 3.3. In Sect. 3.4 and 3.5 we detail SDRaD-FFI, and experimentally compare our prototype with related approaches. We report on a security evaluation, lessons learned, and limitations of our approach in Sect. 3.6. Finally, in Sect. 3.7 and 3.8, we discuss our approach in the context of recent related work and draw conclusions.

3.2 Background

3.2.1 Rust-FFI

Rust is a modern systems programming language that is designed to prevent memory-safety defects through a strong type system combined with built-in compile- and run-time checks. Memory-safe Rust will restrict arbitrary casting, prevent temporal memory-safety bugs through a set of ownership rules on data objects, and perform bounds-checks on static and dynamic data allocations. Together, these properties ensure *safe* Rust code is *sound* and will not exhibit undefined behavior [186].

However, Rust programmers also have access to an *unsafe* superset of Rust which may exhibit undefined behavior. With unsafe Rust it is the programmer (not the compiler) who is responsible for ensuring the soundness of the code. Unsafe Rust is generally required when interfacing directly with hardware, operating systems, or *other languages*.

Foreign function interface The Rust Foreign Function Interface (FFI) enables the sharing of code and functions between Rust and other programming languages. Since the target language may not conform to the memory-safety properties enforced in Rust, FFI calls are considered inherently unsafe. As Rust and other memory-safe system programming languages are being deployed gradually, it is frequently necessary for Rust programs to interface with legacy libraries written in unsafe languages, such as C and C++. Recent work on the security of such multi-language programs [150] demonstrate that the interplay between safe and unsafe languages can undermine existing mitigations against memory attacks.

Different approaches for sandboxing legacy code from the safe Rust code have been proposed [12, 104, 114, 131]. In what follows, we describe the most relevant approaches in detail.

3.2.2 Process-based Isolation

Process-based isolation is an essential concept in most operating systems. It protects the system's integrity and resilience by providing the following features: 1) *integrity*: each process runs in its own virtual memory space that prevents a malicious process from accessing the memory of another process, and 2) *resilience:* each process has its own failure boundary so one process' failure does not affect others.

Multi-process software architectures Compartmentalizing large applications into distinct, isolated processes is a well-known design pattern used both for security and reliability. Multi-process software architectures, such as *site-isolating* browsers [178] come with two significant drawbacks: 1) they are complex to engineer and maintain, and 2) come with associated memory and process-to-process communication overhead that is exacerbated as the number of processes increases.

The engineering challenges with process-based isolation stem from moving from a monolithic program with a single thread of control to a concurrent programming model that necessitate multiple isolated processes to co-ordinate their execution and to communicate results to each other. For example, the incorporation of site isolation to the Chrome browser is the result of a multi-year engineering effort [111].

Automatic software compartmentalization The problem of *automatically* splitting an existing monolithic program into multiple, compartmentalized components requires program transformation of local function calls into *remote procedure calls* (RPCs) that occur across multiple processes. Sensible boundaries for such RPCs are highly application-specific, e.g., for Chrome, they are defined by the interfaces between the browser Chrome and the isolated component, such as the renderer. In most prior work automatic software compartmentalization the boundary is based on existing library APIs that, using source-to-source translation, can be turned into RPC calls. Source-to-source translation is generally invasive and requires changes to the compiler which rarely make their way into upstream toolchains.

Sandcrust In this work, we primarily focus on the use case of automatically compartmentalizing unsafe library interfaces in Rust programs. Sandcrust [114] is an easy-to-use sandboxing solution for compartmentalizing Rust applications to execute unsafe code in a separate, isolated process. Unlike solutions that require compiler-based source-to-source translation Sandcrust builds on the metaprogramming capabilities provided by the Rust macro system. This is the key selling point for Sandcrust's ease-of-use: to compartmentalize functions belonging to an unsafe library interface,

```
sandbox!{
    fn F (...)    {
        unsafe {
            ...  // unsafe code
        }
    }
}
```

Listing 3.1: Transform unsafe *F* with Sandcrust's `sandbox!` macro to a remote procedure executed in an isolated process.

they are simply annotated with a `sandbox!` macro provided by the Sandcrust crate (Listing 3.1).

Under process-based isolation, for different compartments (i.e., processes) to communicate and exchange data with each other, they must do so via inter-process communication (IPC) primitives. IPC typically comes with high overheads due to the context switching associated with scheduling different process and data crossing security and failure boundaries. In Sandcrust, argument passing and communicating return values between the main process and sandboxed code is handled over IPC channels. To allow the passing of Rust (and C) objects across the process boundary, one must be able to serialize, and deserialize, any data type that is to be transferred. As we will show in Section 3.4.4 the overhead associated with serialization and deserialization is the principal source of run-time overhead for compartmentalization in Sandcrust. Consequently, minimizing the overhead of object passing to and from the sandboxed code is an important consideration for making automatic program compartmentalization in Rust more tractable.

3.2.3 In-Process Isolation

In contrast to process-based isolation, *in-process isolation* is based on the notion of creating compartmentalized security domains within the memory space of a single application process. The main perceived benefit of in-process isolation is that since transitions from one domain to another stays within the same process, in-process isolation can significantly reduce the run-time cost of context switching compared to traditional process isolation. This is especially beneficial when domain transitions are frequent.

Software fault isolation In-process isolation is enabled through *software fault isolation* (SFI) [214], a technique for using program transformations to establish *logical*

protection domains. The enforcement of such protection domains can either leverage software-based checks inserted through compiler-based program transformation [31, 132, 144, 232], binary rewriting [68], or hardware-assisted enforcement [33, 89, 98, 104, 109, 116, 148, 181, 190, 207, 224, 230, 233].

Memory protection keys The inclusion of *memory protection keys* (MPK) [125] to commodity processors has prompted advances in sofware-fault isolation (SFI) solutions that leverage hardware assistance. The *protection keys for userspace* (PKU) mechanism in 64-bit x86 processors by both Intel [3] and AMD [6] has received, by far, the most attention from academia.

Protection keys for userspace PKU associates each 4KB memory page with a 4-bit *protection key* which is stored by the operating system (OS) in the page table entry. An additional 32-bit, CPU-specific, user accessible, protection key rights register (PKRU) stores a 2-bit value for each of the 16 possible protection keys, controlling whether associated memory pages are writable or accessible. The validity of the memory accesses is enforced in hardware based on the PKRU configuration.

Because the PKRU is accessible from user space, the access control policy enforced by PKU can be updated without the need to call into the OS kernel. This characteristic makes PKU significantly more efficient compared to OS-enforced memory access control; OS-involvement is only needed when a memory page's protection key is updated. However, as the PKRU policy is entirely controlled from user space, it can also be subverted by adversaries that can control writes to the PKRU. This risks violating the isolation guarantees of PKU-enforced in-process isolation unless PKU is augmented with protection for unauthorized writes to the PKRU. Previous work has explored compiler-based code rewriting [109], binary inspection [224], system call filtering [230], and hardware extensions [190] to harden PKU/PKRU security.

PKRU-Safe and XRust Automatic compartmentalization of multi-language Rust applications has been explored by Liu et al. [131] and Kirth et al. [104]. These prior works focus on portioning the heap into distinct protection domains for safe and unsafe Rust code, including calls occurring via FFIs. While these compartmentalization approaches share similar goals to process-based isolation, they are not directly comparable as process-based isolation encompasses not only the application heap, but all data, including the stack and static allocations. As discussed in Section 3.2.2, process-based isolation can further provide a degree of resilience against memory corruption whereas conventional SFI approaches terminate the application as soon as a violation of a domain boundary is detected.

In this work, we are concerned with comparing process- and in-process isolation for Rust-FFI under settings which provide similar isolation guarantees. Consequently, PKRU-Safe and XRust do not meet our requirements.

Secure Rewind and Discard Secure Rewind and Discard of Isolated Domains [87] allows compartmentalizing an application into distinct domains and restoring the execution state of the application in case of memory corruption in the compartmentalized domain. SDRaD is a C library that realizes this scheme using in-process isolation based on PKU. Developers can enhance their applications with rewinding capability by leveraging SDRaD APIs. For example, a compartmentalized domain can be created by assigning a custom user domain index with `sdrad_init()` call. Memory management for separate domain heaps is provided via `sdrad_malloc()` and `sdrad_free()` calls, leveraging the Two-Level Segregated Fit (TLSF) allocator [147]. Developers can use `sdrad_enter()` to enter and run code in a previously initialized domain and `sdrad_exit()` to exit from the domain. In case a memory corrupting event is detected inside the domain, the execution flows is rewound to the point where the domain was initialized. The application can then take an alternate action to avoid the offending event.

Because restoring the process to a prior point of execution under an adversary model where an attacker has access to the application memory requires strong isolation guarantees that encompass both the application, stack, heap, and static data, SDRaD provides a stronger isolation model compared to XRust [131] and PKRU-Safe [104]. Depending on the configuration, the SDRaD APIs can provide both confidentially and integrity guarantees for the isolated domains.

The main of the drawbacks of SDRaD is that manual effort is required for integrating SDRaD APIs calls into an application and the current implementation only supports C code. Therefore, while SDRaD provides a better match for the isolation properties we require, it requires augmentation in order to be usable with Rust code.

3.3 Problem Statement

The goal of this study is to apply in-process isolation as implemented by the SDRaD library to the Rust FFI. Doing so, will allow developers to isolate foreign functionality in an in-process sandbox with resilience to potential memory safety violations. If such a fault occurs, the sandbox is discarded and an error is signaled to developer-provided handler code that may either take steps to recover from the fault (and avoid it being triggered, if possible) or fail gracefully.

An important objective here is to use Rust's metaprogramming capabilities to allow for automatic software compartmentalization in the same way as Sandcrust does for process isolation. As discussed above, such ease-of-use is crucial for the adoption of a hardening mechanism, as high complexity and cost have proven to be one of the major obstacles to the improvement of software security.

The validation phase then compares the resulting prototype with Sandcrust as a representative of automated process isolation for the Rust FFI. In particular, we look at the performance overhead of the two solutions, micro-benchmarking the context switch cost as well as evaluating real-world applications for different data transfer volumes. Moreover we compare the security posture of the two solutions. All study results are discussed in Section 3.6 along with lessons learned.

3.4 Prototype Implementation

In order to protect the Rust FFI using in-process isolation, we provide *SDRaD for Foreign Function Interfaces (SDRaD-FFI)*, a Rust crate containing a Linux library for the 64-bit x86 architecture. It allows developers to leverage metaprogramming in Rust to conveniently wrap functions that should be executed in an isolated domain. Under the hood, it uses the SDRaD C library API with Protection-Keys for Userspace (PKU) as the underlying isolation primitive and it supports compilation with different serialization crates.

3.4.1 High Level Idea

As in *Sandcrust*, our SDRaD-FFI crate provides a `sandbox!` macro to annotate functions that are to be isolated (cf. Listing 3.1). Whenever a sandboxed function is called at run-time it executes in a nested domain, i.e., isolated from the caller, using SDRaD. In case of rewinding after detecting a fault, the behavior depends on the return type of the function. If Rust's *Result*-type idiom [220] is used, the fault is encapsulated as an *Err* variant. Otherwise, SDRaD-FFI raises an unwinding `panic` [219] that can be caught using the standard library `std::panic::catch_unwind` function [218] (cf. Listing 3.2). In either case the developer decides how to resume: they may try to recover, e.g., by dropping the input that caused the fault, or exit the program gracefully.

3.4.2 SDRaD-FFI design

The `sandbox!` macro is expanded into calls to the *SDRaD* library and additional glue logic for passing parameters and results between the domains as well as error

```
let result = std::panic::catch_unwind(||{      ①
    F(...);    ②
    });
match result {
    Ok(v) => ... , // Process result v ③
    Err(e) => println!("Fault in nested domain!"), ④
}
```

Listing 3.2: Example code snippet demonstrating handling unwinding panics raised by isolated function *F* ②. Function `std::panic::catch_unwind` turns an arbitrary function into a *Result*-type function ①. The result ③ and error ④ handling can be done as if a *Result* was directly returned.

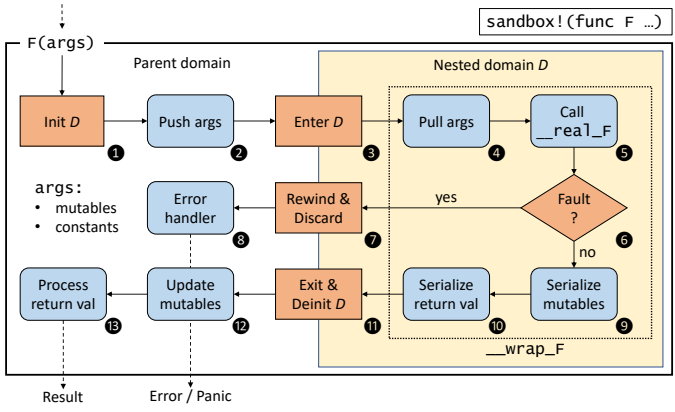


Figure 3.1: Transformation of sandbox macro. The angled orange boxed are SDRaD functionality, while rounded blue boxes are new glue logic for parameter passing and error handling.

handling. Syntactically, the definition of a sandboxed function *F* that takes a number of arguments and returns some value is maintained as `__real_F()`, while an additional wrapper `__wrap_F()` is defined to execute the required glue logic in the nested domain. Figure 3.1 shows the general control flow for executing a call to *F* in a nested domain *D*.

Firstly, domain *D* is initialized as an accessible domain via `sdrad_init()` so that the parent domain can easily push the arguments of *F* into *D*'s memory. To this end, the parent domain serializes the function arguments (cf. Section 3.4.4) and writes them to a dedicated memory area in the nested domain ❷ (cf. Section 3.4.3). Next, domain *D* is entered using `sdrad_enter()` ❸ and wrapper `__wrap_F()` is executed in the nested domain area. It first pulls from memory and rebuilds all function arguments using the

deserialization method ❹ (cf. Section 3.4.4). Then, `__real_F()` holding the original function body is executed ❺.

If a memory access violates the nested domain boundaries ❻ the fault is caught by SDRaD, the domain is rewound and discarded automatically ❼, and control-flow is redirected to an error handler which either returns an error value or raises a panic as described above ❽. Similar to as in SDRaD for C code, also Rust applications can be compiled with stack smashing protections (`RUSTFLAGS='-C stack-protector=strong'` [185]) that can trigger a domain rewind in SDRaD-FFI.

On a successful FFI call the results have to be persisted in the safe part of the program. To this end, any mutable local variables used as arguments ❾ and the return value ❿ are serialized and execution returns to the parent domain via `sdrad_exit` ⓫. During this step, the control state of domain *D* is deinitialized to prevent it from being re-entered without a corresponding sandboxed call. The parent domain then deserializes the variables, updates them in the parent domain ⓬, and returns the deserialized return value ⓭.

3.4.3 In-Process Communication

SDRaD-FFI requires the parent domain to manage a memory area in the nested domain to transfer data between the domains. SDRaD provides a fixed-size array memory management API for nested domains requiring manual size calculation and error handling to use. To hide these details from the programmer, we leverage the Rust vector type, a dynamically sized and resizable array, for data transfers between domains. Rust vector allocation can be customized by defining specific Allocator traits [221] for different use cases. In our case, we need to create a new vector that should grow and shrink using memory from a specific nested domain. To achieve it, we implemented the Allocator trait for *SdradAllocator* which is passed a specific domain identifier to be used with the SDRaD memory management API. Then a dedicated *SdradAllocator* struct is initialized for each nested domain and if arguments need to be passed from parent domain to a nested domain *D*, a new vector is created for that purpose using *D*'s dedicated *SdradAllocator*.

During a call of a function *F*, all input data is copied to that vector by the parent domain and read out by the nested domain when executing `__wrap_F()`. To this end, the nested domain needs to know vector metadata, i.e., the vector's length, capacity, and backend memory information. The parent domain writes this metadata to a reserved region in the nested domain stack. The nested domain can then create a second instance of the same vector using `Vec::from_raw_parts_in()` function by reading the vector info from its own stack.

However, memory allocated for this vector in Rust is automatically freed when it goes out of scope, i.e., when exiting `__wrap_F()`. As the original instance of the vector references the same memory, that memory would be freed once more when the parent function terminates eventually. To solve this problem, we implemented an SDRaDNestedAllocator that does not free any memory area by itself and we use it to construct the new vector in the nested memory area.

Some of the operations above are not natively supported by the SDRaD API, which we extended (cf. Section 3.4.5).

3.4.4 Serialization and Deserialization

Before entering the newly spawned domain, the function arguments should be passed and redefined in the nested domain. To this end, it is required to track the Rust data types of the arguments and copy all related memory areas into the nested domain area. Sandcrust [114] uses the Bincode serialization crate to pass arguments to another process using IPC. Bincode transforms data into a common binary representation that allows passing data between different platforms. However, as Sandcrust and SDRAD-FFI target only a single platform, this is redundant and Bincode serialization introduces unnecessary overhead. Sandcrust provides a macro-level optimization for $Vec\langle u8\rangle$ to reduce this overhead. It uses the macro matching mechanism to directly serialize and deserialize data of type $Vec\langle u8\rangle$ by copying the vector memory area, sending it over the IPC, and later restoring it in another process instead of using Bincode's native serialization scheme. Unfortunately, this solution is not scalable, as it requires providing a dedicated optimized implementation for any other Rust type instance that is not exactly $Vec\langle u8\rangle$, e.g., $Vec\langle u16\rangle$ or $Option(Vec\langle u8\rangle)$.

Abomonation [*sic*] [223] is a serialization crate that, unlike Bincode and other serialization libraries that provide a platform-independent representations of data types, simply produces a *deep-copy* of all reachable memory that belongs to an in-memory object. However, this raw-memory representation reveals architectural variations and can result in undefined behavior if the in-memory representation of serialized types changes between the time of serialization and deserialization. The latter can occur as a result of changes in the type's definition (which would also affect conventional serialization libraries) or changes in the type's underlying representation, e.g., the use of Rust's `repr(C|packed|align(n))` directives. For this reason, Abomonation is typically not suitable for production use [223] and is primarily of interest as a point of comparison in serialization benchmarks. Nevertheless, in SDRaD-FFI, serialized objects are only persisted across domain boundaries, never leave process memory, and their in-memory representation does not change. Consequently, Abomonation does not introduce undefined behavior in our use case, and is fast compared to other serialization crates [107]. SDRaD-FFI also benefits from Abomonation's ability to deserialize

in-place, i.e., deserialization results in a shared reference pointing to the serialized data. However, we observed that in-place deserialization prevents Abomonation from respecting memory alignment restrictions in deserialized data [223]. This can lead to suboptimal performance in code tuned to operate with a specific alignment, e.g., to ensure neighboring fields or elements occupy different cache lines, or misbehavior in code that relies on certain alignment for correctness. For our proof-of-concept implementation this limitation did not exhibit adverse effects.

Rust's specialization feature allows traits to have default implementations that can be overridden for more specific types, allowing for optimized trait implementations. [222]. We used this feature to extend the Abomonation trait implementation, optimizing serialization and deserialization of *Vec⟨u8⟩* in the same way as Sandcrust does. As opposed to the Sandcrust implementation, using specialization for traits is more scalable, as it also applies optimizations for sub-type instances of a given type, e.g., for *Vec⟨u8⟩* within *Option*(*Vec⟨u8⟩*) in our case.

Our case studies also required passing arguments and results of Rust's *slice* type, which represents sections of vectors, strings, or arrays. Slices are not a *collection type*, i.e., their size is undefined at compile-time and ownership of the underlying memory does not transfer with them, thus deserialization into slices is usually not supported by Rust serialization crates. We employed macro matching to solve this issue by first converting such variables into vectors (for slices of vectors and arrays) or strings (for slices of strings) before transferring them between domains using the above-mentioned mechanism. After the transfer, each variable is cast back to its original slice type.

3.4.5 SDRaD Integration and Extension

In the SDRaD-FFI Rust crate, we create a binding to the SDRaD C library using the #[*link*(...)] attribute. The SDRaD APIs are defined in an `extern` block in Rust, and we also define the SDRaD API macros because the C library macro definitions cannot be used directly in the Rust application.

In Rust, the default memory allocator used for dynamic memory management is the system allocator provided by the operating system. However, using alternative allocators such as *jemalloc* or *tcmalloc* is also possible by configuring the Rust compiler. Supporting different allocators may have advantages, such as reducing fragmentation for different applications. However, SDRaD uses the TLSF [43, 147] via interposing libc malloc family in the library load order, which is necessary to allow the subheap memory management in the application. That restricts allowing different memory allocators in Rust.

As we discussed in Section 3.4.3, transferring data between domains requires that the nested domain stack information should be exposed to the parent domain. We

extend the SDRaD API with two new APIs: 1) exposing the stack base pointer to the caller, allowing the parent domain to write argument vector metadata in the nested domain stack area. 2) updating the stack base pointer of the nested domain that prevents overwriting information stored on the nested domain's stack by the parent domain. These extensions were sufficient to support the in-process communication mechanism described above.

3.4.6 Parallel and Nested Domains

The SDRaD library allows to manage up to 15 in-process domains using PKU. These domains come in different flavors, e.g., accessible or inaccessible by the parent domain as well as persistent or transient. Furthermore, different domains can be used in parallel by different libraries and threads and even be nested to achieve more fine-grained isolation.

We restrict the features of domains in favor of a simplified usage model. By default, the nested domain to isolated foreign functionality from Rust is accessible and persistent to simplify data passing and allow foreign functionality with persistent state on the heap. The developer may manage different domains by providing a custom *Secure Domain Identifier (SDI)* to easily separate persistent internal data, e.g., two different libraries would use two different SDIs in their `sandbox!` definitions.

In principle our implementation also supports multi-threading and nesting of sandboxed calls, however, we consider such use cases out of scope of this study.

3.5 Evaluation

To evaluate our SDRaD-FFI prototype, we perform microbenchmarks to measure the latency of domain transition via the SDRaD call gate and SDRaD heap allocator function call overheads. We evaluate the performance of SDRaD-FFI applied to function calls from two unsafe C libraries: the compression library *snappy* and the image codec *libpng*.

We evaluated SDRaD-FFI performance using rustc 1.71.0-nightly, Abomonation 0.7.3, bincode 1.3.3, and 2.0.0-rc.3. We compared SDRaD-FFI with Sandcrust using bincode v1.0.0-alpha7. SDRaD-FFI leverages `allocator_api` unstable features. We run all experiments with the "release" profile on Dell PowerEdge R540 machines with 24-core MPK-enabled Intel(R) Xeon(R) Silver 4116 CPU (2.10GHz) having 128 GB RAM and using Ubuntu 18.04, Linux Kernel 4.15.0.

3.5.1 Microbenchmark

We perform microbenchmarks to measure the overhead of invoking a function in an isolated domain and the overhead of the SDRaD version of `malloc()` and `free()`. All benchmarks were conducted over 1×10^8 iterations and we calculated the mean execution time and standard deviation across all iterations.

To gauge the overall context switch costs, we sandboxed the `empty()` function and measured its execution time, i.e., the time for entering and exiting the sandbox. We compared SDRaD-FFI to a baseline value without compartmentalization, Sandcrust, and PKRU-Safe. Note that the PKRU-Safe microbenchmark value is an estimate based on the description in the paper [104]. Moreover, PKRU-Safe only compartmentalizes heap memory space, and we report the number here as a reference for another PKU-based isolation mechanism. Figure 3.2 shows the execution times for all systems under test. While the sandboxed `empty()` function with SDRaD-FFI takes 177.2*ns* (σ=61*ns*), the baseline without sandboxing takes 23*ns* (σ=21*ns*) and Sandcrust takes 8671*ns* (σ=695.5*ns*). SDRaD-FFI is 48.93*x* much faster than Sandcrust and 7.69*x* slower than the baseline function calls. This result is in the same order of magnitude as context switch overheads reported for PKRU-Safe [104].

We profiled the context switch and a main culprit for the overhead seems to be CPU pipeline flushes due to writes of the PKRU register. In SDRaD, each API call involves two such writes for entering and exiting the security monitor. Due to initialization and deinitialization calls, entering and exiting the sandbox accrues four PKRU writes each.

In the second case, we measured the execution time of calling `malloc()` and `free()` functions by allocating and later freeing memory with sizes uniformly distributed over a range from 0 to 4096 bytes. We compare SDRaD with the Rust default allocator. SDRaD takes 66*ns* (σ=40.5*ns*) for `malloc()` and 48.1*ns* (σ=34.2*ns*) for `free()` on average. In the Rust default allocator, the average run time is 44.4*ns* (σ=31.1*ns*), and 33.5*ns* (σ=25.0*ns*) respectively. Thus, in the SDRaD allocator, `malloc()` is on average 1.49*x* slower and `free()` is 1.44*x* slower compared to the Rust default allocator.

3.5.2 Snappy

Snappy [79] is a fast compression library that is presented as the FFI example in the Rust Book [105], where the raw snappy C APIs are wrapped by Rust interface functions: `compress()` and `uncompress()` as seen in Listing 3.3. We sandboxed these functions with SDRaD-FFI, similar to Sandcrust [114].

Compressing and uncompressing randomly generated data of different sizes, we measured the execution time of each operation for 5×10^5 iterations. We evaluate SDRaD-FFI solutions with different serialization crates: bincode and Abomonation.

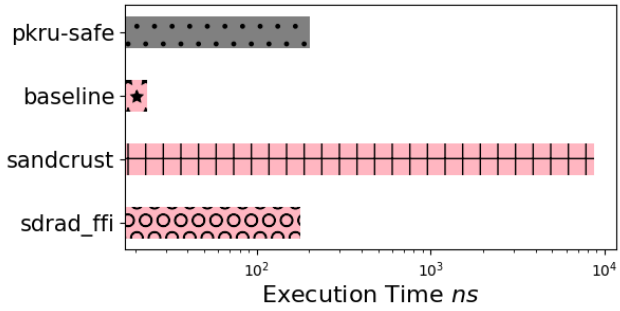


Figure 3.2: Microbenchmark of context switch latency for different sandboxing mechanisms. Numbers for PKRU-Safe have been extrapolated from literature.

```
pub fn compress(src:   &[u8]) -> Vec<u8>;
pub fn uncompress(src:  &[u8]) -> Option<Vec<u8>>;
```

Listing 3.3: Snappy compress and uncompress function signature

Comparing SDRaD-FFI to Sandcrust, we used the latter's "custom vector" feature for optimized $Vec\langle u8\rangle$ serialization.

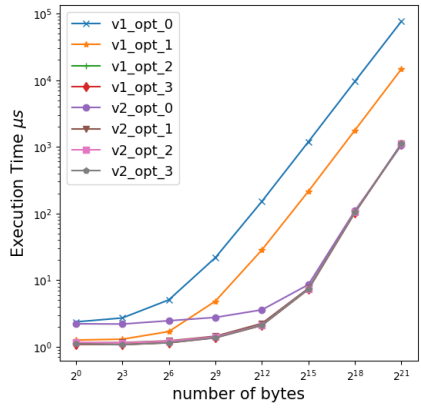
Abomonation optimization In the initial development phase of SDRaD-FFI, we noticed that Abomonation is slow for $Vec\langle T\rangle$ type variables. Abomonation crate has a generic trait for $Vec\langle T\rangle$ variables that are serialized and deserialized item by item and

```
fn decode_png(png_image: &[u8]) -> Result<Vec<Vec<u8>>, String>;
fn is_png(buf:   &[u8]) -> bool;
fn png_init()   -> Result<(),  String>;
```

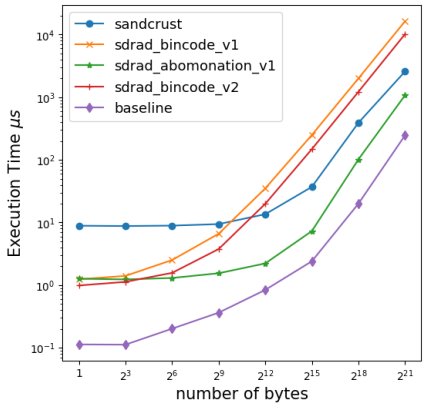
Listing 3.4: Sandboxed functions in libpng

png_create_info_struct()	png_create_read_struct()
png_destroy_read_struct()	png_sig_cmp()
png_get_io_ptr()	png_set_longjmp_fn()
png_read_info()	png_get_image_height()
png_get_rowbytes()	png_read_image()

Table 3.1: List of libpng C functions that are invoked by the sandboxed libpng Rust API in Listing 3.4.



(a) snappy `compress()` and `uncompress()` for SDRaD-FFI with Abomonation (v1: original, v2: specialized), compiled with different optimization levels



(b) snappy `compress()` and `uncompress()` for SDRaD-FFI, Sandcrust, and baseline without any isolation

Figure 3.3: Execution time of snappy with different isolation mechanisms for different numbers of bytes.

the corresponding iterator code takes up around 60% of the run time. However, for *Vec⟨u8⟩* such iteration is redundant: the vector underlying memory area can simply be copied in bulk.

To optimize Abomonation for *Vec⟨u8⟩*, we used the unstable specialization feature as explained Section 3.4.4. Our experiments show that this specialization improved performance significantly as seen in Figure 3.3a.

Later, we observed that building with optimization level 3 (release mode) yields similar

performance for the unmodified Abomonation crate with generic type *Vec⟨T⟩*. The Rust compiler optimizes the iteration for *Vec⟨u8⟩* and inlines Abomonation completely so that each serialization and deserialization step is just a `memcpy()` operation.

Overall, as described in Section 3.4.4, the memory layout presentation used by Abomonation to serialize and deserialize variables is suitable for optimization, unlike Bincode, which uses binary format representation and cannot be optimized automatically by the compiler in the same way.

Snappy benchmark result We perform the following benchmarking using the original version of Abomonation. Figure 3.3 show the snappy `compress()` and `uncompress()` function execution times for different byte sizes: 1 byte to 2^{21} bytes. In Figure 3.3b, Sandcrust performs worst when compressing data smaller than 2^{12} bytes which can be explained by our microbenchmark results from Section 3.5.1.

For data larger than 2^{12} bytes, Sandcrust performs better than SDRaD-FFI with bincode version bincode 1.3.3 and 2.0.0-rc.3. As discussed, the bincode serialization crate is not optimized for serializing and deserializing byte arrays and Sandcrust uses macro-level optimizations for *Vec⟨u8⟩* instead of using the bincode serialization. As a result, serialization crates overheads for large bytes are dominant over overheads from isolation methods. SDRaD-FFI with Abomonation crates is faster than the bincode-based versions and Sandcrust.

In general, all compartmentalization methods have an overhead compared to the baseline of at least one order of magnitude if only little data is transferred between the compartments. The overhead is reduced for larger data, but still significant. We profiled the SDRaD-FFI benchmark using the *perf* tool and found that about 60% of the run time is spent in `memcpy()`.

For uncompressing, Sandcrust has slightly better performance than SDRaD-FFI with bincode 2.0.0-rc.3. Even though Sandcrust is optimized to handle variables of type *Vec⟨u8⟩*, that solution does not apply for other types that contain *Vec⟨u8⟩*. Sandcrust cannot perform the same optimization for the `uncompress()` return value of type *Option⟨Vec⟨u8⟩⟩*.

SDRaD-FFI with Abomonation does not suffer from this problem by integrating optimizations in the serialization mechanism directly. Its performance comes relatively close to the baseline for large inputs but degrades for small inputs due to the high context switch costs.

3.5.3 libpng

The C library *libpng* is used to handle portable network graphic images. We reused the Sandcrust evaluation code published in [113] which sandboxes three Rust functions; their function signatures can be seen in Listing 3.4. These wrapper functions make calls to a total of ten different libpng C functions listed in Table 3.1 to read a PNG image into a vector of row byte vectors. We compared SDRaD-FFI-Abomonation with Sandcrust and measured the execution time of each decoding operation for 1×10^6 iterations and different image sizes.

Table 3.2 shows mean and relative-standard deviation for decoding measurement of different images. SDRaD-FFI introduced a worst-case overhead of 11.72% performance degradation compared to a baseline measurement for 5.5*KB* image decoding and it performed significantly better than Sandcrust in all cases.

Data size (bytes)	Execution Time [μs]			Execution Time Overhead	
	Baseline	**SDRaD-FFI**	**Sandcrust**	**SDRaD-FFI** vs. **Baseline**	**Sandcrust** vs. **Baseline**
5.5K	264.14 (σ±3.30%)	295.10 (σ±5.33%)	461.86 (σ±3.30%)	+11.72%	+74.85%
64K	4430.93 (σ±4.00%)	4749.53 (σ±3.19%)	7694.12 (σ±2.91%)	+7.19%	+73.64%
380K	9085.24 (σ±3.16%)	9295.79 (σ±2.14%)	11375.59 (σ±2.50%)	+2.32%	+25.21%
895K	13415.76 (σ±3.39%)	14008.36 (σ±2.00%)	20513.90 (σ±2.81%)	+4.42%	+52.91%

Table 3.2: *libpng* Decoding Measurements

3.5.4 Security Evaluation

We reproduced CVE-2018-1000810 [55] to verify the SDRaD-FFI error handling mechanisms. The `str::repeat` function in the Rust standard library has an integer overflow that causes a buffer overflow. Integer overflows only occur in the release build; Rust checks for these in debug builds. We sandboxed the `str::repeat` function with a specific return value in case a fault occurs and the sandbox is discarded. We found that the buffer overflow causes a domain violation which triggers the rewinding mechanism. Control is transferred to the error handler which returns the specific return value.

3.6 Discussion

Below we summarize the insights gained in our study, provide a comparative security evaluation of both process-based and in-process isolation, and outline remaining challenges.

3.6.1 Lessons Learned

As we have seen, the in-process isolation approach clearly outperforms process isolation in terms of context switch overhead. This is not surprising, as the development of PKU technology is explicitly designed to reduce context switch cost (be it for switches to the kernel or to other user processes).

Nevertheless, even for modestly sized arguments, the context switch cost starts to get dominated by the cost of data transfer between domains. Here, the data serialization method used can significantly impact performance and thus it is crucial to optimize it for the use case at hand.

One important insight here is that in our scenario, where data is exchanged between isolated code running on the same platform, serialization can essentially be simplified to copying the required data between different memory domains and making sure that data is interpreted using the correct types.

Such an optimization is straight-forward for vector-type variables, however, care must be taken where to implement these optimizations. We found that it is more elegant and scalable to use traits and specializations within the serialization crate instead of implementing optimizations using macro type matching, as is done in Sandcrust.

In this study, we adapted in-process isolation as provided by SDRaD, which involves many options for domains and a rather complicated programming model. We strived to control this complexity and provide a simple interface for isolation of unsafe code, similar to Sandcrust.

The Rust macro system proved to be a powerful tool in this respect as its meta-programming capabilities allow to automate argument and result passing. Moreover, the *Result* type and *panic* concepts could readily be integrated with the sandbox design. If possible, unsafe functions should use the *Result* return type idiom, so that they can easily be sandboxed without the developer having to write a custom panic handler.

Prior work on automated software compartmentalization conventionally relies on compiler-based source-to-source translation which, as discussed in Section 3.2.2, faces a high barrier for adoption due to requiring invasive changes to compilers and toolchains. Compartmentalization APIs such as SDRaD place the burden of integrating compartmentalization capability on the developers. For languages such as C, such changes can become invasive as the amount of data to be passed across the domain boundaries increases. Languages with strong metaprogramming capabilities, such as the Rust macro system can greatly reduce the complexity for repetitive integration effort, such as argument passing. There can also be benefits in converging on interoperable interfaces for language- and library-level compartmentalizing; by adopting a similar interface as Sandcrust, SDRaD-FFI can be used as a drop-in replacement for Sandcrust that provides improved run-time performance.

3.6.2 Security Evaluation of SDRaD-FFI

The security of Sandcrust is based on the mutual isolation of regular user processes. This is a fundamental and well-studied concept of computer systems [57, 100, 188] and a multitude of language, compiler, and OS-based hardening techniques exists to raise the bar for adversaries to breach the confines of a user process. [10, 111, 193]

In comparison, hardware-assisted in-process isolation has only been widely supported since rather recently and is not without caveats. In particular, the PKU mechanism as implemented by Intel and AMD has known security issues that necessitate additional compile-time effort and runtime hardening to ensure that the in-process isolation is watertight.

The required measures have been discussed before in detail [42, 87, 189, 230], in a nutshell: 1. instrumented programs need to be scanned for illicit writes to the PKRU register which controls access to domain memory [42, 109, 224], 2. a control flow integrity mechanism needs to be in place to ensure that legitimate PKRU write instructions of the security monitor cannot be abused as gadgets by an adversary [42,89], 3. run-time modification of code must either be prohibited or subject to binary analysis to verify that no new PKRU write instructions are introduced [224], 4. additional system call filtering and hardening of signal handlers is needed to prevent abuse by an adversary who tampers with the PKU mechanism [42, 189, 230]. While these are not unsurmountable issues, especially requirement 3) makes it tenuous to support functionality that requires just-in-time compilation.

However, improvements of the PKU hardware design have been suggested which would make most of the hardening measures listed above redundant [190]. It would be desirable for processor designers such as Intel and AMD to adopt similar solutions to make in-process isolation more secure and usable.

3.6.3 Security Impact on Sandboxed Application

In our case studies of *snappy* and *libpng*, we focus on Rust API functions that handle potentially unsanitized inputs that are passed to the underlying C library for processing. In Table 3.1 we list the decoder functions that are immediately called by the sandboxed API of *libpng*. If these inputs are attacker-controlled, compromising the security and availability of the application as a whole becomes feasible. A very recent example of such a vulnerability is ilustrated in the recent CVE-2022-3857 [56], where a crafted PNG image can lead to a segmentation fault and denial of service. While our case study does not include the `png_write_png()` call involved in the above CVE (but focuses on `png_read_png()` instead) and we did not try to reproduce this particular vulnerability, our example involves similar risks for the Rust application. Since these

vulnerabilities exist in many libraries and even automated exploit generation [18] is possible, we believe that API-based sandboxing techniques, including our SDRaD-FFI, provide a valuable additional line of defense to protect critical application logic – in our case the Rust application – from being attacked through vulnerabilities in third-party libraries. Moreover, isolating such code from the main program creates a failure boundary, improving software robustness. Approaches to assess and validate libraries regarding security requirements such as the absence of certain classes of vulnerabilities is difficult and complementary to our approach.

3.6.4 Future Work

As a direct consequence of our work, the performance of Sandcrust could be improved by employing a more streamlined serialization crate such as Abomonation. As we have shown, process isolation is inherently slower than in-process isolation due to the context switch cost, which is especially pronounced for low transfer bandwidth between the safe and unsafe parts of the program. Hence we shift focus to in-process isolation.

As discussed before, SDRaD-FFI can natively support parallel and nested isolation of foreign functionality. Future work should investigate which use cases could benefit from these features as well as their performance impact.

Another avenue for improvement is the security of SDRaD-FFI itself. As it is based on the SDRaD C library, our SDRaD-FFI crate is mostly unsafe code itself which increases the TCB. Formal verification could be employed to obtain correctness and security guarantees on the in-process isolation implementation.

One may ask whether SDRaD-FFI can only be used for the Foreign Function Interface or also to isolate unsafe functionality in general. This interface is a natural place to split a program into two, as there is likely no interdependency between functionality written in different languages and it is convenient to split the program stack at a function call. Yet, for unsafe code blocks part of Rust functions, working on shared objects in memory is essential. Here it is more challenging to split a Rust program into two domains without breaking the Rust runtime and its control-flow integrity and memory-safety guarantees.

Moreover, unsafe code in Rust is often used to optimize operations for low latency in ways that would not be legal in plain Rust [105]. Given the significant performance overhead of in-process isolation, it might actually be cheaper to use a safe Rust implementation than to isolate an unsafe code snippet. However, certain performance optimizations might not be easy to achieve without the use of unsafe Rust [186]. Further research is needed to identify classes of “native” unsafe Rust code for which

Table 3.3: Comparison of application compartmentalization schemes discussed in Section 3.7. The first rows shows how many distinct, isolated protection domains are supported. The *isolation type* is either process (⧉) or in-process (◰) isolation. The next seven rows indicate what kind of code modules may be isolated w.r.t. stack and heap memory. *Crash resistance* means that the application can continue execution after protection domain violations are detected and contained. We characterize the *development effort* as follows: the scheme requires invasive code changes, e.g., modifying the arguments of functions (*medium*), superficial code changes suffice (*low*), or the scheme can be incorporated *automatically*. The last rows show the geometric mean of the *performance degradation* for TRust [19], Sandcrust [114] and SDRaD-FFI in the Snappy benchmark (Section 3.5) compress and uncompress operations for input sizes 256B, 1KB, 4KB, 16KB, 64K, 256K, and 1GB as used by Bang et al. [19].

		ERIM [224]	**SDRaD** [87]	**XRust** [131]	**Fidelius Charm** [12]	**Galeed** [180]	**PKRU-Safe** [104]	**TRust** [19]	**Sandcrust** [114]	**SDRaD-FFI**
No. of domains		16[1]	15[1]	2	2	2	2	2	2	15[1]
Isolation type		◰	◰	◰	◰	◰	◰	◰	⧉	◰
Between safe and unsafe code	Isolated stack	–[2]	–[2]	✗	✓	✗	✗	✓	✓	✓
	Isolated heap			✓	✗	✗	✓	✓	✓	✓
Between unsafe and unsafe code	Isolated stack	✓	✓	✗	✗	✗	✗	✗	✓[3]	✓
	Isolated heap	✓	✓	✗	✗	✗	✗	✗	✓[3]	✓
Mixed-language support		✗	✗	✗	✓	✓	✓	✓	✓	✓
Between safe and mixed-language code	Isolated stack	–	–	–	✓	✗	✗	✓	✓	✓
	Isolated heap				✗	✓	✓	✓	✓	✓
Crash resistance		✗	✓	✗	✗	✗	✗	✗	✓	✓
Development effort		med.	med.	low	med.	med.	auto.	auto.	low	low
Performance degradation in Snappy benchmark	Compress	*Numbers not available*						20.5%[4]	1265.70%	155.7%
	Uncompress							50.0%[4]	6340.2%	370.8%

[1] May be less than 16/15 in cases where Linux's MPK support reserves a few domains for specific purposes. [2] Only supports C and / or C++ code.
[3] Due to manipulating the Rust abstract syntax tree (AST) via Rust's macros Sandcrust's sandbox can also be applied to unsafe functions without FFI.
[4] Result based on numbers reported in Table 8 in pre-print of Bang et al. [19] available at time of writing. We limit the comparison to the 256B – 1GB range due to the custom allocator used by SDRaD limiting the size of continuous allocations to slightly under 4GB in size [87].

in-process isolation is practical. In some cases, the Rust standard library may need to be augmented to support execution confined by an isolated domain [131].

3.7 Related Work

Several studies have explored the compartmentalizing of applications based on different underlying primitives: hardware-assisted in-process isolation [33, 87, 89, 98, 104, 109, 116, 148, 181, 190, 207, 224, 230, 233], process-based isolation [114] and software-based isolation [131, 214]. The majority of such approaches address the problem of compartmentalizing applications written in C or C++ [33, 87, 89, 98, 109, 116, 148, 190, 207, 224, 230, 233]. In this work, we focus on multi-language applications, specifically Rust code calling C or C++ code [12, 19, 104, 114, 131]. As noted in Section 3.2, the majority of prior work on using in-process isolation operate under weaker isolation guarantees than those of comparable solutions based on process-isolation [114]; XRust [131], Galeed [180] and PKRU-Safe [104] only protect the Rust heap from unsafe code while Fidelius Charm [12] only protects the Rust stack.

Independently and concurrently with our work, Bang et al. [19] propose TRust, a scheme with similar goals as Sandcrust [114] TRust protects both the Rust heap and stack from unsafe and mixed-language code. However, unlike SDRaD-FFI, which we designed to act as a drop-in replacement for Sandcrust, TRust relies on compiler-driven compartmentalization of Rust code based on the language-level boundary between safe and unsafe Rust. The principal benefit of compiler-based approaches, such as PKRU-Safe and TRust is relieving the developer completely of the burden of defining protection domains. In practice, automated approaches are restricted to boundaries that can be derived from language-level constructs. In the case of Rust, approaches such as TRust are not suitable for isolating unsafe code from other unsafe code. Combined with conservative compartmentalization policies that classify any objects that the untrusted code might use as unsafe, this can result in problematic edge cases where nearly all allocations are delegated to the unsafe domain (cf. Table 10 in [19]), negating the benefit of compartmentalization in the first place. Approaches, such as Sandcrust and SDRaD-FFI, which *minimize* developer effort but leave the developer in control can deal efficiently with such cases. Fully automating compartmentalization using compiler-based rewriting can also, at least in the short term, be counter-productive for the adoption of these solutions as invasive changes to mature toolchains are required, which are unlikely to make their way upstream. Point-solutions that can be deployed as discrete libraries or crates offer a more realistic path to industry adoption. Finally, as discussed in Section 3.2, one of the principal motivations for process-based isolation are improved resilience by means of a failure boundary between processes. Apart from SDRaD and SDRaD-FFI, no prior work in in-process isolation provides such a failure boundary between compartmentalized domains.

Table 3.3 summarizes our comparison between the relevant compartmentalization schemes for commodity 64-bit x86 processors excluding approaches that require kernel modification [12, 89, 224] for deployment. SDRaD-FFI is the only in-process isolation solution that provides similar capabilities as process-based isolation while significantly improving the run-time overhead compared to the similar Sandcrust in the snappy use case. While SDRaD-FFI remains less efficient than in-process isolation that partitions the program stack and heap in a manner which avoids unsafe code from interacting with objects in the safe area altogether, e.g., TRust, SDRaD-FFI allows programmer-selected unsafe functions to operate safely on data originating from the safe stack or heap. Our benchmarks indicate the measured performance overhead to be highly benchmark-specific, which means performance characteristics are influenced by choice of functions to sandbox.

We posit that schemes such as SDRaD-FFI remain a valid alternative to compiler-based rewriting for use cases that require more flexibility than static analysis can achieve and allow taking developer intent and constraints into account. Static and dynamic approaches to program analysis and vulnerability detection, in particular compiled-program analysis, can inform developers about potentially dangerous functions in libraries, which will allow them to use sandboxing effectively. Furthermore, approaches such as DynPTA [171] or CryptoMPK [98] can inform and complement SDRaD-FFI. Potential future work could further investigate combining compiler-based software compartmentalization, that can provide complete coverage of unsafe code, with developer-guided *further* sandboxing, providing the best of both worlds and defense in depth.

3.8 Conclusions

By allowing to include unsafe code, the Foreign Function Interface (FFI) represents a chink in the armor of Rust's memory safety guarantees [12, 104, 114, 131, 150]. A line of recent research tries to protect safe Rust code from the fallout of potential memory safety violations in cross-language functionality, e.g., by running such code as a separate process that may be compromised without affecting the safety of the Rust code. In this chapter we have studied the use of in-process stack and heap memory isolation to secure the Rust FFI.

Building on the SDRaD C library [87] for secure domain rewind and discard, we developed a prototype, SDRaD-FFI, which allows to conveniently sandbox foreign functionality in Rust. The implementation hides domain operations as well as passing arguments and results between the domains using Rust's macro metaprogramming capabilities. Faults within the isolated code are exposed to the programmer using standard error handling idioms such as *panics* and *Result* types.

We compare our prototype to Sandcrust's implementation of process isolation for FFI [114] and find that in-process isolation outperforms the latter due to its lower context-switching costs. This is further improved by optimized serialization methods for data transfer between isolated domains. We also compare the security of in-process isolation to the traditional process-based approach and find that in-process isolation requires quite extensive hardening to protect the PKU isolation mechanism. However, these shortcomings can be alleviated by updates to the hardware design that have been proposed in related work.

Overall, we conclude that in-process isolation is a usable and efficient security solution for Rust FFI, especially when a lot of data is exchanged between the safe and unsafe portions of the program. Our approach allows to run unsafe foreign code alongside safe Rust code while isolating the latter from possible memory safety violations in the former. Our implementation artifacts is available under a BSD license on GitHub:

`https://secure-rewind-and-discard.github.io/`

Chapter 4

Mon CHÉRI: Mitigating Uninitialized Memory Access with Conditional Capabilities

This chapter consists of ℂ3 and was originally published as:

> ***M. Gülmez**, H. Englund, J. T. Mühlberg and T. Nyman, "Mon CHÉRI: Mitigating Uninitialized Memory Access with Conditional Capabilities" in 46th IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2025, pp. 829-847,* https://doi.org/10.1109/SP61157.2025.00133

This chapter addresses $\mathcal{RQ}3$: "How can an architectural solution be designed to prevent uninitialized memory access?"

Abstract Up to 10% of memory-safety vulnerabilities in languages like C and C++ stem from uninitialized variables. This work addresses the prevalence and lack of adequate software mitigations for uninitialized memory issues, proposing architectural protections in hardware. Capability-based addressing, such as the University of Cambridge's CHERI, mitigates many memory defects, including spatial and temporal safety violations at an architectural level. CHERI, however, does not handle undefined behavior from uninitialized variables. We extend the CHERI capability model to include "*conditional capabilities*", enabling memory-access policies based on prior operations. This allows enforcement of policies that satisfy memory-safety objectives

such as "*no reads to memory without at least one prior write*" (`Write-before-Read`). We present our architecture extension, compiler support, and detailed evaluation of our approach on the QEMU full-system simulator and a modified FPGA-based CHERI-RISCV softcore. Our evaluation shows conditional capabilities are practical, with high detection accuracy while adding a small ($\approx 3.5\%$) overhead which is comparable to the cost of baseline CHERI capabilities.

4.1 Introduction

Uninitialized variables, variables that are declared but not assigned a value, are a well-known source of software defects in the C-family of programming languages. In general, all run-time allocated memory in C, C++, and even in modern languages like Rust, starts out as *uninitialized*. In this state, the value of the memory is indeterminate and may not reflect a valid state for the variable type. Attempting to interpret uninitialized memory results in *undefined behavior*, which can cause security vulnerabilities. In particular, uninitialized memory may expose residual data from previously deallocated data structures. This may inadvertently result in information disclosure, ranging from leaked pointer values [37] to the exposure of cryptographic keys [204]. Leaked pointer values can be exploited by attackers to bypass *address-space layout randomization (ASLR)* [37], while the use of uninitialized variables can enable arbitrary code execution attacks [141]. Following recent surveys, use-before-initialized conditions account for a sizable 10% of memory-safety vulnerabilities in the wild [24, 99, 209].

Hardware-assisted defenses against memory-safety issues [257] are motivated by the need to reduce the overhead of run-time defenses through hardware/software co-designs and willingness by processor manufacturers to incorporate mechanisms for software security into their designs [17, 94, 176]. A prominent example is *Capability Hardware Enhanced RISC Instructions (CHERI)* [244], a joint research project of SRI International and the University of Cambridge. CHERI extends instruction-set architectures (ISAs) with fine-grained memory protection and software compartmentalization.

In its default configuration, CHERI provides spatial safety through capability-based addressing. While this allows software to preclude many classes of memory-safety defects, CHERI does not address uninitialized variables. Microsoft Security Response Center (MSRC) conducted a comprehensive analysis of vulnerabilities reported in 2019 to assess the CHERI ISAv7 [99]. The findings indicate that only 31% of the reported vulnerabilities could have been mitigated through the default configuration of CHERI. An additional 24% could be mitigated by CHERI when configured to provide partial temporal safety under the Cornucopia mechanism [240]. The analysis further

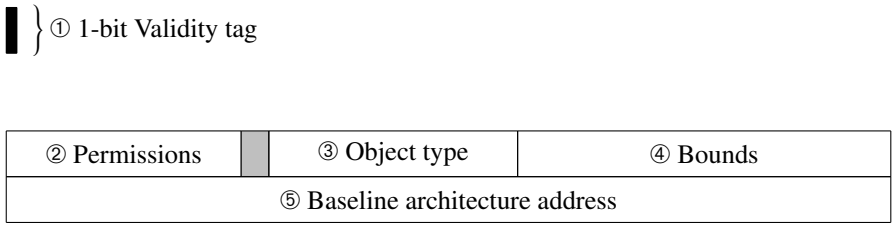


Figure 4.1: In-memory representation of CHERI capabilities adapted from Watson et al. [236]

highlights that at least 12% of the assessed vulnerabilities could have been mitigated if CHERI would protect against uninitialized access. This work explores and evaluates extensions for CHERI to mitigate those 12%.

This chapter and contributions. In this chapter, we extend the CHERI capability model to express memory-access policies that eliminate undefined behavior associated with uninitialized memory. We introduce *conditional permissions (CPs)* to capability-based addressing. CPs can express memory-access policies that take previous operations on memory into account. This enables *conditional capabilities* with fine-grained policies that satisfy different instances of memory-safety objectives at different granularities, such as *"no reads of memory which has not been the subject of at least one write"* (`Write-before-Read`) or *"this memory can be written to only once"* (`Write-Once`). We describe these CPs in section 4.3. CPs are enforced by introducing the notion of *operation-specific bounds* to capability-based addressing:

- We introduce *conditional permissions* and *conditional capabilities* for capability-based addressing (Section 4.3).
- We integrate conditional capabilities to CHERI-RISC-V in prototypes based on the QEMU full-system emulator and an FPGA softcore based on Flute64Cheri IP (Section 4.4).
- We add support for `Write-before-Read` conditional permissions to the CHERI-enabled Clang/LLVM compiler (Section 4.4.3) and embedded memory allocators (Section 4.4.4).
- We evaluate CPs using > 1000 NIST Juliet test suite test cases for uninitialized variables and using EEMBC CoreMark performance benchmarks (Section 4.5).

Our results show that CPs achieve 100% detection with only six false positives ($\approx$ 1%) for the Juliet test suite, where the false positives do exhibit uninitialized

(but non-vulnerable) accesses. We further show that CPs on CHERI-RISC-V impose only ≈ 3.5% performance overhead in addition to that added by CHERI, compared to benchmarks on an unmodified RISC-V softcore (7% combined). Consequently, CP overhead is comparable to that of CHERI. In summary, our work effectively and efficiently addresses uninitialized memory vulnerabilities, combining very high detection accuracy with performance penalties that are substantially lower than other detection mechanisms in hardware or software. Our conditional capability-enhanced QEMU and toolchain prototypes, and evaluation artifacts are available at `https://github.com/conditionalcapabilities`.

4.2 Background

How prevalent are uninitialized memory vulnerabilities? The prevalence of uninitialized memory as a source of vulnerabilities has been highlighted in statistics based on the Common Vulnerability Enumeration (CVE) program. MSRC reports that, between 2017 and 2019, uninitialized memory vulnerabilities accounted for ≈ 5–10% of the CVEs issued by Microsoft [24, 99]. In 2019, uninitialized memory accesses were the fourth largest class of memory defects (10%), after spatial (44%), temporal (29%), and type confusion (14%). Similarly, a more recent survey of memory-safety CVEs between 2015 – 2022 by Sutter [209] indicates that use-before-initialized conditions account for ≈ 9% of all reported memory-safety vulnerabilities, after lifetime safety (49%), bounds safety (18%), and type confusion (11%), indicating that uninitialized memory accesses remain an issue in industrial code bases that impacts system security.

4.2.1 Capability-Based Addressing

Capability-based addressing is a memory access-control paradigm originating from mainframe computers of the late 1950s and 60s [120]. In capability-based addressing, conventional references to locations in computer memory, i.e. *pointers*, are replaced by protected objects called *capabilities* [61]. Capabilities carry, in addition to the referenced memory address, additional permission information that is used by the processor or memory-management subsystem to determine whether the accesses performed through a capability are allowed. While the exact composition of a capability can vary between different hardware instantiations, virtually all capability-based addressing schemes express allowed operations through at least *Read* (`R`), *Write* (`W`), and *Execute* (`X`) permissions that control whether references through a capability are permitted for load and store instructions or instruction fetches respectively. Capabilities also include *bounds information* that limit the range of memory that can be referenced via a particular capability.

Interest in capability-based addressing diminished with the introduction of memory management units (MMUs) that, in addition to performing address translation between virtual and physical memory addresses, also manage access control to virtual memory. However, capabilities differ fundamentally from the access control in MMUs: whereas MMUs associate permissions to *individual memory pages*, capabilities associate permissions to the *references used to address memory*.

4.2.2 The CHERI Capability Architecture

CHERI, which stands for Capability Hardware Enhanced RISC Instructions, is an ISA extension for a capability-based architectural protection model and hardware-software co-design. The CHERI architecture extends an underlying conventional ISA with hardware-supported capabilities that are used to protect virtual addresses used as code or data pointers. The CHERI ISA specification [235] defines the representation of capabilities held in registers and memory, as well as capability-aware instructions to manipulate them. Currently, implementations of CHERI exist for MIPS, RISC-V, and Armv8-A instruction sets. CHERI-enabled processors have been developed by Arm [81], Microsoft [14], as well as in the RISC-V ecosystem.

Figure 4.1 shows the in-memory representation of a CHERI capability. Each capability is double the width of the native integer pointer type of the baseline architecture: 128 bits on 64-bit platforms and 64 bits on 32-bit platforms. One additional bit, the *validity tag* ①, is stored separate from the capability and is protects its integrity: any manipulation of the capability in-memory by non-capability-aware instructions invalidates its tag. Capability-aware instructions maintain the tag as long as certain architectural invariants are met. This prevents direct in-memory manipulations and injection of arbitrary data as capabilities. The *permissions* ② control how the capability can be used and consists of the permissions described in Section 4.2.1. The *object type* ③ allows capabilities to be temporarily "sealed", which renders the capability unusable until it is "unsealed" by a special instruction. Sealing is used by CHERI to implement opaque pointer types and fine-grained in-process isolation. The *bounds* ④ describe a lower and upper bound relative to the baseline architecture address ⑤, which limits the portion of address space the capability is able to access. To reduce the in-memory footprint of capabilities, the bounds are stored in a compressed format [243] with both bounds in 28 bits (for a 64-bit address), losing precision as the object size increases.

New capabilities in the CHERI architecture are always derived from an existing capability. The heritage of all capabilities can thus be traced back to the initial capabilities made available to firmware at boot time. CHERI enforces *monotonicity* on newly created capabilities, ensuring that capabilities constructed by a capability-aware instruction cannot possess permissions or bounds that exceed those of the original capability. The only exceptions to capability monotonicity are facilities for

Table 4.1: Conditional permission types for conditional capabilities

Conditional permission	Description
`Write-before-Read`	Memory must be written to at least once before read-access is granted
`Write-before-Execute`	Memory must be written to at least once before execute access is granted
`Write-before-Read-Only`	Memory must be written to exactly once before read access is granted
`Write-before-Execute-Only`	Memory must be written to exactly once before execute access is granted
`Write-Once`	Memory can be written to exactly once

exception handling and compartmentalization using sealed capabilities, which allow non-monotonicity in a controlled manner to enable software to gain access to additional data capabilities.

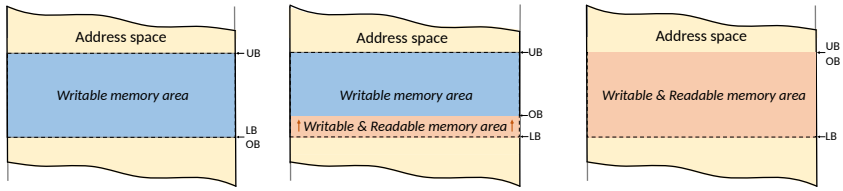
The bounds information stored together with the virtual address forms the basis for the memory-safety properties provided by CHERI. Each allocation made by a program running on a CHERI-capable processor is associated with a capability that describes, in addition to the address, the valid bounds of the object (or sub-object) in memory. This allows CHERI to provide inherent spatial memory-safety properties. Extensions to the CHERI software stack have explored adding temporal-safety properties to heap-based allocations [72, 240, 245] and sandboxing [36].

CHERI does not enforce type safety for capabilities, nor does it prevent software from accessing uninitialized memory using a capability it possesses. Consequently, CHERI must be complemented with compiler-based type-safety analysis, and instrumentation passes that zero local variables before first use [24, 152] as well as heap allocators returning zeroed memory. Security analyses of CHERI (e.g., by MSRC [99]) generally take such mitigations for granted.

4.3 Conditional Capability Design

Consider the uninitialized pointer-dereference vulnerability in the Linux kernel shown in Listing 4.1. This vulnerability allows control-flow hijacking due to the uninitialized `backlog` pointer ❶ being dereferenced at ❹ when `cpg->eng_st != ENGINE_IDLE` ❷. An attacker could exploit this vulnerability to achieve arbitrary code execution by spraying the stack to take control of the value of `backlog` and make it point to attacker-controlled code [141].

Our goal is to detect the first use of the uninitialized pointer in the `if`-clause at ❸. Unlike methods that automatically zero out memory [24, 152], conditional permissions (CPs) offer the same protection against uninitialized memory use but with smaller and more predictable run-time overhead.



(a) Initial state of memory area. (b) State after store between LB and OB (c) Final state when OB reaches UB.

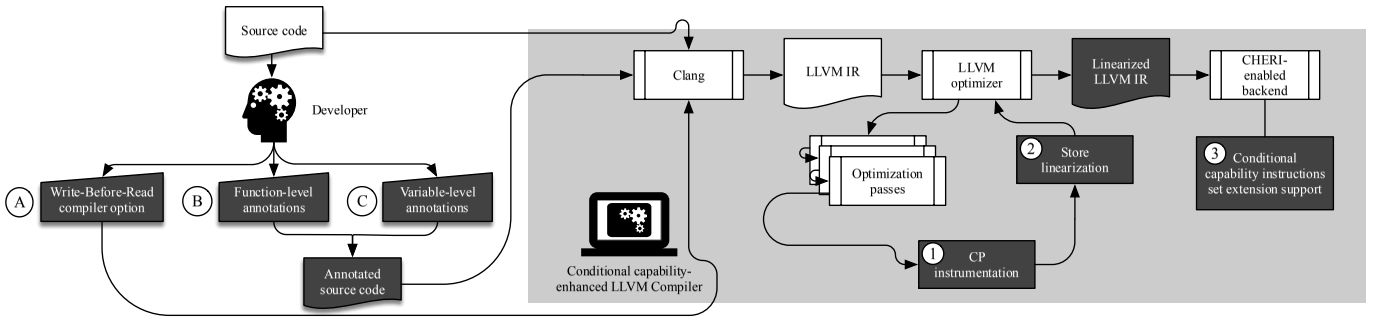
Figure 4.2: `Write-before-Read` conditional capability state transitions: *(a)* newly allocated memory under the `Write-before-Read` CP is write-only between the lower bound (LB) and upper bound (UB) and becomes gradually readable *(b)* after store operations advance the operation bound (OB). In the final state *(c)*, the OB has reached the UB and the conditional capability behaves identical to a corresponding conventional capability.

```
1  static int queue_manag(void *data){
2      cpg->eng_st = ENGINE_IDLE;
3      do {
4          struct crypto_async_request *async_req = NULL;
5        ❶ struct crypto_async_request *backlog;
6          /* ... cpg->eng_state may change state here ... */
7        ❷ if (cpg->eng_st == ENGINE_IDLE) {
8              backlog = crypto_get_backlog(&cpg->queue);
9              /* ... */
10         }
11
12       ❸ if (backlog) {
13       ❹     backlog->complete(backlog, -EINPROGRESS);
14         }
15         /* ... */
16     } while (!kthread_should_stop());
17     return 0;
18 }
```

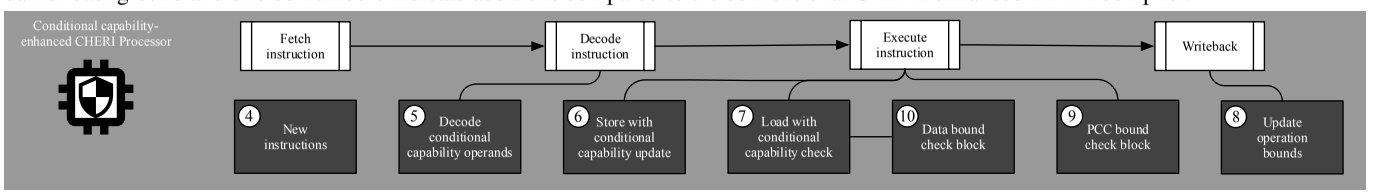
Listing 4.1: Excerpt from Linux' `queue_manag()` function defined in `drivers/crypto/mv_cesa.c` showing an uninitialized pointer dereference patched in April 2015 [102].

4.3.1 Challenges

Integrating CPs into capability-based addressing presents several challenges. A key issue is managing the tracking of uninitialized data. Previous approaches [77, 93, 252] assume that data between the lower bound and address pointed to by the capability is initialized, while uninitialized data falls between the address and the upper bound. However, this approach has limitations, as discussed in Section 4.6. To address this, we propose *operation-specific bounds* (OBs), which precisely track writes. OBs must be updated when a program writes to a region of memory to reflect the new bounds. Otherwise, stale or misaligned bounds can lead to false positives.



(a) Overview of the conditional capability-enhanced LLVM compiler. Circled letters indicate the different usage models for conditional capabilities: (A) `Write-before-Read` compiler option, (B) function-level annotations, and (C) variable-level annotations. Components with a darker background and circled numbers indicate additions compared to the conventional CHERI-enhanced LLVM compiler.



(b) conditional capability-enhanced CHERI processor. Components with a darker background circled numbers indicate additions compared to the conventional CHERI processor.

Figure 4.3: High-level system architecture of the conditional capability-enhanced LLVM compiler and CHERI processor.

Storing and updating operation bounds. To store and update OBs efficiently, we propose leveraging unused bits in the baseline architecture address (Figure 4.1). This minimizes changes to the underlying hardware but makes significant compression of the OB necessary. Further details on the hardware modifications are provided in section 4.3.3 and section 4.4.

Integration with existing architectures. To be practical, CPs must integrate seamlessly with existing capability architectures, such as CHERI, that are gaining industry adoption. As a case study, we integrate CPs into CHERI while ensuring full compatibility with the underlying RISC-V ISA. In section 4.4, we describe the integration process and our hardware extension, "*Mon CHÉRI*", which introduces only minor modifications to the CHERI capability representation (section 4.4.2).

Maintaining capability monotonicity. CPs add, similar to capability sealing (Section 4.2.2), controlled non-monotonicity to the CHERI design. They allow temporary suspension of permissions, which are regained once the associated condition is met. Our design (Section 4.3.2) guarantees that CPs do not to elevate privileges beyond the original capabilities.

Maintaining capability linearity. *Capability linearity* refers to the consistency and integrity of OBs across a program's execution. To maintain this, we propose a new compiler pass to ensure that capabilities are properly updated as the program executes. This pass is implemented in the LLVM-based Mon CHÉRI toolchain, as detailed in section 4.3.3 and section 4.4.3.

Minimizing run-time overhead. A concern with any memory access control mechanism is the potential for performance degradation. For Mon CHÉRI, we ensure that checks involved in monitoring memory accesses do not add significant latency or computational overhead during normal program execution. In Section 4.4.1, we explain how the OB checks are carefully designed to minimize additional cycles per instruction.

4.3.2 Conditional Capabilities

Conditional capabilities are designed to complement the conventional capability permissions with conditional permissions, allowing permissions to be tailored to specific needs or requirements for different sections of data or operations. The conventional capability permissions and CPs, when enabled for a capability, are

evaluated in parallel; both sets of permissions need to be valid for memory access to be allowed. Conditional capabilities consist of two building blocks: 1) *Conditional permissions* enable capabilities to trace whether the corresponding condition is fulfilled. 2) *operation-specific bounds* enable CPs to be enforced at a granularity that allows the differentiation between accessible and inaccessible memory ranges within the conventional bounds of the capability.

Conditional permissions. To describe conditional permissions, we denote conventional permissions that remain *unchanged* throughout the lifetime of a capability, as $\mathcal{P}^{\mathcal{U}}$, operations on memory as $\mathcal{X}$, and CPs as $\mathcal{P}^{\mathcal{C}}$. A $\mathcal{P}^{\mathcal{C}}$ is granted on the condition that $\mathcal{X}$ occurs: $\mathcal{X} \implies \mathcal{P}^{\mathcal{C}}$ (if $\mathcal{X}$, then $\mathcal{P}^{\mathcal{C}}$). If $\mathcal{X}$ does not occur, $\mathcal{P}^{\mathcal{C}}$ is not granted. On an architectural level, operations ($\mathcal{X}$) are limited to load, store and execute, while permissions ($\mathcal{P}^{\mathcal{U}}$ and $\mathcal{P}^{\mathcal{C}}$) can be read, write and execute. For simplicity, we will use the terms `Write`, `Read`, and `Execute` when referring to both $\mathcal{X}$ and $\mathcal{P}^{\mathcal{C}}$ in CP names. The effective permissions ($\mathcal{P}$) are a subset of the conventional and the conditional permissions:

$$\mathcal{P} = \begin{cases} \mathcal{P}^{\mathcal{U}} & if\ \mathcal{P}^{\mathcal{C}} = \emptyset \\ \mathcal{P}^{\mathcal{U}} \wedge \left(\mathcal{X} \implies \mathcal{P}^{\mathcal{C}}\right) & if\ \mathcal{P}^{\mathcal{C}} \neq \emptyset \end{cases}$$

Operation-specific bounds. Tracking the memory range for which $\mathcal{P}^{\mathcal{C}} : \mathcal{X} \implies \mathcal{P}^{\mathcal{C}}$ requires *operation-specific* bounds. Each operation bound (OB) tracks the subset of memory within a capability's conventional bounds for which $\mathcal{X}$ has occurred. To accommodate multiple CPs at the same time, $\mathcal{X} \implies \mathcal{P}^{\mathcal{C}}$ and $\mathcal{Y} \implies \mathcal{Q}^{\mathcal{C}}$ where $\mathcal{X} \neq \mathcal{Y}$ requires two distinct OBs. Conversely, for $\mathcal{X} \implies \mathcal{P}^{\mathcal{C}}$ and $\mathcal{X} \implies \mathcal{Q}^{\mathcal{C}}$, where $\mathcal{P}^{\mathcal{C}} \neq \mathcal{Q}^{\mathcal{C}}$, one OB is sufficient.

Figure 4.2 illustrates an example use case for our conditional capabilities: `Write-before-Read`. In its initial state (Figure 4.2.a), the capability refers to a writable memory area with upper and lower bounds. If a store occurs in the memory area (Figure 4.2.b), the OB increases by the size of the store operand on the hardware level.

An advantage of our design is that it allows capability hardware to enforce a variety of novel access-control permissions beyond conventional `RWX`. We identify the CPs and corresponding use cases in Table 4.1. `Write-before-Execute` is useful for just-in-time (JIT) compilers that reuse memory buffers for generated code, making them targets for control-flow hijacking attacks using executable heap or stack buffers. `Write-before-Execute-Only` enables conditional capabilities to emulate execute-only-memory (XOM), typically limited to firmware, to protect secrets such as stack canary reference values or values used for control-flow enforcement [60] from being read by attackers.

4.3.3 High-Level System Architecture

Figure 4.3 depicts the high-level system architecture for our prototype conditional capability-enhanced LLVM compiler and CHERI processor. Inputs and process blocks in a darker color indicate the changes to a conventional CHERI-enabled compiler and processor architecture. For conciseness, in this description, we focus on `Write-before-Read` CPs.

In our prototype, conditional capabilities are exposed to developers through the three alternative usage models: Ⓐ a `Write-before-Read` compiler option, Ⓑ function-level annotations, and Ⓒ variable-level annotations. Our current design of the compiler option Ⓐ applies `Write-before-Read` to all stack variables, but we discuss possibilities for more intelligent heuristics in Section 4.7. Function- and variable-level annotations are implemented as Clang function and variable attributes, respectively, which gives developers control over which variables `Write-before-Read` is applied to.

The conditional capability-enhanced LLVM compiler (Figure 4.3a) can operate either on original source code or source code annotated by the developer using the aforementioned attributes. This differs from the conventional CHERI-enabled LLVM in three ways: ① the *CP instrumentation* transform pass that marks variable for instrumentation based on the compiler option or developer annotations, and emits intrinsics that interface with the conditional capability hardware to initialize operation-specific bounds for newly created capabilities, ② a *store linearization* transform pass that ensures that accesses to uninitialized objects in memory are performed consistently through a capability that tracks the operation bound, and ③ support for the new conditional capability instructions in the CHERI RISC-V back end, allowing the compiler to interface with the conditional capability-enhanced CHERI processor.

The conditional capability-enhanced CHERI processor (Figure 4.3a) is modified to support the conditional capability instructions (④, Section 4.4.1), and its instruction pipeline is augmented with logic to encode and decode capabilities containing an additional operation bound (⑤, Section 4.3.2). Modified ⑥ store and ⑦ load logic updates and checks the operation bounds when executing store and load instructions, respectively. Finally, ⑧ the pipeline performs writebacks of updated capabilities in register operands after the operations that update the operation bounds. Code capability checks for the program counter capability (PCC) are done in a dedicated bounds check block ⑨ (needed for `Write-before-Execute`, `Write-before-Execute-Only`, while all data CPs use a general-purpose bounds check block ⑩.

Table 4.2: Architectural changes to CHERI-RISC-V by conditional permission. The **CSetOpBounds variants** column shows the corresponding `CSetOpBounds` instruction mnemonic for each CP. The **Pipeline changes** column shows the impact on the *Load*, *Store*, and *Execute* logic inside the processor. A ○ indicates the operation is unaffected, a ◐ indicates the operation is augmented with an OB check, a ◑ indicates the operation is augmented with a writeback that, under certain conditions, updates the conditional capability OB. Finally, a ● indicates the operation is augmented with both.

Conditional Permission	CSetOpBounds variants	Pipeline changes		
		Load	*Store*	*Execute*
`Write-before-Read`	`csetwbrbound`	◐	◑	○
`Write-before-Execute`	`csetwbxbound`	○	◑	◐
`Write-before-Read-Only`	`csetrobound`	◐	●	○
`Write-before-Execute-Only`	`csetxobound`	○	●	◐
`Write-Once`	`csetwtbound`	○	●	○

4.4 Mon CHÉRI Implementation

In this section, we present Mon CHÉRI, our implementation of CPs for the CHERI-RISC-V architecture. We implemented two versions of Mon CHÉRI: 1. a Mon CHÉRI software model for the QEMU-system-CHERI128 full-system emulator, and 2. a Mon CHÉRI softcore based on the CHERI-Flute64 processor intellectual property (IP). The Mon CHÉRI ISA extension is described in Section 4.4.1.

In addition, we extended the existing CHERI-LLVM toolchain [51] with support for the Mon CHÉRI ISA extension and `Write-before-Read` CP instrumentation for stack-allocated variables (Section 4.4.3). Finally, we added run-time `Write-before-Read` CP support for heap-allocated memory in the CheriFreeRTOS [11] and Two-Level Segregated Fit (TLSF) [43] memory allocators (Section 4.4.4).

4.4.1 Mon CHÉRI Extension for CHERI-RISC-V

`CSetOpBounds` instructions. The RISC-V ISA is extensible through portions of the instruction encoding space reserved for ISA extensions. We add a family of `CSetOpBounds` instructions to CHERI-RISC-V to the existing CHERIv9 [235] non-standard `Xcheri` extension in the *custom-2/rv128* opcode space to allow setting an initial, or updating an already set, value for a capability's operation bound. Our `CSetOpBounds` instructions require two operands: an existing capability and a length operand. When invoked, `CSetOpBounds` sets the operation bound to the range $[b, o]$

where is the "*base*" encoded as part of the conventional capability bound, and is the "*operation top*" encoded into the operation bound (see Section 4.4.2). The operational bounds set by `CSetOpBounds` are restricted to be within the original capability bounds.

Additionally, `CSetOpBounds` must identify the CPs which operation bound to set. Lacking free operands in the instruction encodings available in `Xcheri`, we chose to encode the CP information into the `CSetOpBounds` opcode. The `CSetOpBounds` instruction variants are shown in Table 4.2.

Conditional permissions. We assign new *conditional permissions control bits* from the CHERI capability representation (p_{op} in Figure 4.4). The CHERI-Flute64 IP reserves 12-bits for hardware-defined permissions, i.e., those authorizing load, store, etc., and 4-bits for software-defined permissions, which interpretation are left open by the CHERIv9 specification. For the purposes of prototyping CPs, we utilize the four available software-defined bits, treating them as a 4-bit integer that enumerates one of 16 possible, mutually exclusive permission states. Five of those states correspond to the CPs in Table 4.2, ten modes are left unused, and one mode, the zero value, corresponds to the default state in which CP enforcement is disabled. In this default state, protections such as `Write-before-Read` are inactive, allowing uninitialized memory access within the capability's bounds. Invoking `csetwbropbound` on a capability enables `Write-before-Read` enforcement for that capability.

Processor pipeline changes. To avoid increasing processor latency, bounds checks are carefully structured within the execute instruction block of the pipeline (see Figure 4.3b), which has two stages. In the first stage, where single-cycle arithmetic logic unit (ALU) operations are performed, the CHERI implementation checks the conditional permissions and initiates the data capability bounds check, while the actual bounds check occurs in the second stage (used for longer-latency operations like memory access). OB checks run in parallel with the general-purpose bounds checking to minimize their latency. For stores and loads, the first stage checks if the conditional permission allows the operation, and if the target address results in an update of the current operation bound; if it does and is adjacent, the second stage extends the bound to cover it, with the update committed in the writeback stage. Table 4.2 shows how the OB checks and writeback are used by different CPs. Any violation raises a CHERI protection exception in the second stage.

Avoiding data hazards. Conditional Capabilities create dependencies between instructions that are typically independent. This is because stores using conditional capabilities update the OB in the capability operand, while stores on conventional capabilities do not. As a result, when a store using a conditional capability is

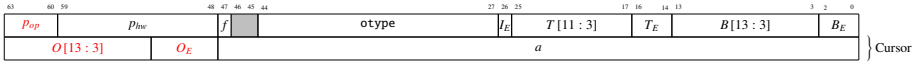


Figure 4.4: Layout of a 128-bit Mon CHÉRI capability with compressed operation bound. The labels indicate the fields for CP control bits (p_{op}), hardware-defined permissions (p_{hw}), flag (f), object type (`otype`), internal exponent (I_E), compressed top (T) including high part of exponent (T_E), compressed base (B) including low part of exponent (B_E), and cursor consisting of the compressed operation bound (O, O_E) and pointer address (a). Adapted from [235] with changes compared to CHERI ISAv9 compressed capability representation marked in red.

immediately followed by a load or another store using the same capability, a *data hazard occurs*: the latter operation requires the updated OB at the ALU stage for the OB check but is not available in the operand register until the writeback stage completes.

To solve this, we implemented a *bypass*—a new data path within the pipeline—that forwards the updated OB from the memory access or writeback stage to the ALU stage. This bypass activates when the processor detects that the next instruction in the pipeline operates on the same conditional capability. This avoids the need to stall the processor or reorder instructions during compilation and is compatible with conventional RISC-V instruction ordering. Appendix Appendix C.1 shows an example of a data hazard that is avoided by the bypass.

4.4.2 Adding Operation Bounds to Capabilities

The CHERI ISA introduced capability compression in ISAv6 [238], with the current CHERI Concentrate [243] compression scheme introduced in ISAv7 [237]. Compression reduces the in-memory size of capabilities, which would otherwise occupy 256 bits, quadrupling the space needed for native 64-bit pointers, and increasing cache footprint and memory bandwidth requirements. Expanding capabilities further to include additional operation bounds is impractical. However, most modern 64-bit operating systems (OSs) use only part of the 64-bit virtual address space. For instance, RISC-V Linux uses 48-bit addresses [78], and 32-bit OSs like FreeRTOS use 32 bits even on 64-bit hardware. We leverage these unused bits to store an additional 16-bit OB.

Conditional capability masking. Masking a portion of the address in cases where OSs do not fully utilize 64-bit addresses has precedents in both conventional processors, such as in Arm's Top Byte Ignore (TBI) [202], Intel Linear Address Masking (LAM) [95], and AMD Upper Address Ignore (UAI) [15] features, as well in the CHERI

design as alternative compression formats described in Appendix E of the CHERI ISAv9 [235]. To encode operation bounds, Mon CHÉRI applies an address mask when a register holds a conditional capability. Since CHERI-RISC-V uses a merged register file, general-purpose registers can hold either a 64-bit integer or a 128-bit capability. Unlike prior RISC-V pointer masking proposals [142], Mon CHÉRI's address mask is limited to conditional capabilities, leaving conventional CHERI capabilities and regular pointers unchanged.

Mon CHÉRI compressed capability representation. A raw 256-bit capability is comprised of three virtual addresses: b, t, and a. The 128-bit representation of capabilities utilizes the redundancy between the three addresses and stronger alignment requirements (proportional to object size) for a more compact representation.

Figure 4.4 shows the capability format for Mon CHÉRI. B and T encode the b and t bounds in one of two formats depending on the I_E bit: if $I_E = 1$ then an E is stored in the lower three bits of B and T (B_E and T_E) reducing their precision by three bits. E determines the position at which B and T are inserted into a to obtain b and t. Otherwise ($I_E = 0$, $E = 0$) the full width of b and t are used. Their width is determined by an encoding parameter: MW that determines the precision of the decoded bounds. The CHERI ISAv9 uses $MW = 14$ for 128-bit capabilities. However, t is further compressed by two bits as the top two bits of t can be derived from the equation $T = B + L$ where the most significant bit of L (L_{msb}) is known from the values of I_E and E and a carry bit is implied if $T\,[11:0] < B\,[11:0]$ since t is known to be larger than b.

When decoding the bounds, b and t are derived from a by substituting MW bits, E to $E + MW$, with B and T and clearing the bottom E bits. To allow a to span a larger region while maintaining the original bounds, the most significant bits of t and b $a_{top} = a\,[63 : E + MW]$ can be adjusted up or down using corrections c_t and c_b. The detailed description of the CHERI Concentrate compression can be found in Section 3.5.4 of the CHERI ISAv9 specification [235].

Encoding the operation bound. We encode the operation bound, o in two fields in the most significant 16-bits of the cursor: $O\,[13:3]$ (11 bits) and O_E (5 bits) which are freed by limiting $a_{top} = a\,[47 : E + MW]$. When $I_E = 0$, o is stored identically to b with its lowest three bits $O\,[2:0]$ derived from the most significant bits of O_E ($O_E\,[4:2]$). When $I_E = 1$, up to five bits from $O_E\,[E + 2 : 0]$ are used to store the least significant bits of o. Our current implementation limits O_E to five bits, limiting o to an I_E of at most 2. We discuss methods to alleviate this limitation in Section 4.7. Expanding o to full 48 bits happens similarly to b with its own correction c_o. Figure C.1 in Appendix C.1.1 illustrates the changes relative to CHERI Concentrate compression.

4.4.3 Mon CHÉRI Support for CHERI-LLVM

CP instrumentation. The CHERI-LLVM compiler adds capability bounds to stack variables in an intermediate representation (IR)-level compiler pass, `CheriBoundAllocas`. This pass replaces every IR stack allocation instruction (`alloca`) with a `llvm.cheri.cap.bounds.set` intrinsic. These are then replaced with CHERI `csetbounds` instructions by the CHERI-RISC-V backend. We extended this pass to add `llvm.cheri.cap.op.bounds.set` intrinsics for variables that are either annotated with `__writebeforeread` (Ⓒ in fig. 4.3a), in functions annotated with `__attribute__(("writebeforeread"))` (Ⓑ), or all stack variables, when compiling with the `-cheri-write-before-read` option (Ⓐ). The `llvm.cheri.cap.op.bounds.set` intrinsic is replaced with `csetwbrbound` instructions by the backend.

Optimizing `Write-before-Read`. The `CheriBoundAllocas` pass checks if `alloca` instructions fall within the original capability bounds and omits `llvm.cheri.cap.bounds.set` in those cases. We disable this optimization for variables that receive `llvm.cheri.cap.op.bounds.set` since it does not guarantee stores to `Write-before-Read` variables fall within the OBs. However, not all variables require run-time CP checks. For `Write-before-Read`, we optimize arrays and scalar variables allocated in function entry blocks by using a simple, non-heuristic analysis that checks if they are fully initialized before being accessed. The analysis inspects each store in the function's first basic block and verifies whether they are preceded by loads to the same allocation. For arrays, we track initialization with a vector, checking store instructions for the base pointer or specific indices, ensuring each load is preceded by a store through dominance analysis [138]; if a load occurs before a store, the index is marked uninitialized in the vector. For scalars, we verify whether a store targets the variable, and each load is dominated by a corresponding store. Variables shown to be initialized do not receive the `Write-before-Read` CP and, therefore, can be checked by `CheriBoundAllocas` for capability-bound optimization. In Section 4.7, we discuss the possibility of using heuristic static analysis to optimize `Write-before-Read` variables further.

Store linearization. LLVM IR uses static single-assignment (SSA) form, where each variable has exactly one assignment. The compiler creates new IR variables to maintain this rule when a variable has multiple assignments. If a variable is accessed through different control-flow paths, a phi (Φ) function is introduced to merge the values from these paths.

During register allocation, the compiler maps variables to processor registers. However, due to register pressure—the number of live variables exceeding the number of available

registers——the compiler generates *spill code* that moves variable contents between memory and registers. Live-range splitting optimizes when variables are spilled and can make use of the fact that the same variable may, at times, be available in multiple registers simultaneously. However, when a conditional capability is stored in multiple registers, the state of its operation bound may become inconsistent across instances. This can occur due to: 1. register spilling, or 2. register forking, where the conditional capability state differs between registers. Inconsistencies arise when a duplicated capability is stored in memory, becomes stale as its copy's operation bound is updated, and is later restored.

To address this, we introduce a *store linearization pass*, which runs after SSA optimizations and ensures a single, canonical conditional capability instance is maintained across stores. Performing store linearization after optimizations guarantees robustness across optimization levels.

Store linearization inserts placeholder function calls for every store operand in the IR to prevent live-range splitting from spilling a canonical conditional capability, causing it to grow stale. This placeholder is an identity function—effectively a no-op—taking a conditional capability operand and returning it unchanged. This, however, creates a data dependency between the input and output operand, signaling to the live-range splitting algorithm that the conditional capability has changed since its use in the store.

Array indexing via the LLVM IR `GetElementPtr` (GEP) instruction requires special handling. GEP takes a pointer and an offset, returning a new pointer (in SSA-form) pointing at the specified offset in the array. Store linearization inserts a placeholder function for the GEP operand, not the indexing pointer. In practice, the CHERI-RISC-V backend generates code using integer-relative store instructions (`s[bhwd]`), where the GEP input is used as a capability operand with an offset. Therefore, the linearization targets the original operand to extend its liveness.

Finally, the pass recursively replaces any variables holding conditional capabilities used store operands with those returned by the corresponding placeholder functions.

Escape value analysis. When a value in SSA form escapes a code block, it must be updated when accessed along different control-flow paths. The `reg2mem` transformation [134] in LLVM handles escaped values by: 1) allocating stack memory for each escaped SSA register, 2) storing SSA values in this memory before exiting a block, and 3) reloading values upon entering a new block. This ensures consistency across code paths. Store linearization modifies `reg2mem` to update conditional capabilities across control-flow paths, similar to how phi functions merge SSA values. One optimization available to store linearization stems from the original `reg2mem` also tracking output operands from the GEP instruction as escaped values. The special

Table 4.3: Detection rate of Mon CHÉRI on uninitialized memory issues from Juliet Test Suite [163] CWE457 test cases. Green cells indicate the true positives and true negatives, while red cells indicate the false positives and false negatives.

Ground truth	Mon CHÉRI	
	Positive	**Negative**
Bad	560 100%	0 0%
Good	6* 1%	554 99%

*Following the "Good" and "Bad" classification in Juliet, Mon CHÉRI reports six false-positives. Yet, these cases do exhibit uninitialized memory access behavior (cf. Section 4.5.1).

handling of array indexing allows us to avoid storing output operands from GEP in memory.

Similarly, conditional capabilities stored explicitly in memory must be updated after being used for store operations, even in straight-line code. Store linearization recursively checks if the operand used in a store is itself stored in memory. If so, an additional store operation is inserted to update the capability stored in memory. Listings C.3 and C.4 in Appendix C.2 illustrate this transformation.

4.4.4 Mon CHÉRI Support for Memory Allocators

We implemented the `cheri_opbounds_set()` application programming interface (API) to provide low-level support for conditional capabilities in system software, such as memory allocators. To enable `Write-before-Read` CPs for heap allocations, the allocator must initialize the capabilities to allocate memory with the appropriate CP.

A CHERI-aware `malloc()` already uses the `cheri_bounds_set()` intrinsic to set bounds of the return pointer, `ptr`, based on the allocation size. To make it conditional capability-aware, we added a call to `cheri_opbounds_set(ptr, 0, WriteBeforeRead)` before returning from `malloc()`. This explicitly sets the operation bounds of `ptr` to zero and configures the CP control bits to indicate `ptr` should be treated as `Write-before-Read` by the hardware.

Table 4.4: Area cost on VCU118 @ 100MHz expressed in number of lookup tables (LUTs) and number of registers.

	LUTs						**registers**		
	logic	Δ		memory	Δ		registers	Δ	
CHERI-Flute64	139109	–		10649	–		134427	–	
MonCHÉRI-Flute64	142069	2960	2%	10705	56	0.5%	135114	687	0.5%

4.5 Evaluation

We evaluated the Mon CHÉRI prototypes for functionality, security, performance, and area cost. The functional and security evaluation (Section 4.5.1) was performed on a conditional capability-enhanced QEMU-system-CHERI128 full-system emulator, CheriFreeRTOS [11], which integrates CHERI-based compartmentalization, and a `Write-before-Read` memory allocator (Section 4.4.4). For performance and area cost, we extended the CHERI-RISC-V field-programmable gate array (FPGA) softcore [52] based on the open-source Bluespec [159] RISC-V processor IP with the CP ISA extension (Section 4.4.1). This prototype, MonCHÉRI-Flute64, was validated against RISC-V specifications using 229 RISC-V ISA tests [157].

4.5.1 Functional and Security Evaluation on QEMU

We used the US National Institute of Standards and Technology (NIST) Juliet Test Suite [163], which includes thousands C/C++ of test cases that demonstrate common programming defects that lead to memory vulnerabilities, to evaluate Mon CHÉRI. These tests are organized by CVE numbers, with "bad" versions exhibiting vulnerabilities and "good" versions showing patched code. We focused on the CWE-457 [153] (Use of Uninitialized Variable) C test cases, covering 560 bad and 560 corresponding good cases. These examples cover a range of realistic scenarios, such as conditional control flows where variables might remain uninitialized in certain branches (e.g., as illustrated in Listing 4.1), function calls that pass variables assumed to be initialized, and cases involving complex data types like arrays, pointers, and structures.

To extensively assess Mon CHERI's detection rate for uninitialized variable accesses, we made specific modifications to the Juliet test suite. Modern compilers like LLVM tend to optimize away uninitialized memory accesses or reject them outright due to their sophisticated static analysis capabilities. To prevent this behavior, we declared all variables as volatile, which stops the compiler from applying optimizations based on

Table 4.5: Performance cost on VCU118 @ 100MHz expressed as CoreMark test results. The CoreMark score for a processor is reported as CoreMark-iterations-per-second-per-core-MHz. The Δ is relative to CHERI-Flute64 nocap results.

					CoreMark									
		Binary size		Δ	Total ticks	Δ		Total time (sec)		Δ	Iterations/sec		Δ	Score
	CHERI-Flute64													
	(baseline) nocap	43728		–	2704279286	–		27		–	370		–	3.7
	purecap	49872	6144	14.05%	2878393823	174114537	6.43%	28	1	3.57%	357	13	3,51%	3.57
	MonCHÉRI-Flute64													
	nocap	43728		–	2704301802	22516	0.42%	27	0	0%	370	0	0%	3.7
	purecap	49872	6144	14.05%	2878516285	174236999	6.44%	28	1	3.57%	357	13	3.51%	3.57
❶	`Write-before-Read` + purecap	50224	6496	14.85%	2949321554	245019752	9.06%	29	2	7.04%	344	26	7.00%	3.44
❷	`Write-before-Read` + purecap excluding store linearization	49232	5504	12.59%	2882957831	178678545	6.61%	28	1	3.57%	357	13	3.51%	3.57

undefined behavior. This allows us to build the tests with optimization level `-O2` while ensuring all test bad cases trigger uninitialized memory accesses.

The tests were executed within CheriFreeRTOS, which uses an instrumented allocation API to track memory accesses. CheriFreeRTOS provides a "compartmentalize and return" mode that isolates each test case in a separate compartment, allowing test cases resulting in a CHERI protection fault to return control to the caller, which records the result and proceeds to the next test case. The test cases were compiled with the Mon CHÉRI-enhanced CHERI-LLVM at optimization level `-O2` and with `Write-before-Read` instrumentation and store linearization but without the `Write-before-Read` optimization described in Section 4.4.3.

The results in Table 4.3 show that Mon CHÉRI detects all "bad" cases (100% true-positive rate) and reports only six false positives (1% false-positive rate). Upon closer inspection, these false positives stem from cases where uninitialized variables are copied, but never used. Although these cases are technically valid violations of the `Write-before-Read` policy, they do not pose a security risk, as data is immediately overwritten. Analysis tools relying on taint propagation [191] might not flag these cases, as the uninitialized memory does not propagate. As Mon CHÉRI enforces policies at an architectural level, distinguishing between these benign cases and actual vulnerabilities is not currently possible. All six false positives share this pattern, where the uninitialized memory is copied but later overwritten before being used. We provide the source code for one of these cases in Appendix C.2, Listing C.2. Further, all six cases have straightforward software workarounds that can be applied, once detected, to initialize variables early with a default zero value to avoid uninitialized access.

We evaluated the effectiveness of the store linearization pass by comparing the detection rate for the Juliet tests instrumented with and without store linearization. Without store linearization 119 out of 560 "good" test cases exhibit false positives (21% false positive rate). This demonstrates store linearization provides a significant improvement to the detection accuracy of Mon CHÉRI. We also verified the `Write-before-Read` optimization did not affect the detection accuracy as it only omits CP instrumentation for variables that are statically verified to be fully initialized.

For comparison, we compiled the Juliet test suite with all relevant warnings enabled in GCC and Clang/LLVM. GCC detected 170 out of 560 uninitialized cases (30%) at `-O0` and 173 cases (31%) at `-O2`. Clang/LLVM detected 117 cases (21%) regardless of optimization level. Detection rates for the Valgrind and Dr. Memory dynamic analysis tools range from below 10% to levels comparable to Mon CHÉRI, depending on the Juliet and compiler configurations. Under the same configuration used for Mon CHÉRI, (volatile variables and Clang with `-O2`), Valgrind and Dr. Memory detected 400 cases (71%) and 388 cases (69%), respectively.

As the Juliet CWE457 tests do not exhibit patterns that would require `Write-before-Execute`, `Write-before-Read-Only`, `Write-before-Execute-Only`, or `Write-Once` CPs, we verified the functionality of these CPs in the conditional capability-enhanced QEMU-system-CHERI128 implementation using purpose-built synthetic test cases.

4.5.2 Performance and Area Evaluation on FPGA

Area cost. We synthesized MonCHÉRI-Flute64 at 100 MHz on an AMD Virtex UltraScale+ VCU-118 FPGA. Compared to the CHERI-Flute64 design, the area cost of MonCHÉRI-Flute64 increased by only 2%, a small cost considering the overhead of adding CHERI. The majority of this additional logic is shared across many CPs.

Performance cost. We integrated the MonCHÉRI-Flute64 softcore into the BESSPIN-GFE security evaluation platform [174], which allows for a full-system evaluation of Mon CHÉRI performance. We measured performance using the EEMBC CoreMark [64] benchmark, running bare-metal on the GFE. Although MonCHÉRI-Flute64 supports all CPs in Table 4.2 we focus in these experiments on `Write-before-Read` as it is applicable to all variables in CoreMark. Consequently, applying it to all variables in the CoreMark benchmark code provides a worst-case estimate of Mon CHÉRI's performance impact. As we expect other CPs to only be applied to a subset of variables, their impact is a fraction of `Write-before-Read`'s.

Table 4.5 compares the performance results MonCHÉRI-Flute64 to the CHERI-Flute64 across different configurations: no capability enforcement (no-cap), pure-capability mode (purecap), and `Write-before-Read` enabled in pure-capability mode (`Write-before-Read` + purecap ❶). The results show that the `Write-before-Read` extension adds a modest ≈ 3.5% overhead over pure-capability mode, with minimal impact on baseline performance (≈ 0.4%) when capability enforcement is disabled. The combined performance impact of `Write-before-Read` and CHERI pure-capability mode over the baseline performance with no capability enforcement is 7%.

We also compared the performance of MonCHÉRI-Flute64 with and without store linearization (❷). Although store linearization is necessary, as explained Section 4.5.1, for the correctness of `Write-before-Read` enforcement, disabling the hardware fault for this experiment allows us to compare the performance impact of the hardware changes with that of the store linearization program transformation. Enabling store linearization resulted in most of the performance degradation observed in earlier tests (≈ 3.5%), with the hardware changes contributing negligible additional overhead (≈ 0.2%). This suggests that performance can be improved further by optimizing the store linearization strategy.

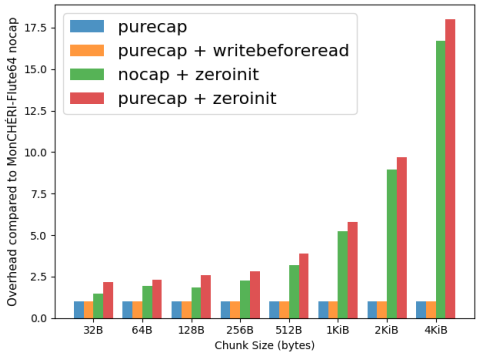


Figure 4.5: TLSF allocator microbenchmark. Bars show overhead relative to TLSF in MonCHÉRI-Flute64 nocap mode.

Microbenchmarks. To assess the impact of our store and load pipeline changes we microbenchmark stores and loads between CHERI-Flute64 in purecap mode and MonCHÉRI-Flute64 in `Write-before-Execute` + purecap mode. The store microbenchmark writes a 256-element array, recording total ticks. The load microbenchmark, we measured read from an of equal size. Here we report the difference in mean times for the experiment over 10 repetitions. We observed a negligible difference between CHERI-Flute64 and MonCHÉRI-Flute64: ≈ 5 ticks for the load and ≈ 30 ticks for the store benchmark.

Finally, we evaluated the impact of Mon CHÉRI on the TLSF allocator through microbenchmarks that allocate and free 1 MB of memory in chunks ranging from 32 bytes to 4 KiB. The results, shown in Figure 5, indicate that the `Write-before-Read`-enhanced TLSF allocator introduces negligible performance overhead: $\approx 0.1\%$ compared to purecap and $\approx 1.5\%$ (g.m.) compared to no-cap. The overhead of `Write-before-Read` + purecap is constant regardless of allocation size, while the overhead of zero-initialization increases linearly with allocation size.

4.6 Related Work

Various techniques for detecting the use of uninitialized variables are routinely used in modern software development. We categorize the existing approaches into six categories, as shown in Table 4.6. In this section, we compare conditional capabilities and Mon CHÉRI to existing approaches.

Table 4.6: Related work

Support for stack allocations	Support for heap allocations	Usable with dynamic analysis	Unaffected by optimization level	Performance overhead	Memory overhead	
Static code analysis[1]						
`-Wuninitialized` (GCC [75]	✓	✓	✓	✗	–	–
`-Wmaybe-uninitialized` (GCC [75])	✓	✓	✓	✗	–	–
`-Wuninitialized` (Clang [135])	✓	✓	✓	✓	–	–
`-Wsometimes-uninitialized` (Clang [135])	✓	✓	✓	✓	–	–
Dynamic analysis						
Valgrind Memcheck [197]	✓	✓	✓	✗	20×	yes
Dr. Memory [26]	✓	✓	✓	✗	10×	yes
Sanitizers						
Memory Sanitizer [203]	✓	✓	✓	✓	2×–4×	yes
Redundant execution						
DieHard [23]	✓	✓	✓	✓	≈ 40%	yes
Differential Replay [30]	✓	✓	?	✓	22×–24×	yes
Automatic initialization						
Secure deallocation [38]	✓	✓	✓	✓	<7%	–
UniSan [139]	✓	✓	✗	✓	≈ 5%	–
SafeInit [152]	✓	✓	✗	✓	≈ 5%	–
STACKLEAK [175]	*[2]	✗	✗	✓	≈ 1% – 5%	–
initAll (MSVC) [24]	✓	✗	✗	✓	≈ 10%	–
`-ftrivial-auto-var-init` (GCC [75], Clang)	✓	✗	✗	✓	≈ 1% [82] – 35% [160]	–
Hardware-based detection						
Uninitialized capabilities [77]	*[4]	✗	✓	✓	?	*[3]
Capstone [252]	*[4]	?	✓	✓	≈ 50%[5]	*[3]
Mon CHÉRI	✓	✓	✓	✓	≈ 3.5% / 7%[6]	*[3]

[1] For conciseness, we include only compiler-based static analyzers focusing on uninitialized variable detection in Table 4.6.
[2] STACKLEAK protects the Linux kernel call stack after system calls.
[3] Memory overhead due to replacement of pointers with CHERI / Capstone capabilities.
[4] Uninitialized capabilities and Capstone protect stack frames at coarser granularity than what uninitialized variable detection for individual variables requires.
[5] Overhead reported for the Capstone isolation model by Yu et al. [252].
[6] Overhead for Mon CHÉRI given both excluding overhead for CHERI (purecap) and including overhead for CHERI, based on Table 4.5.

Static analysis. Static analysis evaluates a program's code without executing it. By analyzing the code's structure and syntax, compilers, and dedicated static analysis tools can detect potential errors, security issues, and coding standard violations. Static analysis is performed early in development, allowing developers to address problems proactively.

Most major C/C++ compilers, including GCC [75], Clang [135], Intel Data Parallel C++ (DPC++), and Microsoft Visual C++ (MSVC) support compile-time checks for uninitialized variables. Static analysis tools such as Adlint [253], Clang-Check [137], Clang-Tidy [136], CodeSonar [41], Coverity Scan [212], CppCheck [146], Flawfinder [241], Frama-C [103], IKOS [25], Infer [71], and LCLint/Splint [70] can also check for uninitialized variables in C/C++ code.

However, all static analysis is limited by Rice's Theorem [179], which states that analyzing non-trivial properties of program behavior is undecidable for Turing-complete languages. As a result, static methods are approximations, balancing verbosity with false positives and analysis time.

Dynamic analysis. Dynamic analysis examines program behavior during execution. Unlike static analysis, which analyzes code without running it, dynamic analysis monitors run-time characteristics such as performance, memory usage, and interaction with system resources. It can, therefore, identify bugs, security vulnerabilities, and performance bottlenecks that may not be evident through static analysis. Tools like Valgrind Memcheck [197] and Dr. Memory [26] use dynamic instrumentation frameworks (e.g, DynamoRIO [27]) to track memory accesses and detect use of uninitialized variables.

Dynamic analysis, however, incurs significant performance overhead, often degrading program speed by 10× to 20×, making it impractical for continuous use. These tools also struggle to accurately identify the origin of uninitialized memory. For example, Memcheck traces uninitialized variables to heap blocks or stack allocations that occur in a particular function, but may not always pinpoint the exact source. Dr. Memory, meanwhile, does not detect uninitialized variables smaller than a machine word.

Sanitizers. Sanitizers are compiler-based tools designed to detect memory-safety, concurrency, and undefined behavior issues in C and C++ programs. They intercept memory accesses via compile-time instrumentation, offering higher efficiency and accuracy than dynamic analysis tools. Sanitizers like MemorySanitizer [203] detect uninitialized stack- and heap-allocated memory at individual bit granularity, with less overhead (2× to 4× slowdown) compared to Valgrind's one-order-of-magnitude slower dynamic analysis [156].

However, like Valgrind Memcheck, MemorySanitizer only reports uninitialized values that affect control flow, which limits its effectiveness in identifying memory issues related to information disclosure to, e.g., uninitialized data that overlaps with previously allocated pointers and which can reveal information to bypass ASLR.

Redundant execution techniques. Redundant execution techniques, such as DieHard [23], enhance memory safety by using randomized memory allocation and replication. DieHard scatters memory allocations across a large heap, reducing the chances of uninitialized memory being adjacent to other active regions. By comparing execution across multiple program replicas, DieHard can detect discrepancies caused by uninitialized memory reads. Similarly, differential replay [30] captures execution traces and replays them with varied initial memory states.

Multi-variant execution [45, 49, 96, 108, 187, 227] generalizes this concept, running multiple functional equivalent, but independently developed, programs in parallel. This principle has been applied to safety-critical software in various domains, such as train switching and flight control systems, electronic voting, and specialized software testing, such as detecting zero-day exploits and kernel information leaks [169]. However, replicating compute instances and I/O across each variant is resource-intensive and impractical for general-purpose use.

Automatic initialization. Automatic initialization approaches, such as UniSan [139] and SafeInit [152], automatically set variables to default values, mitigating uninitialized memory issues. These methods introduce performance overhead, particularly with large allocations [160], and may miss issues with non-stack variables. For example, Microsoft MSVC's initAll [24] and the `-ftrivial-auto-var-init` option in GCC and Clang focus on stack variables, but their their use is limited by their performance penalties [24, 82].

Automatic initialization can also interfere with dynamic analysis tools and sanitizers, masking issues with uninitialized variables that could be detected and fixed, making it less suitable for debugging and software testing [168]. Chow et al. [38] propose a secure deallocation technique to zero memory upon function exit, though it shares similar drawbacks, introducing overhead at the end of object lifetimes. The Linux Kernel implements a similar scheme, STACKLEAK [175], that clears the kernel stack at the end of system calls. This mitigates the impact of information leakage bugs, although it can impact system performance by up to 5%.

Hardware-based detection. Hardware-based detection methods, such as Georges et al.'s uninitialized capabilities [77] and Capstone [252], attempt to address

uninitialized memory issues by simply introducing a new permission to CHERI. However, this approach has significant limitations.

Firstly, the use of uninitialized capabilities requires software to derive new capability with each write operation, introducing complexity in managing memory permissions. This limits the utility of uninitialized capabilities to a secure calling convention that enhances local stack frame encapsulation, first proposed by Skorstengaard et al. [201], by additionally protecting against uninitialized stack reads.

Secondly, uninitialized capabilities only support `Write-before-Read` semantics, meaning they do not provide protection against other forms of uninitialized memory access. This limited expressibility reduces the utility of the approach, particularly when compared to more flexible solutions like Mon CHÉRI, which supports a broader range of access control policies as well as protection for both stack, heap, and other types of memory allocations.

Capstone [252] is a redesign of the CHERI capability model that enables broader memory isolation and attempts to generalize uninitialized capabilities. It addresses the first drawback of Georges et al.'s method by introducing self-incrementing write semantics. However, like Georges et al.'s uninitialized capabilities, Capstone shares the limitation of only addressing `Write-before-Read` scenarios. Capstone also suffers from a major drawback of its own: it incurs a significant performance overhead of up to 50%.

Additionally, in Capstone, fully initialized capabilities must be explicitly promoted to regular capabilities. Either the developer or compiler must understand when a capability is expected to be fully initialized, in order to promote it. In contrast, due to the way conditional capability semantics are designed (Section 4.3.2), a fully initialized conditional capability is equivalent to the corresponding CHERI capability.

These prior hardware-based approaches also face integration challenges. Uninitialized capabilities have only been simulated on the obsolete CHERI-MIPS ISA [93]. Capstone, on the other hand, comes with invasive changes to the established CHERI architecture and high overhead, making it impractical for real-world applications, especially in performance-critical systems.

Comparison with Mon CHÉRI. Our evaluation demonstrates that CPs, particularly `Write-before-Read` capabilities in Mon CHÉRI, offer high detection accuracy with minimal false positives when used in isolation. While static analysis is valuable for early detection, tools like dynamic analysis, sanitizers, and CPs require runtime errors to be exercised. We believe that `Write-before-Read` CPs can complement static analysis by improving the detection of uninitialized memory issues. In Section 4.7, we discuss how static analysis can be used with CP instrumentation to further optimize CP performance.

Our assessment of Mon CHÉRI is based on extensive evaluation using standard public sector and industry benchmarks on prototypes based on the QEMU full-system emulator and the MonCHÉRI-Flute64 softcore (Section 4.5). To our knowledge, no practical implementation nor compiler support for Georges et al.'s uninitialized capabilities is available to enable a fair comparison with Mon CHÉRI on FPGA. We argue that conditional capabilities, carefully designed to impose only minimal changes to the CHERI capability representation (Section 4.4.2), have a better chance of real-world adoption than more invasive proposals, such as Capstone.

Hardware-enforced CPs can improve detection performance for uninitialized memory issues similar to how memory tagging [17] has enhanced memory-safety sanitizers [196]. Moreover, CPs offer an alternative to automatic initialization (see Section 4.7), emulating it without the associated drawbacks. Lastly, CPs enable novel memory access control policies, such as `Write-before-Execute-Only`, which provide similar benefits to memory with special-purpose features.

4.7 Discussion and Future Work

Here, we discuss limitations of the current Mon CHÉRI prototype, alternate designs, and suggest future research.

Leveraging compiler-based static analysis. In Section 4.4.3 we showed how `Write-before-Read` CPs can be omitted for variables that are statically verified to be initialized. Existing compiler heuristics could be used to further optimize `Write-before-Read`, e.g., by omitting `Write-before-Read` CPs when a variable is determined initialized by GCC's `-Wuninitialized`, or instrument only variables identified as potentially uninitialized in certain code paths by GCC's `-Wmaybe-uninitialized`. We leave further optimizations as future work as Clang/LLVM, which our conditional capability-enhanced compiler prototype is based on, does not currently replicate GCC's `-Wmaybe-uninitialized` heuristic. Clang's alternative `-Wsometimes-uninitialized` is more conservative as it only issues warnings when the conditions under which a variable is left uninitialized are known. Developers can already correct such cases based on the emitted warnings. We believe the `Write-before-Read` CPs are useful in complementing compiler-based analysis, covering cases where the analysis is inconclusive.

Inter-function store linearization. The store linearization pass (Section 4.4.3) is currently limited to intra-function analysis, such as when `Write-before-Read` conditional capabilities are used within functions or passed to callees. However,

currently propagating conditional capabilities from a callee back to the caller must be done explicitly to avoid the conditions explained in Section 4.4.3 to occur across function boundaries. Solving inter-function store linearization is simpler in languages that enforce *borrowing*, where only one mutable reference to an allocation can exist at a time. Borrowing is prominent in Rust [105], but similar, compiler-enforced borrow checking has been proposed for C [200] and C++ [20, 112, 208] as well. Future work should explore borrow-checker-aided full program conditional capability linearization.

Emulating automatic initialization. In Section 4.6, we suggest `Write-before-Read` CPs could emulate automatic initialization, avoiding its drawbacks. Instead of issuing a hardware protection fault, a load to an address outside the conditional capability's OB could set the destination register to zero (or a default pattern). This mimics automatic initialization without the overhead of pre-initializing memory.

Another drawback of automatic initialization, discussed in Section 4.6, is interference with dynamic analysis and sanitizers. Initialization emulation could be controlled via a hardware configuration, allowing it to be toggled on or off. This allows the same CP-instrumented program binary to be used in production and for testing, depending on the configuration.

Overlapping capability permission bits. In Section 4.4.1 we explain how Mon CHÉRI reuses software-defined permission bits in the CHERI capability format. A drawback of the enumerator-based p_{op} representation is that it makes CPs, and possible software-defined permissions, mutually exclusive.

To avoid this drawback, we considered an alternative based on that CPs always describe a subset of the conventional CHERI permissions, e.g., `Write-before-Read` confers the same access as `R` when $o = t$. In this alternative, a *conditional control bit*, c is assigned from the reserved, but unused, capability bits to indicate whether the capability is in *conditional* mode. In conditional mode, i.e., when $c = 1$, the p_{hw} bits `R`, `W`, and `X` represent the corresponding CPs: `Write-before-Read`, `Write-Once`, and `Write-before-Read` respectively. This allows CP bits to overlay conventional CHERI permissions bits without losing expressivity. The `Write-before-Read-Only`, and `Write-before-Execute-Only` CPs could be overlayed similarly with an additional *exclusive permission* bit.

Non-sequentially written memory. In Section 4.4.1, we assume that memory accessible via CPs is written sequentially. This holds for most data types and operations like `memcpy()` and `memset()`, but not for C structures that are initialized field-by-field or contain uninitialized padding. Software workarounds like `#pragma pack` with

ordered field-initialization or an initial `memset()` before individual fields are set can prevent false positives from field-by-field access.

Alternatively, conditional capabilities can treat the o value as a bitmap, where each bit describes the initialization state of a memory segment. Memory can be segmented into equal chunks or structured data fields. If the 16-bit o field is insufficient, it could point to a larger bitmap in shadow memory, similar to how MemorySanitizer [203] tracks initialized memory. However, we believe the existence of straightforward software workarounds makes the overhead of complex tracking solutions unnecessary and thus leave exploring solutions for tracking non-sequentially initialized memory outside the scope of this work.

Supporting $I_E > 2$. In Section 4.4.2, we explain that the current operation bounds encoding restricts conditional capabilities to $I_E \leq 2$. Similar to how the CHERI Concentrate encoding sacrifices *alignment* precision as allocation sizes increase, the o can sacrifice *write* precision as I_E increases. At an architectural level, this is achieved by zero-padding the least significant bits of O_E, akin to handling B and T. When the precision of the least significant stored bit in O_E exceeds the ability to express writes to individual bytes, halfwords, or words, corresponding store instructions (`s[bhw]`) are disabled for that conditional capability. Extending writes beyond doubleword precision requires a variant of `SetOpBounds`, which extends o under the condition that the two preceding instructions have been writes reaching a target granularity.

4.8 Conclusion

This chapter presents Mon CHÉRI, a novel extension to the CHERI architecture that addresses uninitialized memory errors that account for ≈ 10% of all memory vulnerabilities.

By introducing conditional capabilities, Mon CHÉRI enables precise run-time detection of uninitialized memory access at instruction-level, with minimal performance overhead. Our extensive evaluation on the Mon CHÉRI QEMU-system-CHERI128 emulator and FPGA-based MonCHÉRI-Flute64 prototype shows that Mon CHÉRI achieves a 100% true-positive rate while maintaining a low, 1%, false-positive rate on the Juliet test suite, and incurs only a ≈ 3.5% performance overhead for the `Write-before-Read` extension.

Our comparison with state-of-the-art solutions, including static analysis tools, sanitizers, and other hardware-based detection techniques demonstrates that Mon CHÉRI complements static analysis and provides additional coverage where static analysis alone falls short. Moreover, conditional permissions (CPs) enable novel memory

access control policies, while being carefully designed to impose only minimal changes to the CHERI capability representation and thus have a better chance of real-world adoption than previous proposals that provide only `Write-before-Read` semantics, with significant limitations or invasive changes.

Looking ahead, we plan to extend Mon CHÉRI capabilities by integrating it with broader real-world use cases, emulation of automatic initialization, explore further optimization strategies, and reducing false positives in edge cases though compiler improvements.

Chapter 5

Conclusions

5.1 Summary of Contributions

This thesis shines a light on previously overlooked critical gaps in state-of-the-art memory-safety approaches—from programming languages to hardware design—and addresses these gaps. It considers three research questions, $\mathcal{RQ}1$–$\mathcal{RQ}3$. By answering these, the thesis addresses the following:

Firstly, $\mathcal{RQ}1$ asks how to provide an efficient solution to recover application state and maintain availability after a memory-corruption attack once it has been detected by state-of-the-art mitigations. Such detection approaches focus on preventing the exploitation of memory-safety vulnerabilities by terminating the application. Ensuring resiliency and availability when facing memory-corruption attacks, therefore, remains a gap. This thesis proposes a novel approach, secure domain rewind and discard (SDRaD), to improve the resiliency of applications. SDRaD allows applications to recover from memory-safety violations detected by existing mitigations by compartmentalizing applications to limit the scope of memory corruption caused by attacks.

Secondly, the adoption of memory-safe languages such as Rust has gained traction within both the open-source community and the wider software industry. However, the presence of unsafe code in Rust applications can still lead to memory-safety issues, remaining a gap in Rust's memory-safety and software-resiliency guarantees. $\mathcal{RQ}2$ asks how approaches that address $\mathcal{RQ}1$ can be adapted to protect Rust code in multi-language applications. To address this gap, this thesis proposes an easy-to-use application programming interface (API) to isolate unsafe parts of Rust applications. This ensures that the memory-safety guarantees of safe Rust code are upheld in the presence of unsafe Rust code or in multi-language applications calling into C libraries,

and additionally improves Rust's resilience against memory-safety violations in such unsafe code at runtime.

Lastly, considering the billions of lines of C and C++ written to date, the transition to memory-safe languages will be challenging. This highlights why hardware-based approaches, such as CHERI, are promising for widespread memory safety: they allow running legacy C and C++ codebases with improved memory safety without a wholesale rewrite. The baseline CHERI design exhibits gaps in the memory-safety properties it provides, such as a lack of initialization safety. $\mathcal{RQ}3$ asks how CHERI can be extended to prevent uninitialized-memory access. To address this gap, this thesis presents Mon CHÉRI, a software–hardware co-design to enhance initialization safety for CHERI.

As cybersecurity organizations continue to advocate for memory-safety practices, this thesis assesses a wide range of current practices and identifies gaps in *improving software resilience and availability* and in *preventing uninitialized-memory access* in both programming languages and hardware design.

While the focus of this thesis is on bridging overlooked gaps in memory-safety defenses, it has also yielded complementary contributions: $\mathbb{C}\mathbb{C}1$ introduces *lazypoline*, an efficient and comprehensive syscall interposition mechanism; $\mathbb{C}\mathbb{C}2$ presents SandCell, a compiler-assisted automated compartmentalization framework for Rust beyond unsafe code as a follow-up to $\mathbb{C}1$ and $\mathbb{C}2$; $\mathbb{C}\mathbb{C}3$ offers an evaluation of memory-safety defenses, including stack canaries and shadow stacks, assessing their effectiveness and performance; $\mathbb{C}\mathbb{C}4$ proposes BLACKOUT, an extension to memory-safe hardware that enforces data-oblivious computation to harden software against timing side channels; and $\mathbb{C}\mathbb{C}5$ discusses environmental sustainability considerations related to $\mathbb{C}1$ and $\mathbb{C}2$.

Taken together, $\mathbb{C}1$ and $\mathbb{C}2$ are especially important for building fault-tolerant, highly available software for critical infrastructure. While availability remains a primary objective alongside confidentiality and integrity, many security mechanisms can, in practice, undermine availability. This thesis highlights software designs that not only mitigate memory-safety issues but also preserve system availability and resiliency when such issues are detected. These results pave the way for more robust, reliable systems.

Further, this thesis recognizes that the industry's shift toward memory-safe languages will be gradual. Simply adopting such languages and allowing safe interoperation with legacy memory-unsafe code is not sufficient. $\mathbb{C}2$ and $\mathbb{C}\mathbb{C}2$ introduce techniques that strengthen the memory-safety guarantees of Rust applications even in the presence of legacy unsafe code, thereby improving the security and reliability of Rust-based systems throughout this transition.

Finally, $\mathbb{C}3$ and $\mathbb{C}\mathbb{C}3$ contribute to the state-of-the-art CHERI memory-safe hardware architecture. They set a precedent for future processors to incorporate more

comprehensive memory-safety features and hardening against side-channel leakage. Looking ahead, feature hardware architecture design should treat security, including memory safety and side-channel leakage, as a first-class, comprehensive design goal from the beginning.

5.2 Limitation & Future Work

Software Resilience. The secure rewind and discard approach (ℂ1 ℂ2) proposed in this thesis relies on Memory Protection Keys (MPK) on the 64-bit x86 (x86-64) architecture as its underlying hardware mechanism for isolation, which is not fully secure by design for the type of in-process isolation necessary for improving software resilience. In the complementary contributions ℂℂ2, we further harden MPK. Even though ℂ1 and ℂ2 evaluations show promising results for developer effort and performance characteristics, future work should be conducted across all hardening for underlying security primitives to ensure a fair comparison. However, the core concept of secure rewind and discard is not limited to MPK and can be applied to a broader range of software- and hardware-assisted compartmentalization techniques.

Although building resilient software using CHERI as the underlying detection and isolation mechanism is not in scope of this thesis, CHERI has characteristics that make it well-suited as a building block for rewind-and-discard solutions. In particular, CHERI enables earlier detection of memory attacks compared to traditional defenses such as stack canaries. This reduces an attacker's ability to perform arbitrary memory writes, thereby reducing the need for isolation. On the other hand, software with built-in resiliency still needs reliable rewind points from which to resume operation should memory errors be encountered. This implies a degree of logical compartmentalization that isolates the effects of faults and recovers from them safely. Future work could further explore the design space of secure rewind and discard, particularly in conjunction with emerging hardware memory-safety features such as CHERI.

In-process Isolation. Independently of its uses for improving software resiliency, in-process isolation continues to receive attention from both academia and industry. For example, RPAL (Run Process As Library) was recently proposed by ByteDance [205] as a way to improve the performance of multi-process applications by running such processes within the same virtual memory space, thus avoiding the need for inter-process communication (IPC) that requires crossing the process and kernel boundaries. By bypassing the Linux kernel for communication and scheduling, RPAL promises performance improvements to IPC-heavy applications. Another similar effort is co-located processes (co-processes), which enhance compartmentalization by improving UNIX IPC and reducing context-switching overhead by placing multiple processes

within the same address space while isolating them using CHERI capabilities. Although removing traditional IPC mechanisms improves performance, challenges remain similar to those highlighted in this thesis (see ℂ2), such as minimizing the overhead associated with passing objects across distinct isolated domains.Future work could explore the relevance of the techniques developed in ℂ2 in such alternative use cases, and optimize the serialization process to reduce this overhead. Another of the limitations of the secure rewind and discard approach is that side effects from certain operations cannot be undone. For example, system calls that perform external I/Os that can not be reversed are not suitable for rewindable compartments. A mechanism of buffering must be implemented to properly execute them outside compartments, thus requiring heavy modifications of the source code that developers might be frightened to attempt. Future work could explore techniques to automatically identify and isolate such side-effecting operations.

An alternative approach to achieving in-process isolation is retrofitting isolation into existing applications that were not originally designed with compartmentalization as a first-class concern. However, this typically results in significant codebase modifications. In contrast, when designing new software, it is essential for developers to treat compartmentalization, modularity, and fault recovery (e.g., secure rewind in the event of a memory error) as foundational design principles.

Rust. Rust is one of the most promising memory-safe languages, gaining widespread adoption in many open-source projects. One reason for Rust's success is that its language constructs are designed to support static analyses and enforce specific memory-safety guarantees. Future work could consider resilience and compartmentalization similarly: what kinds of language constructs would be needed to make them first-class language features rather than retrofitting applications through APIs? However, best practices around the use of `unsafe` in Rust remain poorly defined, and no formal language specification currently exists to guide or evaluate its use. Even as the transition to memory-safe languages like Rust progresses, writing Rust with `unsafe` blocks does not eliminate memory-safety risks. Furthermore, soundness bugs in the Rust compiler can arise even in safe code, due to compiler- or language-level issues.

CHERI. Capabilities offer a useful basis for novel protection mechanisms such as conditional- and blinded capabilities (see ℂ3 and ℂℂ4). However, as a limitation, ℂ3 cannot provide sparse initialization that might be desirable for existing code bases. Future work has room to develop the capability concept for both sparse initialization and other memory-safety properties, e.g., type safety, data-race safety, or use-after-free gap. In the meantime, CHERI is already a promising technology for adoption. Industrial research can aim to reduce CHERI's overhead further through more mature microarchitectural realizations and compiler support.

5.3 Concluding Remarks

Achieving full memory safety has been a challenge for decades, and there is no single solution that offers comprehensive protection. Instead, memory safety must be addressed from multiple angles, beginning in the software design phase and continuing through production-ready code. For example, this can include choosing a memory-safe programming language, enabling compiler-based hardening flags, or selecting memory-safe hardware. Additionally, enhancing software resilience is just as crucial as identifying and mitigating memory-safety vulnerabilities.

Appendix A

Additional Resources for Rewind & Discard: Improving Software Resilience using Isolated Domains

A.1 Supplementary Measurements

Table A.1 shows the detailed results for our rollback latency measurements for Memcached. The *Rollback latency* column gives the mean latency in μs over 1000 rollback iterations and the σ column the standard deviation. We measured both the rollback that destroys the offending domain but leaves the contents of its memory intact and a version of the rollback that "scrubs" (zeroes) the full contents of domain memory before it can be re-allocated, as indicated in the *Zeroing of domain data column*. The latter estimates the upper bound for maintaining confidentiality guarantees for nested domains using a 4GB heap pool and 4MB domain stack thats store sensitive information as discussed in Section 2.6. As rollbacks are exceptional events, we deem the $0.2s$ latency reasonable for use cases with confidentiality requirements. The latency could be reduced by tracking and scrubbing only allocations that contain sensitive data, or reducing the reserved memory for confidential domains.

Table A.3 shows the detailed results of our memory consumption measurements for Memcached. The *Maximum resident set size* column reports the mean maximum resident size (RSS) reported by the Bash shell builtin `time` command over five iterations

of the YCSB benchmark loading phase for the unmodified baseline (*Baseline*) and Memcached equipped with SDRaD (*SDRoB*). The columns marked σ give the relative standard deviation of the RSS. We used used four worker threads for this experiment as indicated in the *#Thr* column.

Table A.5 shows the detailed result of our throughput measurements of the YCSB benchmark for Memcached using different numbers of threads (*#Thr*) that were summarized in Section 2.5. The *Throughput* column gives the throughput in operations / seconds of three versions of Memcached: 1) The unmodified baseline (*Baseline*), 2) Memcached using the TLSF allocator (*TLSF*), and 3) Memcached equipped with SDRaD (*SDRoB*). The columns marked σ give the relative standard deviation of the throughput over ten benchmark runs as percentage for the aforementioned versions. The *Throughput degradation* column gives the degradation of throughput in percentage of (a) Memcached using the TLSF allocator compared to the baseline (*TLSF/Baseline*), (b) Memcached equipped with SDRaD compared to Memcached using the TLSF allocator (*SDRaD/TLSF*), and (c) Memcached equipped with SDRaD compared to baseline (*SDRaD/Baseline*).

Table A.2 shows the detailed results for our rollback latency measurements for NGINX. The columns are similar to those in Table A.1.

Table A.4 shows the detailed results of our memory consumption measurements for NGINX. We report the mean maximum RSS over 10 iterations of the NGINX benchmark for the unmodified baseline (*Baseline*) and NGINX equipped with SDRaD (*SDRoB*). The columns marked σ give the relative standard deviation of the RSS. We used 4 workers processes for this experiment as indicated in the *#Wkr* column.

Table A.7 shows the detailed result of our throughput measurements of the ab tool for NGINX using different file sizes starting from 0KiB to 128KiB with different worker processes that were summarized in Section 2.5. The *Throughput* column gives the throughput in requests / seconds of three versions of NGINX: 1) The unmodified baseline (*Baseline*), 2) NGINX using the TLSF allocator (*TLSF*), and 3) NGINX equipped with SDRaD (*SDRoB*). The columns marked σ give the relative standard deviation of the throughput over five benchmark runs as percentage for the aforementioned versions. The *Throughput degradation* column gives the degradation of throughput in percentage of (a) NGINX using the TLSF allocator compared to the baseline (*TLSF/Baseline*), (b) NGINX equipped with SDRaD compared to NGINX using the TLSF allocator (*SDRaD/TLSF*), and (c) NGINX equipped with SDRaD compared to baseline (*SDRaD/Baseline*).

Table A.6 the detailed result of our throughput measurements of the OpenSSL benchmark tool for different input sizes from 16 to 262144 bytes that were summarized in Section 2.5. The *Throughput* column gives the throughput in 1000s of bytes / seconds of three versions of OpenSSL: SDRaD 1. – 3. correspond to the three different design

Table A.1: Memcached: Rollback latency

Zeroing of domain data	Rollback latency (μs) SDRaD	σ
No (default)	3.46	$\pm 0.9\mu s$
Yes	228376.99 ($0.2s$)	$\pm 2500\mu s$

Table A.2: NGINX: Rollback latency

Zeroing of domain data	Rollback latency (μs) SDRaD	σ
No (default)	3.41	$\pm 0.7\mu s$
Yes	232632,4168 ($0.2s$)	$\pm 2400\mu s$

Table A.3: Memcached: Memory consumption

#Thr	Maximum resident set size (KiB) Baseline	σ	SDRaD	σ
4	14535444.8	±1.19%	14598972.8	±1.18%

Table A.4: NGINX: Memory consumption

#Wkr	Maximum resident set size (KiB) Baseline	σ	SDRaD	σ
4	3135.6	±1.59%	3234.8	±1.71%

choices for the `EVP_EncryptUpdate()` wrapper described in Section 2.4.1: SDRaD 1. – OpenSSL domain has read-only access to parent, SDRaD 2. – the wrapper is responsible for copying input/output via an intermediate data domain, SDRaD 3. – the parent domain sets up a shared data domain which the OpenSSL domain in the wrapper can access directly.

Table A.5: Memcached: Detailed table of throughput measurements.

#Thr	Throughput (ops/sec)						Throughput degradation (%)		
	Baseline	σ	TLSF	σ	SDRoB	σ	TLSF/Baseline	SDRaD/TLSF	SDRaD/Baseline
Loading Phase									
1	83913	±0.69%	84191	±0.46%	78047	±0.89%	+0,33%	-7.30%	-6.99 %
2	117194	±1.05%	117275	±0.18%	111838	±0.40%	+0,07%	-4.64%	-4.57 %
4	157920	±1.24%	157982	±0.43%	153328	±0.63%	+0,04%	-2.95%	-2.91 %
8	161161	±0.80%	161448	±0.75%	160910	±1.13%	+0,18%	-0.33%	-0.16 %
Running Phase									
1	99722	±0.46%	99829	±0,52%	92645	±0,23%	+0,11%	-7,20%	-7,10%
2	155247	±0.54%	155015	±0,51%	146715	±0,17%	-0,15%	-5,35%	-5,50%
4	223958	±0.20%	223886	±0,41%	214671	±0,54%	-0,03%	-4,12%	-4,15%
8	234823	±0.76%	235007	±0,55%	225277	±0,62%	+0,08%	-4,14%	-4,07%

Table A.6: OpenSSL: Detailed table of throughput measurements.

Input	Throughput (1000s of bytes/sec)								Throughput degradation (%)		
bytes	Baseline	σ	SDRaD 1.	σ	SDRaD 2.	σ	SDRaD 3.	σ	1./Baseline	2./Baseline	3./Baseline
2^4	267879	±1.04%	53570	±2.08%	33995	±1.41%	53851	±3.02%	-80.00%	-87.31%	-79.90%
2^6	724060	±1.47%	196647	±1.84%	127678	±1.02%	201048	±2.88%	-72.84%	-82.37%	-72.23%
2^8	1508889	±2.11%	616788	±1.79%	1192933	±1.32%	639910	±2.07%	-59.12%	-71.59%	-57.59%
2^{10}	2515179	±1.85%	1551532	±1.66%	428704	±1.74%	1610508	±1.76%	-38.31%	-52.57%	-35.97%
2^{13}	3073966	±1.96%	2744814	±1.17%	2503868	±1.51%	2842073	±1.10%	-10.71%	-18.55%	-7.54%
2^{14}	3139433	±1.78%	2856443	±1.11%	2567001	±1.18%	3025082	±1.52%	-9.01%	-18.23%	-3.64%
2^{15}	3167069	±1.16%	2855263	±1.13%	2577581	±1.48%	3111608	±1.35%	-9.85%	-18.61%	-1.75%
2^{16}	3184860	±1.18%	2904903	±0.71%	1532381	±1.60%	3169666	±1.23%	-8.79%	-51.89%	+0.28%
2^{18}	3198742	±1.14%	2941696	±0.83%	2637291	±1.54%	3207550	±1.05%	-8.04%	-17.55%	+0.38%

Table A.7: NGINX: Detailed table of throughput measurements.

File size	Throughput (reqs./sec.)						Throughput degradation (%)		
(KiB)	Baseline	σ	TLSF	σ	SDRoB	σ	TLSF/Baseline	SDRaD/TLSF	SDRaD/Baseline
1 worker									
0	69386	±1.40 %	66953	±2.21%	65025	±1.04%	-2.96%	-2.75%	-5.63%
1	55454	±1.01 %	55484	±0.74%	51839	±1.48%	-0.31%	-6.17%	-6.46%
2	55184	±0.81 %	55273	±0.26%	52482	±0.70%	-0.21%	-4.93%	-5.13%
4	55506	±0.70 %	54753	±1.28%	52083	±1.21%	-1.33%	-4.88%	-6.14%
8	53413	±2.33 %	53574	±1.20%	51005	±0.72%	-0.14%	-4.51%	-4.37%
16	51302	±1.13 %	50841	±0.99%	49107	±1.44%	-0.78%	-3.61%	-4.36%
32	47702	±1.83 %	47498	±1.46%	45067	±0.95%	-0.02%	-5.39%	-5.42%
64	37049	±0.83 %	36687	±1.16%	35861	±1.00%	-0.83%	-2.51%	-3.31%
128	25852	±0.55 %	25808	±0.69%	25398	±0.76%	-0.06%	-1.54%	-1.60%
2 workers									
0	132013	±1.80%	131029	±1.97%	123723	±0.48%	-0.68%	-5.09%	-5.73%
1	106254	±0.61%	105192	±1.40%	99769	±1.36%	-1.36%	-4.95%	-6.24%
2	104029	±0.68%	104104	±1.35%	99469	±0.39%	-0.18%	-4.57%	-4.74%
4	104776	±1.21%	105309	±0.73%	99836	±1.02%	+0.30%	-5.37%	-5.09%
8	101995	±1.30%	101960	±0.56%	95991	±0.78%	+0.24%	-5.78%	-5.55%
16	98139	±0.44%	96381	±1.16%	92208	±0.49%	-1.74%	-4.32%	-5.99%
32	89601	±0.43%	89253	±0.62%	84653	±1.51%	-0.16%	-5.01%	-5.17%
64	67910	±1.31%	67249	±1.10%	64342	±1.34%	-0.95%	-4.40%	-5.31%
128	46904	±1.36%	46821	±0.49%	45525	±0.33%	-0.12%	-2.68%	-2.80%
4 workers									
0	258531	±3.90%	257575	±3.03%	243747	±3.03%	-0.37%	-5.37%	-5.72%
1	203020	±2.92%	202384	±2.45%	192917	±2.45%	-0.31%	-4.68%	-4.98%
2	202407	±2.61%	202701	±2.88%	190962	±2.88%	+0.15%	-5.79%	-5.65%
4	203955	±2.69%	204389	±3.70%	194757	±3.70%	+0.21%	-4.71%	-4.51%
8	198404	±3.43%	197599	±2.38%	187675	±2.38%	-0.41%	-5.02%	-5.41%
16	188374	±4.20%	188913	±2.48%	179768	±2.48%	+0.29%	-4.84%	-4.57%
32	172736	±2.38%	169735	±2.61%	161146	±2.61%	-1.74%	-5.06%	-6.71%
64	135227	±3.61%	132880	±1.76%	127736	±1.76%	-1.74%	-3.87%	-5.54%
128	93900	±2.62%	93940	±3.21%	91661	±3.21%	+0.04%	-2.43%	-2.38%

A.2 OpenSSL Example

The excerpts of code in Listing A.1 and Listing A.2 show an example usage of SDRaD with deeply nested domains as explained in Section 2.3.5. The excerpts show a simple file encryption server in an event-driven architecture that encrypts client data using OpenSSL and stores the ciphertext on the server.

Upon receiving a client request the `event_handler` function (Listing A.1) performs the following tasks:

① Reads the encryption key from a file chosen by user and generates a random initialization vector (IV).

② Calls `gcm_encrypt_user_data` (Listing A.2), which:

③ Reads a plaintext message from the user via a file descriptor that corresponds to a communication socket.

④ Encrypts said plaintext with AES GCM using the OpenSSL's "Envelope" (EVP) API.

⑤ Finally (Listing A.1), the `even_handler` stores the ciphertext (and Galois/-counter mode tag) for later retrieval.

The objective to introducing rollback capability to the event handler is to achieve the properties described in Section 2.4.1, namely to 1. protect the event handler running in the root domain from errors in `gcm_encrypt_user_data`, e.g., the possible overflow to the plaintext buffer (Listing A.2, ❾), 2. encapsulate the pointer to the OpenSSL context (`ctx`) within an outer, nested domain in such a way that OpenSSL's key objects remain inaccessible to `gcm_encrypt_user_data` should it malfunction, and 3. simplify error handling for the domain in which the OpenSSL code is run. To this end, the event handler (running in the root domain) creates a persistent domain for OpenSSL execution that is inaccessible from both the root domain and any nested domain (Listing A.1, ❶). This execution domain is immediately deinitialized to invalidate its saved execution context to avoid unintended rollbacks to the beginning of `event_handler`. The actual execution context for rollback (within another nested domain) is established later.

The event handler (still running in the root domain) creates two additional data domains (Listing A.1, ❷ and ❸). The first data domain (❷) is used to store the encryption key after it has been read from the file, and the random initialization vector (①). Both the root domain and the OpenSSL domain can access this domain area.

The second data domain (❸) is used for data that is shared between the OpenSSL domain and the domain `gcm_encrypt_user_data` will run in. This corresponds to

the third design option in Section 2.4.1 with respect to how the OpenSSL wrapper (Listing 2.2) is expected to operate. The OpenSSL domain is granted access to the data domain as the data domain is created (❸) The nested domain used to run the `gcm_encrypt_user_data()` function is first initialized, then granted shared access to the data domain (Listing A.1, ❹). The event handler (still running in the root domain) uses this shared data domain to allocate buffers that hold the plaintext, ciphertext, and the Galois/Counter Mode (GCM) tag which are either used to communicate data to the nested domains, or data back from them (Listing A.1, ❺).

The `gcm_encrypt_user_data()` function is then invoked inside the nested domain (Listing A.1, ②). It then re-initializes the OpenSSL domain to set it to use the nested domain's saved execution context upon an abnormal domain exit (Listing A.2, ❽). This avoids the need to establish individual rollback points for each individual OpenSSL invocation that execute in the dedicated persistent domain.

In case of an abnormal domain exit from *either* domain the execution of the `gcm_encrypt_user_data()` or OpenSSL is rolled back to the point of the second `sdrob_init()` in `event_handler` (Listing A.1, ❹). The return value of that call (if other than `SUCCESSFUL_RETURNED`) indicated the UDI of the domain that initiated the abnormal exit. When this occurs, the SDRaD library has already destroyed the offending execution domain, but the caller uses the returned UDI to determine any remaining cleanup operations, such as destroying any remaining domains (Listing A.1, ❻ and ❼). Note that in ❻, if the nested domain running Listing A.2 experienced and abnormal domain exit, the persistent OpenSSL domain still remains, but cannot be entered again until a new execution context for rollback is established. In principle the event handler could leave the domain intact (but deinitialized) at ❻ and reuse it the next time the event handler is called. For clarity, we show each domain explicitly destroyed in Listing A.1. In ❼, the persistent OpenSSL domain experienced an abnormal exit. As it was initialized by `gcm_encrypt_user_data()` to use the calling domains execution context for rollback (Listing A.2, ❽) SDRaD has automatically destroyed both the offending OpenSSL domain and the nested domain where `gcm_encrypt_user_data()` executed. Data domain are always left intact after rollback and must be explicitly destroyed.

On normal domain exit it is the responsibility of the event handler to deinitialize or destroy any domains before it exits, as explained in Section 2.4.2. Similar to above, the event handler could choose to leave any of the domains intact (but deinitialize) to reuse them the next time it is entered.

```
1  int event_handler(struct event_handler_args *args)
2  {
3      gcm_encrypt_user_data_args_t gcm_args;              // holds read-only arguments for gcm_encrypt_user_data()
4      register gcm_encrypt_user_data_args_t *gcm_args_p  asm("r12") = &gcm_args;  // pointer passed across stacks
5      register int ciphertext_len asm("r13");            // return value from nested domain held in callee-saved
           registers
6
7      if(sdrob_init(OPENSSL_UDI, EXECUTION_DOMAIN | INACCESSIBLE_DOMAIN | RETURN_HERE) !=  SUCCESSFUL_RETURNED) {
8          handleErrors(); /* If the OpenSSL domain initialization fails, don't continue*/
9      }                                                                                                  ❶
10     sdrob_deinit(OPENSSL_UDI);
11
12
13     if(sdrob_init(OPENSSL_PRIVATE_DATA_UDI, DATA_DOMAIN) !=  SUCCESSFUL_RETURNED) {
14         sdrob_destroy(OPENSSL_UDI, NO_HEAP_MERGE);
15         handleErrors();                                                                                ❷
16     }
17     sdrob_dprotect(OPENSSL_UDI, OPENSSL_PRIVATE_DATA_UDI, READ_ENABLE | WRITE_ENABLE);
18
19
20
21     if(sdrob_init(OPENSSL_SHARED_DATA_UDI, DATA_DOMAIN) != SUCCESSFUL_RETURNED) {
22         sdrob_destroy(OPENSSL_PRIVATE_DATA_UDI, NO_HEAP_MERGE);
23         sdrob_destroy(OPENSSL_UDI, NO_HEAP_MERGE);
24         handleErrors();                                                                                ❸
25     }
26     sdrob_dprotect(OPENSSL_UDI, OPENSSL_SHARED_DATA_UDI, READ_ENABLE | WRITE_ENABLE);
27
28
29     gcm_args.key = sdrob_malloc(OPENSSL_PRIVATE_DATA_UDI, AES_GCM_KEY_LEN);
30     if (read_key_from_file(key_p, AES_GCM_KEY_LEN, args->pathname) != 1){
31         handleErrors();
32     }                                                                                                  ①
33
34     gcm_args.iv = sdrob_malloc(OPENSSL_PRIVATE_DATA_UDI, AES_GCM_IV_LEN);
35     if (RAND_bytes(gcm_args->iv, sizeof(AES_GCM_IV_LEN)) != 1){
36         handlerErrors();
37     }
38
39
40     udi_t ret = sdrob_init(NESTED_DOMAIN_UDI, EXECUTION_DOMAIN| ACCESSIBLE_DOMAIN| RETURN_TO_CURRENT);
41                                                                                                        ❹
42     if(ret ==  SUCCESSFUL_RETURNED) {
43         sdrob_dprotect(NESTED_DOMAIN_UDI, OPENSSL_SHARED_DATA_UDI, READ_ENABLE | WRITE_ENABLE);
44
45
46         gcm_args.ciphertext = sdrob_malloc(OPENSSL_SHARED_DATA_UDI,  CIPHER_TEXT_LEN);
47         gcm_args.plaintext = sdrob_malloc(OPENSSL_SHARED_DATA_UDI, args->plaintext_len);               ❺
48         gcm_args.tag = sdrob_malloc(OPENSSL_SHARED_DATA_UDI, TAG_SIZE);
49
50         sdrob_enter(NESTED_DOMAIN_UDI);
51         ciphertext_len = gcm_encrypt_user_data(gcm_args_p); ②
52         sdrob_exit();
53         sdrob_destroy(NESTED_DOMAIN_UDI, NO_HEAP_MERGE);
54     } else {
55         switch (ret) {
56             case NESTED_DOMAIN_UDI:
57                 sdrob_destroy(OPENSSL_UDI, NO_HEAP_MERGE);
58                 sdrob_destroy(OPENSSL_SHARED_DATA_UDI, NO_HEAP_MERGE);                                 ❻
59                 sdrob_destroy(OPENSSL_PRIVATE_DATA_UDI, NO_HEAP_MERGE);
60             break;
61             case OPENSSL_UDI:
62                 sdrob_destroy(OPENSSL_SHARED_DATA_UDI, NO_HEAP_MERGE);                                 ❼
63                 sdrob_destroy(OPENSSL_PRIVATE_DATA_UDI, NO_HEAP_MERGE);
64             break;
65             default:
66                 abort();
67         }
68         return OPERATION_FAILED;
69     }
70
71     // store ciphertext and tag for later retrieval  ⑤
72     sdrob_destroy(OPENSSL_UDI, NO_HEAP_MERGE);
73     sdrob_destroy(NESTED_DOMAIN_UDI, NO_HEAP_MERGE);
74     sdrob_destroy(OPENSSL_SHARED_DATA_UDI, NO_HEAP_MERGE);
75     sdrob_destroy(OPENSSL_PRIVATE_DATA_UDI, NO_HEAP_MERGE);
76 }
```

Listing A.1: An excerpt from an example file encryption server. The except shows the event handler code that has been augmented to perform the SDRaD domain management, executing in the root domain. Some error handling has been omitted for brevity.

```
1  /* The gcm_encrypt_user_data_args_t structure that is passed by reference
2     is actually allocated on the root domain stack and hence read-only. */
3  int gcm_encrypt_user_data(const gcm_encrypt_user_data_args_t *args) {
4      ssize_t len = 0;
5      ssize_t num_bytes_read = 0;
6      ssize_t ciphertext_len = 0;
7      EVP_CIPHER_CTX *ctx;
8
9      if(sdrob_init(OPENSSL_UDI, EXECUTION_DOMAIN| INACCESSIBLE_DOMAIN| RETURN_TO_PARENT) !=  SUCCESSFUL_RETURNED)
           {    ❺
10         return GCM_ENCRYPT_FAILED;
11     }
12
13     if((ctx = EVP_CIPHER_CTX_new()) == NULL) {                          // create and initialize the OpenSSL
           context
14         sdrob_deinit(OPENSSL_UDI);                                      // deinitialize domain before
           returning
15         return GCM_ENCRYPT_FAILED;
16     }
17
18     if(EVP_EncryptInit_ex(ctx, EVP_aes_256_gcm(), NULL, NULL, NULL) != 1)  // initialize cipher for encryption
19         goto err_out;
20
21     if(EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_SET_IVLEN, args->iv_len, NULL) != 1) //  set length of IV
22         goto err_out;
23
24     if(EVP_EncryptInit_ex(ctx, NULL, NULL, args->key, args->iv) != 1)       // load key and iv from private data
           domain
25         goto err_out;
26
27 ┌   while(num_bytes_read < plaintext_len) {
28 ③       num_bytes_read += read(fd, plaintext, 1024); ❻
29 └   }
30
31
32 ┌   if(EVP_EncryptUpdate(ctx, NULL, &len, args->aad, args->aad_len) != 1)  // provide any additional
       │   authentication data
33 │       goto err_out;
34 │
35 │   if(EVP_EncryptUpdate(ctx, args->ciphertext, &len, args->plaintext, args->plaintext_len) != 1)
36 │       goto err_out;
37 │   ciphertext_len = len;
38 ④ │
39 │   if(EVP_EncryptFinal_ex(ctx, args->ciphertext + len, &len) != )        //  finalize the encryption
40 │       goto err_out;
41 │   ciphertext_len += len;
42 │
43 │   if(EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_GET_TAG, TAG_SIZE, args->tag) != 1)  // read the tag to the shared
   │       domain
44 └       goto err_out;
45
46
47     sdrob_deinit(OPENSSL_UDI);                                          // deinitialize domain before
           returning
48     EVP_CIPHER_CTX_free(ctx);                                           // success, cleanup and return
49     return ciphertext_len;
50
51 err_out:                                                                // normal error occurred, cleanup and
       return
52     sdrob_deinit(OPENSSL_UDI);                                          // deinitialize domain before
           returning
53     EVP_CIPHER_CTX_free(ctx);
54     return GCM_ENCRYPT_FAILED;
55 }
```

Listing A.2: The `gcm_encrypt_user_data` function reads a plaintext message from a descriptor provided as argument and encrypts the plaintext with AES GCM using OpenSSL's high-level, envelope (EVP) API. The call to the EVP API functions have been wrapped to execute in domain `OPENSSL_UDI` as shown in Listing 2.2 in Section 2.4.1.

Appendix B

Additional Resources for Friend or Foe Inside? Exploring In-Process Isolation to Maintain Memory Safety for Unsafe Rust

As supplemental material, we provide all measurements for results reported in the this report.

Table B.1 shows execution times for running *snappy* with different versions of Abomonation under different optmization levels for different input sizes. These are the numbers underlying Figure 3.3a.

Similarly, the numbers underlying the comparison of SDRaD-FFI with different serialization crates and Sandcrust w.r.t. performance of compression and uncompression with *snappy* as shown in Figure 3.3b are presented in Table B.2.

Finally, we show the evaluation results for the comparison with TRUST and Sandcrust in Table 3.3. As the evaluation of TRUST uses different input sizes for *snappy* we ran a separate experiment using the sizes from Figure 10 in [19], except for 4GiB and 16GiB that where too large to be allocated contiguously by the SDRaD library. We evaluated the run-time of the baseline and SDRaD-FFI with Abomonation over 5000 iterations for each input size. As Sandcrust runs incredibly slow for large input sizes, we only

Table B.1: Snappy: Detailed table of Execution Times measurements (Figure 3.3a)

Abomonation V1								
Input bytes	Execution Times [μs] (compress)							
	SDRaD-FFI opt-0	σ	SDRaD-FFI opt-1	σ	SDRaD-FFI opt-2	σ	SDRaD-FFI opt-3	σ
2^0	2,36	±21,38%	1,26	±128,68%	1,09	±369,58%	1,09	±55,19%
2^3	2,71	±18,15%	1,30	±29,44%	1,08	±37,72%	1,08	±36,99%
2^6	5,06	±11,92%	1,69	±22,25%	1,16	±41,28%	1,15	±37,77%
2^9	21,72	±5,87%	4,83	±17,84%	1,38	±34,57%	1,37	±33,95%
2^{12}	151,60	±2,56%	28,37	±10,38%	2,10	±25,81%	2,10	±23,71%
2^{15}	1190,04	±2,13%	214,85	±4,63%	7,27	±14,34%	7,32	±14,34%
2^{18}	9586,14	±2,45%	1770,13	±2,99%	104,06	±7,61%	103,06	±6,70%
2^{21}	76327,93	±1,57%	14688,60	±2,41%	1099,66	±2,61%	1100,45	±2,49%
	Execution Times [μs] (uncompress)							
2^0	2,21	±64,37%	1,20	±122,08%	1,01	±281,91%	1,00	±39,60%
2^3	2,52	±18,29%	1,23	±31,15%	0,99	±35,18%	0,99	±36,07%
2^6	4,73	±10,93%	1,59	±22,25%	1,03	±38,94%	1,01	±36,88%
2^9	21,17	±5,91 %	4,60	±18,21%	1,11	±33,51%	1,07	±36,54%
2^{12}	150,54	±2,66 %	27,74	±8,76%	1,49	±31,92%	1,39	±32,79%
2^{15}	1188,71	±2,14 %	213,98	±4,65%	6,07	±15,29%	6,00	±15,35%
2^{18}	9574,67	±2,46 %	1758,30	±3,01 %	91,19	±7,88%	91,58	±8,68%
2^{21}	76266,24	±1,59 %	14561,62	±2,43 %	1025,46	±1,66%	1023,86	±1,81%

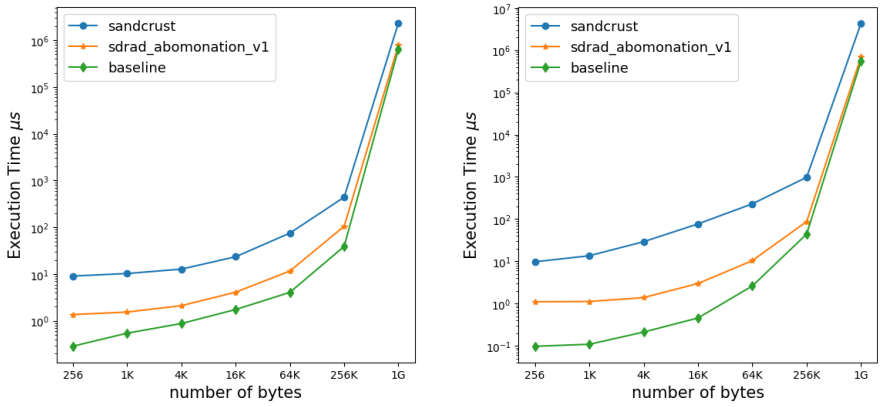
Abomonation V2								
Input bytes	Execution Times [μs] (compress)							
	SDRaD-FFI opt-0	σ	SDRaD-FFI opt-1	σ	SDRaD-FFI opt-2	σ	SDRaD-FFI opt-3	σ
2^0	2,21	±29,13%	1,14	±24,88%	1,16	±343,80%	1,10	±234,90%
2^3	2,19	±14,92%	1,16	±28,66%	1,15	±33,56%	1,08	±32,85%
2^6	2,46	±15,26%	1,23	±23,15%	1,21	±33,60%	1,15	±37,36%
2^9	2,76	±14,37%	1,44	±20,05%	1,40	±31,28%	1,36	±31,84%
2^{12}	3,58	±13,00%	2,24	±19,37%	2,12	±25,50%	2,10	±27,84%
2^{15}	8,55	±12,14%	7,54	±12,71%	7,36	±18,49%	7,38	±19,03%
2^{18}	109,78	±4,77%	104,18	±6,51%	105,98	±8,49%	104,20	±6,73%
2^{21}	1058,57	±1,70%	1108,83	±4,70%	1112,27	±2,07%	1107,01	±1,87%
	Execution Times [μs] (uncompress)							
2^0	2,12	±29,13%	1,07	±102,90%	1,03	±63,94%	1,00	±78,57%
2^3	2,10	±14,92%	1,05	±31,62%	1,03	±32,94%	0,99	±46,59%
2^6	2,20	±15,26%	1,06	±21,61%	1,03	±43,21%	1,01	±37,33%
2^9	2,29	±14,37%	1,13	±20,62%	1,07	±41,57%	1,07	±40,21%
2^{12}	2,67	±13,00%	1,46	±21,55%	1,38	±31,92%	1,40	±33,27%
2^{15}	7,12	±12,14%	6,30	±13,67%	6,08	±14,52%	6,15	±22,39%
2^{18}	93,67	±4,77%	92,20	±6,96%	94,11	±8,28%	90,25	±7,93%
2^{21}	1004,03	±1,70%	1024,66	±2,57%	1022,25	±1,71%	1027,27	±1,70%

measured each input size 50 times at the cost of a larger standard deviation.

The resulting execution times from our experiments are depicted in Figure B.1 based on the raw numbers shown in Table B.3. We calculated the geometric mean of the measured run times and those given in [19] and computed the corresponding average overheads given in Table 3.3.

Table B.2: Snappy: Detailed table of Execution Times measurements (Figure 3.3b)

Input bytes	Execution Times [μs] (compress)									
	Baseline	σ	SDRaD-FFI Abomonation	σ	SDRaD-FFI Bincode	σ	SDRaD-FFI Bincode-2	σ	Sandcrust	σ
2^0	0,11	±102,43%	1,27	±30,41%	1,24	±16,82%	0,99	±26,65%	8,88	±9,44%
2^3	0,11	±97,86%	1,23	±27,37%	1,41	±19,99%	1,13	±25,06%	8,78	±9,17%
2^6	0,20	±62,73%	1,30	±21,69%	2,52	±15,53%	1,57	±30,86%	8,92	±7,67%
2^9	0,36	±6,36%	1,55	±10,99%	6,62	±13,71%	3,77	±17,24%	9,44	±3,18%
2^{12}	0,84	±22,29%	2,22	±25,54%	35,06	±6,97%	19,89	±9,03%	13,57	±7,20%
2^{15}	2,41	±20,36%	7,30	±10,74%	249,71	±4,86%	148,37	±4,76%	37,06	±6,48%
2^{18}	20,00	±30,98%	101,78	±6,31%	1997,97	±3,05%	1222,46	±3,43%	391,08	±3,68%
2^{21}	246,23	±4,31%	1075,22	±2,43%	16318,68	±2,42%	10177,26	±2,14%	2579,06	±8,71%
	Execution Times [μs] (uncompress)									
2^0	0,09	±85,08%	1,19	±37,24%	1,21	±50,42%	0,88	±26,07%	8,86	±8,02%
2^3	0,08	±105,91%	1,16	±34,33%	1,46	±16,82%	1,02	±28,34%	8,78	±9,58%
2^6	0,09	±14,70%	1,17	±19,37%	2,37	±15,58%	1,39	±30,36%	9,05	±7,58%
2^9	0,10	±15,53%	1,27	±9,05%	6,38	±13,84%	3,40	±18,17%	11,28	±2,64%
2^{12}	0,19	±70,87%	1,53	±29,06%	34,66	±7,02%	19,16	±8,86%	29,77	±4,60%
2^{15}	1,08	±30,27%	6,09	±12,32%	248,15	±4,88%	147,02	±4,76%	115,99	±3,33%
2^{18}	17,39	±45,23%	88,07	±7,18%	1987,48	±3,06%	1207,99	±3,33%	905,96	±3,80%
2^{21}	618,29	±59,77%	993,67	±1,72%	16257,90	±2,42%	10166,90	±2,17%	6478,21	±6,17%



(a) compress() benchmark

(b) uncompress() benchmark

Figure B.1: Measuring snappy execution time for SDRaD-FFI

Table B.3: Snappy: Detailed table of Execution Times measurements (Figure B.1)

Input	Execution Times [μs] (compress)					
bytes	Baseline	σ	SDRaD-FFI	σ	Sandcrust	σ
256	0,29	±56,55%	1,36	±24,47%	9,08	±4,73%
1K	0,54	±11,11%	1,54	±2,49%	10,29	±16,71%
4K	0,88	±17,26%	2,11	±3,95%	12,77	±7,95%
16K	1,75	±15,39%	4,10	±7,58%	23,49	±7,23%
64K	4,07	±60,39%	11,61	±4,26%	75,22	±8,26%
256K	39,07	±70,67%	105,19	±2,97%	440,61	±3,85%
1GB	646610,97	±1,07%	791444,04	±0,28%	2343068,37	±0,65%
GeoMean	11,37		29,08		155,28	
	Execution Times [μs] (uncompress)					
	Baseline	σ	SDRaD-FFI	σ	Sandcrust	σ
256	0,10	±7,40%	1,10	±47,15%	9,74	±1,66%
1K	0,11	±4,95%	1,12	±25,85%	13,54	±3,71%
4K	0,21	±9,50%	1,38	±4,28%	29,16	±12,67%
16K	0,45	±6,42%	2,98	±12,65%	76,88	±25,37%
64K	2,61	±78,54%	10,38	±4,75%	229,61	±4,15%
256K	44,29	±82,15%	87,53	±6,23%	979,81	±1,25%
1GB	550920,53	±1,26%	701423,77	±0,34%	4362162,32	±0,68%
GeoMean	4,86		22,80		312,35	

Appendix C

Additional Resources for Mon CHÉRI: Mitigating Uninitialized Memory Access with Conditional Capabilities

C.1 Example of Avoided Data Hazard

```
/* ... */
   csetbounds       ca0, ca0, 4
① csetwbrbound      ca0, ca0, zero
   li               a1, 10
② csw               a1, 0(ca0)
③ clw               a0, 0(ca0)
   clc              cra, 16(csp)
/* ... */
```

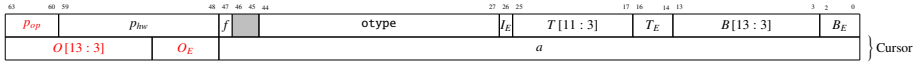
Listing C.1: Listing 2: Example of instruction sequence causing data hazard as described in Section 4.4. At ①, the `csetwbrbound` instructions sets the `Write-before-Read` bound for the capability in register `ca0`, turning it into a conditional capability. At ②, the capability store word (`csw`) instructions performs a store operation on `ca0`, whereupon the operation bound of the capability in `ca0` is increased at Stage-2 of the pipeline. At the same time, the capability load word (`clw`) instruction at ③ has already entered the pipeline and prepares a read via the same capability in `ca0`. Without the the bypass allowing the updated operation bound to be forwarded from from Stage-2 to Stage-1, this instruction would fail due to the operation bound check on the out-of-date bound. With the bypass, this sequence of instructions is valid and does not incur additional latency.

C.1.1 Changes to Capability Encoding

C.2 Examples from the Juliet Test Suite

```
1  void CWE457_Use_of_Uninitialized_Variable__double_64b_goodB2GSink(void *
       dataVoidPtr)
2  {
3      /* cast void pointer to a pointer of the appropriate type */
4      double * dataPtr = (double *)dataVoidPtr;
5      /* dereference dataPtr into data */
6   ③  volatile double data = (*dataPtr);
7      /* FIX: Ensure data is initialized before use */
8   ④  data = 5.0;
9      printDoubleLine(data);
10 }
11
12 static void goodB2G()
13 {
14  ①  volatile double data;
15     /* POTENTIAL FLAW: Don't initialize data */
16     ; /* empty statement needed for some flow variants */
17  ②  CWE457_Use_of_Uninitialized_Variable__double_63b_goodB2GSink(&data);
18 }
19
20 void CWE457_Use_of_Uninitialized_Variable__double_63_good()
21 {
22     goodG2B();
23     goodB2G();
24 }
```

Listing C.2: A "good" example from the Juliet Test Suite [163] where Mon CHÉRI detects an uninitialized variable use. A reference to an uninitialized `volatile double data` ① is passed in a function call ② and dereferenced and copied ③ in the callee where Mon CHÉRI detects an uninitialized load. The memory content of the copy is initialized before use in ④. The example is from `C/testcases/CWE457_Use_of_Uninitialized_Variable/s01/CWE457_Use_of_Uninitialized_Variable__double_63a.c` in the official test suite release.



f: flag p: permissions p_{op}: CP control bits `otype`: object type a: pointer address

If $I_E = 0$:

$$E = 0$$
$$T\,[2:0] = T_E$$
$$B\,[2:0] = B_E$$
$$O\,[2:0] = O_E\,[4:2]$$
$$L_{carry\_out} = \begin{cases} 1, & \text{if } T\,[11:0] < B\,[11:0] \\ 0, & \text{otherwise} \end{cases}$$
$$L_{msb} = 0$$

If $I_E = 1$ and $E = 1, 2$:

$$E = \{T_E, B_E\}$$
$$T\,[2:0] = 0$$
$$B\,[2:0] = 0$$
$$O\,[2:0] = O_E\,[E+2:E]$$
$$L_{carry\_out} = \begin{cases} 1, & \text{if } T\,[11:3] < B\,[13:3] \\ 0, & \text{otherwise} \end{cases}$$
$$L_{msb} = 1$$

Reconstituting the top two bits of T:

$$T\,[13:12] = B\,[13:12] + L_{carry\_out} + L_{msb}$$

Decoding the bounds:

address, a =	$a_{top} = a\,[47 : E + 14]$	$a_{mid} = a\,[E + 13 : E]$	$a_{low} = a\,[E - 1 : 0]$
top, t =	$a_{top} + c_t$	$T\,[13:0]$	$0'E$
bottom, b =	$a_{top} + c_b$	$B\,[13:0]$	$0'E$
			If $I_E = 0$:
			$O'E$
operation o =	$a_{top} + c_o$	$O\,[13:0]$	If $I_E = 1$ and $E = 1, 2$:
			$O_E\,[E-1:0]$

To calculate the corrections c_t, c_b and c_o:

$$A_3 = a\,[E + 13 : E + 11] \tag{C.1}$$
$$B_3 = B\,[13:11] \tag{C.2}$$
$$T_3 = T\,[13:11] \tag{C.3}$$
$$O_3 = O\,[13:11] \tag{C.4}$$
$$R = B_3 - 1 \tag{C.5}$$

$A_3 < R$	$T_3 < R$	c_t
false	false	0
false	true	+1
true	false	−1
true	true	0

$A_3 < R$	$B_3 < R$	c_b
false	false	0
false	true	+1
true	false	−1
true	true	0

$A_3 < R$	$O_3 < R$	c_o
false	false	0
false	true	+1
true	false	−1
true	true	0

Figure C.1: Compressed 128-bit capability format and decoding (adapted from [235] with additions and changes marked in red).

```
__attribute__((writebeforeread, noinline))
void CWE457_Use_of_Uninitialized_Variable__int_array_alloca_partial_init_64_bad()
{
    volatile int *data;
❶   data = (int *)ALLOCA(10*sizeof(int));
    /* POTENTIAL FLAW: Partially initialize data */
    {
        int i;
❷       for(i=0; i<(10/2); i++)
        {
            data[i] = i;
        }
    }
❸   CWE457_Use_of_Uninitialized_Variable__int_array_alloca_partial_init_64b_badSink(&data);
}
```

Listing C.3: An example code from the Juliet Test Suite, demonstrating the use of partially initialized arrays. The function `CWE457_Use_of_Uninitialized_Variable__int_array_alloca_partial_init_64_bad` allocates memory for an integer array using `ALLOCA` ❶ and partially initializes it ❷. The partially initialized array is then passed to `CWE457_Use_of_Uninitialized_Variable__int_array_alloca_partial_init_64b_badSink` ❸, and that function is reading the whole extent of the `data` array. That uninitialized memory access detected by Mon CHÉRI.

```
define dso_local void @CWE457_Use_of_Uninitialized_Variable__int_array_alloca_partial_init_64_bad()
      local_unnamed_addr addrspace(200) #0 !dbg !19 {
entry:
 ① %data = alloca ptr addrspace(200), align 16, addrspace(200), !clang.decl.ptr !28
    %0 = call ptr addrspace(200) @llvm.cheri.bounded.stack.cap.i64(ptr addrspace(200) %data, i64 16)
    %1 = call ptr addrspace(200) @llvm.cheri.cap.op.bounds.set.i64(ptr addrspace(200) %0, i64 0)
 ② %2 = alloca [40 x i8], align 16, addrspace(200), !dbg !30
    %3 = call ptr addrspace(200) @llvm.cheri.bounded.stack.cap.i64(ptr addrspace(200) %2, i64 40), !dbg !30
    %4 = call ptr addrspace(200) @llvm.cheri.cap.op.bounds.set.i64(ptr addrspace(200) %3, i64 0), !dbg !30
 ③ store ptr addrspace(200) %4, ptr addrspace(200) %1, align 16, !dbg !32, !tbaa !33
    ...
 ④ store volatile i32 0, ptr addrspace(200) %4, align 16, !dbg !38, !tbaa !42
 ⑤ store volatile ptr addrspace(200) %4, ptr addrspace(200) %1, align 16, !dbg !37
    ...
}
```

Listing C.4: LLVM IR of Listing C.3 after CP the store linearization. ① allocates memory on the stack for a pointer. ② allocates memory for an array of 40 bytes. ③ stores the capability `%4` at the memory location pointed to by `%1`. After ④, the operation bound is updated at the hardware level, so the capability stored in memory becomes invalid. Our instrumentation adds another store to update the capability in memory ⑤.

Bibliography

[1] *PA-RISC 1.1 Architecture and Instruction Set Reference mMnual, Third Edition*, 1994. https://parisc.wiki.kernel.org/images-parisc/6/68/Pa11_acd.pdf.

[2] *Intel IA-64 Architecture Software Developer's Manual Volume 1: IA-64 Application Architecture. Revision 1.1*, 2000. http://refspecs.linux-foundation.org/IA64-softdevman-vol1.pdf.

[3] *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide*, 2007. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[4] *Intel Software Guard Extensions SDK for Linux OS*, 2016. https://01.org/sites/default/files/documentation/intel_sgx_sdk_developer_reference_for_linux_os_pdf.pdf.

[5] *ARMv8-a Architecture Reference Manual, Version e.a*, 2019. https://static.docs.arm.com/ddi0487/ea/DDI0487E_a_armv8_arm.pdf.

[6] *AMD64 Architecture Programmer's Manual Volume 2: System Programming. Revision 3.38*, 2021. https://www.amd.com/system/files/TechDocs/24593.pdf.

[7] *Programming for AIX 7.3: Storage Protect Keys*, 2022. http://web.archive.org/web/20220509005203/https://www.ibm.com/docs/en/aix/7.3?topic=concepts-storage-protect-keys.

[8] Abadi, M., Budiu, M., Erlingsson, Ú., and Ligatti, J. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (Alexandria VA USA, Nov. 2005), CCS '05, Association for Computing Machinery, pp. 340–353.

[9] Abadi, M., Budiu, M., Erlingsson, Ú., and Ligatti, J. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur. 13*, 1 (Nov. 2009), 4:1–4:40.

[10] Aiken, M., Fähndrich, M., Hawblitzel, C., Hunt, G., and Larus, J. Deconstructing process isolation. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness* (New York, NY, USA, 2006), Mspc '06, Association for Computing Machinery, pp. 1–10.

[11] Almatary, H., Dodson, M., Clarke, J., Rugg, P., Gomes, I., Podhradsky, M., Neumann, P. G., Moore, S. W., and Watson, R. N. M. CompartOS: CHERI Compartmentalization for Embedded Systems. arXiv:2206.02852 [cs], June 2022. http://arxiv.org/abs/2206.02852 (accessed 2025-06-07).

[12] Almohri, H. M. J., and Evans, D. Fidelius Charm: Isolating unsafe Rust code. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy* (New York, NY, USA, 2018), Codaspy '18, Association for Computing Machinery, pp. 248–255.

[13] Alzayat, M., Mace, J., Druschel, P., and Garg, D. Groundhog: Efficient Request Isolation in FaaS. In *Proceedings of the Eighteenth European Conference on Computer Systems* (New York, NY, USA, May 2023), EuroSys '23, Association for Computing Machinery, pp. 398–415.

[14] Amar, S., Chen, T., Chisnall, D., Domke, F., Filardo, N. W., Liu, K., Norton, R. M., Tao, Y., Watson, R. N. M., and Xia, H. CHERIoT: Rethinking security for low-cost embedded systems. Technical Report MSR-TR-2023-6, Microsoft, Feb. 2023.

[15] AMD. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, Mar. 2024. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24593.pdf.

[16] Anderson, J. P. Computer Security Technology Planning Study Volume 1 - Executive Summary. Tech. Rep. AD-758 206, James P. Anderson and Co., L. G. Hanscom Field, Bedford, Massachussets, USA, Oct. 1972. https://apps.dtic.mil/sti/citations/AD0758206.

[17] Arm. Armv8.5-A Memory Tagging Extension. Whitepaper, Aug. 2019. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf.

[18] Avgerinos, T., Cha, S. K., Rebert, A., Schwartz, E. J., Woo, M., and Brumley, D. Automatic exploit generation. *Communications of The Acm 57*, 2 (Feb. 2014), 74–84.

[19] Bang, I., Mayondo, M., Moon, H., and Paek, Y. TRust: A compilation framework for in-process isolation to protect safe Rust against untrusted code. In *32nd USENIX Security Symposium (USENIX Security 23)* (Baltimore, MD, Aug. 2023), USENIX Association.

[20] Baxter, S., and Mazakas, C. Safe C++, Sept. 2024. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p3390r0.html (accessed 2024-10-11).

[21] Ben-Yehuda, A. Keeping Secrets in Rust · Issue #2533 · rust-lang/rfcs. GitHub, Aug. 2018. https://github.com/rust-lang/rfcs/issues/2533 (accessed 2025-03-21).

[22] Bendersky, E. Stack frame layout on x86-64. Eli Bendersky's website, Sept. 2011. https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64 (accessed 2024-11-29).

[23] Berger, E. D., and Zorn, B. G. DieHard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, June 2006), PLDI '06, Association for Computing Machinery, pp. 158–168.

[24] Bialek, J. Solving Uninitialized Stack Memory on Windows | MSRC Blog | Microsoft Security Response Center, Mar. 2020. https://msrc.microsoft.com/blog/2020/05/solving-uninitialized-stack-memory-on-windows/ (accessed 2024-06-27).

[25] Brat, G., Navas, J. A., Shi, N., and Venet, A. IKOS: A Framework for Static Analysis Based on Abstract Interpretation. In *Software Engineering and Formal Methods* (Cham, 2014), D. Giannakopoulou and G. Salaün, Eds., Springer International Publishing, pp. 271–277.

[26] Bruening, D., and Zhao, Q. Practical memory checking with Dr. Memory. In *International Symposium on Code Generation and Optimization (CGO 2011)* (Apr. 2011), pp. 213–223.

[27] Bruening, D. L. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, Sept. 2004.

[28] Burow, N., Carr, S. A., Nash, J., Larsen, P., Franz, M., Brunthaler, S., and Payer, M. Control-Flow Integrity: Precision, Security, and Performance. *ACM Comput. Surv. 50*, 1 (Apr. 2017), 16:1–16:33.

[29] Burow, N., Zhang, X., and Payer, M. SoK: Shining light on shadow stacks. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2019), Sp '19, IEEE, pp. 985–999.

[30] Cao, M., Hou, X., Wang, T., Qu, H., Zhou, Y., Bai, X., and Wang, F. Different is Good: Detecting the Use of Uninitialized Variables through Differential Replay. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, Nov. 2019), CCS '19, Association for Computing Machinery, pp. 1883–1897.

[31] Castro, M., Costa, M., Martin, J.-P., Peinado, M., Akritidis, P., Donnelly, A., Barham, P., and Black, R. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), Sosp '09, ACM, pp. 45–58.

[32] Chen, S., Xu, J., and Sezer, E. C. Non-control-data attacks are realistic threats. In *14th USENIX Security Symposium (USENIX Security 05)* (Baltimore, MD, July 2005), USENIX Association.

[33] Chen, Y., Li, J., Xu, G., Zhou, Y., Wang, Z., Wang, C., and Ren, K. SGXLock: Towards efficiently establishing mutual distrust between host application and enclave for SGX. In *31st USENIX Security Symposium (USENIX Security 22)* (Boston, MA, Aug. 2022), USENIX Association.

[34] Chen, Y., Reymondjohnson, S., Sun, Z., and Lu, L. Shreds: Fine-grained execution units with private memory. In *2016 IEEE Symposium on Security and Privacy (SP)* (2016), pp. 56–71.

[35] Cheng, L., Liljestrand, H., Ahmed, M. S., Nyman, T., Jaeger, T., Asokan, N., and Yao, D. Exploitation Techniques and Defenses for Data-Oriented Attacks. In *2019 IEEE Cybersecurity Development (SecDev)* (Sept. 2019), SecDev '19, IEEE, pp. 114–128.

[36] Chisnall, D., Davis, B., Gudka, K., Brazdil, D., Joannou, A., Woodruff, J., Markettos, A. T., Maste, J. E., Norton, R., Son, S., Roe, M., Moore, S. W., Neumann, P. G., Laurie, B., and Watson, R. N. CHERI JNI: Sinking the Java Security Model into the C. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, Apr. 2017), ASPLOS '17, Association for Computing Machinery, pp. 569–583.

[37] Cho, H., Park, J., Kang, J., Wang, R., Shoshitaishvili, Y., Doupé, A., and Ahn, G.-J. Exploiting Uses of Uninitialized Stack Variables in Linux Kernels to Leak Kernel Pointers. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)* (Aug. 2020), WOOT '20, USENIX Association.

[38] Chow, J., Pfaff, B., Garfinkel, T., and Rosenblum, M. Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation. In *Proceedings of the 14th USENIX Security Symposium* (Baltimore, MD, USA, July 2005), USENIX Security '15, USENIX Association.

[39] CISA. Shifting the Balance of Cybersecurity Risk: Principles and Approaches for Secure by Design Software. Whitepaper, Cybersecurity and Infrastructure Security Agency, Oct. 2023. https://www.cisa.gov/sites/default/files/2023-10/SecureByDesign_1025_508c.pdf (accessed 2024-11-26).

[40] Clang 15.0.0git documentation. Control flow integrity, Feb. 2022. https://web.archive.org/web/20230226000332/https://clang.llvm.org/docs/ControlFlowIntegrity.html.

[41] CodeSecure. CodeSonar. CodeSonar Static Application Security Testing Software Tool, May 2023. https://codesecure.com/our-products/codesonar/ (accessed 2024-05-05).

[42] Connor, R. J., McDaniel, T., Smith, J. M., and Schuchard, M. PKU pitfalls: Attacks on PKU-based memory isolation systems. In *29th USENIX Security Symposium (USENIX Security 20)* (Aug. 2020), USENIX Association, pp. 1409–1426.

[43] Conte, M. Two-Level Segregated Fit memory allocator implementation, Apr. 2016. https://github.com/mattconte/tlsf (accessed 2024-07-10).

[44] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (New York, NY, USA, 2010), SoCC '10, Association for Computing Machinery, pp. 143–154.

[45] Coppens, B., De Sutter, B., and Volckaert, S. Multi-variant execution environments. In *The Continuing Arms Race: Code-Reuse Attacks and Defenses*, vol. 18. Association for Computing Machinery and Morgan & Claypool, Mar. 2018, pp. 211–258. https://doi.org/10.1145/3129743.3129752 (accessed 2024-06-30).

[46] Corbet, J. Sigreturn-oriented programming and its mitigation, Feb. 2016. https://web.archive.org/web/20221117234843/https://lwn.net/Articles/676803/.

[47] Costan, V., and Devadas, S. Intel SGX explained, 2016. https://eprint.iacr.org/2016/086.pdf.

[48] Cowan, C., Pu, C., Maier, D., Walpole, J., and Bakke, P. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium* (San Antonio, Texas, USA, Jan. 1998), USENIX Security '98, USENIX Association.

[49] Cox, B., Evans, D., Filipi, A., Rowanhill, J., Hu, W., Davidson, J., Knight, J., Nguyen-Tuong, A., and Hiser, J. N-Variant Systems A Secretless Framework

for Security through Diversity. In *15th USENIX Security Symposium (USENIX Security '06)* (Vancouver, B.C. Canada, 2006), USENIX Security '06, USENIX Association, p. 16.

[50] CrowdStrike. External Technical Root Cause Analysis — Channel File 291. Tech. rep., Aug. 2024. https://www.crowdstrike.com/wp-content/uploads/2024/08/Channel-File-291-Incident-Root-Cause-Analysis-08.06.2024.pdf (accessed 2024-11-29).

[51] CTSRD. CTSRD-CHERI/Flute. Capability Hardware Enhanced RISC Instructions, June 2024. https://github.com/CTSRD-CHERI/Flute (accessed 2024-07-08).

[52] CTSRD. CTSRD-CHERI/llvm-project. Capability Hardware Enhanced RISC Instructions, July 2024. https://github.com/CTSRD-CHERI/llvm-project (accessed 2024-07-03).

[53] CVE-2009-2629, July 2009. https://nvd.nist.gov/vuln/detail/CVE-2009-2629 (accessed 2023-03-02).

[54] CVE-2011-4971, Dec. 2021. https://nvd.nist.gov/vuln/detail/CVE-2011-4971 (accessed 2023-03-02).

[55] CVE-2018-1000810, Aug. 2018. https://nvd.nist.gov/vuln/detail/CVE-2018-1000810 (accessed 2023-03-09).

[56] CVE-2022-3857, Mar. 2023. https://www.cvedetails.com/cve/CVE-2022-3857/ (accessed 2023-08-30).

[57] Daley, R. C., and Dennis, J. B. Virtual memory, processes, and sharing in MULTICS. *Communications of The Acm 11*, 5 (May 1968), 306–312.

[58] de Clercq, R., and Verbauwhede, I. A survey of hardware-based control flow integrity (CFI). <code>arXiv:1706.07257 [cs.CR]</code>, 2017.

[59] Delshadtehrani, L., Canakci, S., Egele, M., and Joshi, A. SealPK: Sealable protection keys for RISC-v. In *2021 Design, Automation Test in Europe Conference Exhibition (DATE)* (2021), pp. 1278–1281.

[60] Denis-Courmont, R., Liljestrand, H., Chinea, C., and Ekberg, J.-E. Camouflage: Hardware-assisted CFI for the ARM Linux kernel. In *2020 57th ACM/IEEE Design Automation Conference (DAC)* (July 2020), pp. 1–6.

[61] Dennis, J. B., and Van Horn, E. C. Programming semantics for multiprogrammed computations. *Communications of the ACM 9*, 3 (Mar. 1966), 143–155.

[62] Depuydt, H., Gülmez, M., Nyman, T., and Mühlberg, J. T. Do We Still Need Canaries in the Coal Mine? Measuring Shadow Stack Effectiveness in Countering Stack Smashing. In *Availability, Reliability and Security* (2025), Springer, Cham, pp. 193–205.

[63] Designer, S. Bugtraq: Getting around non-executable stack (and fix). https://seclists.org/bugtraq/1997/Aug/63 (accessed 2025-02-13).

[64] EEMBC. Eembc/coremark. Embedded Microprocessor Benchmark Consortium, July 2024. https://github.com/eembc/coremark (accessed 2024-07-03).

[65] ElAtali, H., Gülmez, M., Nyman, T., and Asokan, N. BLACKOUT: Data-Oblivious Computation with Blinded Capabilities. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25)* (Taipei, Taiwan, Oct. 2025), ACM CCS'25.

[66] ElAtali, H., Gunn, L. J., Liljestrand, H., and Asokan, N. BliMe: Verifiably Secure Outsourced Computation with Hardware-Enforced Taint Tracking. In *Proceedings 2024 Network and Distributed System Security Symposium* (San Diego, CA, USA, 2024), Internet Society.

[67] Elnozahy, E. N. M., Alvisi, L., Wang, Y.-M., and Johnson, D. B. A survey of rollback-recovery protocols in message-passing systems. *Acm Computing Surveys 34*, 3 (Sept. 2002), 375–408.

[68] Erlingsson, Ú., Abadi, M., Vrable, M., Budiu, M., and Necula, G. C. XFI: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), Osdi '06, USENIX Association, pp. 75–88.

[69] Erlingsson, Ú., Younan, Y., and Piessens, F. Low-Level Software Security by Example. In *Handbook of Information and Communication Security*, P. Stavroulakis and M. Stamp, Eds. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 633–658. http://link.springer.com/10.1007/978-3-642-04117-4_30 (accessed 2024-11-26).

[70] Evans, D. Splint, Jan. 2002. https://splint.org/ (accessed 2024-05-05).

[71] Facebook. Infer Static Analyzer, May 2016. https://fbinfer.com/ (accessed 2024-05-05).

[72] Filardo, N. W., Gutstein, B. F., Woodruff, J., Clarke, J., Rugg, P., Davis, B., Johnston, M., Norton, R., Chisnall, D., Moore, S. W., Neumann, P. G., and Watson, R. N. M. 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* (La Jolla CA USA, Apr. 2024), ACM, pp. 251–268.

[73] Frassetto, T., Jauernig, P., Liebchen, C., and Sadeghi, A.-R. IMIX: In-Process memory isolation EXtension. In *27th USENIX Security Symposium (USENIX Security 18)* (Baltimore, MD, Aug. 2018), USENIX Association, pp. 83–97.

[74] Fuchs, F. A., Woodruff, J., Moore, S. W., Neumann, P. G., and Watson, R. N. M. Developing a Test Suite for Transient-Execution Attacks on RISC-V and CHERI-RISC-V. In *Fifth Workshop on Computer Architecture Research with RISC-V* (2021), CARRV '21, p. 7.

[75] GCC developer community. *Using the GNU Compiler Collection For GCC Version 14.1.0*, May 2014. https://gcc.gnu.org/onlinedocs/gcc-14.1.0/gcc.pdf (accessed 2024-07-10).

[76] geckodev, m. How much Rust in Firefox. https://4e6.github.io/firefox-lang-stats/ (accessed 2025-09-17).

[77] Georges, A. L., Guéneau, A., Van Strydonck, T., Timany, A., Trieu, A., Huyghebaert, S., Devriese, D., and Birkedal, L. Efficient and provable local capability revocation using uninitialized capabilities. *Proceedings of the ACM on Programming Languages 5*, POPL (Jan. 2021), 6:1–6:30.

[78] Ghiti, A. Virtual Memory Layout on RISC-V Linux. The Linux Kernel documentation, Feb. 2021. https://www.kernel.org/doc/html/latest/arch/riscv/vm-layout.html (accessed 2024-07-03).

[79] Google. Snappy. https://github.com/google/snappy.

[80] Google Project Zero. 0day "In the Wild" dataset, June 2022. https://googleprojectzero.blogspot.com/p/0day.html.

[81] Grisenthwaite, R. Arm Morello Evaluation Platform -Validating CHERI-based Security in a High-performance System. In *2022 IEEE Hot Chips 34 Symposium (HCS)* (Aug. 2022), pp. 1–22.

[82] Guelton, S. Trivial Auto Var Init Experiments. Pythran stories, Dec. 2023. https://serge-sans-paille.github.io/pythran-stories/trivial-auto-var-init-experiments.html (accessed 2024-06-30).

[83] Gülmez, M. Secure Rewind and Discard, 2023. https://secure-rewind-and-discard.github.io/ (accessed 2025-06-07).

[84] Gülmez, M., Englund, H., Mühlberg, J. T., and Nyman, T. Mon CHÉRI: Mitigating Uninitialized Memory Access with Conditional Capabilities. In *2025 IEEE Symposium on Security and Privacy (SP)* (Apr. 2025), IEEE Computer Society, pp. 829–847.

[85] Gülmez, M., Nyman, T., Baumann, C., and Müehlberg, J. T. Exploring the Environmental Benefits of In-Process Isolation for Software Resilience. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S)* (June 2023), pp. 203–205.

[86] Gülmez, M., Nyman, T., Baumann, C., and Mühlberg, J. T. Friend or Foe Inside? Exploring In-Process Isolation to Maintain Memory Safety for Unsafe Rust. In *2023 IEEE Secure Development Conference (SecDev)* (Oct. 2023), pp. 54–66.

[87] Gülmez, M., Nyman, T., Baumann, C., and Mühlberg, J. T. Rewind & Discard: Improving software resilience using isolated domains. In *Proceedings of 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (Washington, DC, USA, June 2023), DSN '23, IEEE Computer Society, pp. 402–416.

[88] Gülmez, M., Nyman, T., Baumann, C., and Mühlberg, J. T. Unlimited Lives: Secure In-Process Rollback with Isolated Domains. arXiv:2205.03205 [cs], Apr. 2023. http://arxiv.org/abs/2205.03205 (accessed 2025-06-14).

[89] Hedayati, M., Gravani, S., Johnson, E., Criswell, J., Scott, M. L., Shen, K., and Marty, M. Hodor: Intra-process isolation for high-throughput data plane libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, July 2019), USENIX Association, pp. 489–504.

[90] Herault, T., and Robert, Y. *Fault-Tolerance Techniques for High-Performance Computing*, 1 ed. Springer Publishing Company, Incorporated, 2015. https://doi.org/10.1007/978-3-319-20943-2.

[91] Horn, J. Speculative execution, variant 4: Speculative store bypass. Project Zero Issue Tracker, Feb. 2018. https://project-zero.issues.chromium.org/issues/42450580 (accessed 2025-07-26).

[92] Hu, H., Shinde, S., Adrian, S., Chua, Z. L., Saxena, P., and Liang, Z. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *2016 IEEE Symposium on Security and Privacy (SP)* (May 2016), S&P '16, IEEE, pp. 969–986.

[93] Huyghebaert, S., Van Strydonck, T., Keuchel, S., and Devriese, D. Uninitialized Capabilities. arXiv:2006.01608 [cs], June 2020. http://arxiv.org/abs/2006.01608 (accessed 2023-10-29).

[94] Intel. A Technical Look at Intel's Control-flow Enforcement Technology. Intel, June 2020. https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html (accessed 2023-10-28).

[95] Intel. *Intel Architecture Instruction Set Extensions and Future Features - Programming Reference*, Mar. 2024. https://cdrdv2-public.intel.com/819680/architecture-instruction-set-extensions-programming-reference.pdf.

[96] Jackson, T., Wimmer, C., and Franz, M. Multi-variant program execution for vulnerability detection and analysis. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research* (New York, NY, USA, Apr. 2010), CSIIRW '10, Association for Computing Machinery, pp. 1–4.

[97] Jacobs, A., Gülmez, M., Andries, A., Volckaert, S., and Voulimeneas, A. System call interposition without compromise. In *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2024).

[98] Jin, X., Xiao, X., Jia, S., Gao, W., Zhang, H., Gu, D., Ma, S., Qian, Z., and Li, J. Annotating, tracking, and protecting cryptographic secrets with CryptoMPK. In *2022 2022 IEEE Symposium on Security and Privacy (SP)* (Los Alamitos, CA, USA, May 2022), IEEE Computer Society, pp. 473–488.

[99] Joly, N., ElSherei, S., and Amar, S. Security Analysis of CHERI ISA. Tech. rep., Microsoft Security Response Center, Oct. 2020. https://msrc.microsoft.com/blog/2020/10/security-analysis-of-cheri-isa/.

[100] Karger, P. A., and Schell, R. R. Thirty years later: Lessons from the Multics security evaluation. In *Proceedings of the 18th Annual Computer Security Applications Conference* (USA, 2002), Acsac '02, IEEE Computer Society, p. 119.

[101] Kashyap, S., Min, C., Lee, B., Kim, T., and Emelyanov, P. Instant OS updates via userspace Checkpoint-and-Restart. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (Denver, CO, June 2016), USENIX Association, pp. 605–619.

[102] King, C. I. Torvalds/linux@1a92b2b · GitHub, Apr. 2015. https://github.com/torvalds/linux/commit/1a92b2ba339221a4afee43adf125fcc9a41353f7 (accessed 2024-07-06).

[103] Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., and Yakobowski, B. Frama-C: A software analysis perspective. *Formal Aspects of Computing 27*, 3 (May 2015), 573–609.

[104] Kirth, P., Dickerson, M., Crane, S., Larsen, P., Dabrowski, A., Gens, D., Na, Y., Volckaert, S., and Franz, M. PKRU-Safe: Automatically locking down the heap between safe and unsafe languages. In *Proceedings of the Seventeenth European Conference on Computer Systems* (New York, NY, USA, 2022), EuroSys '22, Association for Computing Machinery, pp. 132–148.

[105] Klabnik, S., and Nichols, C. *The Rust Programming Language*. No Starch Press, USA, May 2018.

[106] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. Spectre Attacks: Exploiting Speculative Execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy* (May 2019), S&P '19, pp. 1–19.

[107] Koloski, D. Rust serialization benchmark. `https://github.com/djkoloski/rust_serialization_benchmark#rust-serialization-benchmark`.

[108] Koning, K., Bos, H., and Giuffrida, C. Secure and Efficient Multi-Variant Execution Using Hardware-Assisted Process Virtualization. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (June 2016), pp. 431–442.

[109] Koning, K., Chen, X., Bos, H., Giuffrida, C., and Athanasopoulos, E. No need to hide: Protecting safe regions on commodity hardware. In *Proceedings of the Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys '17, Association for Computing Machinery, pp. 437–452.

[110] Koruyeh, E. M., Khasawneh, K. N., Song, C., and Abu-Ghazaleh, N. Spectre Returns! Speculation Attacks Using the Return Stack Buffer. In *Proceedings of the 12th USENIX Workshop on Offensive Technologies* (Baltimore, MD, USA, Aug. 2018), WOOT '18, USENIX Association.

[111] Kosaka, M. Inside look at a modern web browser (part 1), Sept. 2018. `https://developer.chrome.com/blog/inside-browser-part1`.

[112] Lado. Ladroid/CppBorrowChecker, June 2024. `https://github.com/ladroid/CppBorrowChecker` (accessed 2024-07-08).

[113] Lamowski, B. *Automatic Sandboxing of Unsafe Software Components in High Level Languages*. PhD thesis, Technische Universität Dresden, 2017.

[114] Lamowski, B., Weinhold, C., Lackorzynski, A., and Härtig, H. Sandcrust: Automatic sandboxing of unsafe components in Rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems* (New York, NY, USA, 2017), PLOS'17, Association for Computing Machinery, pp. 51–57.

[115] Larsen, P., Homescu, A., Brunthaler, S., and Franz, M. SoK: Automated software diversity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2014), Sp '14, IEEE, pp. 276–291.

[116] Lefeuvre, H., Bădoiu, V.-A., Teodorescu, Ş., Olivier, P., Mosnoi, T., Deaconescu, R., Huici, F., and Raiciu, C. FlexOS: Making OS isolation flexible. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (New York, NY, USA, 2021), HotOS '21, Association for Computing Machinery, pp. 79–87.

[117] Lefeuvre, H., Dautenhahn, N., Chisnall, D., and Olivier, P. SoK: Software Compartmentalization. In *2025 IEEE Symposium on Security and Privacy (SP)* (Nov. 2024), IEEE Computer Society, pp. 3107–3126.

[118] Lenharth, A., Adve, V. S., and King, S. T. Recovery domains: An organizing principle for recoverable operating systems. *SIGARCH Comput. Archit. News 37*, 1 (Mar. 2009), 49–60.

[119] Levy, E. Smashing the Stack For Fun and Profit. *.:: Phrack Magazine ::. 7*, 49 (Nov. 1996).

[120] Levy, H. M. *Capability-Based Computer Systems*. Butterworth-Heinemann, USA, 1984. https://homes.cs.washington.edu/~levy/capabook/ (accessed 2023-10-29).

[121] Li, Z., Wang, J., Sun, M., and Lui, J. C. S. Detecting cross-language memory management issues in Rust. In *Computer Security – ESORICS 2022* (Cham, 2022), V. Atluri, R. Di Pietro, C. D. Jensen, and W. Meng, Eds., Springer Nature Switzerland, pp. 680–700.

[122] Liljestrand, H., Nyman, T., Gunn, L. J., Ekberg, J.-E., and Asokan, N. PACStack: An authenticated call stack. In *30th USENIX Security Symposium (USENIX Security 21)* (Berkeley, CA, USA, Aug. 2021), USENIX Association, pp. 357–374.

[123] Liljestrand, H., Nyman, T., Wang, K., Perez, C. C., Ekberg, J.-E., and Asokan, N. PAC it up: Towards pointer integrity using ARM pointer authentication. In *28th USENIX Security Symposium* (Berkeley, CA, USA, Aug. 2019), USENIX Security '19, USENIX Association, pp. 177–194.

[124] Lind, J., Priebe, C., Muthukumaran, D., O'Keeffe, D., Aublin, P.-L., Kelbert, F., Reiher, T., Goltzsche, D., Eyers, D., Kapitza, R., Fetzer, C., and Pietzuch, P. Glamdring: Automatic Application Partitioning for Intel {SGX}. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (2017), pp. 285–298.

[125] Linux kernel development community. Memory protection keys. https://www.kernel.org/doc/html/next/core-api/protection-keys.html.

[126] Linux manual page. Ptrace(2). https://man7.org/linux/man-pages/man2/ptrace.2.html (accessed 2025-05-24).

[127] Linux manual page. Seccomp(2). https://man7.org/linux/man-pages/man2/seccomp.2.html (accessed 2025-05-24).

[128] Linux Manual page. Setjmp(3), Aug. 2021. https://man7.org/linux/man-pages/man3/longjmp.3.html#:~:text=The%20setjmp()%20function%20saves,case%2C%20setjmp()%20returns%200.

[129] Linux Manual page. Sigaction(2), Aug. 2021. https://man7.org/linux/man-pages/man2/sigaction.2.html.

[130] Litton, J., Vahldiek-Oberwagner, A., Elnikety, E., Garg, D., Bhattacharjee, B., and Druschel, P. Light-weight contexts: An OS abstraction for safety and performance. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (USA, 2016), OSDI'16, USENIX Association, pp. 49–64.

[131] Liu, P., Zhao, G., and Huang, J. Securing unsafe Rust programs with XRust. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (New York, NY, USA, 2020), Icse '20, Association for Computing Machinery, pp. 234–245.

[132] Liu, S., Tan, G., and Jaeger, T. PtrSplit: Supporting general pointers in automatic program partitioning. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (ACM CCS)* (2017).

[133] Liu, Y., Nassar, R., Leangsuksun, C., Naksinehaboon, N., Paun, M., and Scott, S. L. An optimal checkpoint/restart model for a large scale high performance computing system. In *2008 IEEE International Symposium on Parallel and Distributed Processing* (2008), pp. 1–9.

[134] LLVM team. LLVM: Lib/Transforms/Scalar/Reg2Mem.cpp Source File, Jan. 2019. https://llvm.org/doxygen/Reg2Mem_8cpp_source.html (accessed 2024-07-03).

[135] LLVM team. Clang-Check. Clang documentation, 2024. https://clang.llvm.org/docs/ClangCheck.html (accessed 2024-06-30).

[136] LLVM team. Clang-Tidy. Extra Clang Tools documentation, 2024. https://clang.llvm.org/extra/clang-tidy/ (accessed 2024-05-05).

[137] LLVM team. Diagnostic flags in Clang. Clang 18.1.8 documentation, June 2024. https://releases.llvm.org/18.1.8/tools/clang/docs/DiagnosticsReference.html (accessed 2024-07-10).

[138] Lowry, E. S., and Medlock, C. W. Object code optimization. *Commun. ACM 12*, 1 (Jan. 1969), 13–22.

[139] Lu, K., Song, C., Kim, T., and Lee, W. UniSan: Proactive Kernel Memory Initialization to Eliminate Data Leakages. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, Oct. 2016), CCS '16, Association for Computing Machinery, pp. 920–932.

[140] Lu, K., Song, C., Lee, B., Chung, S. P., Kim, T., and Lee, W. ASLR-guard: Stopping address space leakage for code reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), Ccs '15, Association for Computing Machinery, pp. 280–291.

[141] Lu, K., Walter, M.-T., Pfaff, D., Nuernberger, S., Lee, W., and Backes, M. Unleashing Use-Before-Initialization Vulnerabilities in the Linux Kernel Using Targeted Stack Spraying. In *Proceedings 2017 Network and Distributed System Security Symposium* (San Diego, CA, 2017), Internet Society.

[142] Maas, M., and Zabrocki, A. Working Draft of the RISC-V J Extension Specification, June 2024. `https://github.com/riscv/riscv-j-extension` (accessed 2024-07-08).

[143] Maisuradze, G., and Rossow, C. Ret2spec: Speculative Execution Using Return Stack Buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, Oct. 2018), CCS '18, Association for Computing Machinery, pp. 2109–2122.

[144] Mao, Y., Chen, H., Zhou, D., Wang, X., Zeldovich, N., and Kaashoek, M. F. Software fault isolation with API integrity and multi-principal modules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), Sosp '11, ACM, pp. 115–128.

[145] Marathe, V. J., Seltzer, M., Byan, S., and Harris, T. Persistent memcached: Bringing legacy code to Byte-Addressable persistent memory. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)* (Santa Clara, CA, July 2017), USENIX Association.

[146] Marjamäki, D. Cppcheck, May 2024. `https://www.cppcheck.com/` (accessed 2024-05-05).

[147] Masmano, M., Ripoll, I., Crespo, A., and Real, J. TLSF: A new dynamic memory allocator for real-time systems. In *Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004.* (2004), pp. 79–88.

[148] Melara, M. S., Freedman, M. J., and Bowman, M. EnclaveDom: Privilege separation for large-TCB applications in trusted execution environments, 2019.

[149] Memcached, Mar. 2022. `https://memcached.org/`.

[150] Mergendahl, S., Burow, N., and Okhravi, H. Cross-Language attacks. In *Proceedings 2022 Network and Distributed System Security Symposium. NDSS* (2022), vol. 22, pp. 1–17.

[151] Merve, G. Friend or Foe Inside? Exploring In-Process Isolation to Maintain Memory Safety for Unsafe Rust, 2024. https://archive.fosdem.org/2024/schedule/event/fosdem-2024-2632-friend-or-foe-inside-exploring-in-process-isolation-to-maintain-memory-safety-for-unsafe-rust/.

[152] Milburn, A., Bos, H., and Giuffrida, C. SafeInit: Comprehensive and Practical Mitigation of Uninitialized Read Vulnerabilities. In *Proceedings 2017 Network and Distributed System Security Symposium* (San Diego, CA, 2017), Internet Society.

[153] MITRE. CWE-457: Use of Uninitialized Variable (4.14). Common Weakness Enumeration, July 2006. https://cwe.mitre.org/data/definitions/457.html (accessed 2024-07-09).

[154] Mutlu, O., and Kim, J. S. RowHammer: A retrospective. *Trans. Comp.-Aided Des. Integ. Cir. Sys. 39*, 8 (Aug. 2020), 1555–1571.

[155] Nam, H., Kim, J., Hong, S. J., and Lee, S. Secure checkpointing. *Journal of Systems Architecture 48*, 8 (2003), 237–254.

[156] Nethercote, N., and Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, June 2007), PLDI '07, Association for Computing Machinery, pp. 89–100.

[157] Newsome, T., Waterman, A., Lee, Y., Wachs, M., Dabbelt, P., Xia, R., Zhao, J., Mao, H., and Celio, C. Riscv-software-src/riscv-tests. RISC-V International, July 2024. https://github.com/riscv-software-src/riscv-tests (accessed 2024-07-10).

[158] NGINX, July 2022. https://nginx.org.

[159] Nikhil, R. Bluespec System Verilog: Efficient, correct RTL from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.* (June 2004), pp. 69–70.

[160] Nilsson, H. P. Bug 111523 – Unexpected performance regression with -ftrivial-auto-var-init=zero for e.g. systemctl unmask. GCC Bugzilla, Sept. 2023. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=111523 (accessed 2024-07-07).

[161] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., and Venkataramani, V. Scaling memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (USA, 2013), Nsdi'13, USENIX Association, pp. 385–398.

[162] NIST. CVE-2022-3786. https://nvd.nist.gov/vuln/detail/cve-2022-3786 (accessed 2025-06-14).

[163] NIST. Juliet C/C++ 1.3. NIST Software Assurance Reference Dataset, 2017. https://samate.nist.gov/SARD (accessed 2024-12-16).

[164] Niu, B., and Tan, G. RockJIT: Securing just-in-time compilation using modular control-flow integrity. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2014), Ccs '14, ACM, pp. 1317–1328.

[165] Niu, B., and Tan, G. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), Ccs '15, ACM, pp. 914–926.

[166] NSA. Software Memory Safety. Tech. rep., Oct. 2022. https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF (accessed 2025-09-16).

[167] ONCD. Back to the Building Blocks: A Path Toward Secure and Measurable Software. Whitepaper, United States White House Office of the National Cyber Director, 2024. https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf (accessed 2024-11-26).

[168] OpenSSF contributors. Compiler Options Hardening Guide for C and C++. OpenSSF Best Practices Working Group, June 2024. https://best.openssf.org/Compiler-Hardening-Guides/Compiler-Options-Hardening-Guide-for-C-and-C++ (accessed 2024-06-30).

[169] Österlund, S., Koning, K., Olivier, P., Barbalace, A., Bos, H., and Giuffrida, C. kMVX: Detecting Kernel Information Leaks with Multi-variant Execution. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, Apr. 2019), ASPLOS '19, Association for Computing Machinery, pp. 559–572.

[170] Page, B. A Report on the Internet Worm. https://www.ee.torontomu.ca/~elf/hack/iworm.html (accessed 2025-03-09).

[171] Palit, T., Moon, J. F., Monrose, F., and Polychronakis, M. Dynpta: Combining static and dynamic analysis for practical selective data protection. In *2021 IEEE Symposium on Security and Privacy (SP)* (2021), IEEE, pp. 1919–1937.

[172] Park, S., Lee, S., Xu, W., Moon, H., and Kim, T. Libmpk: Software abstraction for Intel Memory Protection Keys (Intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, USA, July 2019), USENIX Association, pp. 241–254.

[173] Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. P., and Saraiva, J. Energy efficiency across programming languages: How do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering* (New York, NY, USA, 2017), Sle 2017, Association for Computing Machinery, pp. 256–267.

[174] Podhradsky, M., Tadros, R., and Roach, A. GaloisInc/BESSPIN-GFE. Galois, Inc., Oct. 2022. `https://github.com/GaloisInc/BESSPIN-GFE` (accessed 2024-07-03).

[175] Popov, A. How STACKLEAK improves Linux kernel security, Nov. 2018. `https://a13xp0p0v.github.io/2018/11/04/stackleak.html` (accessed 2024-07-05).

[176] Qualcomm. Pointer Authentication on ARMv8.3: Design and Analysis of the New Software Security Instructions. Whitepaper, Jan. 2017. `https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf`.

[177] Randell, B. System structure for software fault tolerance. *SIGPLAN Not. 10*, 6 (Apr. 1975), 437–449.

[178] Reis, C., Moshchuk, A., and Oskov, N. Site isolation: Process separation for web sites within the browser. In *28th USENIX Security Symposium (USENIX Security 19)* (Santa Clara, CA, Aug. 2019), USENIX Association, pp. 1661–1678.

[179] Rice, H. G. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society 74*, 2 (1953), 358–366.

[180] Rivera, E., Mergendahl, S., Shrobe, H., Okhravi, H., and Burow, N. Keeping safe rust safe with Galeed. In *Annual Computer Security Applications Conference* (New York, NY, USA, 2021), Acsac '21, Association for Computing Machinery, pp. 824–836.

[181] Rivera, E. E. Preserving memory safety in safe Rust during interactions with unsafe languages. Master's thesis, Massachusetts Institute of Technology / Department of Electrical Engineering and Computer Science, 2016.

[182] Roemer, R., Buchanan, E., Shacham, H., and Savage, S. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Trans. Inf. Syst. Secur. 15*, 1 (Mar. 2012), 2:1–2:34.

[183] Roessler, N., Atayde, L., Palmer, I., McKee, D., Pandey, J., Kemerlis, V. P., Payer, M., Bates, A., Smith, J. M., DeHon, A., and Dautenhahn, N. μSCOPE: A Methodology for Analyzing Least-Privilege Compartmentalization in Large Software Artifacts. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses* (New York, NY, USA, Oct. 2021), RAID '21, Association for Computing Machinery, pp. 296–311.

[184] Ruchlejmer, S. Secure Rewind and Discard on ARM Morello. arXiv:2407.04757 [cs], July 2024. `http://arxiv.org/abs/2407.04757` (accessed 2024-07-10).

[185] Rust Team. GitHub rust-lang/rust pull request #84197: Add codegen option for using LLVM stack smash protection. `https://github.com/rust-lang/rust/pull/84197`.

[186] Rust Team. The rustonomicon —- the dark arts of advanced and unsafe rust programming. `https://doc.rust-lang.org/nomicon/`.

[187] Salamat, B., Jackson, T., Gal, A., and Franz, M. Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the 4th ACM European Conference on Computer Systems* (New York, NY, USA, Apr. 2009), EuroSys '09, Association for Computing Machinery, pp. 33–46.

[188] Saltzer, J., and Schroeder, M. The protection of information in computer systems. *Proceedings of the IEEE 63*, 9 (1975), 1278–1308.

[189] Schrammel, D., Weiser, S., Sadek, R., and Mangard, S. Jenny: Securing syscalls for PKU-based memory isolation systems. In *31st USENIX Security Symposium (USENIX Security 22)* (Boston, MA, Aug. 2022), USENIX Association.

[190] Schrammel, D., Weiser, S., Steinegger, S., Schwarzl, M., Schwarz, M., Mangard, S., and Gruss, D. Donky: Domain keys – efficient in-process isolation for RISC-V and x86. In *29th USENIX Security Symposium (USENIX Security 20)* (Aug. 2020), USENIX Association, pp. 1677–1694.

[191] Schwartz, E. J., Avgerinos, T., and Brumley, D. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *2010 IEEE Symposium on Security and Privacy* (May 2010), pp. 317–331.

[192] Schwartz, E. J., Avgerinos, T., and Brumley, D. Q: Exploit hardening made easy. In *20th USENIX Security Symposium (USENIX Security 11)* (San Francisco, CA, Aug. 2011), USENIX Association.

[193] Schwarzl, M., Borrello, P., Kogler, A., Varda, K., Schuster, T., Gruss, D., and Schwarz, M. Dynamic process isolation, 2021. https://arxiv.org/abs/2110.04751.

[194] Sehr, D., Muth, R., Biffle, C., Khimenko, V., Pasko, E., Schimpf, K., Yee, B., and Chen, B. Adapting software fault isolation to contemporary CPU architectures. In *19th USENIX Security Symposium (USENIX Security 10)* (Washington, DC, Aug. 2010), USENIX Association.

[195] Serebryany, K. ARM Memory Tagging Extension and how it improves C/C++ memory safety. *The USENIX Magazine 48*, 2 (2019), 12–16.

[196] Serebryany, K., Stepanov, E., Shlyapnikov, A., Tsyrklevich, V., and Vyukov, D. Memory Tagging and how it improves C/C++ memory safety. arXiv:1802.09517 [cs], Feb. 2018. http://arxiv.org/abs/1802.09517 (accessed 2024-06-30).

[197] Seward, J., and Nethercote, N. Using Valgrind to detect undefined value errors with bit-precision. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)* (Anaheim, CA, Apr. 2005), USENIX Association.

[198] Shepherd, C., Markantonakis, K., van Heijningen, N., Aboulkassimi, D., Gaine, C., Heckmann, T., and Naccache, D. Physical fault injection and side-channel attacks on mobile devices: A comprehensive analysis. *Computers & Security 111* (2021), 102471.

[199] Shillaker, S., and Pietzuch, P. FAASM: Lightweight isolation for efficient stateful serverless computing. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference* (USA, 2020), Usenix Atc'20, USENIX Association.

[200] Silva, T., Bispo, J., and Carvalho, T. Foundations for a Rust-Like Borrow Checker for C. In *Proceedings of the 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems* (Copenhagen Denmark, June 2024), ACM, pp. 155–165.

[201] Skorstengaard, L., Devriese, D., and Birkedal, L. Reasoning about a Machine with Local Capabilities: Provably Safe Stack and Return Pointer Management. *ACM Trans. Program. Lang. Syst. 42*, 1 (Dec. 2019), 5:1–5:53.

[202] Spickett, D. Top Byte Ignore For Fun and Memory Savings. Linaro Blog, Feb. 2023. https://www.linaro.org/blog/top-byte-ignore-for-fun-and-memory-savings/ (accessed 2024-07-08).

[203] Stepanov, E., and Serebryany, K. MemorySanitizer: Fast detector of uninitialized memory use in C++. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (USA, Feb. 2015), CGO '15, IEEE Computer Society, pp. 46–55.

[204] Sullivan, N. Answering the Critical Question: Can You Get Private SSL Keys Using Heartbleed? The Bloudflare Blog, Nov. 2014. https://blog.cloudflare.com/answering-the-critical-question-can-you-get-private-ssl-keys-using-heartbleed (accessed 2024-07-05).

[205] Sun. [RFC] optimize cost of inter-process communication - Jiadong Sun, 2025. https://lore.kernel.org/lkml/CAP2HCOmAkRVTci0ObtyW=3v6GFOrt9zCn2NwLUbZ+Di49xkBiw@mail.gmail.com/ (accessed 2025-06-01).

[206] Sun, M., and Tan, G. JVM-portable sandboxing of java's native libraries. In *Computer Security – ESORICS 2012* (Berlin, Heidelberg, 2012), Springer Berlin Heidelberg, pp. 842–858.

[207] Sung, M., Olivier, P., Lankes, S., and Ravindran, B. Intra-unikernel isolation with intel memory protection keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2020), Vee '20, Association for Computing Machinery, pp. 143–156.

[208] Sutter, H. Lifetime safety: Preventing common dangling. Technical Report P1179 R1 – version 1.1, Microsoft, Nov. 2019. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1179r1.pdf (accessed 2024-07-08).

[209] Sutter, H. C++ safety, in context. Sutter's Mill, Mar. 2024. https://herbsutter.com/2024/03/11/safety-in-context/ (accessed 2024-03-14).

[210] Swift, M. M., Annamalai, M., Bershad, B. N., and Levy, H. M. Recovering device drivers. *ACM Trans. Comput. Syst. 24*, 4 (Nov. 2006), 333–360.

[211] Swift, M. M., Bershad, B. N., and Levy, H. M. Improving the reliability of commodity operating systems. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), Sosp '03, Association for Computing Machinery, pp. 207–222.

[212] Synopsys. Coverity Scan. Coverity Scan Homepage, Nov. 2023. https://scan.coverity.com/ (accessed 2024-05-05).

[213] Szekeres, L., Payer, M., Wei, T., and Song, D. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy* (May 2013), S&P '13, IEEE, pp. 48–62.

[214] Tan, G. *Principles and Implementation Techniques of Software-Based Fault Isolation*. Now Publishers Inc., Hanover, MA, USA, 2017. https://www.cse.psu.edu/~gxt29/papers/sfi-final.pdf.

[215] Tarkhani, Z., and Madhavapeddy, A. μTiles: Efficient intra-process privilege enforcement of memory regions, 2020. https://arxiv.org/pdf/2004.04846.pdf.

[216] Team, R. The rustonomicon —- foreign function interface. https://doc.rust-lang.org/nomicon/ffi.html.

[217] The Linux Kernel documentation. Syscall User Dispatch. https://docs.kernel.org/admin-guide/syscall-user-dispatch.html (accessed 2025-05-24).

[218] The Rust Standard Library. Function std::panic::catch_unwind. https://doc.rust-lang.org/std/panic/fn.catch_unwind.html.

[219] The Rust Standard Library. Macro std::panic. https://doc.rust-lang.org/std/macro.panic.html.

[220] The Rust Standard Library. Module std::result. https://doc.rust-lang.org/std/result/.

[221] The Rust Standard Library. Trait std::alloc::Allocator. https://doc.rust-lang.org/std/alloc/trait.Allocator.html.

[222] The Rust Standard Library Developers Guide. Using spcialization. https://std-dev-guide.rust-lang.org/code-considerations/using-unstable-lang/specialization.html.

[223] Timely Dataflow. Abomonation. https://github.com/TimelyDataflow/abomonation.

[224] Vahldiek-Oberwagner, A., Elnikety, E., Duarte, N. O., Sammler, M., Druschel, P., and Garg, D. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)* (Santa Clara, CA, Aug. 2019), USENIX Association, pp. 1221–1238.

[225] Van Bulck, J., Oswald, D., Marin, E., Aldoseri, A., Garcia, F. D., and Piessens, F. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (2019), pp. 1741–1758.

[226] van der Veen, V., Andriesse, D., Göktaş, E., Gras, B., Sambuc, L., Slowinska, A., Bos, H., and Giuffrida, C. Practical context-sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), Ccs '15, ACM, pp. 927–940.

[227] Volckaert, S., Coppens, B., Voulimeneas, A., Homescu, A., Larsen, P., De Sutter, B., and Franz, M. Secure and efficient application monitoring and replication. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference* (USA, June 2016), USENIX ATC '16, USENIX Association, pp. 167–179.

[228] von Neumann, J. First Draft of a Report on the EDVAC. Tech. rep., University of Pennsylvania, June 1945. `https://web.mit.edu/sts.035/www/PDFs/edvac.pdf` (accessed 2025-09-16).

[229] Voulimeneas, A., Song, D., Parzefall, F., Na, Y., Larsen, P., Franz, M., and Volckaert, S. Distributed heterogeneous n-variant execution. In *Detection of Intrusions and Malware, and Vulnerability Assessment* (Cham, 2020), C. Maurice, L. Bilge, G. Stringhini, and N. Neves, Eds., Springer International Publishing, pp. 217–237.

[230] Voulimeneas, A., Vinck, J., Mechelinck, R., and Volckaert, S. You shall not (by)pass! Practical, secure, and fast PKU-based sandboxing. In *Proceedings of the Seventeenth European Conference on Computer Systems* (New York, NY, USA, 2022), EuroSys '22, Association for Computing Machinery, pp. 266–282.

[231] Wagle, P., and Cowan, C. StackGuard: Simple stack smash protection for GCC. In *Proceedings of the GCC Developers Summit* (Jan. 2003).

[232] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1993), Sosp '93, ACM, pp. 203–216.

[233] Wang, X., Yeoh, S., Olivier, P., and Ravindran, B. Secure and efficient in-process monitor (and library) protection with Intel MPK. In *Proceedings of the 13th European Workshop on Systems Security* (New York, NY, USA, 2020), EuroSec '20, Association for Computing Machinery, pp. 7–12.

[234] Watson, R. N., Woodruff, J., Neumann, P. G., Moore, S. W., Anderson, J., Chisnall, D., Dave, N., Davis, B., Gudka, K., Laurie, B., Murdoch, S. J., Norton, R., Roe, M., Son, S., and Vadera, M. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *2015 IEEE Symposium on Security and Privacy* (May 2015), pp. 20–37.

[235] Watson, R. N. M., Barnes, G., Clarke, J., Grisenthwaite, R., Sewell, P., Moore, S. W., and Woodruff, J. Arm Morello Programme: Architectural security goals and known limitations. Technical Report UCAM-CL-TR-982, Computer Laboratory, University of Cambridge, 15 JJ Thomson Avenue Cambridge CB3 0FD United Kingdom, July 2023. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-982.pdf (accessed 2023-11-03).

[236] Watson, R. N. M., Moore, S. W., Sewell, P., and Neumann, P. G. An Introduction to CHERI. Technical Report UCAM-CL-TR-941, Computer Laboratory, University of Cambridge, 15 JJ Thomson Avenue Cambridge CB3 0FD United Kingdom, Sept. 2019. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-941.pdf (accessed 2023-11-03).

[237] Watson, R. N. M., Neumann, P. G., Woodruff, J., Roe, M., Almatary, H., Anderson, J., Baldwin, J., Chisnall, D., Davis, B., Filardo, N. W., Joannou, A., Laurie, B., Markettos, A. T., Moore, S. W., Murdoch, S. J., Nienhuis, K., Norton, R., Richardson, A., Rugg, P., Sewell, P., Son, S., and Xia, H. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 7). Technical Report UCAM-CL-TR-927, Computer Laboratory, University of Cambridge, 15 JJ Thomson Avenue Cambridge CB3 0FD United Kingdom, June 2019. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-927.pdf.

[238] Watson, R. N. M., Neumann, P. G., Woodruff, J., Roe, M., Anderson, J., Baldwin, J., Chisnall, D., Davis, B., Joannou, A., Laurie, B., Moore, S. W., Murdoch, S. J., Norton, R., Son, S., and Xia, H. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 6). Technical Report UCAM-CL-TR-907, Computer Laboratory, University of Cambridge, 15 JJ Thomson Avenue Cambridge CB3 0FD United Kingdom, Apr. 2017. http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-907.pdf.

[239] Webster, A., Eckenrod, R., and Purtilo, J. Fast and service-preserving recovery from malware infections using CRIU. In *27th USENIX Security Symposium (USENIX Security 18)* (Baltimore, MD, Aug. 2018), USENIX Association, pp. 1199–1211.

[240] Wesley Filardo, N., Gutstein, B. F., Woodruff, J., Ainsworth, S., Paul-Trifu, L., Davis, B., Xia, H., Tomasz Napierala, E., Richardson, A., Baldwin, J., Chisnall, D., Clarke, J., Gudka, K., Joannou, A., Theodore Markettos, A., Mazzinghi, A., Norton, R. M., Roe, M., Sewell, P., Son, S., Jones, T. M., Moore, S. W., Neumann, P. G., and Watson, R. N. M. Cornucopia: Temporal Safety for CHERI Heaps. In *2020 IEEE Symposium on Security and Privacy (SP)* (May 2020), pp. 608–625.

[241] Wheeler, D. A. Flawfinder. Flwfinder Homepage, Jan. 2007. https://dwheeler.com/flawfinder/ (accessed 2024-05-05).

[242] wiki, g. Overview of malloc, May 2019. https://sourceware.org/glibc/wiki/MallocInternals.

[243] Woodruff, J., Joannou, A., Xia, H., Fox, A., Norton, R. M., Chisnall, D., Davis, B., Gudka, K., Filardo, N. W., Markettos, A. T., Roe, M., Neumann, P. G., Watson, R. N. M., and Moore, S. W. CHERI Concentrate: Practical Compressed Capabilities. *IEEE Transactions on Computers 68*, 10 (Oct. 2019), 1455–1469.

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

[245] Xia, H., Woodruff, J., Ainsworth, S., Filardo, N. W., Roe, M., Richardson, A., Rugg, P., Neumann, P. G., Moore, S. W., Watson, R. N. M., and Jones, T. M. CHERIvoke: Characterising Pointer Revocation using CHERI Capabilities for Temporal Memory Safety. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, Oct. 2019), MICRO '52, Association for Computing Machinery, pp. 545–557.

[246] Xiong, W., and Szefer, J. Survey of transient execution attacks and their mitigations. *Acm Computing Surveys 54*, 3 (May 2021).

[247] Yang, F., Im, B., Huang, W., Kaoudis, K., Vahldiek-Oberwagner, A., Tsai, C.-C., and Dautenhahn, N. Endokernel: A Thread Safe Monitor for Lightweight Subprocess Isolation. In *33rd USENIX Security Symposium (USENIX Security 24)* (2024), pp. 145–162.

[248] Yasukata, K., Tazaki, H., Aublin, P.-L., and Ishiguro, K. Zpoline: A system call hook mechanism based on binary rewriting. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)* (2023), pp. 293–300.

[249] Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. Native client: A sandbox for portable, untrusted x86 native code. In *2009 30th IEEE Symposium on Security and Privacy* (2009), pp. 79–93.

[250] Young, J. W. A first order approximation to the optimum checkpoint interval. *Communications of The Acm 17* (1974), 530–531.

[251] Yu, J., Hsiung, L., El'Hajj, M., and Fletcher, C. W. Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing. In

Proceedings 2019 Network and Distributed System Security Symposium (San Diego, CA, 2019), Internet Society.

[252] Yu, J. Z., Watt, C., Badole, A., Carlson, T. E., and Saxena, P. Capstone: A Capability-based Foundation for Trustless Secure Memory Access. In *32nd USENIX Security Symposium (USENIX Security 23)* (2023), pp. 787–804.

[253] Yutaka, Y. AdLint. Adlint Homepage, Aug. 2015. `https://sourceforge.net/projects/adlint/` (accessed 2024-05-05).

[254] Zhang, I., Denniston, T., Baskakov, Y., and Garthwaite, A. Optimizing VM checkpointing for restore performance in VMware ESXi. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)* (San Jose, CA, June 2013), USENIX Association, pp. 1–12.

[255] Zhang, J., Gülmez, M., Nyman, T., and Tan, G. SandCell: Sandboxing Rust Beyond Unsafe Code. arXiv:2509.24032 [cs], Sept. 2025. `http://arxiv.org/abs/2509.24032`.

[256] Zhang, Y., and Swanson, S. A study of application performance with non-volatile main memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)* (2015), pp. 1–10.

[257] Zhao, L., Shuang, H., Xu, S., Huang, W., Cui, R., Bettadpur, P., and Lie, D. A Survey of Hardware Improvements to Secure Program Execution. *ACM Comput. Surv.* (June 2024), 35.

Statement on the use of Generative AI

I did not use generative AI assistance tools during the research/writing process of my thesis, except for mere language assistance.
The text, code, and images in this thesis are my own (unless otherwise specified). Generative AI has only been used in accordance with the KU Leuven guidelines and appropriate references have been added. I have reviewed and edited the content as needed and I take full responsibility for the content of the thesis.

List of Publications

Contributions:

- **M. Gülmez**, T. Nyman, C. Baumann and J. T. Mühlberg, "Rewind & Discard: Improving Software Resilience using Isolated Domains" in 53rd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Porto, Portugal, 2023, pp. 402-416, https://doi.org/10.1109/DSN58367.2023.00046.
- **M. Gülmez**, T. Nyman, C. Baumann and J. T. Mühlberg, "Friend or Foe Inside? Exploring In-Process Isolation to Maintain Memory Safety for Unsafe Rust" in 2023 IEEE Secure Development Conference (SecDev), Atlanta, GA, USA, 2023, pp. 54-66, https://doi.org/10.1109/SecDev56634.2023.00020.
- **M. Gülmez**, H. Englund, J. T. Mühlberg and T. Nyman, "Mon CHÉRI: Mitigating Uninitialized Memory Access with Conditional Capabilities" in 46th IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2025, pp. 829-847, https://doi.org/10.1109/SP61157.2025.00133

Complementary Contributions:

- A. Jacobs, **M. Gülmez**, A. Andries, S. Volckaert and A. Voulimeneas, "System Call Interposition Without Compromise" in 54th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Brisbane, Australia, 2024, pp. 183-194, https://doi.org/10.1109/DSN58291.2024.00030
- J. Zhang, **M. Gülmez**, T. Nyman, and G. Tan. "SandCell: Sandboxing Rust Beyond Unsafe Code" arXiv:2509.24032 [cs.SE] (2025) https://doi.org/10.48550/arXiv.2509.24032
- H. Depuydt, **M. Gülmez**, T. Nyman, J. T. Mühlberg, "Do We Still Need Canaries in the Coal Mine? Measuring Shadow Stack Effectiveness in Countering Stack

Smashing" in Availability, Reliability and Security (ARES 2025) Lecture Notes in Computer Science, vol 15993. Springer, Cham. https://doi.org/10.1007/978-3-032-00627-1_10

- H. ElAtali*, **M. Gülmez***, T. Nyman, and N. Asokan. 2025. "BLACKOUT: Data-Oblivious Computation with Blinded Capabilities" In Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3765169

- **M. Gülmez**, T. Nyman, C. Baumann and J. T. Mühlberg, "Exploring the Environmental Benefits of In-Process Isolation for Software Resilience" in 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S), Porto, Portugal, 2023, pp. 203-205, https://doi.org/10.1109/DSN-S58398.2023.00056.

*Both authors contributed equally to this work.

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
Celestijnenlaan 200A box 2402
B-3001 Leuven