

On the Spatial and Temporal Safety of Multi-Language Applications

by

Samuel Mergendahl

A dissertation accepted and approved in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in Computer Science

Dissertation Committee:

Stephen Fickas, Chair

Boyana Norris, Co-Chair

Thanh Nguyen, Core Member

Michal Young, Core Member

Sara Hodges, Institutional Representative

University of Oregon

Spring 2024

© 2024 Samuel Mergendahl

This work is openly licensed via **CC BY 4.0**.



DISSERTATION ABSTRACT

Samuel Mergendahl

Doctor of Philosophy in Computer Science

Title: On the Spatial and Temporal Safety of Multi-Language Applications

While the introduction of memory-safe programming languages into embedded, Cyber-Physical Systems (CPS) offers an opportunity to eliminate many system vulnerabilities, a pragmatic adoption of memory-safe programming languages often necessitates incremental deployment due to practical development constraints, such as the size of many legacy code bases. This incremental deployment of memory safety leads to a new type of system configuration, called Multi-Language Applications (MLA), where memory-safe and memory-unsafe programming languages are co-resident on the system.

Unfortunately, the spatial and temporal safety of Multi-Language Applications (MLA) remains understudied which contradicts the strict confidentiality, integrity, and availability constraints of embedded, Cyber-Physical Systems (CPS). Therefore, this dissertation investigates the new paradigm of MLA, in which this report enumerates novel spatial and temporal safety violations that can arise in this setting, and proposes a series of defense methodologies to ensure spatial and temporal isolation between potentially compromised components. Namely, because the memory-unsafe languages in an MLA offer an entry point for an attacker, the system must adopt *cyber-resilience* to prevent an attacker from spreading throughout the system and causing a critical system failure.

In particular, this report first introduces a new type of code-reuse attack that specifically appears in Multi-Language Applications (MLA), called Cross-

Language Attacks (CLA). CLA takes advantage of conflicting assumptions between languages to maneuver around deployed defenses. Correspondingly, this report suggests two techniques to prevent CLA. First, a system should provide language-aware memory allocation and second, adopt a newly proposed language construct, called Pseudo-Pointers, to provide spatial isolation between the languages in the MLA. However, even with the temporal safety benefits gained from the thread isolation of Pseudo-Pointers, this report further demonstrates that the system must account for advanced Denial-of-Service (DoS) attacks, called Manipulative Interference Attacks (MIA), in which a compromised component *manipulates* another component into delaying a third, victim component. Additionally, an advanced form of MIA can arise, called Thundering Herd Attacks (THA), that specifically targets kernel mechanisms which exist to ostensibly *enable* temporal isolation as a means to inadvertently delay other high-priority threads in the system; consequently, the required temporal isolation mechanisms themselves act as an attack vector. Finally, in order to overcome this system coordination dilemma, this report proposes an analysis framework to automatically identify instances of MIA in a configured system. Specifically, the analysis uses a hybrid approach that first leverages static analysis to identify software components with influenceable execution times, and second, automatically generates a formal, system-wide model to determine which compromised protection domains can manipulate the influenceable components and trigger Manipulative Interference Attacks (MIA).

This dissertation includes previously published and unpublished co-authored material.

CURRICULUM VITAE

NAME OF AUTHOR: Samuel Mergendahl

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene
University of Wisconsin, Madison

DEGREES AWARDED:

Doctor of Philosophy, Computer Science, 2024, University of Oregon
Master of Science, Computer and Information Science, 2020, University of Oregon
Bachelor of Science, Mathematics and Computer Science, 2015, University of Wisconsin - Madison

AREAS OF SPECIAL INTEREST:

Cybersecurity
Code-reuse Attacks
Denial-of-Service Attacks
Real-time Analysis
Requirements Engineering
Linear Temporal Logic

PROFESSIONAL EXPERIENCE:

Full Technical Staff, Secure Resilient Systems and Technology, MIT Lincoln Laboratory, 2024
Associate Staff, Secure Resilient Systems and Technology, MIT Lincoln Laboratory, 2020-2024
Teaching Assistant, College of Computer and Information Sciences, University of Oregon, 2016-2020
Teaching Assistant, College of Mathematics, University of Wisconsin - Madison, 2013-2016

GRANTS, AWARDS AND HONORS:

Erwin & Gertrude Juilfs Scholarship in Computer and Information Science,
2018

PUBLICATIONS:

- Mergendahl, S., Fickas, S., Norris, B., & Skowrya, R. (2024, May). Manipulative Interference Attacks. *In 2024 ACM Conference on Computer and Communications Security (CCS)*,(In Submission).
- Mergendahl, S., Jero, S., Ward, B. C., Furgala, J., Parmer, G., & Skowrya, R. (2022, May). The thundering herd: Amplifying kernel interference to attack response times. *In 2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*,(pp. 95-107).
- Mergendahl, S., Burow, N., & Okhravi, H. (2022). Cross-Language Attacks. *Network and Distributed System Security (NDSS) Symposium*.
- Rivera, E., Mergendahl, S., Shrobe, H., Okhravi, H., & Burow, N. (2021, December). Keeping safe rust safe with galeed. *In Proceedings of the 37th Annual Computer Security Applications Conference (ACSAC)*,(pp. 824-836).
- Mergendahl, S., & Li, J. (2020, June). Rapid: Robust and adaptive detection of distributed denial-of-service traffic from the internet of things. *In 2020 IEEE conference on communications and network security (CNS)*,(pp. 1-9).
- Sisodia, D., Li, J., Mergendahl, S., & Cam, H. (2024). A Two-Mode, Adaptive Security Framework for Smart Home Security Applications. *ACM Transactions on Internet of Things (TIOT)*,5(2), 1-31.
- Hu, Z., Li, J., Mergendahl, S., & Wilson, C. (2022, April). Toward a resilient key exchange protocol for IoT. *In Proceedings of the Twelfth ACM Conference on Data and Application Security and Privacy (CODASPY)*,(pp. 214-225).
- Shi, L., Mergendahl, S., Sisodia, D., & Li, J. (2020). Bridging Missing Gaps in Evaluating DDoS Research. *In 13th USENIX Workshop on Cyber Security Experimentation and Test (CSET)*.

- Mergendahl, S., Sisodia, D., Li, J., & Cam, H. (2018, July). FR-WARD: Fast retransmit as a wary but ample response to distributed denial-of-service attacks from the Internet of Things. *In 2018 27th International Conference on Computer Communication and Networks (ICCCN)*,(pp. 1-9).
- Sisodia, D., Mergendahl, S., Li, J., & Cam, H. (2018, August). Securing the smart home via a two-mode security framework. *In 14th International Conference of Security and Privacy in Communication Networks (SecureComm)*,(pp. 22-42).
- Mergendahl, S., Sisodia, D., Li, J., & Cam, H. (2017, November). Source-end DDoS defense in IoT environments. *In Proceedings of the 2017 workshop on internet of things security and privacy (IoT-S&P)*,(pp. 63-64).

To my parents

TABLE OF CONTENTS

Chapter	Page
I. INTRODUCTION	19
1.1. Motivation	19
1.2. Dissertation Overview	22
1.2.1. Cross-Language Attacks (CLA)	23
1.2.2. Pseudo-Pointers	24
1.2.3. Manipulative Interference Attacks (MIA)	25
1.2.4. Thundering Herd Attacks (THA)	26
1.2.5. MIA Discovery	27
1.3. Contributions	28
II. BACKGROUND	31
2.1. Code Reuse Attacks	31
2.2. Rust	32
2.3. Go	34
2.4. Component Isolation	35
2.5. Temporal Interference	36
2.6. μ -kernels	38
2.7. seL4	39
2.8. Requirements Engineering	42
III. THREAT MODEL	44
IV. CROSS-LANGUAGE ATTACKS	46
4.1. CLA Model	48
4.1.1. Single-Language Applications (SLA) Threat Models	49

Chapter	Page
4.1.2. Multi-Language Applications (MLA) Threat Models	51
4.1.3. CLA Attack Construction	53
4.2. CLA using Revenant Vulnerabilities	55
4.2.1. Overview	55
4.2.2. Rust Bounds Check Bypass	58
4.2.3. Rust Lifetime Bypass	59
4.2.4. C/C++ Hardening Bypasses	60
4.3. CLA using Multi-Language-Specific Vulnerabilities	61
4.3.1. Corrupting Rust Dynamic Bounds	62
4.3.2. Double Frees	63
4.3.3. Intended Interactions over FFI	64
4.3.4. Concurrency and CLA	66
4.4. Evaluation	66
4.4.1. Methodology	67
4.4.1.1. Source Language	67
4.4.1.2. Metrics	68
4.4.2. Results	69
4.5. Discussion	73
4.5.1. CLA in Go	74
4.5.2. CLA In Other Languages	74
4.5.2.1. Interpreted Languages	75
4.5.2.2. Multiple Safe Languages	75
4.5.2.3. CLA and Verified Code	76
4.5.3. CLA Beyond Memory Safety	76
4.5.4. Defense Strategies for CLA	77

Chapter	Page
4.5.4.1. Preventing Unintended Interactions	77
4.5.4.2. Securing Intended Interactions	77
4.5.4.3. Alternative Defenses	78
V. PSEUDO-POINTERS	79
5.1. Pseudo-Pointers Design	80
5.1.1. Preventing Unintended Interactions via Heap Isolation	82
5.1.1.1. Heap Isolation	83
5.1.1.2. Heap Splitting	83
5.1.1.3. Access Policy	83
5.1.2. Securing Intended Interactions via Pseudo-Pointers	84
5.1.2.1. Pseudo-pointer Properties	86
5.1.2.2. Rust API	87
5.1.2.3. External Function Transformation	88
5.1.3. Pseudo-Pointers Security Guarantees	89
5.2. Pseudo-Pointers Implementation	90
5.2.1. Heap Isolation	90
5.2.1.1. Heap Creation	90
5.2.1.2. Access	90
5.2.2. Pseudo-Pointers	91
5.2.2.1. Rust API	92
5.2.2.2. External Function Transformation	93
5.3. Evaluation	93
5.3.1. Pseudo-pointers	94
5.4. Practical Lessons Learned	95
5.4.1. Mixed-Language Application Security	95

Chapter	Page
5.5. Limitations	95
VI. MANIPULATIVE INTERFERENCE ATTACKS (MIA)	97
6.1. Manipulative Interference	100
6.1.1. Overview of MIA	100
6.1.2. MIA Primitives	102
6.2. Case Studies	103
6.2.1. seL4 Microkit	104
6.2.2. DARPA CASE	105
VII. THUNDERING HERD ATTACKS (THA)	108
7.1. Traditional IPC Interference	109
7.1.1. Overview	109
7.1.2. FIFO Endpoint Flood Interference	110
7.1.3. Priority Ceiling Processing Interference	112
7.1.4. Budget Drain Interference	113
7.1.5. Relationship to the System-Coordination Dilemma	114
7.2. Thundering Herd Attacks	115
7.2.1. Overview	115
7.2.2. Endpoint Queue Sorting Attack	116
7.2.3. Replenishment Queue Sorting Attack	118
7.2.4. Replenishment Wakeup Processing Attack	120
7.3. Evaluation	122
7.3.1. Experimental Setup	123
7.3.2. Traditional IPC Interference Results	123
7.3.3. Thundering Herd Attack Results	124
7.3.4. Red-Black Tree Mitigation Results	126

Chapter	Page
7.3.5. Queue-per-priority Mitigation Analysis	127
7.4. Implications for System Provisioning	128
7.4.1. Experimental Design	128
7.4.2. Results	130
7.5. Discussion and Related Work	130
7.5.1. Implications for μ -Kernel Design	131
7.5.2. Track metadata with $O(\log(n))$ data-structures	131
7.5.3. Track wait queues with queue-per-priority data structures	132
7.5.4. Expanded use of preemption points	132
7.5.5. Partial processing of wakeups in interrupts	133
7.5.6. Other μ -Kernel Designs	133
7.5.7. Fiasco and Nova	134
7.5.8. Thread migration in Composite	134
7.5.9. Static Partitioning Hypervisors	135
7.5.10. Summary	136
VIII.MIA DISCOVERY	137
8.1. Identifying MIA	137
8.1.1. Cycle Detection	139
8.1.2. Requirements Analysis	140
8.1.3. Triage	141
8.2. Evaluation	142
8.2.1. Static Analysis	143
8.2.2. Goal-Conflict Analysis	145
IX. CONCLUSION	147

APPENDICES

Chapter	Page
A. CROSS-LANGUAGE ATTACKS (CLA)	149
B. MANIPULATIVE INTERFERENCE ATTACKS (MIA)	152
REFERENCES CITED	155

LIST OF FIGURES

Figure	Page
1. Cross-Language Attacks (CLA) transfer back and forth between languages to circumvent deployed defenses.	48
2. Language Threat Models	50
3. Our baseline attack variant on a program with Rust (protected by its lifetimes, borrow checker, and dynamic bounds checks) and C (protected by stack canaries and CFI) colored over a graphical description of all Cross-Language Attacks (CLA).	53
4. C/C++ is not subject to the Rust type system, so it can dereference a pointer out-of-bound to a Rust function pointer and make it point to an attacker chosen gadget.	56
5. Since Rust does not deploy a Shadow Stack due to its memory safety, C/C++ can corrupt returns of previously called Rust functions that will never be checked.	57
6. Sample code to illustrate how CLA can circumvent the Rust type system to cause a OOB error.	58
7. Sample code to illustrate how CLA can coerce Rust into causing a UaF error.	59
8. Sample code to show how CLA can corrupt a Rust function pointer to execute a weird machine and circumvent CFI.	60
9. Example of C/C++ using an arbitrary write to corrupt the size of Rust vector.	62
10. Sample code to illustrate how CLA can corrupt even data intended to cross the FFI boundary.	65
11. Stacked bar plots to breakdown different types of statistics found in Table 3. Y-axis is logarithmic.	71
12. The Cumulative Distribution Function (CDF) of each CLA building block metric.	72

Figure	Page
13. Possible memory accesses in Rust-C++ applications	81
14. Protections on memory accesses	82
15. Pseudo-Pointers restricts all accesses by default.	84
16. In our design, C/C++ uses pseudo-pointers (<i>e.g.</i> , <code>id(p)</code>) to request that Rust dereference Rust memory.	86
17. Transforming an example C++ function to use pseudo-pointers.	93
18. Pseudo-Pointer micro-benchmarks	94
19. Manipulative Interference Attacks (MIA) leverage a corrupt, low-criticality component to influence a higher priority component to cause interference on its behalf.	100
20. Mixed Criticality Scheduling used on <code>seL4-MCS</code> for Microkit offers better system utilization, but still cannot prevent MIA.	104
21. Domain Scheduling used on <code>seL4</code> for the DARPA CASE program leads to complex IPC issues and ultimately instances of MIA.	106
22. Legend used throughout attack examples.	110
23. FIFO Endpoint Flood Interference example schedule.	110
24. Priority Ceiling Processing Interference example schedule.	112
25. Budget Drain Interference example schedule.	114
26. Endpoint Queue Sorting Attack example schedule.	117
27. Replenishment Queue Sorting Attack example schedule.	119
28. Replenishment Wakeup Processing Attack example schedule.	121
29. HPI from the traditional IPC interference issues with different numbers of threads and quantities of work for each request.	124
30. HPI from the Thundering Herd attacks with a linked-list kernel data structure.	125
31. HPI from the Thundering Herd attacks with a red-black tree kernel data structure.	126

Figure	Page
32. Sample schedulability graphs. All distributions uniformly distributed. Medium per-task utilizations in $[0.1, 0.4]$	129
33. We breakdown a system into three hierarchical views to a facilitate a practical search for Manipulative Interference Attacks (MIA) in which we limit the overhead of each of our analysis components to only one system view and combine their results.	138
34. Our analysis first identifies cycles that can be manipulated and influenced, second automatically generates a model to verify LTL properties related to MIA, and third triages the identified failures by those that are triggered by components with weak code reuse protection.	139
35. Cumulative Distribution Function (CDF) of our MIA LLVM analysis compared to the baseline LLVM benchmark test-suite.	143
36. Influence analysis leads to large processing times, but triggerable analysis is more performant and can still cause MIA.	144
37. LTL analysis is exponential with respect to the number of protection domains and the number of clients to a IPC server.	145
1. Sample code to illustrate how CLA can circumvent Go to cause a OOB error.	149
2. Sample code to illustrate how CLA can coerce Go into causing a UaF error.	150
3. Sample code to show how CLA can corrupt a Go function pointer to execute a weird machine and circumvent CFI.	150
4. Example of C/C++ using an arbitrary write to corrupt size of a Go slice.	151
5. Sample code to illustrate how CLA can corrupt even data intended to cross the FFI boundary.	151

LIST OF TABLES

Table	Page
1. CLA Variants using Revenant Vulnerabilities	55
2. CLA Variants using Multi-Language-Specific Vulnerabilities	61
3. The prevalence of CLA building blocks. The number is bold is the total number of each item in the binary. (X, Y) represents the breakdown of the counts where X is the fraction of the total items the specific item accounts for, and Y is the fraction of those that come from the specific language. For example, there are 12,118 transfer points in Rust that constitute 3.70% of all call sites in Rust, and 5.32% of the transfer points in the entire binary come from Rust.	69
4. Heavy hitter functions in Firefox.	73

CHAPTER I

INTRODUCTION

To begin, this dissertation outlines the motivation for spatial and temporal safety of embedded, Cyber-Physical Systems (CPS), as well as the need for spatial and temporal isolation of Multi-Language Applications (MLA) in this setting. Moreover, this section provides an overview of the remainder of this dissertation, that describes the spatial and temporal safety issues that can arise specifically due to MLA, as well as the proposed defense methodologies to provide the needed cyber-resilience for MLA on a CPS system. Finally, this section organizes the collective contributions of this dissertation.

1.1 Motivation

While embedded systems traditionally must overcome strict size, weight, and power (SWaP) constraints, more recently, these systems have also begun to instate cybersecurity requirements [156] to protect the confidentiality, integrity, and availability (*i.e.*, the CIA triad [23]) of the system. In particular, some embedded systems, such as Cyber-Physical Systems (CPS), interface with the physical world using hard real-time (HRT) tasks where an unexpected increase in execution latency can cause deadline misses that can have catastrophic physical consequences. Without cybersecurity protection, a compromised component in the system could maliciously extract sensitive sensor data (confidentiality), modify actuation commands (integrity), or trigger execution delay (availability) to cause a catastrophic failure.

Unfortunately, malicious actors can compromise software components within a system. For instance, memory safety vulnerabilities consistently pervade software [57, 171], with which attackers can launch *code-reuse* attacks where the attacker

stitches together gadgets of code found within the instruction set in an unintended execution order to achieve arbitrarily malicious execution from the compromised software component. In fact, malicious actors continue to identify new, advanced code-reuse attacks that can circumvent modern defenses [205, 71, 228]. For example, popular defenses to code-reuse attacks, such as control-flow integrity [14, 207], address space randomization [31, 32, 111, 28, 112], run-time monitoring [160, 54], and stack protection [56, 60, 48, 100, 103], all have been shown to be insufficient against a modern attacker [45, 76, 78, 186, 193, 91, 75]. Moreover, recent data-only attacks can achieve arbitrary execution without modifying the control flow of a program [51, 101, 106]. While memory-safe programming languages have the potential to mitigate the heart of code-reuse attacks (*i.e.*, memory corruption) [16, 63], their practical adoption into large and prevalent legacy code bases requires incremental updates, which has been shown to be an ineffective strategy against code-reuse attacks [143, 158]. Lastly, software supply chain attacks, such as the SolarWinds [62], NetPetya [61], or the recent backdoor found in the widely used compression utility, xz Utils [178], can corrupt the code provided by the software vendor which can also lead to compromised software components. Therefore, proper cybersecurity risk management for an embedded system must assume that a compromised software component may exist and could execute arbitrary malicious behavior in an attempt to trigger a catastrophic failure in the system.

Component isolation is a defensive technique that can potentially deliver the required cyber resilience against compromised software components. With component isolation, each logical software module is placed into its own compartment, often with the intention of maintaining *spatial isolation* between

components. Namely, a compromised component cannot directly read or modify the memory associated with another compartment (which helps protect the confidentiality and integrity of the system). The compartment boundaries may be manually defined [94, 141], or even automatically derived [133, 58, 176, 136]. Moreover, these boundaries can be enforced with process isolation [40, 34, 152], additional software checks [216, 74, 150, 229, 187], special hardware primitives [231, 52, 161, 211, 98, 183, 174, 113, 141, 68, 222, 142], or constructs found in the programming language [230, 126, 37, 153, 88]—each with different trade-offs.

However, embedded systems with hard real-time (HRT) tasks also require that the compartments provide *temporal isolation* in order to guarantee tasks can meet their strict deadline requirements (which helps protect the availability of the system). In particular, the execution context of each component must be separate to prevent priority interference on highly critical components in the system. Typically, thread separation realizes the isolation of execution contexts, but other designs without an operating system have also been proposed [73, 25].

One popular system design to provide both spatial and temporal isolation is a μ -kernel [130]. Unlike a monolithic OS, such as Linux, a μ -kernel reduces overall system privilege by deploying most OS functionality within isolated, userspace protection domains. The majority of the OS instead operates as a “personality” with lower privilege on top of the core kernel [50, 84, 124, 107]. Moreover, a μ -kernel can dictate the allowed memory access of defined protection domains and schedule threads based on priority and execution budget. In particular, the delicate balance between Inter-Process Communication (IPC) efficiency and temporal isolation on μ -kernels has motivated over five decades of research [96, 155, 200, 191, 129, 38, 72]. For example, `seL4` [119] is a capability-based

μ -kernel [67, 194, 87] that has seen extensive adoption in real-world systems [213, 29, 97] due to its unique and comprehensive guarantees of formal verification [118]. Namely, **seL4** has formally proven its functional correctness (*i.e.*, **seL4** correctly implements its specification [190]), its integrity and information flow security (*i.e.*, **seL4** has no spatial side channels [189, 147]), and offers high assurance timing bounds and modern mixed-criticality scheduling mechanisms [188, 137].

Unfortunately, real-time schedulability analysis often only adopts a semi-honest threat model, in which each thread is assumed to execute from only a well-defined set of actions. However, given the threat and prevalence of code-reuse attacks, a malicious threat model must be considered where a thread can execute potentially arbitrary behavior. In fact, recent research demonstrates that low criticality, isolated components can sometimes trigger large execution times elsewhere in the system [164, 127, 87, 144].

1.2 Dissertation Overview

In my dissertation, I plan to analyze the spatial and temporal safety of Multi-Language Applications (MLA) on an embedded, cyber-physical system. In many systems, the incremental development of parts of the application in the safe programming language is performed to ‘enhance’ its security. For example, in Firefox, the Servo CSS style calculation [12], the Dogear bookmark merger [6], the MP4 metadata parser [9], and the neqo QUIC implementation [10] are all implemented in Rust, while many other parts of Firefox are in C and C++, among other languages. However, because the memory-unsafe languages in an MLA offer an entry point for an attacker, the system as a whole must adopt *cyber-resilience* to prevent an attacker from spreading throughout the system and causing a critical

system failure. Therefore, this dissertation will study this new paradigm of MLA, in which it will identify issues with spatial and temporal safety in this setting, and propose frameworks to ensure spatial and temporal isolation between potentially compromised components.

1.2.1 Cross-Language Attacks (CLA). First, in order to study the spatial safety of MLA, I will introduce a new type of code-reuse attack called **Cross-Language Attacks (CLA)**. In this attack, an attacker can leverage an incompatible set of assumptions made by various languages where typical control-flow hijacking is prevented by each language individually. In particular, I will build a model of how various runtime exploit mitigation checks and language safety checks attempt to break different stages of the code-reuse attack chain. With this model, I will demonstrate that these differences create an incompatible set of assumptions on each side such that an attacker can maneuver between the languages in a way that allows the exploit to succeed without violating the safety checks on either side. Namely, the introduction of a safe language creates a conflicting set of assumptions in which CLA results in control-flow hijacks that are otherwise prevented on each language individually. These findings illustrate that incremental deployment of safe languages, if not done with extreme care, can indeed be detrimental to security. Interested readers can find a series of Cross-Language Attacks (CLA) attack examples online¹. Chapter IV covers this topic, and was previously published in *Mergendahl, S., Burow, N., & Okhravi, H. (2022). Cross-Language Attacks. Network and Distributed System Security (NDSS) Symposium.* with myself and the help of Nathan Burow and Hamed Okhravi as authors.

¹<https://github.com/mit-ll/Cross-Language-Attacks/>

1.2.2 Pseudo-Pointers. In order to restore spatial safety in MLA, this dissertation will then describe a defense methodology to mitigate Cross-Language Attacks (CLA). In practice, CLA arises for two reasons. First, when memory-safe and memory-unsafe languages share an address space, with no abstraction or isolation between them at runtime, an arbitrary-write vulnerability in an unsafe language can alter memory that notionally belongs to the safe language. Secondly, strict isolation between a safe and unsafe language is challenging. Because the different programming languages reside in the same application, the two languages must inevitably interact, often through a shared pointer to memory. This shared pointer provides another opportunity for unsafe code to cause CLA. Correspondingly, there are two aspects to the proposed defense methodology to preserve the memory and type safety guarantees of a safe language in Multi-Language Applications (MLA). First, the system must allocate heap memory in a language-aware manner in which an unsafe language will by default, not receive access to the safe language’s heap. Additionally, stack memory should be isolated between languages, potentially achieved with thread separation. This memory isolation by-default prevents *unintended* interactions between the languages, but when interactions exist through a shared pointer, the second aspect of the proposed defense methodology will introduce **Pseudo-Pointers** to secure the *intended* interactions between the languages. Pseudo-Pointers are implemented as an LLVM compile-time sanitizer that replaces raw memory pointers passed across the language boundary with identifiers to memory objects, and turns dereferences of such pointers into Inter-Process Communication (IPC) back to the other language’s thread with the object ID and requested operation. Interested

readers can find an implementation of Pseudo-Pointers online². Chapter V covers this topic, and was previously published in *Rivera, E., Mergendahl, S., Shrobe, H., Okhravi, H., & Burow, N. (2021, December). Keeping safe rust safe with galeed. In Proceedings of the 37th Annual Computer Security Applications Conference (ACSAC), (pp. 824-836).* with myself and the help of Elijah Rivera, Howard Shrobe, Hamed Okhravi, and Nathan Burow as authors. Elijah Rivera was the primary author of this work.

1.2.3 Manipulative Interference Attacks (MIA). Next, in order to study the spatial safety of MLA, I plan to introduce a new type of Denial-of-Service (DoS) attack called **Manipulative Interference Attacks (MIA)**. While thread isolation separates the memory of the two languages, and further, provides the basis for temporal isolation of execution context, intended interactions between languages require Inter-Process Communication (IPC). If not analyzed carefully, IPC can disrupt the temporal isolation of the system—critical to embedded, cyber-physical systems. In particular, while DoS attacks on inter-component messaging are not new [131, 195], MIA defines a novel type of attack in which a compromised component *manipulates* another component into delaying a third, victim component. In particular, an untrusted, malicious component creates unexpectedly large amounts of processing for a trusted, high-priority component in the system. Because the trusted component executes *on behalf of* the compromised component, this higher-priority component may unknowingly delay a co-resident victim task. When the victim task is a hard real-time task, MIA can cause devastating, critical system failures. Chapter VI covers this topic, and is currently unpublished, but submitted in *Mergendahl, S., Fickas, S., Norris, B., & Skowrya,*

²<https://github.com/mit-ll/galeed/>

R. (2024, May). *Manipulative Interference Attacks*. In *2024 ACM Conference on Computer and Communications Security (CCS)*, (In Submission). with myself and the help of Stephen Fickas, Boyana Norris, and Richard Skowrya as authors.

1.2.4 Thundering Herd Attacks (THA). Additionally, I will introduce an advanced form of Manipulative Interference called **Thundering Herd Attacks (THA)**. In particular, these attacks target the synchronous IPC and execution budget *management* mechanisms of an OS kernel (*i.e.*, the attack manipulates specifically the *kernel* to trigger long processing times). In the classical Thundering Herd problem [180], many threads waiting on some event are woken up but only one is actually able to proceed, causing the other threads to consume resources before blocking again. Similarly, in our Thundering Herd Attacks (THA), a large number of malicious application threads methodically use IPC facilities and carefully consume budget in a manner that causes kernel execution commensurate with the number of threads. Namely, the mechanisms added to ostensibly *improve* temporal isolation inadvertently enable this class of attacks. Because most kernels employ non-preemptive execution to control concurrency, when the kernel execution caused by THA runs non-preemptively, long stretches of non-preemptive execution interfere with and delay the activation of high-priority threads. This can further threaten a high-priority thread’s ability to meet deadlines. This introduces what I call the **System Coordination Dilemma**. Either the system deploys *without* kernel-based temporal isolation mechanisms, and suffers from traditional temporal interference, or the system deploys *with* the kernel-based defenses, and potentially suffers from Thundering Herd Attacks (THA). Chapter VII covers this topic, and was previously published in Mergendahl, S., Jero, S., Ward, B. C., Furgala, J., Parmer, G., & Skowrya, R. (2022, May). *The thundering herd: Amplifying kernel*

interference to attack response times. In 2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS), (pp. 95-107). with myself and the help of Samuel Jero, Bryan C. Ward, Juliana Furgala, Gabriel Parmer, and Richard Skowrya as authors.

1.2.5 MIA Discovery. Because the types of interactions that lead to MIA (and subsequently THA) are potentially complex and easily overlooked, I plan to present an analysis framework to automatically identify instances of MIA in a configured system. Specifically, I will propose a hybrid approach that first leverages static analysis to identify software components with influenceable execution times, and second, automatically generates a system-wide model to determine which compromised protection domains can manipulate the influenceable components and trigger Manipulative Interference Attacks (MIA). I implement the static analysis as an LLVM compiler pass and leverage the Labeled Transition Set Analyzer (LTSA)—a formal system model capable of goal-conflict analysis using Linear Temporal Logic (LTL)—to sort through complex system behavior and identify any interactions that lead to MIA. However, because LTL analysis typically requires expensive, technical expertise, I will additionally provide a tool, called **FSPGen**, that *automatically* generates the required, formal system model using widely available system build artifacts. In this way, the hybrid approach can avoid typical pitfalls of static and LTL analysis, and offer a practical compile-time tool for mixed-criticality systems. Interested readers can find MIA attack examples and the analysis tools available online³. Chapter VIII covers this topic, and is currently unpublished, but submitted in *Mergendahl, S., Fickas, S., Norris, B., & Skowrya, R. (2024, May). Manipulative Interference Attacks. In 2024 ACM*

³<https://github.com/smergendahl/manipulative-interference-attacks>

Conference on Computer and Communications Security (CCS), (In Submission).

with myself and the help of Stephen Fickas, Boyana Norris, and Richard Skowyra as authors.

Throughout this dissertation, the new attacks and proposed defense frameworks are instantiated on the `seL4` μ -kernel, but the problem is fundamental to mixed-criticality, multi-language, co-resident execution contexts. Moreover, the presented attack examples will focus on Multi-Language Applications (MLA) that mix Rust with C/C++ for simplicity of exposition, but generalize to Go and other MLA combinations. In particular, Cross-Language Attacks (CLA) and Manipulative Interference Attacks (MIA)—and their corresponding defense frameworks—are extensible to any other system designs that attempt to spatially and temporally isolate co-resident system components.

1.3 Contributions

In particular, my dissertation will make several contributions:

- It studies the spatial and temporal safety of **Multi-Language Applications (MLA)**. In particular, it develops a model to reason about the stages of an exploit broken by each type of security check and illustrates that different languages have incompatible defense assumptions.
- It introduces a new type of code-reuse attack called **Cross-Language Attacks (CLA)** and illustrates that by leveraging conflicting defense assumptions, an attacker can maneuver between languages in a way that allows control-flow hijacking where none is possible in individual languages alone.

- It automatically analyzes Firefox, a large, popular, and open source code base to highlight the prevalence of opportunities for Cross-Language Attacks (CLA). The findings illustrate that incremental deployment of safe languages, if not done with proper care, can indeed be detrimental to security.
- It organizes spacial safety issues that lead to CLA in Multi-Language Applications (MLA) into unintended and intended interactions.
- It proposes a runtime defense methodology for isolating and protecting the Rust heap from unsafe code with a novel sanitizer to verify the security of intended interactions between safe Rust and unsafe code called **Pseudo-Pointers**.
- It benchmarks these defense techniques to evaluate their security and performance impact, finding that Multi-Language Applications (MLA) can achieve spacial isolation between languages with less than 1% overhead.
- It introduces a new type of availability attack called **Manipulative Interference Attacks (MIA)**, that can cause temporal safety violations and deadline misses in spatially isolated, embedded system.
- It leverages the recent DARPA Cyber Assured Systems Engineering (CASE) program as a case study into the feasibility of MIA for real-world systems.
- It introduces a special form of MIA called **Thundering Herd Attacks (THA)** in which an attacker can control non-preemptive kernel execution to cause large temporal safety violations.

- It quantifies the impact of three instances of Thundering Herd Attacks (THA) in `seL4-MCS` that target the improved temporal facilities for properly prioritizing IPC and implementing budgets.
- It identifies two mitigations to Thundering Herd Attacks (THA), implementing and empirically characterizing the performance of one, and qualitatively examining the other.
- It proposes an admission-control test that determines if a system is schedulable, despite the presence of Manipulative Interference Attacks (MIA) or Thundering Herd Attacks (THA).
- It creates an analysis framework to automatically detect instances of MIA at compilation time with a model generation tool called **FSPGen**.
- It evaluates our analysis framework and demonstrate that a risk analysis of MIA is indeed efficient enough for compilation time analysis.

CHAPTER II

BACKGROUND

Memory corruption attacks and research into real-time schedulability have both had extensive amounts of published literature. Therefore, in this section, we provide an introductory background that focuses on topics required to understand the rest of this report. For more information, interested readers should refer to surveys and systematization of knowledge papers in these areas for a more thorough discussion on memory corruption attacks and defenses [205, 197, 44, 199], μ -kernels [55, 72], and mixed-criticality scheduling [41, 38].

2.1 Code Reuse Attacks

Memory corruption attacks have plagued computer systems for decades [66]. Early defense strategies such as Data Execution Prevention (DEP) [21] helped mitigate code injection attacks [157], but attacks evolved into more advanced code reuse techniques such as return-to-libc [209] and return-oriented programming (ROP) [192, 185]. In fact, this “Eternal War” between attack and defense research has continued over the last few decades with modern attackers continuously discovering new ways to circumvent state-of-the-art defenses [205]. For example, Control Flow Integrity (CFI) [14, 207] and stack defenses, such as stack canaries [60], shadow stacks [56], and SafeStack [48, 103], are popular run-time mitigation mechanisms against code reuse attacks, but these defenses have been shown to only provide partial protection at best. These exploit mitigations enforce policies loose enough to allow an attacker to mount a successful attack without violating their policy [45, 76, 78]. In particular, in order for an enforcement policy to mitigate non-control, data-only attacks [51, 101, 106], it would need to be as fine-grained as Data Flow Isolation (DFI) [47] which is often considered too expensive for real-

world deployment. Similarly, randomization techniques [31, 32, 111, 28, 112] are another type of run-time exploit mitigation, but can suffer from various forms of information-leakage attacks that limit their effectiveness [193, 186].

Because memory corruption is at the heart of code reuse attacks, a recent trend is to move away from memory-unsafe programming languages like C/C++ that delegate security checks to the developer, and toward safer programming languages, such as Rust or Go [140, 145]. Similar to sanitizers [199], these languages have special checks to prevent spatial (*e.g.*, buffer overflows) and temporal (*e.g.*, use-after-free) memory corruption bugs that attackers exploit to launch code reuse attacks [227, 117, 27, 110].

2.2 Rust

Rust [140] is a multi-paradigm programming language that provides strong performance and safety properties. Rust has a C-like syntax, but a strong type system combined with compile-time and runtime checks to prevent large classes of bugs, such as memory corruption and concurrency bugs. Rust’s small language runtime makes it appropriate for systems programming, which has resulted in multiple operating systems and low-level code being developed using it [90, 18, 208, 92, 11, 8].

Rust has a strong type system and enforces both spatial and temporal memory safety [116]. For spatial safety, Rust has a two pronged approach. For statically-sized objects, it performs a compile-time size check to avoid out-of-bound accesses. For dynamically-sized objects or for static objects with unknown indices (*e.g.*, an array with a variable index), Rust inserts proper instructions in the binary to perform bounds checking at runtime. Rust also has a strong type system that prevents raw pointers and unsafe casting.

For temporal memory safety, Rust’s solution is more innovative and at the same time restrictive. Rust has a notion of *ownership*. Each value in Rust has a variable that is its owner. Only one owner of a value can exist at a time and when the owner goes out of scope, the value is destroyed. To allow values to be passed between different parts of a code, Rust uses borrowing, which is a temporary transfer of ownership. As a generalization of this principle, Rust only allows one ‘mutable reference’ (*i.e.*, ‘pointer’ in other languages) or multiple immutable references to exist to an object, but not both. The advantage of this design is that when a value is destroyed, Rust can easily nullify any reference to it without the need for heavy-weight garbage collection. This, in addition to other factors, make Rust appropriate for system programming. By the same token, this rule is also too restrictive at times. For example, in a doubly-linked list, there needs to be two mutable references to each value at a time.

Rust’s mechanism for breaking out of these rules is the `unsafe` keyword. Code enclosed in an `unsafe` block (herein simply referred to as ‘unsafe Rust’) can dereference raw pointers and avoid ownership rules. Unsafe Rust is necessary when interacting with low-level devices (*e.g.*, writing to a memory-mapped I/O device). It can also be used to develop data structures that are internally unsafe (*e.g.*, doubly-linked lists), but only expose safe interfaces in what is known as *interior mutability* [140]. Many such data structures are formally shown to be indeed safe [109]. The dangers of unsafe Rust have been acknowledged in the literature [132, 122], and independently investigated [26]. Here, we primarily focus on novel vulnerabilities from mixing *safe* Rust with unsafe languages.

Rust allows interactions with other languages through its foreign function interface (FFI). FFI is inherently unsafe in Rust, and allows the exchange of

arbitrary data, including pointers, across the language boundary. FFI allows the exchange of arbitrary data (including raw pointers) between Rust and other languages, most notably C. Rust has additional rules to make FFI less dangerous; for example, dynamically-sized types cannot be used for FFI. Having said that, the boundary between Rust and a language like C is, by its very nature, unsafe. Indeed, a call through Rust FFI requires the `unsafe` keyword.

We refer to transfers of control flow between languages in a MLA through FFI as *intended* interactions. *Unintended* interactions are also possible as applications within the MLA share an address space. While intended interactions have been the subject of some prior work [152], and similarly, some previous work encourages sandboxing safe language memory from unsafe languages [175], we believe the relationship between intended/unintended interactions and the preservation of language threat models in MLA have to date been under investigated by the community.

2.3 Go

Go [145] is a multi-paradigm programming language that is statically-typed. It provides spatial safety primarily through runtime bounds checks. It also performs various optimizations at compile-time to eliminate unnecessary bounds checks such as redundant checks inside a loop.

In order to provide temporal safety, Go deploys garbage collection (GC). Go does not have any limitation on the number or usage of pointers, which allows the development of complex data structures. On the other hand, the down side of GC is latency and CPU utilization, which can be substantial ($\sim 25\%$ CPU utilization) depending on the code [173]. This also makes Go's language runtime significantly

more complex than that of Rust. By the same token, Go binaries are generally larger than those of Rust.

Similar to Rust, Go can also interact with other languages. For example, CGo [4] allows calling C code from Go. This includes passing Go pointers to C, which can indeed become dangling [179]. Dangerously, Go also hides the inclusion of C code during compilation without warning.

2.4 Component Isolation

Due to the prevalence of memory corruption [57, 171] and the aforementioned “Eternal War” in memory [205], another strategy to limit an attacker is through component isolation. Rather than prevent the attack, this defense philosophy aims to contain the spread of an attack to a well-defined boundary and instead, provide cyber resilience to the attack. In particular, security research often aims to achieve *spatial* isolation in which the attacker cannot read or write to memory associated with another component. Historically, these boundaries are manually defined [94, 141], but more recently, ways to automatically define component boundaries have also been proposed [133, 58, 176, 136].

Additionally, there are multiple strategies to enforce the defined boundaries at run-time each with different trade-offs. For example, a common strategy to provide isolation among cooperating software modules is to place each software component in its own address space and let the operating system catch malicious memory accesses [40, 34, 152]. With address space separation (*i.e.*, process isolation), an operating system can prevent an untrusted module from reading or writing application data associated with a different software component. In particular, both the heap and stack are isolated, and the software requires a context switch to interact with another component. However, when software modules are

closely intertwined (*e.g.*, they share a global state), process isolation can become difficult to apply to legacy code bases. Instead, another popular strategy is to insert additional software run-time checks that verify memory access is correct [216, 74, 150, 229, 187]. This allows software components to remain in the same address space, but the compiler-inserted checks can sometimes incur prohibitive run-time costs.

Recently, special hardware primitives have been proposed to reduce run-time cost, as software can rely on the hardware to perform the needed checks at a lower cost [68]. For example, Intel Memory Protection Keys [105] have been shown to offer an intra-process isolation primitive [161, 211, 98] which is especially helpful to help prevent corruption from spreading in multi-language applications [174, 113]. Similarly, intra-process isolation primitives have been proposed for ARM-32 [231, 52], ARM-64 [141], and RISC-V [183] architectures.

As the lowest cost option, constructs found in the programming language can also act as an isolation primitive [230, 88]. In fact, there have been a number of proposals to pursue this strategy within an operating system [126, 37, 153]. While language-defined boundaries offer strong compile-time guarantees, when a software component itself contains memory corruption, it can still spread at run-time.

2.5 Temporal Interference

Real-time systems require system responsiveness, such that too slow of a response (*i.e.*, a missed deadline) may lead to critical system failure. In order to maintain these strong temporal guarantees for tasks—or the schedulable entities of the system—software separation must also provide *temporal* isolation to limit the impact of malicious software. In particular, a real-time system will often differentiate tasks based on priority where higher-priority tasks should take

precedence over lower-priority tasks. *Priority interference* occurs if a low-priority task executes, while a higher-priority task is ready to execute. Namely, hard real-time (HRT) tasks should be able to meet their real-time requirements even in the presence of high demands placed on other aspects of the system.

Systems must often constrain the execution of low-assurance code to prevent it from unduly interfering with other functionalities. Budget-driven servers¹ are a traditional mechanism for limiting execution interference over time. For example, deferrable servers [200] limit execution over fixed windows of time. A thread’s budget has an initial value, and the budget is depleted corresponding to a thread’s execution. When budget is exhausted, the thread is suspended awaiting replenishment. A *replenishment policy* determines when the budget is increased for a thread. For example, deferrable servers replenish up to the initial budget value periodically.

However, isolated components may request services from other isolated components through synchronous Inter-Process Communication (IPC) [129] which can complicate budget-management policy. If not provisioned carefully, IPC contention between multiple clients may impact the system’s response times [195]. For example, when a server computes on behalf of a client, it can deplete its own budget or inherit budget from the client [202, 137]. The former can lead to budget attacks [131], in which a client attempts to drain a server’s budget to prevent other clients from accessing the service. The latter requires a policy for when the client provides an insufficient budget which can complete the execution of the server’s functionality. Additionally, IPC should process clients in priority order [137], and servers may leverage priority inheritance [202, 201] from clients. If the system does

¹“Servers” here refers to the logic associated with the budget. To disambiguate, we’ll refer explicitly to budgets and budget management.

not adopt priority inheritance, the system designer must carefully assign priorities to a server as the ceiling of each potential client where a server should never block [99]. Such a policy represents the Immediate Priority Ceiling Protocol (I-PCP) [191].

A relatively new type of temporal interference has been shown to occur when a system service unexpectedly requires large amounts of processing on behalf of a client. For example, a malicious component may carefully craft many timers that all expire at the same time, and if not carefully processed, the higher-priority server that processes these timers can delay other tasks [164]. Similarly, previous research identified that even kernel processing of execution budgets [144], system calls from another core [87], or virtualized network traffic [127] can delay tasks, even when the system is deemed schedulable. While these examples of interference are anecdotal, complex, and previously manually identified using system expertise, they indicate a larger issue that may pervade many different real-time systems.

2.6 μ -kernels

μ -kernels move the majority of Operating System (OS) functionality into user-level processes as servers. Unlike a monolithic OS design, a μ -kernel reduces overall system privilege because the majority of OS services no longer need to reside in highly privileged kernel space. Moreover, the OS can deploy as a patina of userspace servers that follows the principle of least privilege (PoLP) for even more security [107].

Typically, a userspace application will deploy on a μ -kernel within process-isolated boundaries above the OS servers. Correspondingly, when software needs the services provided by another server (either an OS server or other isolated application compartments), the software will make a request using IPC. Because

of the context switch overhead of process isolation, μ -kernels have put significant focus IPC optimization. In particular, L4 μ -kernel variants implement IPC as synchronous rendezvous between threads [129, 130, 182, 72]. Synchronous IPC features a control flow that mimics a function call such that the client thread blocks until the server thread returns.

2.7 seL4

One L4 μ -kernel variant that has seen significant adoption is **seL4** [119]. Its popularity stems from its comprehensive formal guarantees of correctness [118]. In particular, **seL4** has formally proven that its implementation correctly derives its specification [190]. In fact, this proof extends to the binary implementation, which means that **seL4** is free of software bugs on multiple different architectures (*i.e.*, ARM-32, ARM-64, x86_64, and RISC-V). Moreover, **seL4** has additional proofs for integrity and information flow [189, 147], as well as high assurance timing bounds [188]. **seL4** has made several design decisions to facilitate such a complete proof. For example, kernel execution is not concurrent nor parallel such that kernel logic always executes a single sequential flow. On multicore systems, this forces the kernel to run within a lock (*i.e.*, the big kernel lock [166]). Because default **seL4** does not provide budget mechanisms to rate-limit component execution, **seL4** offers extensions to the kernel (which we refer to as **seL4-MCS**) that provide mixed-criticality systems additional flexibility for temporal isolation [137]. Namely, IPC servers on **seL4-MCS** can inherit budget (but not priority) and **seL4-MCS** priority sorts IPC queues based on client priority.

seL4 has a number of design decisions that represent trade-offs between IPC efficiency, predictability, and the functional verifiability of the kernel code-base [72]. One important design consideration is that kernel execution is not

concurrent nor parallel (*i.e.*, a single sequential execution flow executes kernel logic at a time). This makes functional verification possible and in practice, means that the kernel executes with interrupts disabled (*i.e.*, it is non-preemptible). On multicore systems, it executes within a lock (*i.e.*, a big kernel lock). This lock is a demonstrated attack vector [87] whereby computation on one core can delay processing on another core.

There is a tension that exists between the mutually exclusive execution of the kernel and kernel operations that execute for a potentially unbounded number of iterations [35] (*e.g.*, revocation of capabilities). **seL4** uses *preemption points* to solve this, whereby kernel execution will back out of a loop after a fixed number of iterations and process any pending interrupts. The thread resumes the loop where it left off. This effectively adds controlled and explicit kernel preemptions.

Clients that use IPC to request service from a server awaiting IPC are queued. The waiting server may be currently executing a request or blocked on another operation. **seL4**'s default policy uses FIFO queueing of these client threads. This FIFO order does *not* represent a priority-sorted order, but it does guarantee client progress. Each client must only wait for a fixed number of threads before it receives service. Servers do *not* inherit client priorities. Therefore the system designer must carefully assign priorities at the ceiling of the clients should predictable service be required. Such a policy represents the Immediate Priority Ceiling Protocol (I-PCP) [191].

Default **seL4** does not provide rate-limiting policies. However, the **seL4** MCS extensions (henceforth referred to as **seL4-MCS**) [137] that are intended to replace the existing **seL4** mechanisms include mechanisms and policies for budget management. Importantly, servers can be *passive*, inheriting the budget (though

not the priority) of client threads upon IPC. These budgets are implemented as sporadic servers. If a budget is depleted while executing a server, a *temporal exception* activates a policy server that can decide to provide enough budget to finish server execution, or to take other remedial action (*e.g.*, extending the budget to include the rest of the server’s execution).

Additionally, `seL4-MCS` uses priority-sorted IPC wait queues. This has an impact on IPC performance as it converts a simple constant-time operation to enqueue a thread into an iterative operation. The priority-sorted wait queues use simple linked lists, resulting in $O(n)$ complexity.

`seL4`’s capability system provides the means to create and control threads. All memory in the system is initially `untyped` and has to be retyped into capabilities for use by the system, including as new threads, known as TCB capabilities. To be usable, a thread will probably need an IPC buffer and stack, which can also be created from untyped memory. Hence, untyped memory is required to create new threads in `seL4`.

Using the previously mentioned TCB capability, a new thread can be started. However, to adjust the priority of this thread, another TCB capability must be used. This second TCB capability must have a maximum priority greater or equal to the desired priority for the new thread. In practice, this means that a thread with access to untyped memory and its own TCB capability can start more threads of equal or lesser priority.

`seL4-MCS` extends the capability system with `SchedContext` capabilities. These capabilities describe and track the budget and period of a thread. In `seL4-MCS`, a `SchedContext` capability must be added to each TCB capability in order for the thread to be runnable. While the `SchedContext` capability can be created

from untyped memory, configuring it requires access to a `SchedControl` capability given to the root-task at boot. In practice, any component of a system that starts new threads must either have a copy of this `SchedControl` capability or be able to issue a request to an admission-control server that can populate the `SchedContext` budget and period.

2.8 Requirements Engineering

Correct system software relies on a well-formed specification or set of goals that the program should achieve. In fact, significant research has demonstrated the benefits of formal, goal-oriented approaches to software development [123, 65, 64]. *Goals* are prescriptive statements of how the system should behave that can guide the refinement of a specification and support the derivation of software operations [19]. However, goals themselves are often initially too ideal and require refinement [20]. For example, the goal may overly assume the benevolence of a system component [64]. In particular, goals need inconsistency management in which the system must identify contradictory low-level requirements [149]. Because inconsistency management is difficult, a weaker form of conflict, called divergence, for goals has been proposed [212]. *Divergence* identifies goals that are not contradictory (*i.e.*, they can be simultaneously satisfied) but can become inconsistent when certain conditions hold.

One way to represent the divergence of goals is to use Linear Temporal Logic (LTL) [168]. LTL is a popular formalism that can state the properties of a reactive system. While a transformational program is a more conventional type of program that produces a final result at the end of a terminating computation, a reactive system maintains an ongoing interaction with its environment. A canonical example of a reactive program is an embedded cyber-physical system. LTL assumes

a lineal topology of time where each instant of the system is followed by a unique future instant. LTL formulas are then evaluated over infinite system traces that represent system execution [65]. In particular, situations that lead to divergent goals can be captured formally as LTL assertions called boundary conditions.

Once a conflict (or divergence) is identified, goal-conflict analysis assesses the likelihood and severity of the inconsistency. Rather than assume the system designer should manually supply these likelihoods, modern approaches will restrict to goal divergence and generate probabilistic information on the problem [64].

CHAPTER III

THREAT MODEL

In this work, we consider an embedded system implemented in multiple programming languages. For example, the system may include both memory-unsafe languages, such as C/C++, and memory-safe languages, such as Rust or Go that interact using shared pointer references. Additionally, memory safety vulnerabilities may exist within the memory-unsafe code regions with which an attacker may attempt to launch a code-reuse attack to execute arbitrarily malicious actions.

Moreover, the system operates with mixed-criticality components in which component boundaries aim to provide both spatial and temporal isolation. In particular, the system operates on a μ -kernel that can separate software components using process (and correspondingly thread) isolation. Namely, each component resides in its own address space without access to memory within another component, but with Inter-Process Communication (IPC) primitives available for components to request service from each other and system calls to interact with the μ -kernel. Moreover, each component receives a unique execution context in which the system enforces assigned execution budgets on each component. However, when desired by a system designer, IPC servers may initially obtain no execution budget, and instead, inherit budget from clients. We also assume that execution budgets are correctly provisioned to each component, such that the system is deemed schedulable when each thread executes from its well-defined set of actions.

As a mixed-criticality system, some of the application components may be highly critical (*e.g.*, a vehicle braking system) and some of the application components may be less critical (*e.g.*, a vehicle infotainment system). In fact, this

is a common way to design a Cyber-Physical System (CPS), as system functionality must be consolidated in order to meet strict Size, Weight, and Power (SWaP) constraints.

While the system we study maintains isolation between each component, we also assume that each isolated component deploys state-of-the-art code reuse mitigations such as an *ideal* Control Flow Integrity (CFI) policy [14] and Stack Canaries [60] or other stack protections. While many embedded systems may adopt a less-than-ideal CFI policy for performance reasons [207], we show that Manipulative Interference Attacks (MIA) are still possible even under an ideal policy in which the control flow graph (CFG) of benign behavior is fully identifiable (and enforceable) statically.

In this setting, we study how a compromised, low-criticality, low-priority component contained in its compartment can cause spatial or temporal safety violations in the system. While the attacker cannot directly modify the highly critical processes, it seeks to influence them indirectly. In particular, the adversary may use memory corruption to send maliciously formed IPC messages to other components. Furthermore, we assume that all the other components in the system are benign and not compromised.

CHAPTER IV

CROSS-LANGUAGE ATTACKS

As seen in *Mergendahl, S., Burow, N., & Okhravi, H. (2022). Cross-Language Attacks. Network and Distributed System Security (NDSS) Symposium.*

A new generation of modern, safe programming languages have been developed that perform security checks natively [116], motivated partly by the limitations of defenses applied to unsafe languages (like C/C++) [46, 77, 106, 102, 205, 89]. Rust [140] and Go [145] are two such languages that prevent the introduction of memory corruption bugs by virtue of having a strong type system and by performing proper compile-time and runtime checks. For example, Rust's type system prevents arbitrary casting, performs compile-time ownership checks to prevent temporal memory safety bugs, and enforces compile-time bounds checks on static data combined with runtime bounds checks on dynamic data to prevent spatial memory corruption bugs [140]. As another example, Go has a garbage collector to provide temporal memory safety [145]. While these languages provide keywords to ignore the safety checks of the language when necessary (*e.g.*, the `unsafe` keyword in Rust is used to interact with low-level hardware devices), within the confines of the safe code, the applications written in these languages are considered generally safe. In fact, these languages have been touted as the 'best chance' to develop safe systems [108] and their gradual deployment is underway in multiple popular applications and code bases. These include, but are not limited to: Firefox [90], Tor [18], Microsoft Windows operating system [208], Google Fuchsia OS [92], and multiple flavors of Linux [11, 8] developed in part in Rust, as well as, Docker [5], Kubernetes [7], CockroachDB [2], and BoltDB [3] developed in part in

Go. This has resulted in the deployment of Multi-Language Applications (MLA), in which two or more languages are used in development.

In this section, we analyze the spatial safety of MLA. Since unsafe languages without additional protections are trivially vulnerable to memory corruption attacks, we specifically focus on the case where some protection is applied to the unsafe side (*e.g.*, CFI for C/C++) and the safe side does not contain `unsafe` code. In these cases, the incremental development of parts of the application in the safe programming language is performed to ‘enhance’ its security. For example, the Servo CSS style calculation in Firefox [12], Dogear (a bookmark merger for Sync in Firefox) [6], the MP4 metadata parser in Firefox [9], and the neqo QUIC implementation in Firefox [10] are all implemented in Rust, while many other parts of Firefox are in C and C++, among other languages. We build a model of how various runtime exploit mitigation checks and language safety checks attempt to break different stages of an exploit. We further illustrate that these checks create an incompatible set of assumptions on each side. Leveraging these incompatibilities in the safety checks performed, we show that an attacker can maneuver between the languages in a way that allows the exploit to succeed without violating the safety checks on either side. In other words, the introduction of a safe language creates a conflicting set of assumptions that indeed weakens the security of both sides. We illustrate that a new vector of attack, Cross-Language Attacks (CLA), becomes possible in such settings which results in control-flow hijacks that are otherwise prevented on each language individually.

We study different variants of CLA with concrete code samples based on the stages of an exploit that are broken by the security checks. Our examples focus on Rust for simplicity of exposition, but generalize to Go and other language

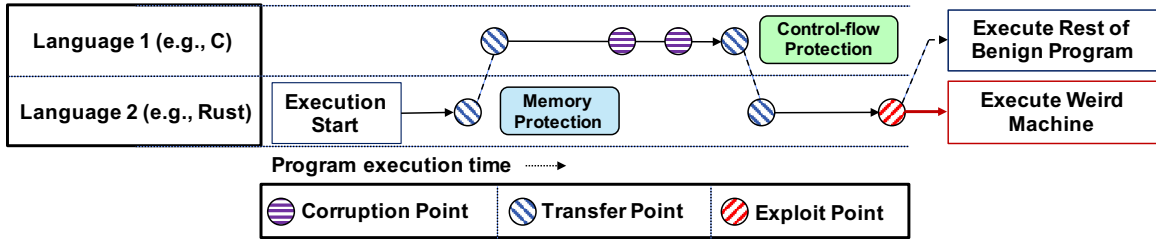


Figure 1. Cross-Language Attacks (CLA) transfer back and forth between languages to circumvent deployed defenses.

combinations (see discussion in §4.5.1 and §4.5.2 and our Go code samples in §A). Moreover, to illustrate the extent of this problem, we perform automated analysis on Firefox, which we believe is representative of large, commonly-used code bases and quantitatively assess the conditions that make CLA possible. Additionally, we make our analysis and concrete code examples available online¹.

Our findings illustrate that, incremental deployment of safe languages, if not done with extreme care, can indeed be detrimental to security. An attacker can leverage the incompatible set of assumptions made by various languages to craft CLA where typical control-flow hijacking is prevented by each language individually.

4.1 CLA Model

We build on existing work [205] presenting high-level threat models for software security, and extend these models to hardened and Multi-Language Applications (MLA). In particular, we show that the threat model for a MLA is the union of the threat models of the constituent languages. In graphical terms, creating a MLA threat model involves adding edges from each node in the threat model to a new “language transfer” node. This can lead to MLA being weaker than their constituent parts due to CLA, a concerning negative synergy.

¹<https://github.com/mit-ll/Cross-Language-Attacks>

At a high-level, a CLA is illustrated in Figure 1. The CLA starts its execution in one language (in this example, Rust). Because of the memory safety checks in the safe language, corruption is not possible, so the CLA proceeds by **transferring** to the unsafe language (in this example, C) for the actual memory corruption. However, because of the protections applied to the unsafe language (in this example, CFI), control-flow hijacking is not possible there, so the CLA transfers back to the safe language to execute the weird machine [196]. The unsafe language assumes that the hardening (*e.g.*, CFI) prevents the hijacking of control and the safe language assumes that the initial corruption is not possible, so it does not check the transfer of control to a weird machine. Consequently, by carefully maneuvering between the languages, the CLA can succeed in a MLA even when it is not possible in individual languages separately. We describe the details of such attack further in the upcoming sections.

In this section, we first discuss the threat models for prevalent programming languages, focusing on compiled languages. We then present a novel graph-based analysis of the threat models that demonstrates that MLA have the pair-wise weaknesses of their constituent languages. The composition of language threat models is illustrated in Figure 2.

4.1.1 Single-Language Applications (SLA) Threat Models.

Figure 2a illustrates the basic chain of events in a memory corruption based software attack, and is modeled off C with only DEP [22], Stack Canaries [215], and ASLR [165] protections (*i.e.*, standard C with no added security). An attacker steers execution towards a memory corruption, which is used to modify the application’s memory layout per an attacker’s specifications (*i.e.*, inject gadgets). These gadgets are then used by the attacker to assume control over the application,

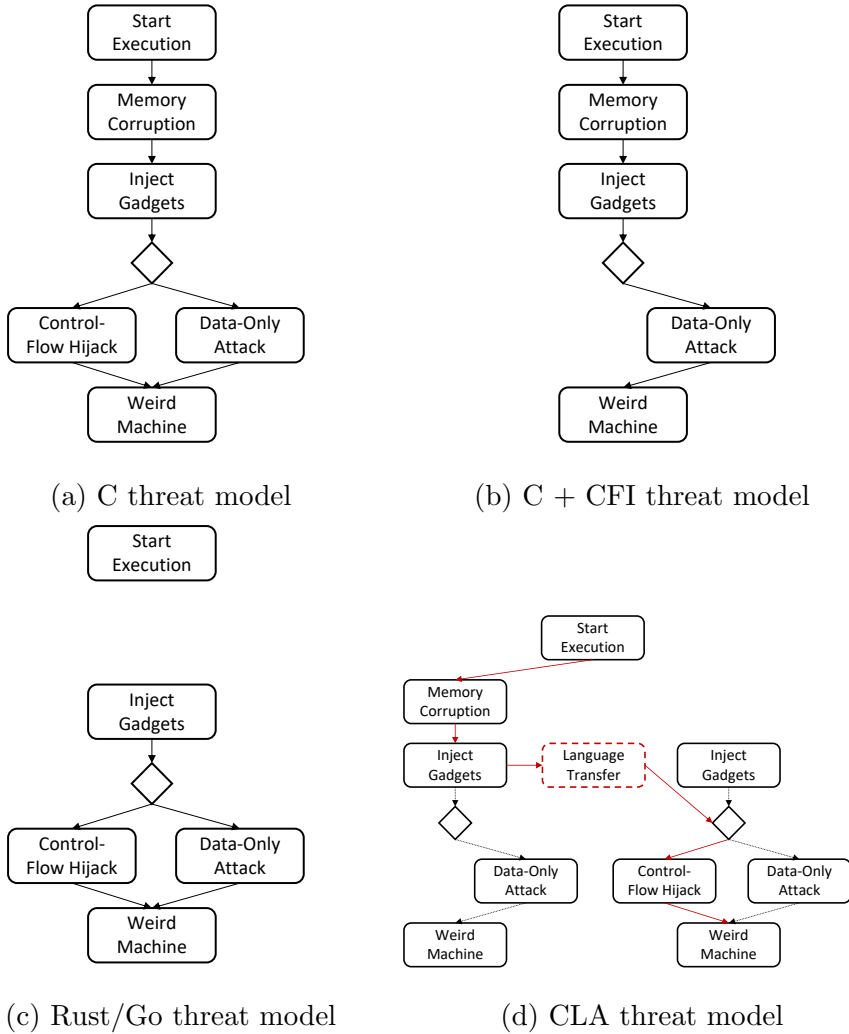


Figure 2. Language Threat Models

either directly by overwriting a code pointer in a control-flow hijack or more subtly and indirectly in DOP [102] attacks. Once the attacker assumes control, they execute the weird machine [196] that their memory corruption set up, and achieve their goals. Attacks thus have four essential phases: i) memory corruption, ii) gadget injection, iii) control-flow assumption, and iv) weird machine execution. To stop an attack, it is sufficient for a defender to disrupt any of these steps, though in practice defenses have focused on steps i and iii [205].

Figure 2b shows the updated threat model for C with (ideal) CFI [42] hardening. Note that the “Control-Flow Hijack” node has been deleted, which is the result of a perfect pointer protection defense (in practice, however, CFI falls short of this standard [46, 77, 79]). Removing this node forces attackers to rely on DOP [102] attacks to execute their weird machines, significantly raising the bar for attackers.

Memory safety, as provided by modern languages such as Rust and Go, offers a strong defense by removing the “memory corruption” node, see Figure 2c. Removing the root cause of an attack removes all of the downstream variants, but experience has shown it must be designed into the language; decades of attempts to retrofit memory safety into C [205, 89, 199] have essentially resulted in only partial protection at best.

4.1.2 Multi-Language Applications (MLA) Threat Models.

Given the SLA threat models in Figure 2a, Figure 2b, and Figure 2c, it is important to correctly compose these underlying threat models for CLA. MLA introduce a new primitive to the threat model: *Language Transfer* nodes. Language transfers occur when an application deliberately interacts with a component in another language (*e.g.*, through FFI).

Conservatively, each node in the constituent language threat models must connect to the Language Transfer node, as there is no way of knowing when language transfers occur in an application. We cannot say, for example, that all language transfers happen before any possible memory corruption. Consequently, the threat models for MLA are fully connected and attacks eliminated by hardening one language may become possible when composing languages.

Figure 2d illustrates how SLA threat models compose for a Rust, Figure 2c, and CFI hardened C, Figure 2b, MLA enabling an attack that is not possible in either component. CFI hardened C applications prevent control-flow hijacking by validating code pointers before they are used. Rust applications prevent the same attack by enforcing memory safety. Note, however, that CFI hardened C is not memory safe, and Rust does validate code pointers before they are used, as it assumes memory safety. Consequently, an attacker can use a memory corruption in C and a non-validated indirect call site in Rust to create a control-flow hijacking attack in a Rust-C MLA.

The fundamental problem here is a mismatch of assumptions in the individual language constituents of a MLA. MLA have the security of their weakest constituent language. While we have illustrated the issue here with a classic control-flow hijacking attack, the problem is much deeper than that. The weakest link principle holds for any element of an application’s threat model that varies across languages. For instance, if Rust were to introduce code signing and validation to mitigate supply-chain attacks and C libraries did not, then a MLA composed of those two languages would remain completely vulnerable to supply chain attacks.

The most insidious case of the MLA threat model composition is when both constituent languages have eliminated a threat, but have done so using different assumptions. The MLA then undermines both sets of assumptions, resulting in the combination of two “safe” languages itself being unsafe. Even for the common scenario of hardening a legacy codebase, such as a C codebase, with a new component written in a safe programming language, such as Rust, this “hardening” can actually end up *weakening* the application’s security.

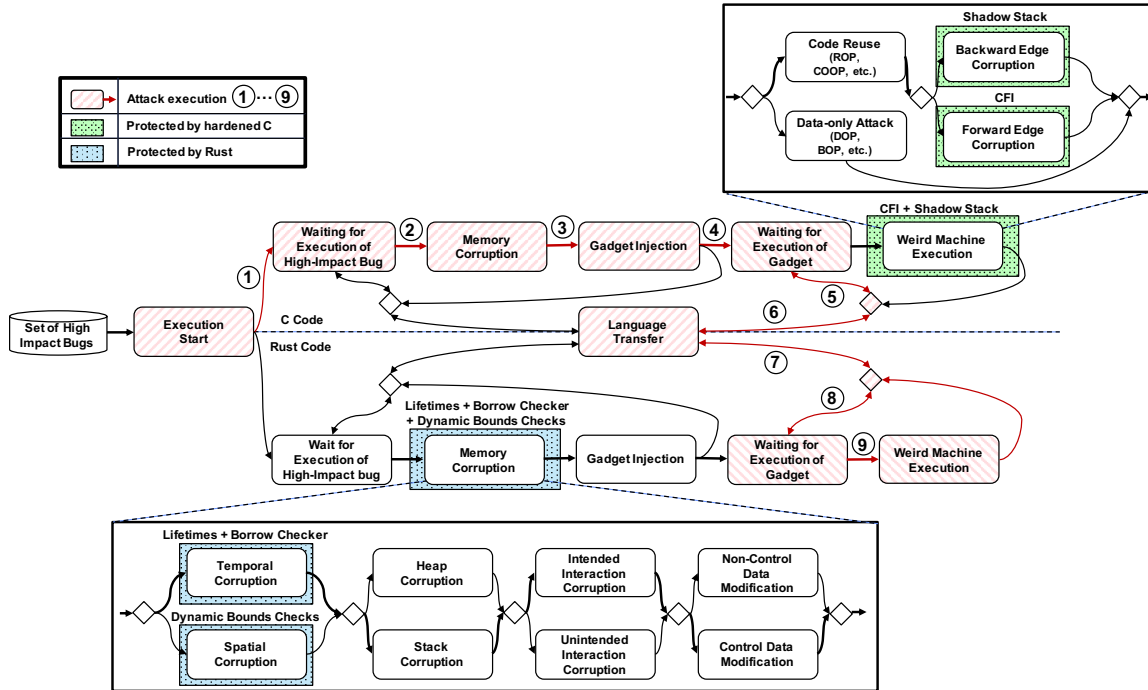


Figure 3. Our baseline attack variant on a program with Rust (protected by its lifetimes, borrow checker, and dynamic bounds checks) and C (protected by stack canaries and CFI) colored over a graphical description of all Cross-Language Attacks (CLA).

4.1.3 CLA Attack Construction.

Given the CLA threat model, we next discuss more concretely the construction of an end-to-end CLA attack. Then, in the next section, we focus on components of such attacks that are unique to the CLA scenario, and introduce new attack primitives. To this end, we next present a more detailed graphical description of CLA in Figure 3. Each node in the graph represents a potential step in the attack, with arrows as indication of possible sequences of steps. A successful attack is a traversal from Execution Start to Weird Machine Execution. Any such traversal that contains the Language Transfer node is a CLA.

As in Figure 2d, we encode the defensive guarantees of Rust and CFI hardened C in Figure 3. Rust’s type system, and in particular its borrow checker,

lifetimes, and dynamic bounds checks, provide memory safety which defends the `Memory Corruption` node (shaded blue). The expansion of the `Memory Corruption` node for Rust shows that the initial steps of memory corruption, the temporal or spatial vulnerabilities, are removed by Rust. Similarly for CFI hardened C, the `Weird Machine Execution` node is defended (shaded green). As the expansion shows, CFI prevents an indirect call / jump from using an arbitrary attacker controlled code-pointer². Combined with a shadow stack [44] to protect returns, hardened C on its own is also largely immune to control hijacking attacks, though data-only attacks [101] remain a threat.

The red nodes in Figure 3 are a concrete instantiation of a CLA attack for illustration, following the concrete attack presented by Papaevripides et al. [159]. The attacker first steers execution towards a known memory safety bug, ①. This leads to the attacker obtaining a “write what where” vulnerability, ②, and using it to inject gadgets, ③. The attacker then steers execution towards a language transition, ④–⑧, and finally, the attacker uses the corrupted code pointer to launch their code reuse attack, ⑨. The only complication here from a classic code reuse attack is the need to find a language transfer point. At the binary level where attacks are constructed, however, this problem is simplified to finding an unprotected indirect call for the attack to target. With such a point and a vulnerability, existing techniques such as Block Oriented Programming [106] can successfully construct attacks.

The simplicity of constructing such attacks at the binary level makes CLA significantly more dangerous. Namely, such attacks can be constructed unknowingly by adversaries looking for classic attack patterns, as opposed to COOP [184] or

²An attacker can still redirect within the set of allowed targets, which has been shown to be sufficient for attacks [46].

Table 1. CLA Variants using Revenant Vulnerabilities

Targeted Language	Bypassed Defense	Memory Corruption Used		Weird Machine Execution Origin	
		Spatial	Temporal	Forward Edge Corruption	Backward Edge Corruption
Rust	Bounds Checks	✓		✓	✓
	Lifetimes		✓	✓	
C++	Shadow Stack	✓			✓
	CFI	✓	✓	✓	

DOP [102] attacks that require new primitives. We next discuss numerous CLA variants, including first, using old vulnerabilities brought back to life by CLA (that we denote as CLA using Revenant Vulnerabilities found in §4.2), and second, using new vulnerabilities that only exist in MLA (that we denote as CLA using Multi-Language-Specific Vulnerabilities found in §4.3).

4.2 CLA using Revenant Vulnerabilities

We present a series of attack variants demonstrating that vulnerabilities typically mitigated by safety check/hardening techniques re-emerge as revenant vulnerabilities using CLA in MLA. We focus our exposition on Rust-C/C++ applications, and show that key defensive primitives either built into or commonly applied to Rust and C/C++ respectively are completely bypassed by CLA. An overview of the attacks we present in this section is contained in Table 1. We show that the spatial and temporal memory safety defenses of Rust are bypassed by CLA, as are Shadow Stacks [44] and CFI [42] for C/C++. We facilitate our discussion with a series of code examples—found in Figure 6,7,8,9, and 10—for exposition. While our examples focus on Rust for simplicity of exposition, we point the reader to §A for an illustration of similar examples in Go. Moreover, we typically use C and C++ interchangeably throughout the following sections.

4.2.1 Overview. A key feature of Rust is memory safety, which rests on two pillars: Rust’s expressive type system and its automatically inserted, dynamic checks. The type system can prove many accesses to be spatially safe

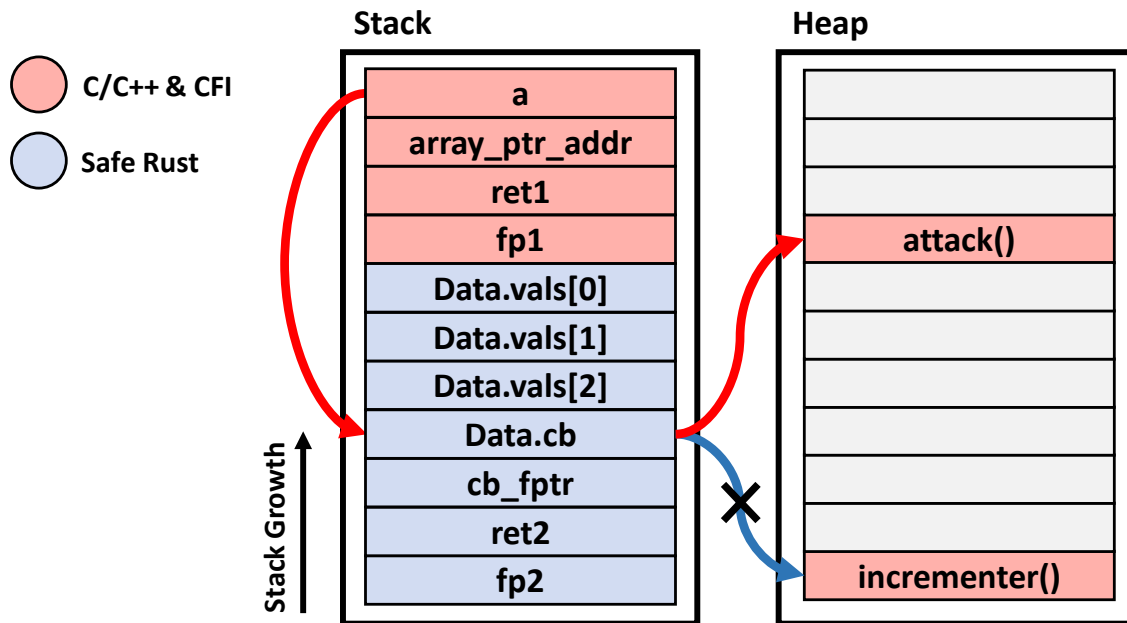


Figure 4. C/C++ is not subject to the Rust type system, so it can dereference a pointer out-of-bound to a Rust function pointer and make it point to an attacker chosen gadget.

at compile time, but some accesses require simple checks at runtime against a constant size bound (*e.g.*, random access into a fixed size array). However, for objects whose size is not known at compile time, such as **vectors**, Rust stores the bounds information in memory, and performs bounds checks against it. All indexes into objects are unsigned, meaning Rust only has to perform upper bounds checks and not lower.³ All of this machinery is lost in MLA, however, as arbitrary write vulnerabilities in C/C++ can effect *any* memory in the shared application’s address space. Such attacks are simplified with a pointer to Rust memory, such as a Rust heap object that contains a function pointer or the Rust stack, but by no means require such a pointer. An example of such an attack is in Figure 4.

³It is interesting to note that these three categories of checks map nicely to the CCured [154] type system that is now 16 years old, highlighting how long and winding the road to practical memory safety has been.

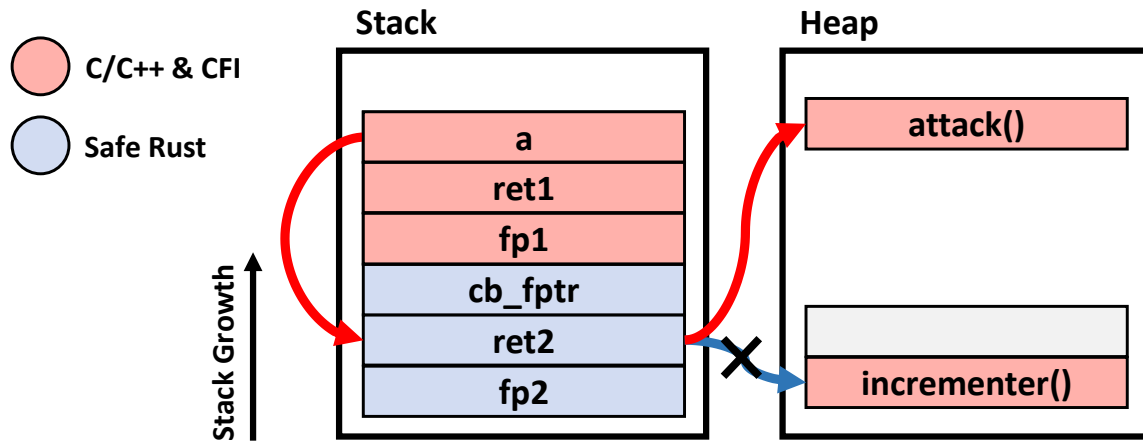


Figure 5. Since Rust does not deploy a Shadow Stack due to its memory safety, C/C++ can corrupt returns of previously called Rust functions that will never be checked.

Rust’s temporal memory safety relies on the ownership model of its type system, which is used for automatic memory management. While programmers can force heap allocations, they are usually oblivious to whether a variable is stack or heap allocated, and deallocation is handled automatically when the variable goes out of scope. However, there is nothing preventing double frees as a result of FFI—as we illustrate below—or preventing programmer error from causing a Use-after-Free (UaF) as a result of FFI. Given that FFI requires unsafe code, responsibility for memory management returns to the programmer, reintroducing such errors.

CFI is entering widespread usage in C/C++ applications, and is designed to provide partial memory safety by protecting the integrity of code pointers. In combination with Shadow Stacks, CFI offers the best combination of strength and performance among runtime defenses to date. As discussed previously, Figure 2, Rust does not use CFI as it provides full memory safety, rendering partial memory safety redundant. Consequently, an arbitrary write vulnerability in C/C++ can

corrupt a code pointer used in Rust, bypassing CFI verification or shadow stack protection. See Figure 5 for an example.

Note that all the examples contained below are simplified for discussion here. One can find the full working versions online⁴.

```
1 fn rust_fn(cb_fptr: fn(&mut i64)) {
2     // Initialize some data
3     let mut x = Data {
4         vals: [1,2,3],
5         cb: cb_fptr,
6     };
7
8     unsafe{ vuln_fn(**Ptr to x.vals**) }
9
10    // Uses corrupted function pointer
11    (x.cb)(&mut x.vals[0]);
12 }
```

(a) Rust code that calls C/C++ to modify a Rust struct.

```
1 // This function modifies a given array
2 // Can cause an OOB vulnerability
3 void vuln_fn(int64_t array_ptr_addr) {
4     // These values are set by a corruptible
5     // source, e.g., user input
6     int64_t array_index = 3;
7     int64_t array_value = get_attack();
8
9     int64_t* a = (void *)array_ptr_addr;
10    a[array_index] = array_value;
11 }
```

(b) C/C++ code that performs an Out-of-Bounds (OOB) error.

Figure 6. Sample code to illustrate how CLA can circumvent the Rust type system to cause a OOB error.

4.2.2 Rust Bounds Check Bypass. We first demonstrate a variant of CLA that can bypass the simple bounds checks that Rust inserts on certain memory accesses. As mentioned previously, for statically-sized objects in memory, such as arrays, Rust will perform bounds checks on associated memory accesses. In Figure 6a, the `Data` struct contains a field `vals` that is a statically sized array. If Rust were to attempt to access the fourth element of `x.vals`, say on line 13, the

⁴<https://github.com/mit-ll/Cross-Language-Attacks>

```

1 fn rust_fn(cb_fptr: fn(&mut i64)) {
2     let heap_obj: /* Rust heap allocation */
3
4     unsafe{ vuln_fn(/*Ptr to heap_obj*/) }
5
6     heap_obj[0] += 5; // UaF
7 }

```

(a) Rust code that uses a pointer wrongfully freed by C/C++.

```

1 // Frees object it does not own
2 void vuln_fn(int64_t obj_ptr_addr) {
3     int64_t* a = (void *)obj_ptr_addr;
4
5     //C/C++ frees Rust allocated object!
6     free(a);
7 }

```

(b) C/C++ code that leads to a Use-after-Free (UaF) error in Rust.

Figure 7. Sample code to illustrate how CLA can coerce Rust into causing a UaF error.

program would either completely fail to compile or panic at runtime depending on the optimizations of the Rust compiler. However, when Rust calls `vuln_fn` on line 8, the unsafe C/C++ function is free to access (and modify) the fourth element of `x.vals`. Because the “fourth” element of `x.vals` is actually the function pointer, `x.cb`, in memory, C/C++ is able modify the Rust function pointer, achieving a control-flow hijack and executing a weird machine when Rust later uses the function pointer at line 11. Therefore, when Rust interfaces with FFI, the typical *spatial* memory safety guarantees of Rust may silently fail.

4.2.3 Rust Lifetime Bypass. We next demonstrate how CLA can bypass Rusts temporal memory safety guarantees in Figure 7. Rather than relying on the programmer to properly allocate and free memory, the Rust type system attaches a lifetime to each object and frees the object when it goes out of scope. By default, Rust uses the `libc malloc()` implementation, meaning that Rust-C/C++ applications in practice share a heap managed by the same allocator.

Thus, C/C++ may deallocate memory without Rust’s knowledge. On line 2 in Figure 7a, Rust allocates a heap object.⁵ When C/C++ frees this object on line 6 in Figure 7b, Rust still believes this object is alive, and valid for use. Consequently, Rust has no problem with the object being used at line 6 of Figure 7a, leading to a UaF vulnerability despite Rust’s temporal memory safety guarantees. CLA can thus cause Rust to silently perform a UaF.

```
1 fn rust_fn(cb_fptr: fn(&mut i64)) {
2     let fptr: /* Function pointer */
3
4     //C++ code overwrites fptr
5     unsafe{ vuln_fn() }
6
7     // No CFI checks!
8     fptr();
9 }
```

(a) Rust code that uses a function pointer.

```
1 void vuln_fn() {
2     int64_t a[1] = {0}; // C/C++ array
3     // These values are set by a corruptible
4     // source, e.g., user input
5     int64_t array_index = 47;
6     int64_t array_value = get_attack();
7
8     // Arbitrary Write to Rust fptr
9     a[array_index] = array_value;
10 }
```

(b) C/C++ that overwrites a Rust function pointer.

Figure 8. Sample code to show how CLA can corrupt a Rust function pointer to execute a weird machine and circumvent CFI.

4.2.4 C/C++ Hardening Bypasses. While we have demonstrated that CLA can bypass the memory safety of Rust, we now show that CLA can also circumvent hardening techniques applied to C/C++ code. In particular, in Figure 8, we illustrate how C/C++ can corrupt a Rust function pointer on the stack which will lead to an opportunity for C/C++ to bypass CFI checks, which

⁵This can happen automatically, or be forced via the Box<> data structure.

Table 2. CLA Variants using Multi-Language-Specific Vulnerabilities

New Attack	Memory Corruption Used		Weird Machine Execution Origin	
	Spatial	Temporal	Forward Edge Corruption	Backward Edge Corruption
Corrupt Dynamic Bound	✓		✓	✓
Double Free		✓	✓	
Intended FFI Interactions	✓	✓	✓	✓
Concurrency Safety	✓	✓	✓	✓

are *not* present in Rust code. On line 9 in Figure 8b, C/C++ performs a typical OOB error and corrupts the `fptr` on the Rust stack to point to an attacker chose location. When Rust uses this function pointer on line 8 of Figure 8a, the attacker has successfully hijacked the application’s control flow to an arbitrary location to execute a weird machine. If the corrupted pointer had instead been used in C/C++, a CFI check would have detected the deviation from the application’s CFG. The memory effects of this bypass are illustrated in more detail in Figure 4, and we can note that Figure 5 demonstrates a similar attack, but one that bypasses the Shadow Stack instead of CFI by overwriting a Rust return value.

While previous work has introduced similar C/C++ hardening bypasses [159], we note that this is only one variant of our presented Cross-Language Attacks (CLA). Most importantly, our work demonstrates that not only can we bypass C/C++ hardening with CLA, but we illustrate how CLA causes Rust memory safety guarantees to be violated. In fact, our goal is to demonstrate that the philosophy of incrementally hardening memory unsafe code with memory safe code can have serious flaws—beyond C/C++ hardening bypasses—if not handled properly.

4.3 CLA using Multi-Language-Specific Vulnerabilities

Beyond reviving the threat of memory safety vulnerabilities in “safe” languages, and bypassing existing partial memory safety defenses in unsafe languages, MLA are vulnerable to variants of CLA that only arise in the context of

```
1 fn rust_fn(cb_fptr: fn(&mut i64)) {
2     //Rust vectors have dynamic bounds
3     let mut vecs: vec![4];
4
5     unsafe{ vuln_fn(*Ptr to vecs*) }
6
7     // C++ changed vecs size to 128!
8     let vec_fp_addr: i64 = x.vecs[55];
9 }
```

(a) Rust code that passes a vector to C/C++.

```
1 void vuln_fn(int64_t vec_ptr_addr) {
2     // These values are set by a corruptible
3     // source, e.g., user input
4     int64_t array_index = 2;
5     int64_t array_value = 128;
6
7     int64_t* a = (void *)vec_ptr_addr;
8     a[array_index] = array_value;
9 }
```

(b) C/C++ code with an arbitrary write vulnerability.

Figure 9. Example of C/C++ using an arbitrary write to corrupt the size of Rust vector.

MLA. In particular, we highlight four such new vulnerabilities. First, Rust’s spatial memory safety can rely on bounds stored in memory which is only safe if the *entire* application is memory safe. Second, Rust’s automatic memory management relies on it being the only entity controlling the allocation status of memory. However, Rust commonly uses the libc `malloc()` implementation under the hood, giving rise to vulnerabilities in MLA. Third, we highlight two additional ways intended interactions via FFI over the language barrier can go wrong: passing bad values and more complex serialization/deserialization errors. Finally, we describe how multi-threaded programs heighten vulnerabilities. An overview of the attacks we present in this section is contained in Table 2. Moreover, we again point the reader to §A for an illustration of similar examples in Go.

4.3.1 Corrupting Rust Dynamic Bounds. For objects whose size is determined at runtime and may change, such as `vectors`, Rust stores the current

size of the object in memory. That value is then loaded and used in any required bounds checks. By corrupting the recorded size of the object, an attacker can enable a buffer-overflow of arbitrary length in Rust. While this attack is indirect, we note that Rust is seeing adoption in input processing libraries precisely because of its safety features. Consequently, corrupting the bound of a user facing object may be an efficient way to achieve an arbitrary write in practice. Regardless, this attack primitive is useful for attackers and undercuts Rust’s security guarantees in MLA.

Concretely, we demonstrate this attack in Figure 9. Rust allocates a vector on line 3 of Figure 9a. This vector is then passed by reference to the vulnerable C/C++ function, `vuln_fn` in Figure 9b. Because a `vector` in Rust allocates a pointer to the heap for the data in the `vector`, a `capacity` field that denotes the total possible length of the `vector`, and a `len` field that denotes the current length of the `vector` all on the stack at initialization, C/C++ can set the current length of the vector arbitrarily high on line 8 in Figure 9b. Thus, when Rust accesses the 55th element of the `vector` on line 8 of Figure 9a—an obvious OOB access—Rust will not panic as it normally should. Therefore, Rust now operates in the same level of spatial memory safety as C/C++. **Namely, the Rust program—solely written in Safe Rust outside its call to C/C++—can no longer claim spatial memory safety if it successfully compiles.**

4.3.2 Double Frees. In §4.2.3 we showed how UaF can arise in MLA. Here we generalize that to other temporal errors. In particular, if C/C++ frees a Rust object, Rust will still try to free that object at the end of its lifetime, giving rise to a double free vulnerability. Prior work has shown that double frees can lead to exploits in practice [70].

Looking back to our example in Figure 7, even if Rust did not directly use the `heap_obj` on line 6 after calling the `vuln_fn` C/C++ function, when the scope of `rust_fn` finishes, Rust will cleanup any memory associated with `heap_obj`. While Rust lifetimes can become more complex than default with explicit lifetimes and custom `Drop` traits that define the cleanup behavior, it will inevitably lead to memory being freed twice. Thus, Rust could then overwrite the memory it just freed, leading to a series of propagating UaF errors, and inevitably, undefined behavior. Therefore, Rust now operates in the same level of temporal memory safety as C/C++. **Namely, the Rust program—solely written in Safe Rust outside its call to C/C++—can no longer claim temporal memory safety if it successfully compiles.**

4.3.3 Intended Interactions over FFI. Interactions over FFI, where Rust-C/C++ *intend* to share data can also give rise to CLA. While we saw that the attacks in §4.2 are simplified when Rust shares a pointer with C/C++, we observe more complex attacks that can occur when C/C++ shares data with Rust. One version of this is where C/C++ hands Rust a pointer to a buffer to populate (*e.g.*, for user input sanitized by Rust). Another scenario is when Rust receives a function pointer from C/C++ (*e.g.*, for a callback function triggered on some event). In either case, Rust has no way of verifying that the shared pointer—or its contents—is valid, and must trust C/C++. This trust can be abused, leading to CLA.

For example, in Figure 10a, Rust calls `vuln_cb_fptr` to ask C/C++ to return a function pointer for its new callback function. If C/C++ returns malicious information, as we see on line 3 in Figure 10b, then when Rust uses that function pointer at line 4 of Figure 10a, the attacker successfully hijacks the application’s


```

1 // Uses a function pointer provided by C/C++
2 fn rust_fn(cb_fptr: fn(&mut i64)) {
3     unsafe { let mut fp_ptr = vuln_cb_fptr(); }
4     fp_ptr();
5 }

```

(a) Rust code that calls C/C++ to receive a callback pointer.

```

1 // Returns a call back function to register
2 int64_t vuln_cb_fptr() {
3     int64_t fp_ptr = get_attack();
4     return fp_ptr;
5 }

```

(b) C/C++ code that corrupts a return value to Rust.

Figure 10. Sample code to illustrate how CLA can corrupt even data intended to cross the FFI boundary.

control and can execute their weird machine. Note that any unsafe function could potentially be used to corrupt the return value from `vuln_cb_fptr`, perhaps on a different thread, or a callee of that function. Rather than pass Rust corrupted data as a return value, another variant of this attack is to have C/C++ directly pass corrupted data to Rust as a function parameter when it invokes a Rust function.

However, the Rust compiler does emit warnings about function pointers crossing the FFI boundary. In particular, Figure 10a is simplified; the programmer must explicitly transmute the function pointer received from C/C++ to a function pointer type within an unsafe block (this requirement is unique to function pointers, as other data can often be coerced with the `as` keyword in Safe Rust). While this helps identify which regions the programmer should likely sanitize, the function pointer conversion can reside within the same unsafe block used to call the C/C++ function, which may lead to bugs being overlooked.

More complex intended interaction errors are of course possible over the FFI interface. For instance, C/C++ strings and Rust strings have different representations, forcing conversion through null terminated C/C++ strings for

compatibility over FFI. This illustrates the need for *serialization* over the FFI interface. Serialization is a well known source of errors [167], but it has primarily been considered in inter-application scenarios (*e.g.*, I/O to networks, Files, or IPC), not intra-application scenarios where it arises as a type of CLA.

4.3.4 Concurrency and CLA. The preceding examples are all single threaded. This imposes constraints on CLA attacks; typically Rust must call into C/C++. However, in real applications such constraints are less relevant. All threads have access to the entire memory space, meaning that a C/C++ function executing on one thread that contains an arbitrary write can attack a Rust function operating on a separate thread. This effectively removes ordering constraints.

CLA is thus more general than just FFI issues when the MLA is multi-threaded.

4.4 Evaluation

In order to demonstrate the applicability of CLA, we collect a series of metrics that demonstrate the prevalence of CLA building blocks in real-world open source code bases. We focus on Mozilla Firefox [90], a large, open source project that has been consistently ported piece-by-piece from C/C++ to Rust. As Mozilla contributes to the creation of both Firefox and Rust itself, we believe characteristics of Firefox will act as a representative showcase for features of Rust and C/C++. We perform static analysis determine the order of magnitude prevalence of CLA building blocks to assess the scope of the problem; our goal is not to build proof-of-concept exploits. Thus, our two research questions for the evaluation are:

RQ1 How prevalent are language transitions?

RQ2 What is the distribution of language transitions across functions (*i.e.*, are language transitions widespread among all functions or centralized in a few)?

4.4.1 Methodology. We analyze Firefox version 92.0a1 using a debug build with optimizations disabled and the Rust v0 name mangling scheme, which allows us to distinguish Rust and C++ mangled function names. Our analysis utilizes `pyelftools` [30] to parse debug information, and `objdump` [203] to find callsites. We opt to analyze the compiled file as we can extract all needed information for our evaluation at the binary level and it is simpler than source level analysis. Additionally, we make our analysis openly available online⁶.

4.4.1.1 Source Language. We use a novel set of fingerprinting techniques to determine the source language of each function. Our fingerprinting technique is based on the differences in name mangling between languages during the compilation process. Name mangling is used by Rust and C++ to support function overloading, while still presenting unique function names to the linker. Name mangling encodes metadata about the function (*e.g.*, return value and argument types into the function name). For Rust v0 mangling, included with the nightly compiler, the function type (*i.e.*, normal, closure, or monomorphized) and the types of the function signature are encoded in the function name. The standard Rust mangling scheme is the same as C++, so we use the v0 scheme to differentiate the source languages. Note that FFI between Rust and C++ uses C style unmangled function names as the call target. Consequently, we assume that unmangled calls in a known Rust or C++ function represent language transitions (which will include a transition from C++ to C as a language transition). We do

⁶<https://github.com/mit-ll/Cross-Language-Attacks>

not assess the accuracy of this technique; we aim only to obtain a rough order of magnitude understanding of the prevalence of CLA building blocks.

4.4.1.2 Metrics. As discussed in §4.1, CLA leverages control flow transitions between source languages which motivates RQ1, and thus, the metrics we collect must quantify the behavior and interactions of functions with other languages. Namely, we want to observe how frequently, to which degree, and in what manner Rust and C/C++ invoke each other.

We collect a total of seven metrics to quantify language transitions in Firefox. First, we observe the *total number* of functions in each language to get a sense of how significant the role of new languages such as Rust are in development. Second, we collect the set of functions that each function calls, which we denote as *call targets*, as well as the set of *call sites* (*i.e.*, where control flow changes) within the function. An important subset of call sites are the *transfer points* in which one language is calling a function in a different language. Transfer points are key building blocks of CLA. For each call site, we determine the type of call used. Calls can be: *indirect* (*i.e.*, the call target is in a register), *dynamic* (*i.e.*, indirect through the program lookup table (PLT) used for functions in dynamically linked libraries), or *direct* (*i.e.*, to a constant address). Note that metrics on direct calls are not reported. Indirect calls are frequently targeted by code reuse attacks in the Weird Machine Execution phase, Figure 3. The PLT can similarly be corrupted, leading to the same effect [221]. For each function, we also collect its *invocation points*, or the set of functions that call it. We are particularly interested in invocation points that come from a different language, which we denote as *visitor points*.

Table 3. The prevalence of CLA building blocks. The number is bold is the total number of each item in the binary. (X, Y) represents the breakdown of the counts where X is the fraction of the total items the specific item accounts for, and Y is the fraction of those that come from the specific language. For example, there are 12,118 transfer points in Rust that constitute 3.70% of all call sites in Rust, and 5.32% of the transfer points in the entire binary come from Rust.

(a) Total function metrics.

	Rust	C/C++	Entire Binary
Total Functions	487,763 (100%, 26.68%)	1,340,347 (100%, 73.32%)	1,828,110 (100%, 100%)

(b) Call site metrics

	Rust	C/C++	Entire Binary
Call Sites	327,653 (100%, 9.23%)	3,220,415 (100%, 90.77%)	3,548,068 (100%, 100%)
Transfer Points	12,118 (3.70%, 5.32%)	215,778 (6.70%, 94.68%)	227,896 (6.42%, 100%)
Indirect Calls	179,598 (54.81%, 64.04%)	100,843 (3.13%, 35.96%)	280,441 (7.90%, 100%)
Dynamic Calls	126,710 (38.67%, 22.15%)	445,418 (13.83%, 77.85%)	572,128 (16.13%, 100%)

(c) Invocation metrics

	Rust	C/C++	Entire Binary
Invocations	346,469 (100%, 10.25%)	3,032,583 (100%, 89.75%)	3,379,052 (100%, 100%)
Visitor Points	184,799 (53.34%, 81.09%)	43,097 (1.42%, 18.91%)	227,896 (6.74%, 100%)

4.4.2 Results.

First, we observe the raw totals of our metrics collected in Firefox, including constituent libraries built from the Mozilla repo, in Table 3. Table 3 is made of three subtables that each breakdown related classes of

statistics. In each cell of Table 3, we show the total number of each metric collected in bold, the percentage of its class of metrics (*i.e.*, transfer points are a subclass of call sites) on the left, and the percentage of the metric that comes from each language with respect to the entire binary on the right. For example, we found 12,118 transfer points in Rust functions with which we can conclude that 3.70% of call sites in Rust are transfer points, but only 5.32% of transfer points come from Rust. These results make sense when paired with the fact that only 9.23% of call sites in Firefox are written in Rust, but have powerful implications for CLA in that nearly one in 25 of all Rust function calls will call back to an unsafe language. In fact, these results demonstrate significant opportunity for CLA. While a function call within a Rust function is typically the target of the memory corruption found in C/C++ for a CLA, in order for the unsafe language to corrupt Rust data, the Rust data must remain “live” while the unsafe language operates. Namely, when Rust calls C/C++, C/C++ has an opportunity to corrupt Rust data and create exploit points. These results indicate this “liveliness” requirement will often be met. Multi-threaded programs, such as Firefox, also relax this requirement as a C/C++ function can corrupt a Rust function on another thread. Moreover, we present a visualization of these statistics in a series of stacked bar plots found in Figure 11. Note that the y-axis in these figures is logarithmic.

We highlight more results from Table 3 notable to CLA as follows. We observe that 54.81% of call sites in Rust are indirect calls which indicate a use of a function pointer to make the call. This is important because these function pointers are the target of corruption in the forward-edge version of the CLA. Similarly, 38.67% of call sites in Rust are dynamic, which unsafe languages can corrupt for more serious loss of memory safety in Rust, similar to the attack presented in

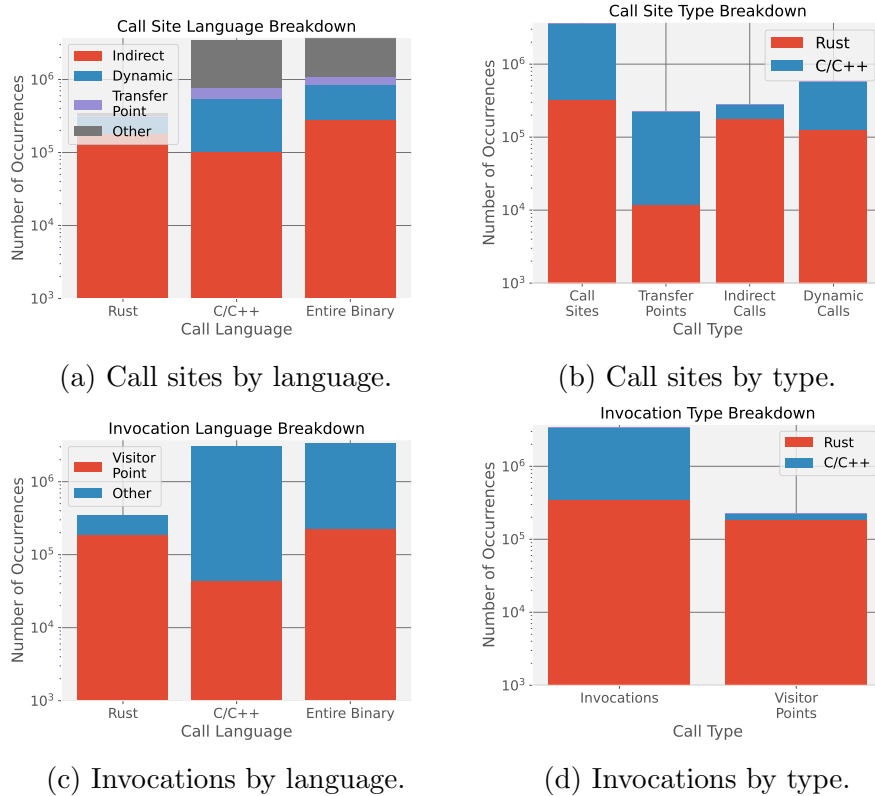


Figure 11. Stacked bar plots to breakdown different types of statistics found in Table 3. Y-axis is logarithmic.

Figure 3. On the other hand, we see that an overwhelming number of invocations of Rust functions come from memory unsafe languages. Specifically, 53.34% of invocations of Rust functions are visitor points, which indicates that C/C++ functions call Rust functions just as often as other Rust functions. Lastly, we point out that while the percent of these calls are convincing, the magnitude of transfer points (12,118) and visitor points (184,799) in Rust are also significant.

Next, we investigate Figure 12 to illuminate the trends of our metrics over the entire set of functions to compliment the aggregate values of our metrics found in Table 3. For example, from Figure 12a and Figure 12d, we see that the degree of unique functions called by Rust and towards Rust respectively overall tends to be less than C/C++ functions, while in Figure 12b and Figure 12c, we see that

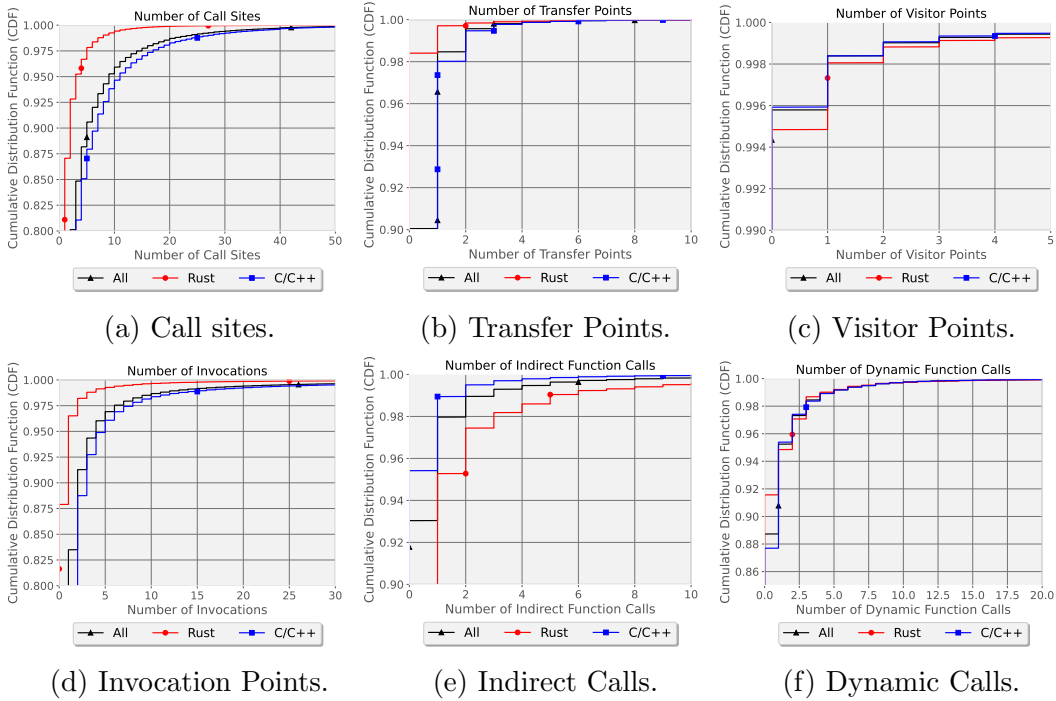


Figure 12. The Cumulative Distribution Function (CDF) of each CLA building block metric.

the subclass of functions we care about (*i.e.*, transfer and visitor points), tends to be similar between Rust and C/C++. Moreover, in Figure 12c, less than 1% of C/C++ functions contain a visitor point. Thus, we can conclude that while frequent, the number of unique functions that Rust calls that cross the language boundary is limited. However, it is important to remember that there is a large number of unique C/C++ functions (1,340,347), so even 1% of these functions being relevant to CLA is still significant.

Because we observe a small number of influential functions, which we denote as *heavy hitters*, with respect to CLA, we next present the top heavy hitters for call sites, transfer points, invocation points, and visitor points in Table 4. In this table, we present the top three heavy hitters for each language with its respective count. The function name is followed by an '@' symbol to indicate in

Table 4. Heavy hitter functions in Firefox.

	Rust	C/C++
Top Functions with Call Sites	<ol style="list-style-type: none"> 1. <code>assert_initial_values_match@libxul</code> (588) 2. <code>get_longhand_property_value<alloc>@libxul</code> (464) 3. <code>get_longhand_property_value<nsstring>@libxul</code> (459) 	<ol style="list-style-type: none"> 1. <code>CreateInstance@libxul</code> (1,631) 2. <code>generateBodyEv@libxul</code> (1,160) 3. <code>run@libxul</code> (846)
Top Functions with Transfer Points	<ol style="list-style-type: none"> 1. <code>main@crashreporter</code> (55) 2. <code>main@modutil</code> (25) 3. <code>main@logalloc-replay</code> (24) 	<ol style="list-style-type: none"> 1. <code>Unified_cpp_protocol_http3@libxul</code> (84) 2. <code>UICrashUI@crashreporter</code> (54) 3. <code>nsWindow@libxul</code> (49)
Top Functions with Invocations	<ol style="list-style-type: none"> 1. <code>as_bytes@libxul.so</code> (930) 2. <code>state@libxul</code> (554) 3. <code>_Unwind_Resume@plt</code> (520) 	<ol style="list-style-type: none"> 1. <code>AnnotateMozCrashReason@libxul</code> (134,254) 2. <code>ReportAssertionFailure@libxul</code> (131,545) 3. <code>Array_RelocateUsingMemutil@libxul</code> (17,475)
Top Functions with Visitor Points	<ol style="list-style-type: none"> 1. <code>_Unwind_Resume@std</code> (488) 2. <code>as_str_unchecked@libxul</code> (25) 3. <code>qcms_transform_data@libxul</code> (24) 	<ol style="list-style-type: none"> 1. <code>__assert_fail@GLIBC</code> (4388) 2. <code>ostream@GLIBC</code> (3326) 3. <code>strlen@GLIBC</code> (1294)

which binary the function resides. Of note, we can see that the top visitor points in C/C++ functions come from GLIBC, and have a large magnitude (*e.g.*, 4,388 for `__assert_fail`). This large magnitude indicates a long tail on the corresponding CDF in Figure 12c and suggests many other functions in C/C++ will have only one or two visitor points. Further, the library *libxul* tends to dominate the heavy hitters list which indicates an ample opportunity for CLA.

Our findings indicate that the building blocks for CLA are abundantly available and an attacker has a very large catalog of options when building such attacks. These results highlight the prevalence of opportunities for CLA and the fact that existing countermeasures are not sufficient to prevent them. New countermeasures are necessary when applications are written in multiple languages with mismatching threat models, which we further discuss in the next section.

4.5 Discussion

We have focused our discussion of CLA primarily on Rust and Go in combination with C/C++. However, the issue of threat model mismatches extends beyond these languages, and beyond issues of memory safety in MLA. In this section, we discuss future research directions and the broader implications of CLA. We also provide some thoughts on securing MLA.

4.5.1 CLA in Go. In contrast to Rust, which was designed as a systems programming language that would have to integrate with C/C++ to be adopted, Go is intended to largely be a stand alone programming language. While Go supports MLA via CGo, and there are certainly projects that leverage CGo (*e.g.*, Docker, Kubernetes, and CockroachDB), several Go language developers discourage use of CGo have written publicly about issues created by CGo [53] for Go applications. In fact, some projects that initially leveraged CGo, later chose to remove it (*e.g.*, the Go implementation of git [1]). What direction Go will go in the future remains to be seen.

Regardless, the vulnerabilities in §4.2 and §4.3 nearly all directly translate to Go from Rust (§A). The exception are the temporal safety vulnerabilities. Go leverages garbage collection (GC) to provide temporal safety, in contrast to Rust’s lifetimes. This opens the door to use-after-gc errors and more complicated double free scenarios when Go interacts with C memory management.

4.5.2 CLA In Other Languages. CLA arises anytime two or more languages are used in an application and have different threat models. This extends beyond just Rust-C/C++ or Go-C/C++. For instance, the C++ language standards since C++11 have introduced a growing eco-system for safe(r) programming practices, such as smart pointers. However, applications written in C++11 and newer are still backwards compatible with older C++ standards and C applications, and often feature code with older code standards (*e.g.*, in included libraries). Such older, and unsafer, components of the application can subvert the additional safety guarantees offered by newer C++ features. Another instance of this in language dichotomy is applications in Rust that contain unsafe Rust, which has seen recent research interest [26, 132].

Note that while the FFI interface requires the `unsafe` keyword in Rust, CLA is fundamentally much more dangerous than normal uses of `unsafe`. CLA opens the door to the entire gamut of vulnerabilities present in unsafe languages, whereas `unsafe` is usually used for relatively simple operations that can't be statically proven safe by Rust's type system, and so are excluded. As Rustbelt [109] has shown, such uses of `unsafe` are simple enough to be amenable to formal verification, as opposed to the exponentially larger amount of code exposed via FFI than can be leveraged in CLA attacks.

4.5.2.1 Interpreted Languages. Interpreted languages add another dimension to possible threat model mis-matches. The Java Virtual Machine (JVM) has been implemented in both C and C++, and the original Python interpreter is written in C, though others exist today. Indeed – attacks against Java have targeted the JVM [93]. All browsers have JavaScript interpreters, though attempts are made to sandbox them [69, 17]. We leave a deep inspection of appropriate threat models for mixed interpreted/compiled applications, particularly when the interpreted language has safety guarantees that do not align with its interpreter, as future work.

4.5.2.2 Multiple Safe Languages. CLA is possible in MLA even if all component languages are themselves safe. For instance, Rust and Go both provide memory safety, but because their *strategy* to provide memory safety differs, an attacker could launch a CLA on such a multi-language program. In particular, Rust and Go prevent temporal memory corruption with lifetimes and garbage collection respectively. Should these different systems disagree on the state of memory, double frees or UaFs are possible. We believe other subtle vulnerabilities are likely, but do not explore them here.

4.5.2.3 CLA and Verified Code. Formal methods increasing maturity is evidenced by the seL4 [119] microkernel and recent verification of KVM [128]. Formal verification offers mathematical proof of certain properties, as long as the assumptions used in the analysis hold and the underlying model is accurate. The current approach is to formally verify a critical subset of code, such as the OS/hypervisor or cryptographic libraries [232]. Such verified code is then used as part of a larger, unverified application, enabling CLA. Note that the mixture of verified and unverified code opens up new attack possibilities not explored here, and the unverified code can be used to undermine any of the assumptions/modeling used during formal verification.

4.5.3 CLA Beyond Memory Safety. CLA results from the mixed assumptions of the different threat models in a MLA, and this extends beyond memory safety. Different assumptions and threat models around type safety enable type confusion [204, 95] attacks in MLA. This endangers schemes for other purposes, including language-based isolation [138, 153, 36], general verification [114], and information-flow control [148]. Correctness violations are also possible—Rust’s type system notably provides significant concurrency safety guarantees. Indeed, a selling point of Rust is “fearless concurrency” [116] in which the type system removes many hard-to-debug concurrency errors. In MLA, CLA can reintroduce data races as Rust’s guarantees no longer hold. While more benign than memory safety violations, such errors can still lead to denial-of-service attacks.

Our insight that differing assumptions leads to adverse effects holds generally—MLA must be explicit about the assumptions each component is making, and care must be taken to ensure that the application as a whole is not weaker than its constituent parts.

4.5.4 Defense Strategies for CLA. Next, we offer some thoughts on general approaches for defending against CLA, and break the problem into two components: preventing unintended interactions and securing intended interactions. We end with a discussion on alternative approaches.

4.5.4.1 Preventing Unintended Interactions. Preventing unintended interactions requires isolating—to the greatest extent possible—the memory of the different language components. One such approach could place each language component in its own process and rely on virtualization and explicit shared memory to control interactions. In fact, this has been proposed by Sandcrust [122]. Doing so is likely impractical, however, particularly if more than two languages are involved. Consequently, intra-process isolation techniques will need to be refined for this use case. For example, recent work shows that isolating the heap between languages is a significant step in the right direction [175]. However, doing so will not be free, as there will be performance costs both for the isolation technique and to allow data-flow across the language boundary. Isolating the stack may require separate stacks per language, which will increase the overhead of language transitions and requiring more intrusive changes. Finally, while code isolation already exists outside of JIT systems courtesy of DEP [22], preventing the corruption of generated code has also been studied for JIT [224], and is largely a separate problem.

4.5.4.2 Securing Intended Interactions. Securing intended interactions requires verifying that interactions across the FFI interface cannot violate the security properties of either language. To date, there has been work on using formal methods [109, 172, 217] to do so in a variety of contexts. In particular, Rust FFI has been studied, building on existing literature (*e.g.*, the

java native interface (JNI) [206]). Beyond formal methods, new sanitizers that target interactions across FFI should be developed by the community, as well as runtime defenses. For example, as discussed further in Chapter V, in order to sanitize data meant to be shared between Rust and C/C++, Rivera et al. propose a new pointer construct called *pseudo-pointers* [175]. Moreover, tagged architectures [198, 225, 81, 177] that enable metadata to be added to data provide a promising hardware based approach for mitigating CLA as well. Indeed, the Cheri project [226] has proposed such an extension [223].

4.5.4.3 Alternative Defenses. Some recent work attempts to solve specific subsets of CLA. For example, there is an ongoing effort to add CFI checks to Rust code (*e.g.*, Control Flow Guard (CFG) [24]). Such a defense will help mitigate specific variants of CLA, such as the attack outlined in §4.2.4, since defense assumptions between the two languages will now match for this particular variant. However, as we have mentioned, CLA proposes a problem much deeper than control-flow hijacking (*e.g.*, loss of Rust memory safety guarantees) that such a defense cannot solve. Similarly, incremental deployment of randomization techniques, such as Address Space Layout Randomization (ASLR) [165, 33], offers another alternative approach, but previous work has shown randomization can be bypassed [181, 193]. Moreover, it is unclear if ASLR can address each variant of CLA detailed in §4.2 and §4.3. Thus, we believe that while these defenses have merit, they address a symptom of CLA rather than the cause.

CHAPTER V

Pseudo-Pointers

As seen in *Rivera, E., Mergendahl, S., Shrobe, H., Okhravi, H., & Burow, N. (2021, December). Keeping safe rust safe with galeed. In Proceedings of the 37th Annual Computer Security Applications Conference (ACSAC), (pp. 824-836).*

Next we consider ways to ensure spatial safety against Cross-Language Attacks (CLA) in Multi-Language Applications (MLA) that consist of both a memory and type safe component (Rust) and an unsafe component (C/C++). As seen in Chapter IV, such applications pose two threats against their “safe” components. The first is that in practice safe and unsafe languages share a heap, with no abstraction or isolation between them at runtime, such as virtual addresses between processes. This allows efficient communication, but also means that an arbitrary-write vulnerability in C/C++ can alter memory that notionally belongs to Rust. The second is through the programmer intended interactions, in which Rust gives C/C++ a pointer to use.

In this chapter, we consider a defense that consists of two components: 1) a runtime defense that isolates Rust’s heap from manipulation by C/C++, thereby preventing *unintended* interactions, and 2) a sanitizer, that we call **Pseudo-Pointers**, that secures *intended* interactions between the safe and unsafe programming language. This style of defense has also been proposed elsewhere [175], and is not the main contribution of this dissertation. However, due to popularity of this isolation methodology, we describe its structure in detail in order to describe the state-of-the-art for spatial memory safety in Multi-Language Applications (MLA).

In particular, Pseudo-Pointers protect safe code from possible corruptions in the unsafe code with which it interacts. In this report, the considered runtime defense is built on top of `seL4`, which enables the use of *capabilities* to remove read/write access to the heap. However, the Pseudo-Pointers design is generic and can work with any enforcement mechanism. For example, previous work has also used Pseudo-Pointers in conjunction with Intel’s Memory Protection Keys (MPK) [104] as access control to the heap [161, 211, 98, 183, 175]. The Pseudo-Pointers sanitizer replaces pointers passed across the language boundary with identifiers to Rust objects, and turns dereferences of such pointers into function calls back into Rust with the object ID and request operation. The security guarantees of the sanitizer are demonstrated on micro-benchmarks, which show reasonable overhead.

5.1 Pseudo-Pointers Design

In this section, we describe the design of Pseudo-Pointers. Pseudo-Pointers has two components: a runtime defense for isolating the Rust heap from unintended interactions, and a sanitizer for securing intended interactions. We describe each component separately for ease-of-understanding, but they are both part of the overall technique for preserving the memory safety guarantees of safe Rust when it interfaces with unsafe code.

Rust is primarily being incrementally deployed: a longstanding codebase written in a different unsafe language (most often C/C++) is typically converted piece-by-piece to the equivalent Rust code. For example, the ubiquitous web browser Firefox, started its migration from C/C++ to Rust in 2016 [146]. Mozilla, the maintainers of Firefox, list Rust’s memory safety as primary reason for the switch [146].

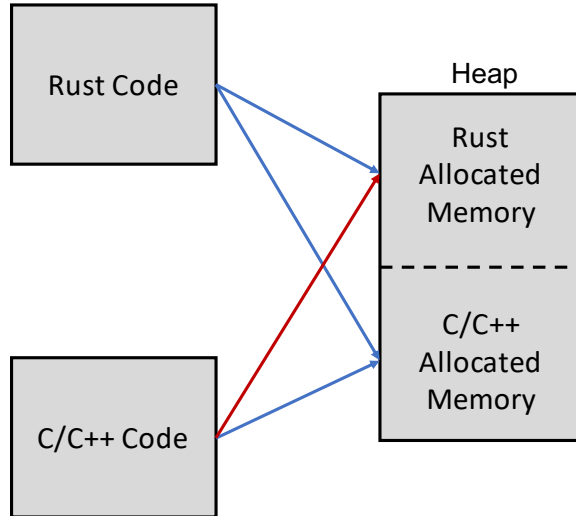


Figure 13. Possible memory accesses in Rust-C++ applications

Mixing Rust with another language (*e.g.*, C/C++) breaks the Rust memory safety model, and leaves the mixed-language application more vulnerable to exploit than a CFI [14, 43] hardened C/C++ implementation [159]. Our work is general to any unsafe language that interfaces with Rust, but for the sake of simplicity in the discussion below we focus on C/C++. C/C++ is not bound by the Rust memory model, nor does it have to obey the restrictions of the Rust compiler. Calling into C/C++ from Rust breaks any promises of memory safety, and thus such calls must always be marked as *unsafe* in Rust.

In a mixed Rust-C/C++ application, there are 4 possible patterns of memory access: Rust code accessing Rust-allocated memory, Rust code accessing C/C++-allocated memory, C/C++ code accessing C/C++ allocated memory, and C/C++ code accessing Rust allocated memory (see Figure 13). Rust code accessing Rust memory should never be able to break Rust memory safety (by definition). Additionally, Rust memory safety is independent of accesses to C/C++ memory.

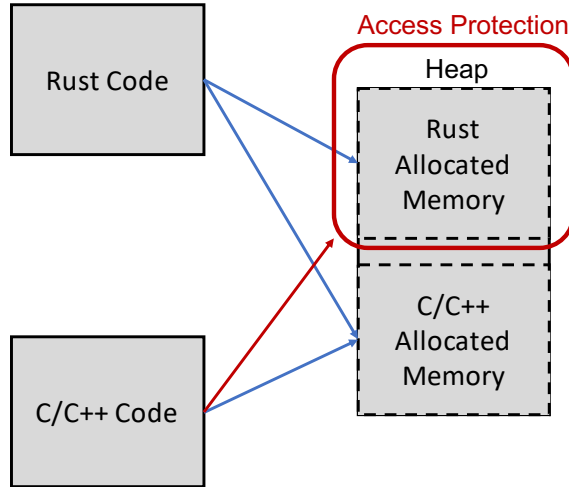


Figure 14. Protection via page-level memory isolation and permissions switching

In contrast, C/C++ accessing Rust memory (*i.e.*, the red arrow in Figure 13) could cause any number of violations to Rust memory safety guarantees, up to and including full control-flow hijacking [159]. We separate these memory accesses further into two cases: *intended* and *unintended* accesses. An intended access occurs when C/C++ is explicitly given the location of some part of Rust memory by Rust code and then accesses that Rust memory, while any other access is considered unintended. An example of an intended interaction is when Rust parses a message and passes a pointer to it to C/C++ for further processing. An example of an unintended interaction is when an arbitrary write gadget (*e.g.*, a dangling pointer in C/C++) is used to modify a data structure in Rust memory when such an interaction was not conceived of by the developer.

5.1.1 Preventing Unintended Interactions via Heap Isolation.

First, we focus on preserving memory safety in the presence of unintended accesses, and then we extend our discussion to secure intended accesses in §5.1.2.

In order to preserve Rust memory safety in the Rust component of a MLA, we must isolate and restrict Rust memory such that it cannot be accessed by a

component written in another language. If only Rust can access Rust memory, Rust memory safety is preserved.

5.1.1.1 Heap Isolation. Pseudo-Pointers approach to Rust memory isolation is to make sure that all of the pages of Rust-allocated memory are in the same page group, and then to set permissions on these pages in such a way that external functions are unable to access the Rust memory. If only the given Rust component can access its own memory, and accesses from other non-Rust components to Rust memory are forbidden, then the program stays consistent with the Rust memory model despite executing untrusted code in another language. General memory safety for non-Rust components is out of scope of this work, and is well-studied in literature [199, 150, 151, 187].

5.1.1.2 Heap Splitting. In order to isolate the Rust heap, we split the unified program heap into safe language component heaps that are protected, and a remaining unified unsafe language heap. Each safe language heap comprises a distinct set of pages with its own access capability. This allows safe language permissions to be controlled. Note that if, *e.g.*, a page used for Rust heap contains a C/C++ allocation, then access permissions that operate on the page-level can no longer distinguish between the language heaps and thus appropriate permissions regimes. Note also that the pages for each heap can be interleaved, so long as each page in a language heap is dedicated exclusively to that heap.

5.1.1.3 Access Policy. Whenever a safe language is executing, its heap and the unified unsafe language heap have full read and write permissions. Leaving the unified unsafe heap accessible still prevents unintended interactions while maintaining safety, see Figure 13. On language transitions, which happen on function calls (and correspondingly thread context switches), permissions are

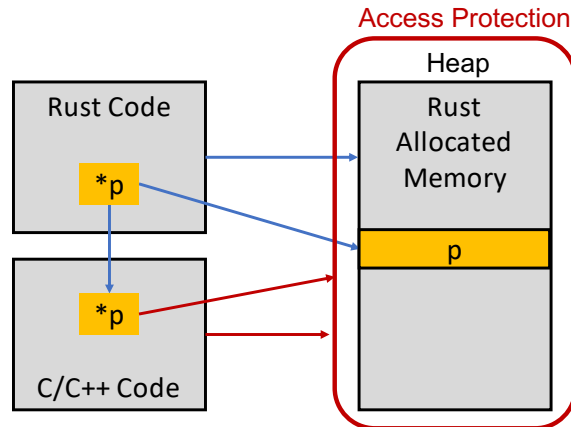


Figure 15. Pseudo-Pointers restricts all accesses by default.

removed for the calling safe-language heap. This permission change is inverted upon return. The Pseudo-Pointers policy invariant is that a safe language heap is accessible if and only if a thread associated with that safe language is currently executing. This policy results in full isolation of the language heaps, which is overly restrictive in practice. We next discuss how to relax this regime while maintaining safety with Pseudo-Pointers.

This heap isolation technique is a *runtime defense* (*i.e.*, exploit mitigation); a technique that is meant to run continuously when the application is running in order to provide the protection discussed above. As such, its small performance footprint is crucial for its adoption.

5.1.2 Securing Intended Interactions via Pseudo-Pointers.

Pseudo-Pointers default policy intentionally excludes *intended accesses* (*i.e.*, times when C/C++ is explicitly and intentionally given a pointer to Rust memory).

This most commonly occurs in FFI function calls, by passing a pointer as an argument to a structure in memory instead of passing directly by value. In fact, this pattern is employed by many Firefox modules, often due to performance or

storage considerations. Pseudo-Pointers’s default behavior breaks this intended behavior, illustrated in Figure 15.

In particular, we present an option for data flow between safe Rust and unsafe code that does not require breaking the safety guarantees provided by Pseudo-Pointers’s default heap policy. Instead, when external functions need access to Rust memory, we will force the external function to request that Rust make the change in its own memory—a request that Rust can safety-check and reject. We present a design for both the interfaces and underlying machinery required in both the Rust and external functions, followed by an implementation of this design specialized to Rust and C/C++.

We introduce the idea of Pseudo-Pointers as identifiers that pass to an external function instead of raw pointers. Pseudo-Pointers keep an internal mapping of identifiers to raw pointers. Any time a non-Rust component attempts to dereference a Rust pointer, it must present a valid, non-expired identifier to Rust via an exposed API, along with the information for the change it wishes to make (if applicable). Rust verifies that the Pseudo-Pointer is valid and non-expired. In the case of a write request, Rust also verifies that the value to write represents a valid member of the type associated with the memory location. Once verified, Rust executes the request. Since only Rust directly accesses Rust memory, we can keep our heap isolation in place and ensure memory safety (see Figure 16).

In contrast to our heap isolation that is a runtime defense, our Pseudo-Pointer technique is a *sanitizer* [199] meant to be used by the developer to detect and remove vulnerability pre-release. Accordingly its performance budget is much higher [199].

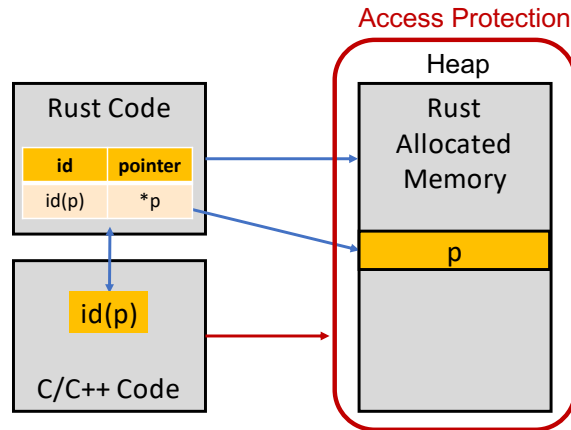


Figure 16. In our design, C/C++ uses pseudo-pointers (e.g., $\text{id}(p)$) to request that Rust dereference Rust memory.

We break the design into three components to discuss further: necessary properties of Pseudo-Pointers, the API which Rust exposes to other external components, and the requirements on external functions.

5.1.2.1 Pseudo-pointer Properties. Pseudo-Pointers need to have certain properties in order to function correctly as safe pointer identifiers: uniqueness, automatic expiration, and forgery resistance.

Pseudo-Pointers must be unique to the memory they represent: each Pseudo-Pointer must represent exactly one real memory location, and each memory location must be represented by at most one Pseudo-Pointer. Not only is this necessary for being able to look up the corresponding memory location, but also it is necessary to comply with the Rust borrow-checker.

Pseudo-Pointers must automatically expire when the corresponding memory is freed at the latest. If a Pseudo-Pointer is still treated as valid and used to access memory even after its corresponding memory location has been freed, we have violated Rust memory safety with a use-after-free error.

Pseudo-Pointers must be difficult to guess or forge. Ideally this applies even between different runs of the same program, which requires some level of randomization. It should be noted that while forging a valid Pseudo-Pointer could potentially cause information leaks or even information replacement (both important security risks), neither one has the possibility of breaking memory safety, since the operations are still controlled by safe Rust and are valid operations within the Rust memory model.

Pseudo-Pointer management should be automated, and transparent to the developer. This is not a requirement for correct functionality, but is still critical in the push to incorporate these safety changes into existing applications. The more of the process that can be automated, the lower the burden on the developer. A fully-automated, transparent system for introducing and using Pseudo-Pointers reduces the possibilities for potential mistakes.

5.1.2.2 Rust API. Pseudo-Pointers are functionally useless without the corresponding external-facing Rust API, consisting of functions which can be called by another language in order to read from or write to the memory represented by a Pseudo-Pointer. For any given structure that will be used in the FFI, the Rust API will have a getter and setter for each field within that struct. The function names for these getters and setters are automatically generated using a naming strategy that includes both the struct type and the field name. These functions will either be *NOPs* or raise errors when asked to perform a memory operation that is inconsistent with its current internal understanding of that memory location, including both type errors and expired Pseudo-Pointers. These functions must also be entirely in safe Rust, where compile-time and run-time checks automate most of this for us.

5.1.2.3 *External Function Transformation.* Pseudo-Pointers

are passed in place of pointers in every call to an external function, to avoid ever passing a Rust memory location to another language. Before each external function call, we create a Pseudo-Pointer for the pointer that would normally be passed, and pass that instead. We invalidate the Pseudo-Pointer once the function returns, for the reasons mentioned in §5.1.2.1.

If we rewrite calls to external functions to use Pseudo-Pointers, we will also need to rewrite the external functions themselves to accept and use these Pseudo-Pointers everywhere that they would have had a real pointer instead. Pointer dereferences and writes need to be converted into the equivalent Rust API calls as seen in §5.1.2.2.

Ideally, these rewrites can be done automatically, which would once again mitigate the burden on the developer. In fact, full automation of these external rewrites would allow secure calls to large existing legacy libraries with little to no change, allowing for this technique to be used in cases like migration from a legacy codebase in an unsafe language (*e.g.*, Firefox, originally in C/C++). Additionally, since developers are often hesitant to make changes (even automated ones) to working legacy code, these rewrites should be able to be performed at compile time instead of modifying the source file.

Aliasing in unsafe languages could stand as a barrier to the full automation above, as it may be impossible to completely determine the full set of pointer dereferences for a Rust object. We note that ours is a conservative approach prioritizing guaranteed safety. In cases where alias analysis fails and a pointer dereference is not transformed, that pointer dereference will be disallowed by the memory access permissions and will not violate memory safety. The developer can

then debug the code to ensure that all the necessary pointers are transformed. We did not encounter such cases in our analysis, although their possibility remains open.

5.1.3 Pseudo-Pointers Security Guarantees. Pseudo-Pointers have two security aims: 1) to prevent unintended interactions on the heap in Multi-Language Applications (MLA) by providing a runtime defense, and 2) helping developers identify vulnerabilities in intended interactions between languages by providing a sanitizer. Heap isolation in turn has two components—the isolation policy and the mechanism used to enforce it. The isolation policy is simple: a safe-language heap is only accessible when code in that language is executing. Enforcing this requires changing permissions whenever the executing language changes, which is precisely what Pseudo-Pointers do. Such a policy is as sound as its underlying enforcement mechanism. For example, Pseudo-Pointers could start a new thread on a language transfer and use separate capabilities passed to the separate threads on `seL4` to enforce heap isolation due to `seL4`'s low context switch overhead. Additionally, if a new memory-unsafe language thread is started for a language transfer, Pseudo-Pointers could start a new memory-safe language thread to receive the Inter-Process Communication (IPC) requests when the memory-unsafe language makes getter and setter requests. This would provide additional temporal safety (see Chapter VI) and stack isolation for further spatial safety.

The sanitizer ensures that all pointers given to C/C++ are still used in a memory and type safe manner. It relies on Rust's built in guarantees to do so, by referring all pointer dereferences back to safe Rust for verification before they are completed, and the result returned to C/C++. By removing all pointers to the

Rust heap from C/C++, Rust maintains the integrity of its heap against intended interactions.

5.2 Pseudo-Pointers Implementation

We implemented a prototype for Pseudo-Pointers, specialized to interactions between Rust and C/C++. The full source code for our implementation is available online ¹.

5.2.1 Heap Isolation.

5.2.1.1 Heap Creation. Rust provides machinery for writing a custom allocator that can be imported as a crate and used in place of the default. Our implementation does not separate the allocator into its own crate out of convenience, but doing so would allow a developer to switch to this allocator with a handful of lines of code.

Because `seL4` does not provide an implementation for a page table, we also assume there is an isolated memory allocation service that receives Inter-Process Communication (IPC) requests for more memory, and can return `seL4` capabilities to these memory pages to the requesting thread. This service must be isolated to prevent a compromised thread from corrupting memory allocation. Because IPC requests are *badged* on `seL4` (*i.e.*, the memory allocation service knows the identity of the IPC sender), the memory allocation service can return access to the heap in a language-aware manner.

5.2.1.2 Access. Code to switch heap permissions is included on either side of all external function call sites. Namely, the code immediately preceding the call site to another language sets up an Inter-Process Communication (IPC) call to the other language thread. This effectively switches the selected heap permissions,

¹<https://github.com/mit-ll/galeed>

as `seL4` can track which thread is executing and consequently update memory access based on the capability provided. Similarly, the code immediately following the call site switches all permissions back on by returning from the IPC call. We currently switch the Rust memory permissions to read-only at all call sites, but permissions could be selected at each call site by swapping named constants.

5.2.2 Pseudo-Pointers. Pseudo-Pointers extend our heap isolation mechanism to secure intended interactions between Rust and C/C++. We implement Pseudo-Pointers for user-defined *structs* that are intended to be passed across the language boundary as those are the primary vehicle Rust and C/C++ use to exchange data. For primitives like booleans, integers, and floating-point numbers, we would normally expect these to be passed by value directly. For other constructs in the language and/or standard library, further work is required to implement the necessary transformations.

Pseudo-Pointers are implemented as a transparent `struct` containing a single field, the ID of the Pseudo-Pointer as a signed 32-bit integer. The `struct` also contains a `PhantomData` field that is the type of the pointed data. `PhantomData` is used in Rust for fields that exist at compile time but not at runtime. This allows us to make distinctions in code between Pseudo-Pointers that represent different types, while still having confidence that they will still compile down to 32-bit identifiers once all of the compiler checks are passed.

We also define a specific map `struct` for Pseudo-Pointers, including a function that takes a Rust `struct`, adds it to the map, and returns the corresponding Pseudo-Pointer, and the reverse function that takes a Pseudo-Pointer, removes it from the map, and returns the Rust `struct`. Every time a `struct` is added to the map, it will be added with a different ID, and every time

a `struct` is retrieved, that ID becomes invalid. This prevents external functions from attempting to access a `struct` after Rust has reclaimed it.

It is worth noting that creating Pseudo-Pointers using this interface requires having ownership of the object. One cannot just have a writable reference to the object. This is how we ensure temporal memory safety, as Rust requires the object's lifetime must extend for at least as long as it is in the map.

Pseudo-Pointer support is implemented as an attribute macro that can be added to a `struct`. This attribute macro creates the global map that will hold all Pseudo-Pointers of this `struct` type. Additionally, the macro automatically creates the API that will be exposed to external functions, as described in the next section.

5.2.2.1 Rust API. The attribute macro is able to generate getter and setter functions for each field of a `struct` by name. The macro has access to the type information of each field, so these functions are able to carry that type information in their return value and arguments respectively.

These functions use the Pseudo-Pointer provided as an argument, and go to the appropriate Pseudo-Pointer map to request access. If the Pseudo-Pointer is valid, the function proceeds as expected, either reading or writing the appropriate value. If the Pseudo-Pointer is invalid, the function will panic. Other reactions to an invalid pointer (*e.g.*, a NOP instruction instead of a panic) can also be easily used as appropriate depending on the application.

In addition to generating the getter and setter functions based on the name of a field, we also generate equivalent functions based on that field's position in the `struct`. This enables some of the low-level automation described in the next section.

```

1 int add5(MyStruct* const p) {
2     p->x += 5;
3 }

```

(a) Before

```

1 int add5(ID<MyStruct> const p) {
2     x = get_x_in_MyStruct(p);
3     set_x_in_MyStruct(p, x+5);
4 }

```

(b) After

Figure 17. Transforming an example C++ function to use pseudo-pointers.

5.2.2.2 External Function Transformation. In order to use this new Pseudo-Pointer interface, external C/C++ functions that once accepted pointers to `structs` in memory must be modified to instead accept Pseudo-Pointers, and operations on those pointers must be replaced with the appropriate Rust API getters and setters above. Figure 17 shows an example of this transformation.

Instead of placing the burden on developers to manually perform these transformations, we automate this transformation process. We introduce a module-level pass into the LLVM compiler which is enabled by a command-line flag. This pass transforms identified functions by replacing the expected pointer argument with a Pseudo-Pointer argument. It then traces usages of that argument through the function, replacing load instructions with calls to the correct getter function and store instructions with calls to the setter function. The information needed to determine the correct function can be found in the type information that LLVM preserves.

5.3 Evaluation

In this section, we evaluate our safety claims and calculate the performance overhead costs for our prototype.

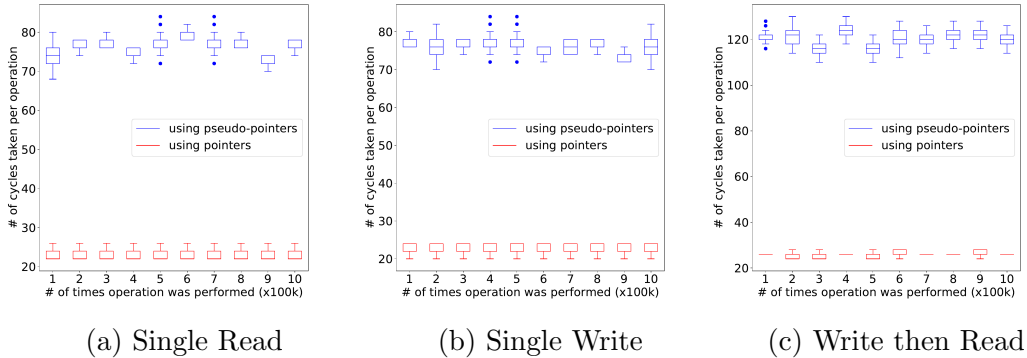


Figure 18. Pseudo-Pointer micro-benchmarks

5.3.1 Pseudo-pointers. To evaluate the functionality and performance of Pseudo-Pointers, we developed a proof-of-concept application in which the C/C++ side has a “library” of functions which took in a pointer to a Rust `struct` and read from and/or wrote to that struct. We are able to show that the compiled unit for this application had replaced all pointer dereferences and writes for the Rust struct with the corresponding Rust function calls for that struct. Rust pointers were never accessed from C/C++, while other pointers not from Rust were left unaffected.

We evaluate the performance overhead of adding these additional function calls using micro-benchmarks (see Figure 18). We found that there is $\sim 3x$ overhead for each individual read/write operation; however, when operations are chained the overhead is not $\sim 6x$ as expected but instead $\sim 4.5x$, indicating that the compiler toolchain is inserting additional optimizations post-transformation.

This overhead is considered quite practical for sanitizers, many of which have overheads ranging from $3x$ to over $10x$ [199].

5.4 Practical Lessons Learned

In this section, we discuss some of the lessons we learned during this effort for practical deployment of a technique like Pseudo-Pointers, how alternative design choices could impact them, and the directions for future work. It is our hope that these lessons not only inform the reader about these practical considerations when using Rust and Pseudo-Pointers, but also they can help researchers be cognizant of some of the practical challenges and pitfalls when developing similar technologies.

5.4.1 Mixed-Language Application Security. This work is a first attempt to address the security of mixed-language applications, and only considers interactions between compiled languages. Future work should consider interactions between compile and JIT compiled languages such as Python, or with the JVM. Further work is also needed to examine, both statically and dynamically, the full relationship between Rust and C/C++ applications.

5.5 Limitations

Pseudo-Pointers also has a number of limitations that we discuss here. In our prototypes, we intentionally focused on preserving memory safety first, sometimes to the detriment of performance. In the Pseudo-Pointer sanitizer, we replace C/C++ pointer access with external thread-separated Inter-Process Communication (IPC) calls, performing this step before either compiler has a chance to potentially optimize some of these accesses away. In addition, in both cases, we made no attempts to allow for LLVM's cross-language link time optimization (LTO).

Pseudo-Pointers can also be further automated, with the end goal being a fully automated compiler process that requires little to no developer input. We have already achieved this on the C/C++ end with the LLVM pass that

automatically replaces Rust struct pointers with pseudo-pointers and inserts the correct function calls, but many opportunities are still available on the Rust side.

Moreover, in our Pseudo-Pointers prototype, we currently support flat user-defined `structs`. This covers a large amount of use cases, but must be expanded to accommodate current Rust/C/C++ interactions. For example, we do not support strings, which are used in the parsing modules that Firefox has migrated to Rust.

Lastly, our prototype currently depends on having access to the original source code for Rust, and at minimum the LLVM bytecode for C/C++. Future work can investigate how to retrofit security in cases where only Rust/C/C++ binaries are available.

CHAPTER VI

Manipulative Interference Attacks (MIA)

As seen in the unpublished submission *Mergendahl, S., Fickas, S., Norris, B., & Skowrya, R. (2024, May). Manipulative Interference Attacks. In 2024 ACM Conference on Computer and Communications Security (CCS), (In Submission).*

While the spatial safety (*i.e.*, memory safety) is important to embedded, Cyber-Physical Systems (CPS) systems, timing predictability and correctness is also a paramount design consideration in these systems. Many such systems include hard real-time (HRT) tasks, where unexpected jitter or latency can cause deadline misses that can have catastrophic physical consequences. It is therefore imperative to maintain temporally correct execution for such tasks, even while processing other workloads with less stringent temporal requirements. This fundamental challenge has motivated over four decades of research dating back to seminal results such as the sporadic server [200], and up through current research on mixed-criticality scheduling (MCS) [214, 41].

A guiding philosophy of much work on temporal budgeting and mixed-criticality scheduling is that of *temporal isolation*. Hard real-time safety-critical tasks should be capable of meeting their real-time requirements even in the presence of high demands placed on other aspects of the system. In its original formation, the sporadic server was presented to improve the quality of service (QoS) of aperiodic tasks while ensuring periodic HRT tasks could still safely execute. More recently, much work on MCS is motivated by the fact that determining task worst-case execution times is notoriously difficult, especially on modern complex architectures. In the (hopefully) exceedingly rare case that high-criticality jobs run longer than expected (*e.g.*, due to rare microarchitectural

effects), MCS provides a means of maintaining the temporal-correctness guarantees of high-criticality tasks.

While temporal-isolation mechanisms were principally designed for reliability and QoS, they are also important *for security*. Without temporal isolation, malicious actors could trivially perform denial of service (DoS) attacks to consume processing time needed by HRT tasks, delaying or even preventing the execution of safety-critical tasks.

The need for temporal isolation is also evident in the sharing of functional services. Some Operating Systems (OSes) that focus on providing a secure execution environment for embedded and real-time computation are structured as μ -kernels. Services are implemented as user-level servers accessed by multiple clients through Inter-Process Communication (IPC). Fast IPC between processes is often *synchronous* [129] and mimics the control flow of function calls, switching from the clients to server, and back. As synchronous IPC binds potentially untrusting clients and servers, it is particularly security sensitive and has a history of challenges [195]. Early performance-driven IPC mechanisms avoided scheduler interaction on IPC, thus eliding proper client/server accounting [182]. Pairing temporal isolation facilities based around limited thread execution budgets with synchronous IPC opened systems to budget attacks in which clients attempt to expend a server's budget to prevent it servicing other clients [137, 202]. Additional improvements to the temporal properties of synchronous IPC have included priority-order execution of clients [137] and adding priority [202, 201] and budget [202, 137] inheritance.

As such, next, we introduce a new type of denial-of-service (DoS) attack that we call **Manipulative Interference Attacks (MIA)**. While DoS attacks

on inter-component messaging are not new [131, 195], MIA defines a novel type of attack in which a compromised component *manipulates* another component into delaying a third, victim component. In particular, an untrusted, malicious component creates unexpectedly large amounts of processing for a trusted, high-priority component in the system. Because the trusted component executes *on behalf of* the compromised component, this higher-priority component may unknowingly delay a co-resident victim task. When the victim task is a hard real-time task, MIA can cause devastating, critical system failures.

Because the types of interactions that lead to MIA are potentially complex and easily overlooked, later in Chapter VIII, we present an analysis framework to automatically identify instances of MIA in a configured system. Specifically, we propose a hybrid approach that first leverages static analysis to identify software components with influenceable execution times, and second, automatically generates a system-wide model to determine which compromised protection domains can manipulate the influenceable components and trigger MIA. We implement our static analysis as an LLVM compiler pass and leverage the Label Transition Set Analyzer (LTSA)—a formal system model capable of goal-conflict analysis using linear temporal logic (LTL)—to sort through complex system behavior and identify any interactions that lead to MIA. However, because LTL analysis typically requires expensive, technical expertise, we provide a tool that *automatically* generates the required system model using widely available system build artifacts. In this way, our hybrid approach can avoid typical pitfalls of static and LTL analysis, and offer a practical compile-time tool for mixed-criticality systems. Interested readers can find our analysis tools available online¹.

¹<https://github.com/smergendahl/manipulative-interference-attacks>

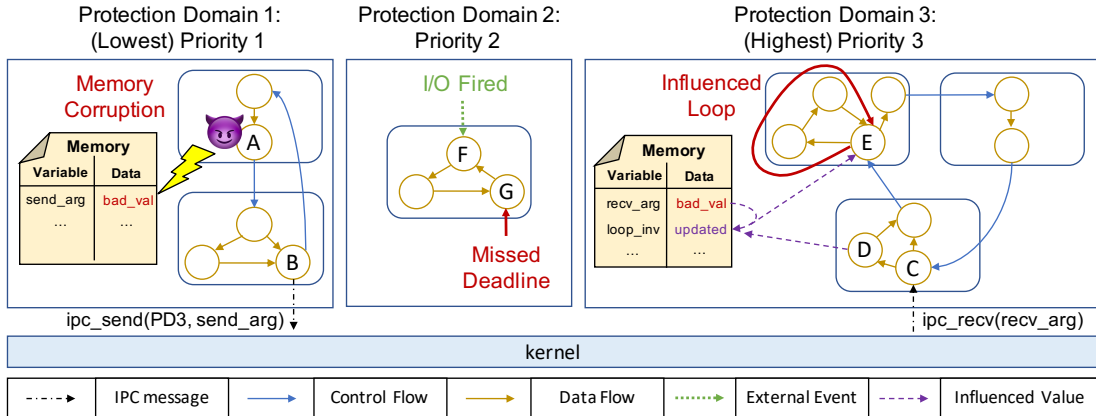


Figure 19. Manipulative Interference Attacks (MIA) leverage a corrupt, low-criticality component to influence a higher priority component to cause interference on its behalf.

6.1 Manipulative Interference

In this section, we introduce a new type of attack that we call Manipulative Interference Attacks (MIA), and describe the primitives required to launch such an attack. Namely, we discuss how cycles in high-priority software components can be influenced and/or triggered by other corrupt software components in the system. In particular, MIA occurs when a corrupt software component maliciously increases the number of times a cycle is taken as a means to cause unexpectedly long processing at a high-priority component such that another critical component in the system exhibits long delays.

6.1.1 Overview of MIA.

We define Manipulative Interference Attacks (MIA) as a scenario where an isolated, low-priority, untrusted software component leverages an approved communication pathway to *manipulate* another, higher-priority, trusted software component into executing for unexpectedly large amounts of time. Because this targeted, trusted software component maintains a higher priority than the compromised component, the attacker essentially achieves privilege escalation with respect to

the temporal properties of the system. With this higher privilege, the attacker can now delay other components with lower priority than the manipulated component. In particular, the compromised component carefully crafts IPC messages to send to the high-priority component such that the received IPC data influences a loop invariant in the IPC receiver. Because IPC in a system must follow the Immediate Priority Ceiling Protocol (I-PCP) [191, 99], the priority of the manipulated component may be high enough to delay another critical component.

Figure 19 provides an overview of MIA. In this example, three software components are separated by budget and priority. However, the lowest priority component has an IPC path to the highest priority component. Additionally, the control and data flow of each protection domain (PD) is shown. When the untrusted component, PD1, contains a memory corruption vulnerability, it can corrupt its own memory, but not the memory in a different PD. As such, at **A**, the attacker uses memory corruption to change the value of *send_arg* in its local memory. Correspondingly, when PD1 sends an IPC message to PD3 at **B**, the IPC message argument now contains the maliciously crafted *bad_val*. When PD3 receives this IPC message, it internally tracks *bad_val* in its local memory. Later in the data flow of PD3 at **D**, *loop_inv* is updated with respect to *bad_val*. Because *loop_inv* determines if the data flow cycle found at **E** should continue, PD1 has effectively *influenced* PD3 to perform an unexpectedly long number of iterations around a cycle. While PD3 continues to process this cycle, an external event fires for PD2 (in this case, an I/O event at **F**), in which PD2 has strict timing requirements to process this event at **G** within a deadline. However, because PD3 has a higher priority than PD2, the kernel will not schedule PD2 to handle this event until PD3 completes its processing on behalf of PD1. When this execution continues a

sufficient number of times, PD1 has successfully caused PD2 to miss its deadline, which may result in a critical system failure.

6.1.2 MIA Primitives. In order for MIA to occur, several primitives must exist within the system. First, there must exist an approved communication path between a compromised component and another, higher-priority component. Because of the complexity of IPC optimization with budget and priority assignment in the system, such a communication path is difficult to identify manually. Moreover, this attack is notably not a budget drain type of attack [131]: MIA does not attempt to drain the budget of a server as a means to prevent other components from accessing the service. Instead, the attack stems from a privilege escalation of the attacker’s execution context. Even if the IPC receiver is a *passive* server (*i.e.*, the receiver inherits budget from the sender), the compromised component spends its budget at a higher priority than expected.

Secondly, this approved communication path must contain a server with a *manipulable* path of execution. In particular, we define three features of a data flow cycle that can meet this requirement:

- **Influenceable Cycle:** This is a cycle in which the invariant that determines whether the cycle will exit or not is dependent on input from an external source.
- **Triggerable Cycle:** This is a cycle in which the cycle exists on a control or data flow path that directly follows after input from an external source.
- **Unbounded Cycle:** This is a cycle in which the cycle may never exit.

While this is a traditional issue typically identified in schedulability analysis, malicious input may be able to cause a cycle to unexpectedly never exit.

In particular, these features can compound on each other. For example, a cycle may be influenceable and triggerable if both the control flow to reach the cycle and the cycle itself are dependent on the received data from IPC. Similarly, an influenceable and unbounded cycle may be a cycle such that the maximum number of iterations is also influenceable by received IPC data.

6.2 Case Studies

In this section, we introduce two systems built on `seL4` and discuss how Manipulative Interference Attacks (MIA) can threaten both system designs. As a μ -kernel, `seL4` offers a trusted core for a system, but application design requires system orchestration (*e.g.*, for spatial and temporal isolation properties), so we investigate two popular choices for orchestration on `seL4`.

First, we describe the `seL4` Microkit [125], formally known as the `seL4` Core Platform, that facilitates system deployment on the `seL4-MCS` version of the kernel with mixed-criticality scheduling extensions. Second, we describe the series of build artifacts created in the DARPA Cyber Assured System Engineering (CASE) program [29] that facilitates system deployment on the original `seL4` version of the kernel.

Each version of the kernel has different benefits, and moreover, offers different ways to provide temporal isolation, so we study the impact of MIA in both situations. Specifically, the goal of this section is to discuss how MIA can arise in both system designs, regardless of the temporal isolation strategy used. Correspondingly, we create a toy example system that suffers from MIA, and discuss how neither system design can inherently prevent the attack, and instead, a system designer must check and account for MIA in either case. In particular, Figure 20 and Figure 21 show our toy example system under mixed-

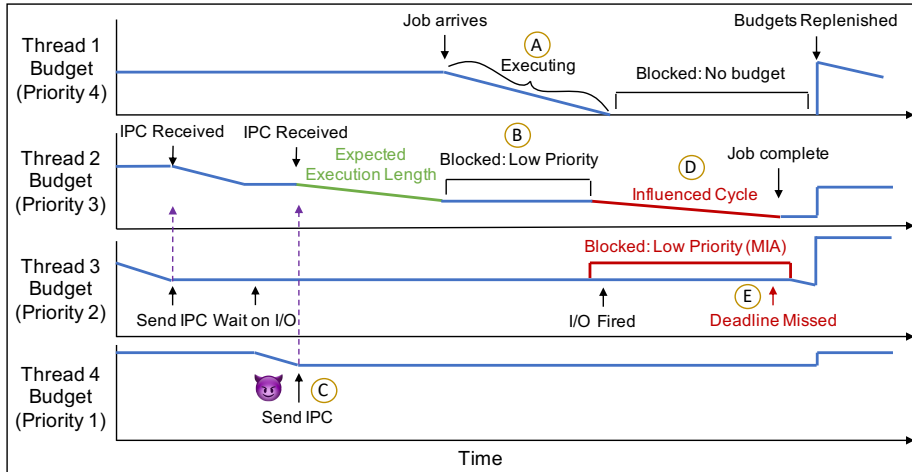


Figure 20. Mixed Criticality Scheduling used on `seL4-MCS` for Microkit offers better system utilization, but still cannot prevent MIA.

criticality scheduling and domain scheduling, respectively. Afterwards, in Section 8.2, we complementarily evaluate the performance of our automatic analysis tool. The studied toy example system can be found in Appendix B. Moreover, the automatically generated FSP model for the two systems can be found online, and alternatively, one can recreate the vulnerable system on each kernel in our experiments found with our analysis tool².

6.2.1 `seL4` Microkit. The Microkit system is an operating system framework that provides a small set of simple abstractions to ease the design and implementation of statically structured systems on the `seL4-MCS` extensions of the `seL4` kernel. In particular, Microkit helps deploy individual programs designed to be isolated with the fundamental abstraction of a protection domain. Because Microkit uses `seL4-MCS` [137], each protection domain is assigned a budget, period, priority, and series of notification and protected procedure objects. Notification objects facilitate shared memory channels, whereas protected procedure objects

²<https://github.com/smergendahl/manipulative-interference-attacks>

facilitate IPC. Importantly to MIA, systems built on Microkit deploy within protection domains scheduled with respect to priority and budget.

In Figure 20, we discuss how **seL4-MCS** schedules threads. Namely, when a job arrives for a high-priority thread, and the thread has a remaining budget, it will execute until its budget is depleted (see **A** at Thread 1 in Figure 20). Once the budget is depleted, the thread will block until **seL4-MCS** replenishes its budget. However, when a job arrives for a lower-priority thread, and a higher-priority thread is currently executing, **seL4-MCS** will not schedule the lower-priority thread until the higher-priority thread blocks (see **B** at Thread 2 in Figure 20). Following from research into MCS systems, Microkit can achieve high system utilization (*e.g.*, the system is never idle in Figure 20).

However, when a high-priority thread acts as an IPC server (in this case, Thread 2), a compromised thread has an opportunity to trigger MIA. When Thread 4 sends a maliciously crafted IPC message to Thread 2 (see **C** in Figure 20), the message could influence a cycle at the receiver that triggers unexpectedly long processing time (see **D** in Figure 20). During this processing, when Thread 2 receives an external event, **seL4-MCS** will not schedule Thread 2 due to the server's higher priority which will cause Thread 2 to miss its deadline (see **E** in Figure 20). It is important to note that under a semi-honest threat model, where maliciously crafted IPC messages are not considered, this system will operate as desired, and meet all required deadlines.

6.2.2 DARPA CASE. The DARPA CASE program [15] introduced several assurance tools to aid with the orchestration of critical systems. This program was a giant leap forward for the usability of assurance tools and developed a series of model-based systems engineering tools [134, 135, 59] for system

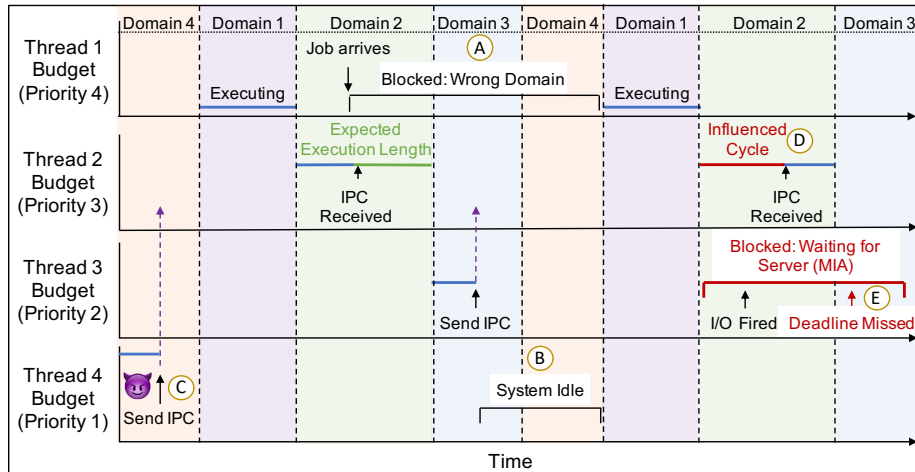


Figure 21. Domain Scheduling used on `seL4` for the DARPA CASE program leads to complex IPC issues and ultimately instances of MIA.

assurance. In particular, these tools realize cyber-resiliency requirements based on an initial Architecture Analysis and Design Language (AADL) model [80] in which components are automatically generated from formal specifications with verified system design properties, such as information flow, using formal methods. One of the tools, HAMR [29], automatically stitches the system together to deploy within the popular CAMkes `seL4` configuration environment [120].

However, unlike Microkit, these tools leverage the original `seL4` kernel. This is largely because `seL4-MCS` does not support the same level of system verification as the original kernel. Since the program aims for system verification, the original `seL4` kernel acts as a stronger trusted compute base (TCB) to guarantee system properties. But, without mixed-criticality scheduling, the CASE program instead must rely on the domain scheduler offered by `seL4` to ensure the temporal safety of the system.

In Figure 21, we discuss how domain scheduling works on the original `seL4` kernel. Like `seL4-MCS`, threads are assigned priority, notification objects, and

protected procedure objects, but are not assigned execution budgets. Instead, with the domain scheduler, threads operate in a round-robin fashion where each thread is guaranteed a timeslice to execute. Namely, a compromised thread that attempts to exceed its scheduled timeframe cannot starve other tasks. However, this strict schedule can lead to loss of system utilization. For example, if a job arrives before the relevant timeslice, the thread remains blocked until the appropriate domain is scheduled (see Figure 21 at **A**) which can lead to periods when the entire system is idle (see Figure 21 at **B**). As such, a common strategy to achieve temporal isolation on the formally verified version of the kernel is to deploy conservative timing estimates and accept the loss of system utilization.

However, similar to with `seL4-MCS`, system IPC leads to an opportunity for compromised components to trigger MIA. As before, when Thread 4 sends a maliciously crafted IPC message to Thread 2 (see **C** in Figure 21), its message can influence a cycle at the receiver that triggers an unexpectedly long processing time (see **D** in Figure 21) which ultimately leads to a missed deadline (see **E** in Figure 21). Moreover, under a semi-honest threat model, this system will again operate as desired, and meet all required deadlines.

Unfortunately, this toy example of a domain-based system highlights a subtle insight previously unidentified. With domain scheduling, shared IPC **cannot** follow the required IPC guidelines of `seL4` [99]. Namely, a shared IPC server should never block as it allows for low-priority IPC clients to be processed while a higher-priority client requires service. However, the strict nature of domain scheduling **necessitates** this behavior. We believe this is often overlooked, and system designers need an analysis tool to be able to discover these subtle problems.

CHAPTER VII

Thundering Herd Attacks (THA)

As seen in *Mergendahl, S., Jero, S., Ward, B. C., Furgala, J., Parmer, G., & Skowyra, R. (2022, May). The thundering herd: Amplifying kernel interference to attack response times. In 2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS), (pp. 95-107).*

This chapter introduces Thundering Herd Attacks on the synchronous-IPC and budget-management mechanisms in OS kernels. In the classical Thundering Herd problem [180], many threads waiting on some event are woken up but only one is actually able to proceed, causing the other threads to consume resources before blocking again. Similarly, in our Thundering Herd Attacks, a large number of normal application threads methodically use IPC facilities and carefully consume budget in a manner that causes kernel execution commensurate with the number of threads. Many of the mechanisms added to ostensibly improve temporal isolation inadvertently enable this class of attacks. Most kernels employ non-preemptive execution to control concurrency. However, when the kernel execution caused by the Thundering Herd runs non-preemptively, long stretches of non-preemptive execution interfere with and delay the activation of high-priority threads. This can threaten the high-priority thread's ability to meet deadlines.

This leaves us with what we call the *system-coordination dilemma*: if we use simple IPC mechanisms, then the system suffers from inter-client interference, thread accounting is unpredictable, and execution is unconstrained; if we counter inter-client interference by making IPC priority aware, then attacks on the kernel's priority mechanisms cause significant interference; if we counter unpredictable system accounting and lack of execution isolation by using budgets, then the

budget-accounting mechanisms cause significant interference. The techniques employed to increase the intelligence of thread coordination also lead to significant attacks. Many of these issues are quite nuanced and not immediately obvious, especially to application engineers instead of systems researchers. This chapter therefore sheds light on this fundamental scientific dilemma, and illustrates specific tradeoffs.

7.1 Traditional IPC Interference

In this section, we describe a series of well-known, synchronous-IPC-based interference issues that delay high-priority tasks that use shared services. These issues serve as the motivation for the mechanisms that are exploited by our Thundering Herd Attacks in §7.2. In particular, though the academic community has introduced these issues previously, in order to fully understand the system-coordination dilemma that drives our Thundering Herd Attacks, we provide a detailed analysis of these relevant interference issues known by the community. For each interference issue, we provide an overview of the issue, describe the issue in detail, and finally, discuss existing mitigations. Moreover, we later quantify the negative effect of these interferences in §8.2.

7.1.1 Overview. In order to create these interferences, we leverage a set of low-priority threads to cause a high-priority shared server to perform work on behalf of low-priority threads, instead of either a) working for the high-priority victim task or b) scheduling the high-priority victim task itself. We consider any such work that the server executes *on behalf of* a low-priority thread to be High-priority interference (HPI). Note that we assume that the shared server is executing at a higher priority than any client. Lastly, for each attack example, we refer the reader to Figure 22 for the notation found in each figure.















	Job Scheduled		Work for a Low-Priority Thread		Blocked on I/O		Budget Replenished
	Work on Behalf of Victim		Blocked on IPC Endpoint		Replenishment Period		Budget Spent
	Synchronous IPC to Server		I/O Fired		Replenishment Queue Sort		
	Request I/O		IPC Endpoint Queue Sort		Replenishment Processing		

Figure 22. Legend used throughout attack examples.

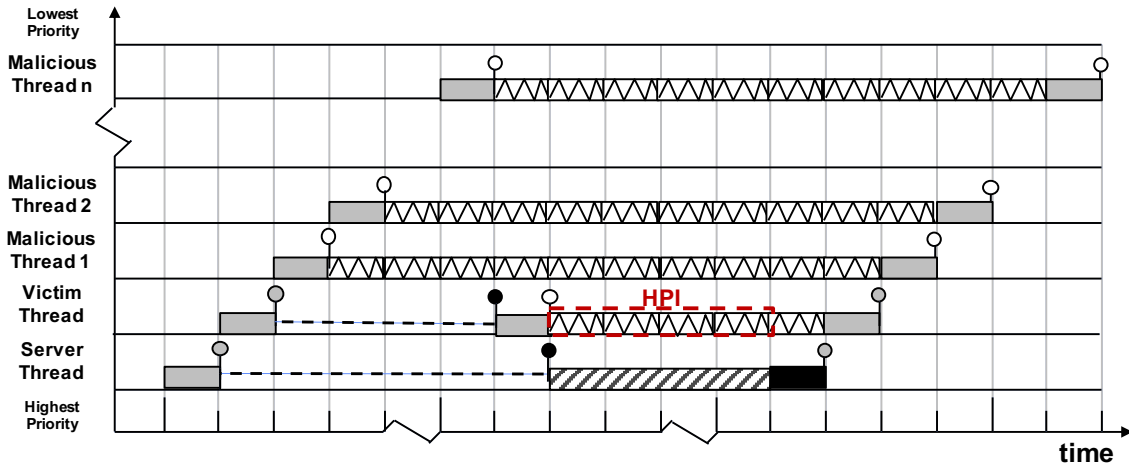


Figure 23. FIFO Endpoint Flood Interference example schedule.

7.1.2 FIFO Endpoint Flood Interference. The first HPI we describe is the FIFO Endpoint Flood Interference discussed by Liedtke et al. almost 25 years ago [131]. We depict this in Figure 23. In this HPI, a set of low-priority threads all make synchronous IPC requests to a currently blocked shared server before a high-priority victim task also makes a synchronous IPC request. If the IPC endpoint queue is a first-in, first-out (FIFO) queue, as in `seL4` and other common μ -kernels, then this forces the shared server to receive and process the (low-priority) requests from the low-priority threads before the request from the high-priority task. This period in which the server executes on behalf of the low-priority threads instead of the higher-priority client is HPI.

We note that a server executing on behalf of a client shares some commonalities, at least analytically, with executing a critical section within a lock. In the seminal work on the Priority Ceiling Protocol (PCP) [191], it was identified that self-suspensions within critical sections invalidate the PCP analysis. This interference effectively exploits this same phenomenon, and reinforces that neither critical sections nor servers executing on behalf of a blocked client should suspend. While `seL4` has never, to the best of our knowledge, documented that servers should never suspend, it has been part of the informal developer knowledge in the `seL4` community for some time [99]. Additionally, we note that self suspensions are difficult to analyze, and can have surprising consequences, as show by Chen et al. [49] who demonstrated that many papers on the topic going back as early as 1994 are incorrect due to common misconceptions about suspensions.

Interference Mitigations. In order for this HPI to occur, there must be a moment when both the victim task and the shared server are blocked. Thus, one potential mitigation is to not allow either the victim task or the shared server to block. For the victim task this is often an unreasonable request, as it may be periodically activated or block waiting on some long-duration operation. For the shared server, this is more reasonable, but may still be problematic, especially in the case of a server for an I/O device, which may need to occasionally block waiting for the I/O device to catch up. This momentary suspension of the shared server, however, allows the lower-priority threads the opportunity to run and queue on the server’s endpoint, resulting in this interference.

Instead, in order to mitigate this interference, `seL4-MCS` introduces priority sorting for its IPC endpoint queues. Because the victim task has higher priority

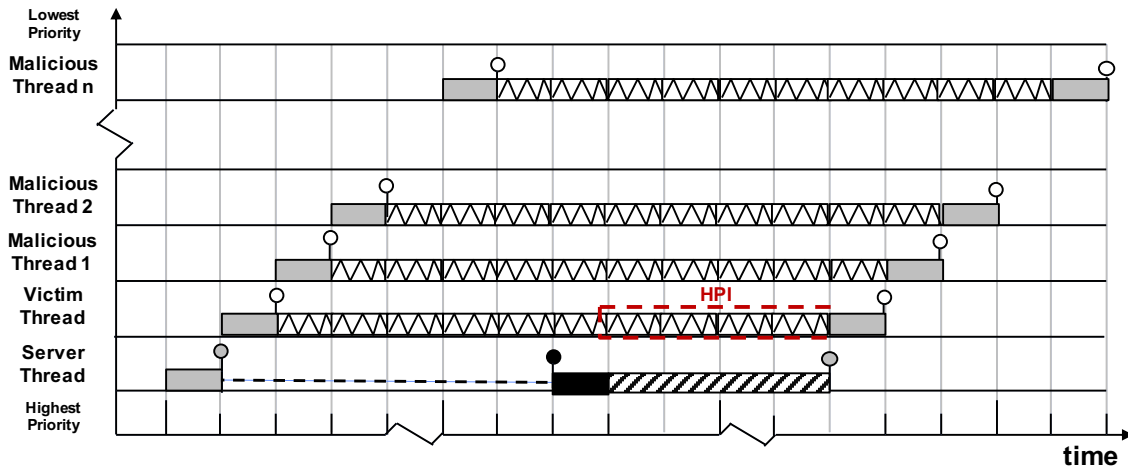


Figure 24. Priority Ceiling Processing Interference example schedule.

than the malicious threads, it will sort to the front of the IPC endpoint queue, and eliminate this particular set of HPI.

7.1.3 Priority Ceiling Processing Interference. We now describe Priority Ceiling Processing Interference, depicted in Figure 24. This interference again leverages the results from the Priority Ceiling Protocol (PCP) [191] work. However, in this case, the IPC endpoint queue does not have to be FIFO; it may be priority-sorted or arbitrarily ordered. Without loss of generality, we assume a priority-sorted endpoint queue, like the one provided by `seL4-MCS`, as that protects against the previous FIFO Endpoint Flood Interference. Similar to the previous FIFO Endpoint Flood Interference, the victim is a high-priority task that performs a synchronous IPC request to a shared server and a set of low-priority threads to introduce interference. Again, the shared server executes at a higher priority than all clients and we assume that it blocks, possibly due to handling I/O.

When the shared server is blocked, the victim task can run and perform its IPC request to the server, resulting in it being queued on the endpoint. Now, all the low-priority threads run and make IPC requests and queue on the endpoint.

Note that whatever order these clients run, the victim task will be sorted to the front of the priority-ordered queue. When the server wakes up, it will process the request from the victim task first. However, once that request is completed, the server will go on to process the requests from the low-priority threads instead of allowing the high-priority victim thread to be scheduled, because the shared server runs at a higher priority than all its clients. Thus, this period between when the shared server completes processing the request from the victim thread and when that thread is actually scheduled is HPI.

Interference Mitigations. Previous studies have investigated the issues with priority ceiling processing [191], and the general recommendation is that, in order to prevent this problem, shared-server threads should be designed to never block. When high-priority servers do not suspend, the low-priority threads are handled as they arrive and there is no opportunity for a large queue to form. Although `seL4-MCS` does not contain an explicit defense against this interference, its design philosophy follows this recommendation that servers never block, as evident in its sporadic-server design [210, 99].

7.1.4 Budget Drain Interference. Finally, we describe Budget Drain Interference, which enables low-priority threads to delay a high-priority victim task by making repeated requests to a shared server. In particular, as discussed by Shapiro [195], low-priority threads can make repeated requests to a shared server to drain its budget. Then, when the victim task makes a request to the shared server, it must wait for the server’s budget to be replenished before its request is processed. This requires an opportunity for the malicious threads to make many requests to the shared server, but can have very large impacts. See Figure 25 for more details of this HPI. The exact HPI that results scales with

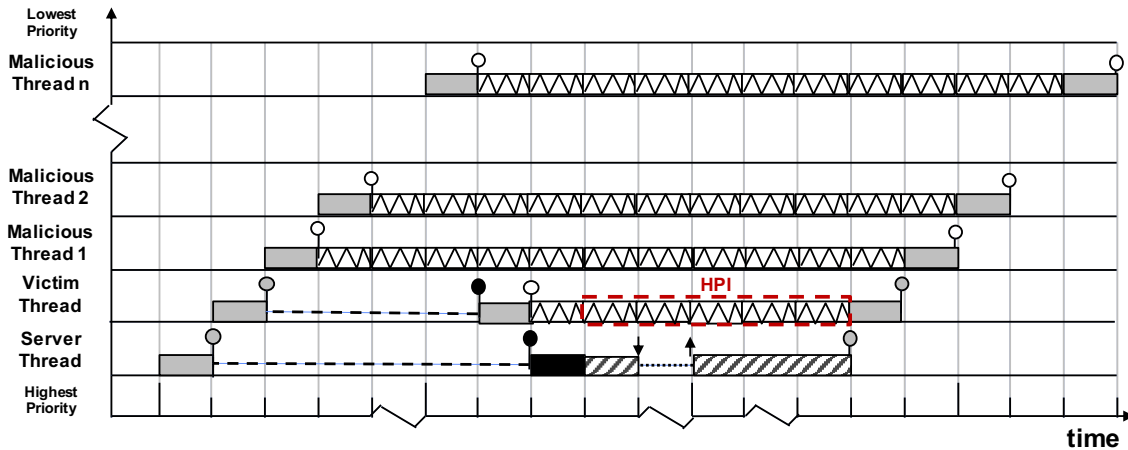


Figure 25. Budget Drain Interference example schedule.

the replenishment period of the server’s budget. We refer to prior work on this interference for a more exact measure of its impact [195].

Interference Mitigations. Mitigations to this HPI include sporadic servers as well as budget inheritance/donation systems that allow servers to use a client’s budget for processing their requests. `seL4-MCS` includes support for passive threads, requiring budget donation, as well as support for sporadic servers that can track a bounded number of replenishments [210].

7.1.5 Relationship to the System-Coordination Dilemma. As discussed, a variety of mitigations exist to prevent these interferences. However, in many cases, particularly priority-sorting IPC endpoint queues, providing sporadic servers, and providing budget inheritance, these mitigations result in additional complexity in μ -kernel implementations. Many μ -kernels, including `seL4-MCS`, implement these additional features to protect against these kinds of HPI.

We show next that this additional complexity can be attacked, resulting in our Thundering Herd Attacks. Thus, the system-coordination dilemma requires

that a system designer must choose between HPI from these traditional interference issues, and HPI from Thundering Herd Attacks.

7.2 Thundering Herd Attacks

In this section, we introduce and describe our Thundering Herd Attacks. These attacks target the mitigations deployed against the traditional IPC interference issues (discussed in §7.1) to introduce additional non-preemptive kernel processing into the system with the aim of causing schedule overruns for high-priority tasks. We first provide an overview of these attacks and then discuss each of them in detail.

7.2.1 Overview. We refer to these attacks as Thundering Herd Attacks because they use many attacker-controlled threads to perform IPC and consume budget in ways that force the kernel mechanisms necessary for handling budgets and prioritized queues to do large quantities of non-preemptive work. This kernel-level work supersedes execution of all user-space threads, and, when combined with a non-preemptable kernel, like `seL4` (see Chapter II), these attacks can even supersede interrupts.

The kernel mechanisms we exploit are necessary to mitigate the previously discussed IPC interference issues. In particular, we leverage priority-sorted IPC endpoint queues and sorted queues of threads waiting for budget replenishment. This forces a system-coordination dilemma on the user: either deal with the client-interference issues illustrated in the traditional IPC interference issues or deploy defense mechanisms against the traditional IPC interference issues and be exposed to our Thundering Herd Attacks.

For each Thundering Herd attack, the attacker leverages a set of low-priority threads. However, unlike the traditional IPC interference issues, a shared server

is no longer required. By manipulating its own threads, the attacker can cause extended non-preemptive kernel processing that will delay any thread or interrupt in the system. Similar to §7.1, we consider this kernel-level processing *on behalf* of a low-priority thread, when the high-priority victim thread is ready, to be HPI. We emphasize that this is possible even when the attacker threads and the victim thread are *completely disjoint* with no shared servers or resources.

7.2.2 Endpoint Queue Sorting Attack. The first attack we describe is the Endpoint Queue Sorting Attack. This attack takes advantage of the fact that μ -kernels, such as **seL4-MCS**, often priority sort the threads blocked on synchronous IPC endpoints. Recall from §7.1.2 that this is an essential technique to mitigate the FIFO Endpoint Flood Interference. **seL4-MCS** uses a simple linked list to implement this priority queue, making it $O(n)$ to insert a new thread in sorted order. Threads are added to the back of the queue and sorted forward in increasing priority. The only requirement for an attacker to launch this attack is the ability to either have or create both a single endpoint and many schedulable threads at three different priorities. This is consistent with our threat model as a malicious thread may spawn other threads. We reiterate that both the shared server and its clients in this attack can be attacker controlled tasks of low priority and disjoint from the rest of the benign system.

The Endpoint Queue Sorting Attack works as shown in Figure 26. First, the attacker finds or creates an IPC endpoint and creates a server thread to listen on this endpoint. This thread should have the lowest priority. Then the attacker iteratively starts a series of attack threads with the middle priority and causes them to perform a synchronous IPC on the IPC endpoint. This generates a long queue of threads on the endpoint, but without any sorting as they are added. The

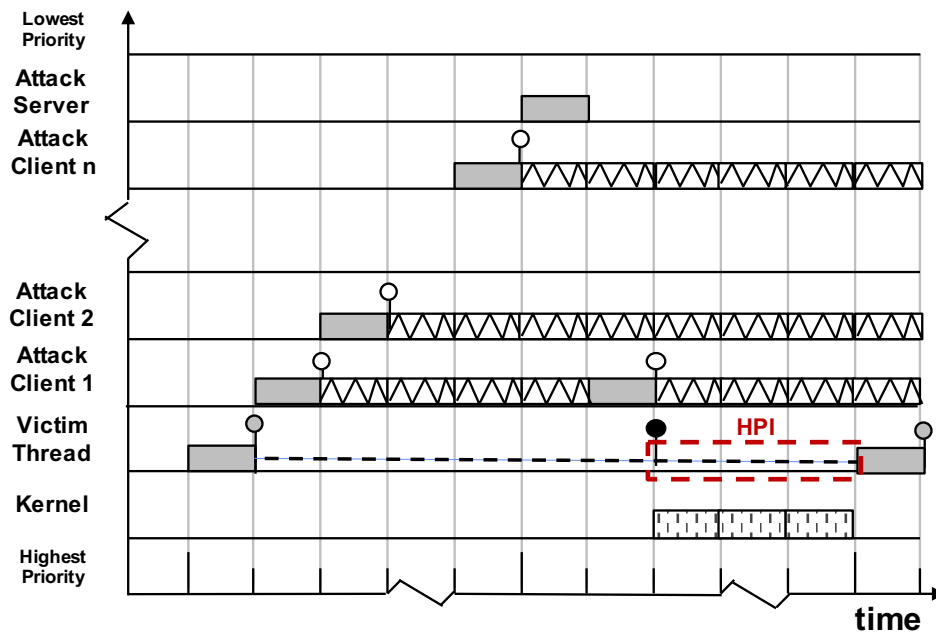


Figure 26. Endpoint Queue Sorting Attack example schedule.

attacker then creates another thread with its higher priority that will repeatedly perform a synchronous IPC on the IPC endpoint. When this occurs, the kernel will sort this attacker thread to the front of the endpoint queue. At this point, the attacker's server thread runs, handling the attacker's higher-priority thread and allowing the attacker's higher-priority thread to run again. The higher-priority thread will immediately make another IPC call which will force it to be sorted to the front of the endpoint queue again. When the high-priority victim thread becomes runnable, often as a result of an interrupt, it will be delayed due to the kernel non-preemptively sorting the endpoint queue, resulting in HPI.

Attack Mitigations. Because the kernel uses a linked list for its priority queue, insertion is $O(n)$. One way to mitigate this attack would be to use a different data structure with smaller insertion complexity for the endpoint priority queues. For example, a red-black tree would offer $O(\log(n))$ worst-case performance

against our attack, which could reduce the impact of the attack, but not eliminate it altogether (see §8.2).

Another option would be for the kernel to maintain a separate queue for each priority level on each IPC endpoint (similar to the `plist` data structure found in Linux [13]). If there were a separate queue for each priority, there would be no need to sort the queues, which would eliminate the vector for this attack. With no queues to sort, insertion would again be $O(1)$, eliminating the HPI caused by the Endpoint Queue Sorting Attack at the cost of a small amount of memory.

7.2.3 Replenishment Queue Sorting Attack. Next we introduce the Replenishment Queue Sorting Attack. This attack leverages the fact that when a thread expends its budget, it is placed on a queue of pending replenishments. Replenishments are essential to budgets which, as discussed in Chapter II, are critical to constrain the execution of low-assurance threads and prevent their interference with other threads.

`seL4-MCS` implements replenishment as set of replenishment queues, one for each core, with each queue sorted based on soonest period expiration (*i.e.*, the thread that will be replenished soonest). In `seL4-MCS` these are, again, implemented as priority-ordered linked lists, making insertion of new threads $O(n)$. As a result, we can linearly increase the computation required to enqueue a thread by increasing the length of this queue. The only requirements for an attacker are the ability to either have or create many schedulable threads at the same priority and to have some control over their budgets and periods. Note that this attack does not rely on IPC.

The Replenishment Queue Sorting Attack works as shown in Figure 27. The attacker creates and runs a large number of attack threads that execute infinite

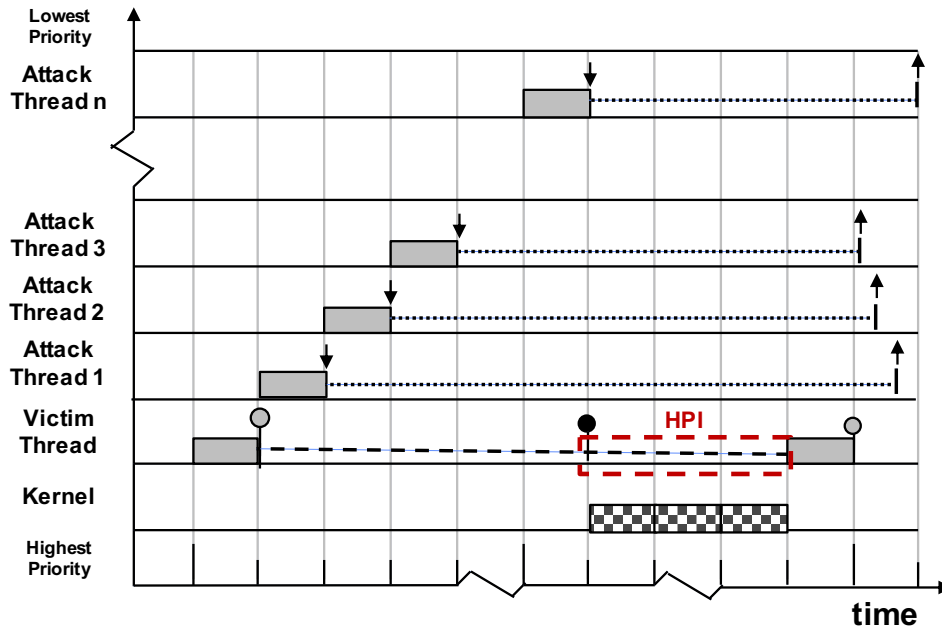


Figure 27. Replenishment Queue Sorting Attack example schedule.

loops to spend their budgets and be queued in the replenishment queue. The periods of these threads, however, are carefully chosen. Since `seL4-MCS` inserts a new thread at the front of the replenishment queue and then sorts it towards the back, inserting threads with longer periods results in more sorting while threads with shorter periods are $O(1)$. As a result, each of the attack threads should have a shorter period than the one before it, thus generating a large replenishment queue without sorting. Finally, to trigger the attack, the attacker creates and executes a thread with a period longer than any other thread. As a result, once this thread exceeds its budget, it will be sorted all the way to the back of the replenishment queue. When the high-priority victim thread becomes runnable, often as a result of an interrupt, it will be delayed due to the kernel non-preemptively sorting the replenishment queue, resulting in HPI.

The specifics of exactly how the replenishment queue is sorted are immaterial to this attack. For example, if the kernel instead inserted threads at the

tail of the replenishment queue and sorted the soonest-to-be-refilled replenishments to the front, the attacker would initialize periods in the opposite manner: the first threads would have increasingly larger periods, with the last attack thread having the shortest period.

Attack Mitigations. As with the Endpoint Queue Sorting Attack, potential mitigations include data structures with lower insertion complexity (*e.g.*, red-black trees). However, unlike the Endpoint Queue Sorting Attack, because the Replenishment Queue Sorting Attack does not target a queue sorted by priority or leverage threads of different priorities, separate queues per priority will not affect the outcome of this attack. In particular, the replenishment queue is sorted based on time-until-replenishment, not the priority of the respective thread.

7.2.4 Replenishment Wakeup Processing Attack. Finally, we introduce the Replenishment Wakeup Processing Attack. Similar to the previous attack, this attack takes advantage of the queue of pending budget replenishments maintained by `seL4-MCS`. However, rather than focusing on the sorting that occurs when a new thread is added to the replenishment queue, this attack focuses on the processing that occurs when threads are ready to be replenished. In particular, this attack attempts to cause a large number of attacker-controlled threads to be replenished at the same moment, directly prior to the execution of the high-priority victim thread. This will cause HPI as the low-priority attacker threads are replenished instead of running the high-priority victim thread. The only requirements for an attacker are the ability to either have or create many schedulable threads and to have some control over their budgets and periods. This could be realized by either having a copy of the `SchedControl` capability,

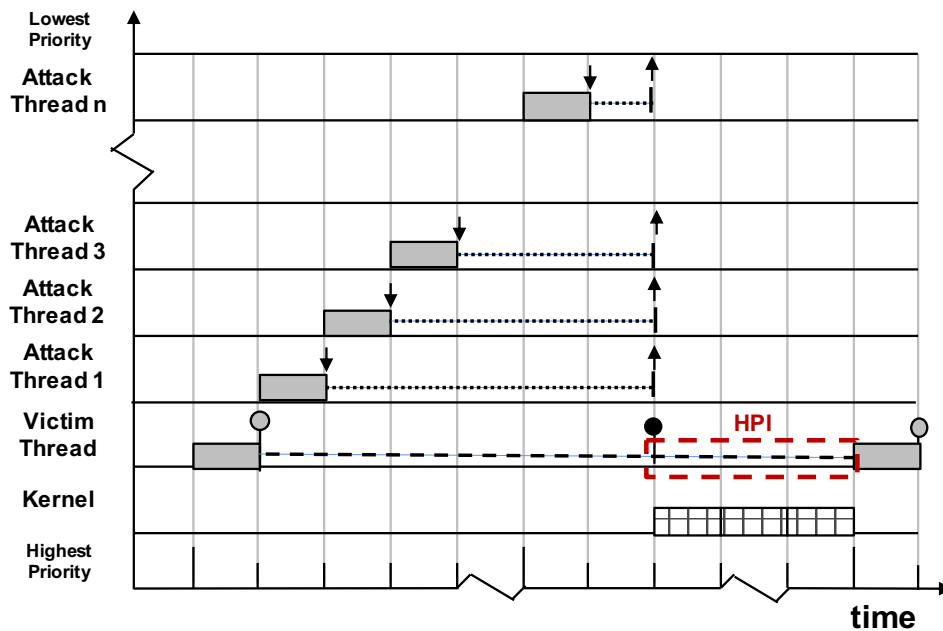


Figure 28. Replenishment Wakeup Processing Attack example schedule.

or issuing a request that is serviced by an admission-control server to populate the SchedContext budget and period.

The Replenishment Wakeup Processing Attack works as shown in Figure 28. The attacker creates and runs a large number of attack threads. These threads execute infinite loops to spend their budgets and be queued in the replenishment queue. However, their budgets and periods need to be carefully chosen. In particular, the attacker must choose the budget and period of each thread such that all threads will be replenished at the same moment (or at least within the same timer tick). This moment when all the threads need to be replenished is chosen to be just before the high-priority victim thread is ready. As a result, when the high-priority victim thread becomes runnable, often as a result of an interrupt, it will be delayed due to the kernel non-preemptively processing the replenishment queue, resulting in HPI.

Attack Mitigations. The Replenishment Wakeup Processing Attack appears to illustrate a fundamental instance of the system-coordination dilemma. Kernel processing of replenishments is fundamental to proper enforcement of temporal budgets. Moreover, as we will see in §8.2, if a system designer deploys a logarithmic data structure to alleviate the Replenishment Queue Sorting Attack, such a logarithmic data structure will *increase* the effect of the Replenishment Wakeup Processing Attack. Namely, while the logarithmic data structure decreases insert processing from $O(n)$ to $O(\log(n))$, it also *increases* deletion from $O(1)$ to $O(\log(n))$.

Lastly, because the kernel processes all available replenishments at once, a separate queue per priority will not mitigate the Replenishment Wakeup Attack alone. In particular, even if the replenishments were in different queues by priority, the current design of the kernel requires it to process all replenishments to avoid replenishment starvation (including replenishments associated with low-priority threads). However, while separate queues per priority alone would not mitigate the attack, such an implementation could enable a new priority-aware replenishment processing scheme in the kernel. Unfortunately, such a design would require significant structural changes to the kernel (*e.g.*, a significant redesign of most aspects of the system including the IPC path). Nonetheless, such a redesign could prove fruitful, with the caveat of a careful examination of the trade-offs of such an implementation.

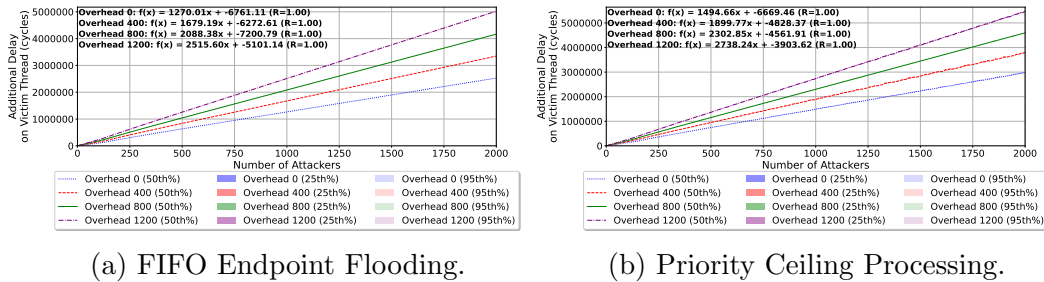
7.3 Evaluation

In this section, we implement and quantify both the synchronous-IPC-based interference issues that delay high-priority tasks using shared services (§7.1) and our Thundering Herd attacks that introduce additional non-preemptive

kernel processing into the system (§7.2). We then empirically evaluate a possible mitigation strategy for our Thundering Herd Attacks that modifies the `seL4-MCS` kernel to use red-black trees to priority sort both the IPC endpoint queues and replenishment queues that the kernel maintains with logarithmic complexity. Finally, we analyze the impact of another mitigation strategy that uses a queue-per-priority data structure.

7.3.1 Experimental Setup. We implement each studied attack on `seL4` (or `seL4-MCS`) version 12.0.0 and quantify their impact using the popular Zynq-7000 XC7Z020 SoC, which includes a dual-core Arm Cortex-A9 processor running at 667 MHz and a Xilinx FPGA. We use only a single core for this evaluation and do not use the FPGA at all. We use `gcc` version 8.3.0 (Debian 8.3.0-2) for `arm-linux-gnueabi-gcc` to compile `seL4` and our test code. We use the on-chip performance counters to determine overheads. Unless otherwise noted, all results are computed from 100 iterations. Because of the high accuracy of our testbed and test suites standard deviations are frequently very small and may not be visible in all graphs.

7.3.2 Traditional IPC Interference Results. We implemented FIFO Endpoint Flood Interference in our testbed using `seL4` and minimal client and server applications and report its impact in Figure 29a. We observe that significant impacts are possible, with a thousand low-priority threads causing over 1 million cycles (1.5ms) of interference. Further, as the number of low-priority threads increases we see that HPI increases linearly. We additionally perform experiments where the server performs different amounts of work for each request, which we refer to as overhead. For example, with overhead 400, the server performs approximately 400 cycles of work for each request, while with overhead 1200 the



(a) FIFO Endpoint Flooding.

(b) Priority Ceiling Processing.

Figure 29. HPI from the traditional IPC interference issues with different numbers of threads and quantities of work for each request.

server performs approximately 1200 cycles of work for each request. Thus, the imposed delay also increases linearly with respect to the amount of work the server performs for each request.

We also implemented Priority Ceiling Processing Interference in our testbed using `seL4-MCS`, which priority sorts the endpoint queue, and report the impact of this HPI in Figure 29b. Much like the prior interference issue, we observe that HPI scales linearly with both the number of low-priority threads and the amount of work the server performs on behalf of each client. Compared to the FIFO Endpoint Flood Interference, we observe that this interference is slightly more powerful in terms of the HPI from each low-priority thread.

7.3.3 Thundering Herd Attack Results.

Next, we empirically evaluate the impact of the Endpoint Queue Sorting Attack on `seL4-MCS` in Figure 30a. We observe that significant impacts are possible, with 1,000 attack threads introducing about 100,000 cycles (150 μ s) of HPI. The introduced HPI is also linear with the number of attack threads. It is also interesting to note that, unlike the IPC interference issues discussed previously, it takes a number of attacker threads (~ 125) before an impact is noticeable. This is likely due to a combination of cache effects, where for very small numbers of threads everything

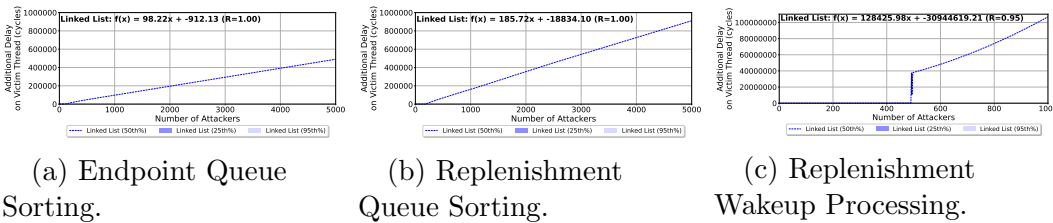


Figure 30. HPI from the Thundering Herd attacks with a linked-list kernel data structure.

may be in L2, and the difficulty of targeting the attack when it introduces only a small delay.

Similarly, we empirically evaluate the impact of the Replenishment Queue Sorting Attack on `seL4-MCS` in Figure 30b. We see that the impact of this attack grows linearly with the number of attacker threads and that it requires a number of attack threads before the attack becomes noticeable. However, this attack has a larger impact for each individual attacker thread of approximately 185 cycles.

Lastly, we implemented the Replenishment Wakeup Processing Attack on `seL4-MCS` and demonstrate its impact in Figure 30c. This attack is more challenging than the others to properly orchestrate since we must predict execution times very accurately so that all attacker threads will be replenished at the same instant as the victim thread becomes runnable. For a small number of attacker threads, we are unable to precisely align the attack with the victim thread, as seen on the left of Figure 30c. However, once we get above about 450 attackers, the impact of the attack becomes large enough that we can reliably impact the victim. While this attack is harder to execute, it is also much more powerful, as each thread causes about 100,000 cycles of HPI. The impact also grows linearly with the number of threads, meaning that 1,000 attack threads can introduce approximately 100 million cycles (150ms) of interference.

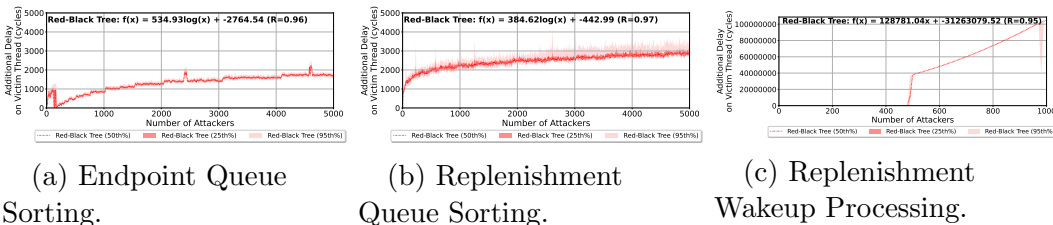


Figure 31. HPI from the Thundering Herd attacks with a red-black tree kernel data structure.

7.3.4 Red-Black Tree Mitigation Results. We now consider one possible mitigation for our Thundering Herd Attacks: the use of red-black trees instead of linked lists for the sorted queues in the kernel. Because a red-black tree maintains a $O(\log(n))$ insert complexity, this would seem to be an attractive means to mitigate the Endpoint Queue Sorting and Replenishment Queue Sorting Attacks. As a result, we replaced the linked list implementation for kernel metadata processing in `sel4-MCS` with a red-black tree and repeated the same tests from §7.3.3.

Indeed, in Figure 31a and Fig. 31b, we can see that the red-black tree limits the Endpoint Queue Sorting Attack and Replenishment Queue Sorting Attack respectively to a logarithmic increase in HPI based on the number of attack threads. However, these data structures do not mitigate the attack completely, as an increase of up to 2,000 cycles and 3,000 cycles respectively is still demonstrated.

In contrast, we see from Figure 31c that using a red-black tree still results in similar impacts for the Replenishment Wakeup Processing Attack. Just as for the linked-list version (Fig. 30c) we see the attack become effective at about 450 attackers and cause about 100,000 cycles of HPI per attacker above that point. If we investigate the differences between the linked list and red-black tree versions, we find that the red-black tree *increases* the impact of the attack by around

1,500 cycles. We posit this effect comes from the fact that while the red-black tree decreases insert complexity from $O(n)$ to $O(\log(n))$, it also increases removal complexity from $O(1)$ to $O(\log(n))$, but we leave a detailed investigation for future work.

7.3.5 Queue-per-priority Mitigation Analysis. We consider one final mitigation strategy for our Thundering Herd Attacks: the use of a data structure with separate queues per priority level, similar to the `plist` structure found in Linux [13]. A common implementation would be an array of queues, one for each priority level. `seL4-MCS` has 256 priority levels (0-255), so such a data structure would have 256 queues.

For our Endpoint Queue Sorting Attack, this data structure would completely eliminate the attack. This is because if there were a separate queue for each priority, there would be no need to sort the queues. With no queues to sort, insertion would be $O(1)$. The only cost is a small amount of additional memory.

For the Replenishment Queue Sorting Attack, a data structure with separate queues for each priority level would have no impact. This is because the queue being attacked is *not* sorted by priority, but rather by time of next replenishment. As a result, a data structure with separate queues per priority would merely move the sorting into the per-priority queues rather than eliminate it. Further, an attacker would still be able to create a large number of threads with the same priority to cause extensive sorting in these per-priority queues.

Finally, for the Replenishment Wakeup Processing Attack, this kind of data structure could provide benefits, but only with extensive kernel-wide modifications. In particular, it would be desirable to only process wakeups for processes of greater or equal priority to the currently running process. This would prevent wakeups

for lower-priority processes from causing HPI and those lower-priority processes would not be run immediately anyway. While this is a promising idea, it requires extensive redesign of `seL4-MCS`, which we leave for future work, to check for pending wakeups on every context switch, including along the IPC hot path.

7.4 Implications for System Provisioning

A promising mitigation to these attacks is to explicitly provision a system to be resilient to the HPI that attackers can induce using these attack techniques. Specifically, if a system is provisioned with sufficient slack time, it is resilient to attacker-induced HPI. Next we demonstrate through simple schedulability experiments the utilization loss associated with such provisioning. Systems that are not provisioned with such slack are vulnerable to Thundering Herd Attacks, which can potentially maliciously cause timing violations.

7.4.1 Experimental Design. For our schedulability experiments, we consider a simple sporadic task system scheduled with fixed-priority scheduling, as is implemented in `seL4`. In particular, we consider a task system as a system of n tasks $\Gamma = \{\tau_1, \dots, \tau_n\}$. Each task $\tau_i = (C_i, T_i)$ is a recurring sequence of jobs, each of which has an execution time of C_i , which are released sporadically with a minimum separation of T_i time units. We assume implicit deadlines, and thus the deadline of each job is T_i time from its release. The utilization of τ_i is defined as $u_i = C_i/T_i$, and the utilization of the task system $U(\Gamma) = \sum_{\tau_i \in \Gamma} u_i$.

We randomly generated task systems using commonly used task-system distributions. We considered all combinations of task-utilization and period distributions used in Brandenburg’s dissertation [38] and codified in the associated Schedcat library [39]. This resulted in 54 unique task-utilization and period combinations. For each such combination, we varied $U(\Gamma) \in \{0.01, 0.02, \dots, 1.0\}$,

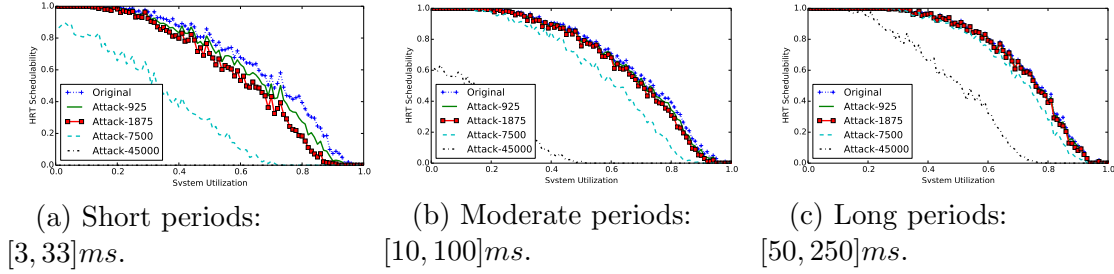


Figure 32. Sample schedulability graphs. All distributions uniformly distributed. Medium per-task utilizations in $[0.1, 0.4]$.

which resulted in 54 unique schedulability graphs. Three sample graphs from this larger study are depicted in Figure 32.

We evaluated schedulability using the classic response-time analysis. We model the High-priority interference induced by our attack techniques as the highest-priority task in the system, $\tau_{\text{High-priority interference}}$. The period of $\tau_{\text{High-priority interference}}$ is defined to be hyperperiod, or the least-common multiple of all task periods. This represents the possibility that an attacker can induce HPI at any time, but cannot necessarily trigger such attacks multiple times to trigger compounding effects. The execution time $\tau_{\text{High-priority interference}}$ is chosen to reflect the amount of HPI that an attacker is assumed to be able to induce. Based on our previous results, we chose to evaluate $C_{\text{High-priority interference}} \in \{0.975, 1.875, 7.5, 45\}ms$. These values are chosen from the FIFO Endpoint Flood Attack hpi with 500 and 1000 attackers ($0.975ms$, 1.875 , resp.), and the Replenishment Wakeup Attack with 500 and 1000 attackers ($7.5ms$, $45ms$, resp.), as representative values to demonstrate the potential range of consequences of Thundering Herd Attacks.

An admission-control test that considers this interference derives this overhead from the number of threads in the system paired with the results in §8.2.

7.4.2 Results. These results demonstrate that, for task systems with shorter periods and hence tighter timing constraints, the HPI that these attacks can induce can significantly impact schedulability. Indeed, the short periods considered (uniformly distributed among $[3, 33]ms$) are less than the HPI from the Replenishment Wakeup Processing Attack ($45ms$), and therefore no such tasks can be guaranteed to meet their deadlines if such attacks are possible. Even if the attacker is assumed to only be able to spawn fewer threads and therefore induce less HPI, there can still be significant utilization loss.

In addition to enabling a proper admission-control test to mitigate the Thundering Herd Attacks, these results can also be interpreted as demonstrating what systems are *vulnerable* to Thundering Herd Attacks. Any task system that is deemed schedulable without considering HPI but is not schedulable with HPI is vulnerable to an attack that can trigger a deadline overrun. This is true even in budgeted systems where temporal interference from one task to another is controlled, as the HPI can be induced by low-priority tasks with minimal budgets.

These evaluations demonstrate the schedulability-related implications of the system-coordination dilemma in light of our Thundering Herd Attacks. We next discuss alternative means of mitigating these attacks and resolving this dilemma.

7.5 Discussion and Related Work

Synchronous IPC mechanisms have been studied as an attack vector [195] as synchronous IPC ties the execution of clients to a server’s computation and introduces inter-client interference when execution is serialized through server threads. We have demonstrated in §7.2 and §8.2 that the kernel mechanisms for maintaining the necessary IPC and execution metadata – which track the state of communication and properly schedule threads – can themselves become attack

targets. A common characteristic of each attack is that kernel processing on this metadata is not constant-time, and can thus be targeted by attackers. These attacks are enabled by the non-preemptive nature of the kernel, made worse by multicore systems that prevent parallel kernel execution using a lock.

While we study attacks on `seL4-MCS`, the core challenges generalize to other systems with non-preemptive processing of IPC and budget-management data structures. For example, μ -kernels commonly use non-preemptive spin-locks to protect data-structures and disable interrupts during timer processing. Thus these attacks might be more broadly impactful.

In the rest of this section we discuss implications of these findings for μ -kernel design and highlight related work.

7.5.1 Implications for μ -Kernel Design. §7.4 discusses how to integrate the measured overheads from the various attacks into schedulability analysis. This test can be integrated into the system’s admission controller. Unfortunately, we demonstrate that doing so can cause significant utilization loss. Here we qualitatively suggest and assess a number of alternate kernel-design options.

7.5.2 Track metadata with $O(\log(n))$ data-structures. IPC endpoint wait queues are tracked with linked lists and are sorted in `seL4-MCS`. Replacing these with balanced binary trees will asymptotically decrease the cost of adding threads to the queue, which would comparably decrease the amount of non-preemptible execution, as shown in §7.3.4. This does *not* defeat the attacks, but does lessen their impact. This effect has been observed when using $O(\log(n))$ data structures for other potentially contended kernel data structures such as timers [164] and futexes [233].

Logarithmic structures can also be used to track replenishments, which would similarly decrease the asymptotic overheads for replenishment attacks. Unfortunately, this is not a clear benefit. Though adding threads to the replenishment queue on budget depletion would benefit, the overhead for processing replenishments during `seL4`'s timer will *increase*, as shown in §7.3.4. Each of the n replenishments that require processing will increase from constant to logarithmic overhead. Our results suggest this may be a reasonable trade off.

7.5.3 Track wait queues with queue-per-priority data

structures. Instead of using a balanced tree, a constant-time structure common in fixed-priority scheduling implementations could be used for IPC endpoint wait queues. This is an array with one entry per priority, each containing a list of waiting threads. A bitmap (or nested bitmaps) are used to track which priorities have waiting threads. The constant-time overhead of this approach would defeat the attacks on kernel IPC endpoint queues. The primary cost of this approach would be a minor increase in IPC endpoint memory consumption commensurate with the number of potential priorities.

7.5.4 Expanded use of preemption points.

Preemption points are explicit closures that capture a kernel in-progress operation and allow interrupts to be processed, later continuing kernel execution from the closure. If they could be applied to bound the cost of the wait-queue and replenishment operations, they could be an important part of a solution. Preemption points add significant complexity to the system and require kernel operations to be iteratively computed. For example, preemption points are used to enable capability revocation to revoke a limited number of resources, enable preemptions, and later resume revocation from where it had previously left off. Unfortunately, iterating through a wait queue, or

through a queue of replenishments, does not fall into the traditional type of logic that preemption points are designed for, as each iteration does not remove work from the computation to be done after a preemption point. Such an iteration could not be resumed after a preemption point as the preemption implies that the queue's structure could have been updated by intervening operations (*e.g.*, removing the thread from the wait queue referenced by the current iterator). Preemption point logic would need to increase in complexity to handle wait-queue operations and budget depletion.

Preemption points also *cannot* be added to the replenishment processing. Preemption points rely on being able to resume a thread that continues kernel processing from where it had previously left off. Despite preemption point's superficial applicability to these problems, unfortunately it won't help with all attacks, particularly with replenishment processing in timer-interrupt context.

7.5.5 Partial processing of wakeups in interrupts.

TimerShield [164] represents a potential solution to the replenishment-processing problem. The key insight is that if replenishments can be tracked per-priority, then all replenishments for time t do not need to be processed at that time. Instead, at each scheduling decision, the replenishment queue can be consulted and processed if the highest-priority threads requiring replenishments have the same or higher priority than the highest-priority thread in the run queue. Though this approach is appealing, it complicates the system, requiring the design of the replenishment logic to be considered more broadly within the system as a whole.

7.5.6 Other μ -Kernel Designs. Although seL4 is based on the L4 μ -kernel heritage, it is a unique μ -kernel with functional verification as its primary goal. Below we discuss other μ -kernel designs.

7.5.7 Fiasco and Nova. Fiasco [202] takes a different view on priority and budget management during IPC. In Fiasco, servers execute using the budget of the client requesting their service (as in `seL4-MCS`). However, in both Fiasco and Nova [201] the server *inherits* the highest priority of any client transitively waiting on the service. This can add overhead to the scheduling path as the dependencies of the highest-priority thread (and its dependencies’ dependencies, etc.) are traversed to find the server to execute. Additionally, Fiasco adds vCPU budgets [121], which require depletion and replenishment processing.

Generally, these kernels execute *preemptively*, which prevents Thundering Herd attacks on kernel structures from causing global interference. However, both use spin-locks to protect kernel structures, which selectively disable interrupts while processing data structures, and both disable interrupts for interrupt execution. We have not assessed if Fiasco or Nova exhibit similar attacks on their non-preemptive access to data-structures. The lessons of this research should inform the assessment of their vulnerability to such attacks.

7.5.8 Thread migration in Composite. Thread-migration-based IPC [82, 84, 162] is a different mechanism for synchronous coordination between client and server. A server process is the target of the IPC, not a server thread. IPC from a client triggers execution in the server that proceeds within the same scheduler context as in the client process. It is called “thread migration” because the same thread simply *continues* execution in the server, though spatial isolation is maintained by splitting client/server execution across separate stacks and register contents. Since the same schedulable client thread executes in the server, the same scheduler abstractions such as priority and budget are maintained. This structure imposes a few requirements:

1. servers are concurrent by default and thus require synchronized access to shared data structures, and
2. server stacks must be allocated upon IPC to the server as the first action within the server's computation.

Both of these challenges can be addressed by efficient, predictable mechanisms for stacks and mutual exclusion for both fixed [219], and dynamic [220] sets of threads. Thread migration can avoid blocking semantics in the kernel; instead, schedulers can be implemented in user-level processes [163, 86]. Even where budgets are tracked in the kernel [85], replenishments are exported from the kernel to user-level schedulers.

As thread migration enables the policies for contention, scheduling, and budget management to be extracted from the kernel, all kernel operations can be constant-time as demonstrated by [218]. However, within the scheduling processes that maintain wait queues, budgets, and priorities, it is possible that Thundering Herd Attacks could be impactful. Though interrupts are never disabled for user-level processing, critical sections within the scheduler might comparably be attacked, delaying necessary scheduling decisions.

7.5.9 Static Partitioning Hypervisors. Another potential solution to these attacks is to use a static partitioning hypervisor, such as Jailhouse [115] to statically isolate untrusted parts of the system from one another. This is perhaps a reasonable solution for coarse-grained isolation of untrusted and uncooperating components, provided that there are sufficient hardware resources to be dedicated to individual partitions. However, μ -kernels such as `seL4` enable more fine-grained isolation, enabling the principle of least privilege to be employed in system design, such as in Patina [107]. For example, many trusted, but not trustworthy,

components may communicate and collaborate and therefore need to be co-located within a single static partition. But even a trusted component may be compromised, and a component-based architecture, and resource sharing and strong isolation enabled by a trustworthy μ -kernel such as **seL4** limit the damage an attacker can do. A static partitioning hypervisor can therefore help ameliorate some of these concerns, but does not fundamentally solve the system-coordination dilemma.

7.5.10 Summary. While we've discussed several potential solutions to Thundering Herd Attacks in **seL4**, they all present trade-offs. We've also analyzed different systems and shown that Thundering Herd Attacks are a more general concern.

CHAPTER VIII

MIA DISCOVERY

As seen in the unpublished submission *Mergendahl, S., Fickas, S., Norris, B., & Skowrya, R. (2024, May). Manipulative Interference Attacks. In 2024 ACM Conference on Computer and Communications Security (CCS), (In Submission).*

8.1 Identifying MIA

In this section, we describe a methodology to automatically identify when the primitives required to successfully execute Manipulative Interference Attacks (MIA) exist within a system configuration. Due to the complexity of MIA primitives (*e.g.*, the delicate interplay between budget, priority, and IPC or the dependence of data flow), we argue that an automated approach will have the most success in identifying instances of MIA, rather than solely relying on manual, technical expertise. Moreover, similar to the triage of software vulnerabilities, we must prioritize any instances of MIA for a system designer to perform a proper risk assessment of the system.

A key insight of our methodology is to leverage different tools where they excel and avoid common pitfalls where the tools may fail. In particular, Figure 33 describes three different views of a system configuration and what tools we use to identify the different MIA primitives. First, because of the difficulty to manually determine if a component can become manipulated, we leverage static analysis to automatically identify any cycles in each software component in the system, and label each of these cycles as influenceable, triggerable, and/or unbounded, with respect to an external entry point in the component (*e.g.*, an IPC receive path). However, the performance of *inter-procedural* static analysis may become challenging, and further, performing static analysis across binaries is often

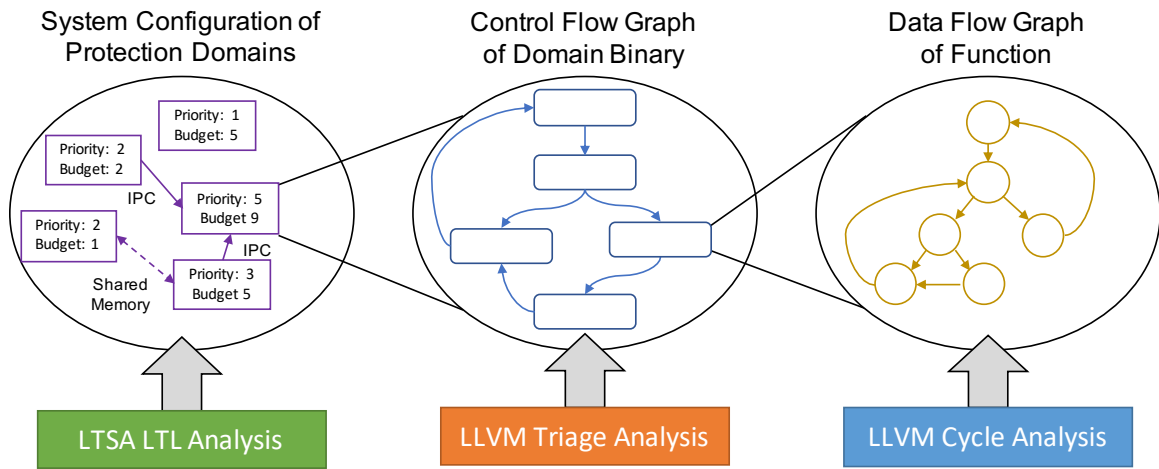


Figure 33. We breakdown a system into three hierarchical views to a facilitate a practical search for Manipulative Interference Attacks (MIA) in which we limit the overhead of each of our analysis components to only one system view and combine their results.

prohibitive. As such, we scope our static analysis to only identify *data flow* cycles, and do not attempt to study the complex interplay between budget, priority, and IPC assignment using solely static analysis.

Instead, we make a key connection that goal-oriented conflict analysis is better suited to identify conflicts at the *system configuration* level. Namely, we track priority, budget, and IPC assignment as *goals* in the system, and instantiate these goals using Linear Temporal Logic (LTL) to identify divergent goals that arise after considering the results from our static analysis. However, if we were to leverage LTL analysis *comprehensively* to identify conflicts and manipulable components, it would also suffer from prohibitively long analysis times. Additionally, creating system models capable of LTL analysis requires expensive technical expertise, so we take a revolutionary approach that *automatically* generates the system model with readily available system build artifacts.

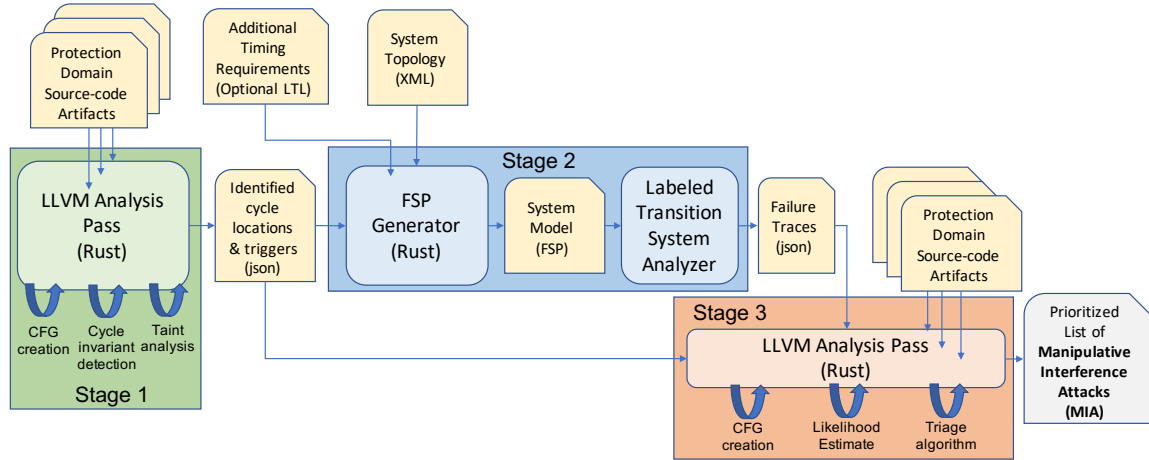


Figure 34. Our analysis first identifies cycles that can be manipulated and influenced, second automatically generates a model to verify LTL properties related to MIA, and third triages the identified failures by those that are triggered by components with weak code reuse protection.

Finally, in order to assess the severity of goal inconsistencies, we draw from previous work to create a metric of the likelihood that a component could become compromised and trigger MIA. Technical expertise may not be available to digest the results of our LTL analysis, so an automated triage process is instead preferred. Namely, we leverage the analysis of CFInsight [83] to prioritize which MIA instances are more likely.

8.1.1 Cycle Detection. In the first stage of our analysis, as seen in Figure 34, we leverage the LLVM analysis framework to find cycles in high-priority components that could become manipulated. LLVM is a set of compiler and toolchain technologies designed around a language-independent intermediate representation (IR) portable to any instruction set architecture. LLVM is structured to perform a variety of transformations over multiple passes. For cycle detection, we write a LLVM pass in Rust that first identifies the control flow graph (CFG) of the analyzed component, and then studies the CFG for cycles

and their features. In particular, after we identify a list of cycles in the CFG, we use the CFG to also identify which cycles follow directly after an IPC receive call. Because IPC is typically a system call on the μ -kernel, we can find the cycles that are *triggerable* from an external component. Additionally, we also leverage taint analysis on the data flow of the program to identify which cycles are *influenceable* from the data received on IPC. We mark the variable that holds the IPC message, and track which other variables this message impacts. When the taint analysis touches a loop invariant, we consider that cycle to be influenceable. However, in order to identify a more complete set of influenced values, multiple passes are required for this process. Notably, we do not need to transform the program to perform this analysis, so we can simply ingest the system build artifacts.

8.1.2 Requirements Analysis. In the second stage of our analysis, as seen in Figure 34, we contextualize the identified cycles from the first stage of our analysis with respect to the priority, budget, and communication paths of the system configuration. To accomplish this, we model the protection domains, their interactions, and the μ -kernel in the formal modeling language, Finite State Processes (FSP) [139]. FSP is a language used to represent software processes (and compositions of processes), including constants, ranges, sets, actions, and safety/progress properties of a system, defined in terms of a Labeled Transition System (LTS). Moreover, FSP can be analyzed using the Labeled Transition System Analyzer (LTSA) [139]. In particular, we represent the system topology (*i.e.*, IPC communication paths as well as shared memory communication paths), budgets, and priorities in FSP, and convert the timing requirements for a system into FSP *fluents* that act as asserts in LTL. LTSA then searches the possible execution paths for a conflict in the LTL asserts. For example, we can check if each

protection domain successfully executes (to prevent starvation that might occur under MIA):

$$\forall pd, \Box(pd_{blocked} \implies \Diamond pd_{executing}) \quad (8.1)$$

This LTL formula verifies that for all protection domains in the system, it is always the case, that if a domain is blocked, it will eventually execute again.

Additionally, we take a revolutionary approach in that we *automatically* generate the FSP model of the system given appropriate system configuration. Namely, we wrote a tool in Rust, *FSPGen*, that consists of a front-end to ingest and convert the system configuration into an intermediate Rust representation, and a back-end that converts the Rust representation into FSP based on a desired μ -kernel. In particular, our front-end currently supports the XML system configuration used in the `seL4` microkit [125] and the system build artifacts from the DARPA CASE study [213, 29, 97]. Moreover, our back-end currently supports the original `seL4` kernel and its `seL4-MCS` extensions. However, our tool is flexible enough to add support for additional μ -kernels, system configuration, or other LTL languages in the future.

8.1.3 Triage. In the final stage of our analysis, as seen in Figure 34, we compute a stand-in metric for the likelihood of the identified failure found by LTSA. Of course, all identified instances of MIA are attacks that should be addressed, but we take a hierarchical approach to triage instances of MIA. First, because we run LTSA iteratively where we only assume one compromised component, we can order failures based on the priority of the compromised component. In particular, the lower the priority (and thus, less trusted) of a compromised component, we raise the severity of the failure. This follows a typical risk assessment of a system in that higher trusted components must be more

trusted to support the system, whereas lower trusted components are more assumed to be isolated.

Secondly, for failures that arise from components with the same priority, we draw from previous work that assesses the risk posture of code-reuse attacks. Originally, CFInsight builds on other metrics that evaluate CFI policies by introducing a novel metric called CFGInsulation that quantifies how easy it is for an attacker to build an exploit under CFI [83]. Namely, CFGInsulation considers the number and length of paths in the CFG to a system call with the insight that a common attacker goal in a code-reuse attack is to invoke a system call with controlled parameters and perform the next stage of the attack. Indeed, since a critical primitive for MIA is to send a maliciously crafted IPC message (*i.e.*, a system call), CFGInsulation is a reasonable stand-in for the “likelihood” of an attack.

8.2 Evaluation

In this section, we evaluate the performance of our MIA analysis tool. In particular, we want to show reasonable analysis times, as we hope our tool can integrate into the compilation process for an embedded system. For example, we imagine system designers could integrate our analysis into their Continuous Integration (CI) test infrastructure. Because Manipulative Interference Attacks (MIA) are a new type of attack, we do not have other analysis tools to directly compare our performance, so instead, we compare to baseline analysis times of similar aspects of a CI test infrastructure. However, we hope to inspire others to find novel methodologies to identify new instances of MIA,

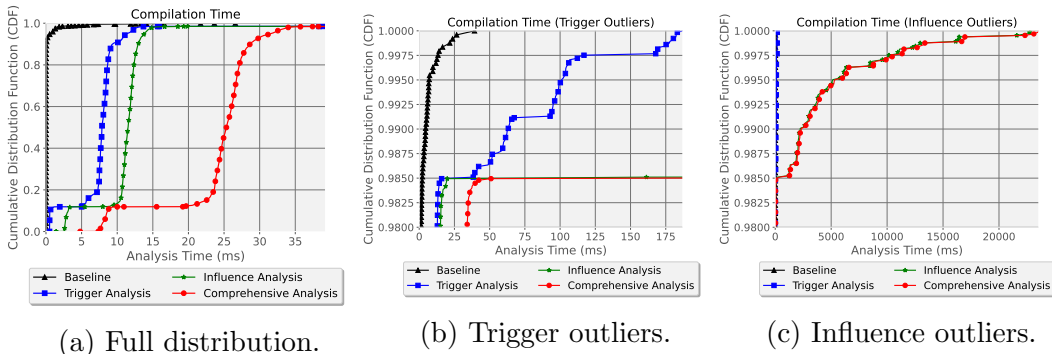


Figure 35. Cumulative Distribution Function (CDF) of our MIA LLVM analysis compared to the baseline LLVM benchmark test-suite.

or improve our analysis times. In particular, we make our tool (and evaluation artifacts) available online¹.

8.2.1 Static Analysis. There are two places where we leverage static analysis to identify MIA. First, in Stage 1 of our tool, we leverage the LLVM compiler infrastructure to find influenceable, triggerable, and unbounded cycles. Additionally, unlike the original CFInsulation metric [83], we analyze LLVM IR rather than the binary for our triage analysis.

Figure 35 shows a Cumulative Distribution Function (CDF) of our analysis times compared to the baseline LLVM test-suite used by the compiler infrastructure project [169, 170]. In particular, Figure 35a shows that while the highly optimized baseline test-suite for LLVM is generally faster than our analysis, the strong majority of our influenceable, triggerable, and unbounded cycles analysis along with our triage analysis occurs in a reasonable amount of time. Namely, 98.5% of our analysis occurs under 35ms. While compilation times should be fast, we believe 35ms is indeed reasonable to identify primitives for MIA. Moreover, in Figure 35b, we see that even the highly optimized analysis times for vanilla LLVM tests can

¹<https://github.com/smergendahl/manipulative-interference-attacks>

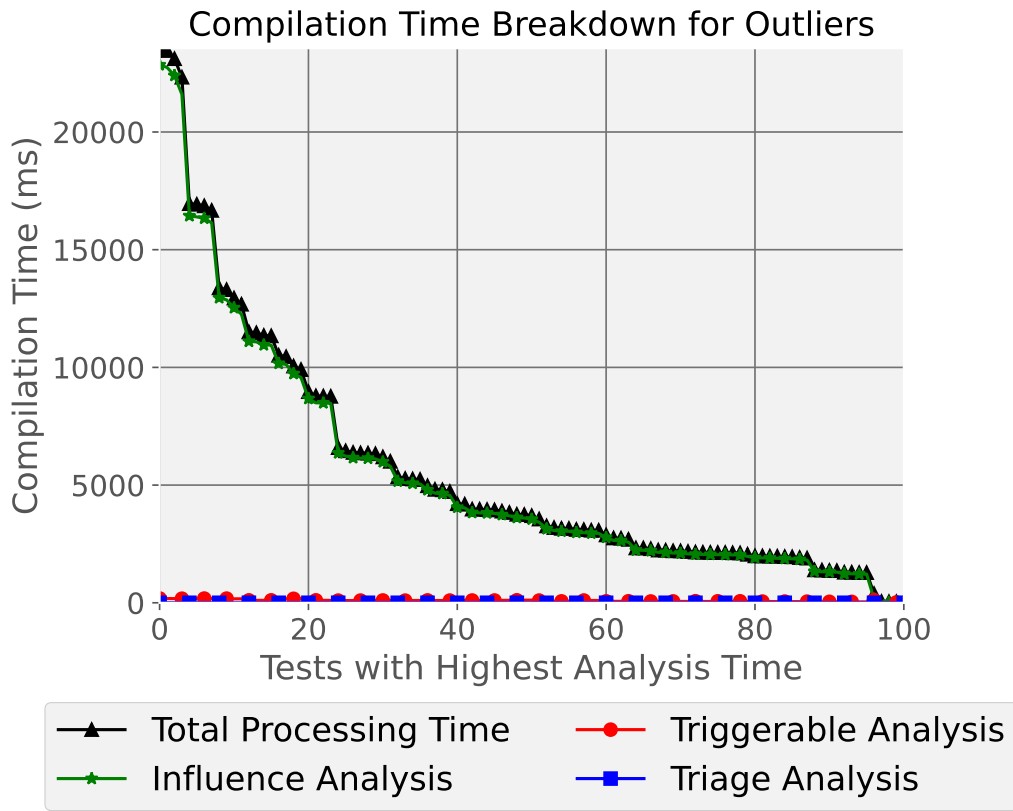


Figure 36. Influence analysis leads to large processing times, but triggerable analysis is more performant and can still cause MIA.

also reach this length of analysis times. However, in Figure 35c, we see that roughly 1% of the time, our analysis can trigger exceptionally long analysis times: MIA analysis can take up to 25s.

We investigate these outlier analysis times in Figure 36. In particular, for the top 100 test times, influenceable analysis is the cause of decrease in performance. As such, for users of our tool that can afford to identify some, but not all, instances of MIA, the system designer can turn off the analysis pass that searches for influenceable cycles. Instead, only triggerable cycles will be identified, but at times that compare to the baseline tests. Notably, if MIA only invokes a

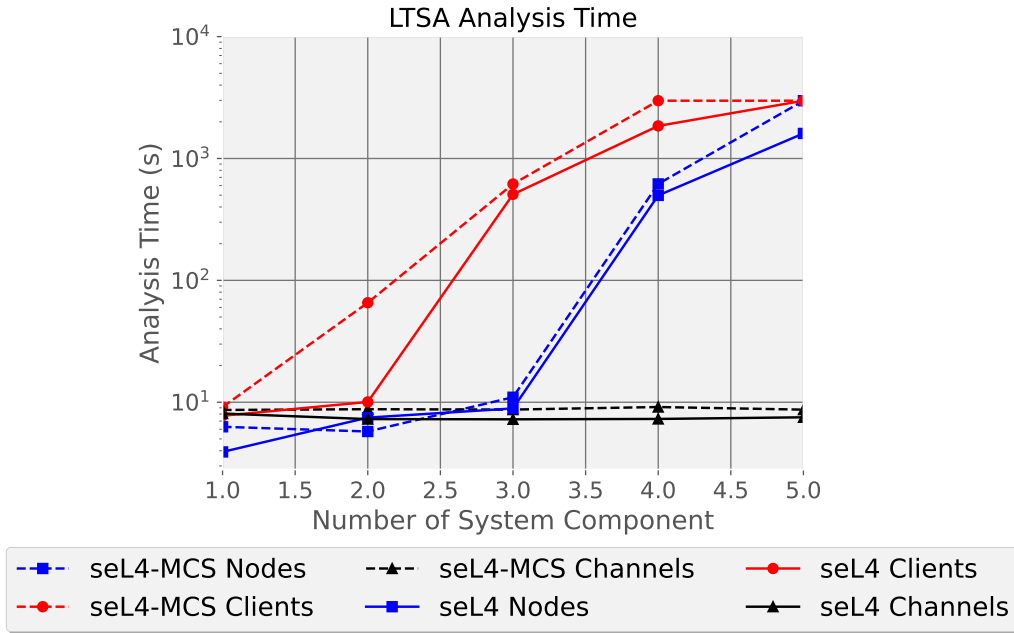


Figure 37. LTL analysis is exponential with respect to the number of protection domains and the number of clients to a IPC server.

triggerable cycle (but not necessarily an influenceable cycle) missed deadlines can still arise in the system.

8.2.2 Goal-Conflict Analysis. We also study the analysis time of the second stage of our tool that ingests the results from our cycle analysis and identifies timing requirement failure traces due to MIA. Namely, we create and benchmark 30 different example systems built on both `seL4` and `seL4-MCS`. This suite of benchmark systems helps us differentiate the impact of different system features on analysis time. In particular, our benchmark systems vary in the number of protection domains, number of IPC channels, and *degree* of each IPC channel (*i.e.*, the average number of clients to a particular IPC server) from 1 to 5, and we record the length of analysis time for each system with a timeout of 3000s. We pause to highlight the success of our *FSPGen* tool. Manually generating 30 different models of 30 different system configurations for two different μ -kernels

would be extremely daunting, and the fact that we can leverage our *FSPGen* tool to *automatically* generate the FSP models of each system under test facilitates the possibility for such an evaluation.

Specifically, in Figure 37, we load each model into LTSA, and benchmark the time to compile each system. First, all studied systems with 3 or less protection domains require roughly 10s or less to analyze. Because many embedded systems will only need a handful of protection domains, we believe this tool will be usable by `seL4` systems. However, we do note that analysis time is roughly exponential with respect to the number of protection domains, so analysis may become expensive for very complex systems. Moreover, as the number of IPC channels (as well as shared memory channels) increases, the analysis time is linear rather than exponential. Additionally, these trends also hold as the degree of IPC endpoints increases: increasing the degree of a channel increases the number of protection domains and channels to analyze, but a linear increase in the degree of a channel is only linear plus exponential increase, rather than doubly exponential. Lastly, we note that analysis is slightly more expensive for `seL4-MCS`, as the system model has complex budget and period management to track.

CHAPTER IX

CONCLUSION

In this dissertation, I presented the first security model for applications developed in multiple programming languages, a type of system I refer to as Multi-Language Applications (MLA). I specifically focused on cases where the application is written well (limited or no usage of *unsafe* code in the parts written in the safe language) and where advanced protections are applied to the parts written in the unsafe language. I illustrated that because of mismatching threat models, an attacker can maneuver between various stages of an exploit in such a way that avoids triggering safety/hardening checks, while succeeding in hijacking control. Dubbed Cross-Language Attacks (CLA), these attacks can non-intuitively result in weakening of the overall application security when parts of the application are ported to a safe programming language. I illustrated different variants of CLA and performed automated analysis on large code bases to measure the prevalence of CLA building blocks. My findings illustrate that CLA building blocks are abundantly found in Firefox, and that a new class of countermeasures must be developed to secure MLA, and sketch their design goals for such future defenses.

Moreover, I introduced a new type of attack to cause temporal safety violations that can circumvent spatial and temporal isolation boundaries in a real-time, embedded system in order to delay hard real-time tasks and cause critical system failure. I call these attacks Manipulative Interference Attacks (MIA), and introduce a special form of MIA when the attacker can create many malicious threads, called Thundering Herd Attacks (THA). I further proposed a methodology to automatically identify instances of MIA given readily available system build artifacts. Our analysis takes a hybrid approach that combines static analysis with

goal-oriented, conflict analysis to identify situations where conflict can arise in the budget and priority assignments of a mixed-criticality system when a low-criticality component may become compromised. We instantiate our tool on the seL4 μ -kernel, and show that both the original seL4 kernel and seL4-MCS extensions are vulnerable to MIA.

APPENDIX A

Cross-Language Attacks (CLA)

In this section, we provide code examples of Cross-Language Attacks (CLA) in Go-C/C++ applications that correlate to the code examples presented in §4.2 and §4.3. In particular, Figure 1, Figure 2, and Figure 3 correspond to the attacks presented in §4.2.2, §4.2.3, §4.2.4 respectively while Figure 4, and Figure 5 correspond to the attacks presented in §4.3.1 and §4.3.3 respectively.

```
1 func go_fn(cb_fptr *func(*int64)) {
2     // Initialize some data
3     x := Data {
4         vals: [3]int64{1,2,3},
5         cb: cb_fptr,
6     }
7
8     C.vuln_fn(/*Ptr to x.vals*/)
9
10    // Uses corrupted function pointer
11    (*x.cb)(&x.vals[0])
12 }
```

(a) Go code that calls C/C++ to modify a Go struct.

```
1 // This function modifies a given array
2 // Can cause an OOB vulnerability
3 void vuln_fn(int64_t array_ptr_addr) {
4     // These values are set by a corruptible
5     // source, e.g., user input
6     int64_t array_index = 3;
7     int64_t array_value = get_attack();
8
9     int64_t* a = (void *)array_ptr_addr;
10    a[array_index] = array_value;
11 }
```

(b) C/C++ code that performs an Out-of-Bounds (OOB) error.

Figure 1. Sample code to illustrate how CLA can circumvent Go to cause a OOB error.

```

1 func go_fn(cb_fptr *func(*int64)) {
2     heap_obj := new(/* Go heap allocation */)
3
4     C.vuln_fn(/*Ptr to heap_obj*/)
5
6     heap_obj[0] += 5 // UaF
7 }

```

(a) Go code that uses a pointer wrongfully freed by C/C++.

```

1 // Frees object it does not own
2 void vuln_fn(int64_t obj_ptr_addr) {
3     int64_t* a = (void *)obj_ptr_addr;
4
5     //C/C++ frees Go allocated object!
6     free(a);
7 }

```

(b) C/C++ code that leads to a Use-after-Free (UaF) error in Go.

Figure 2. Sample code to illustrate how CLA can coerce Go into causing a UaF error.

```

1 func go_fn(cb_fptr *func(*int64)) {
2     fptr := /* Function pointer */
3
4     //C++ code overwrites fptr
5     C.vuln_fn()
6
7     // No CFI checks!
8     (*fptr)()
9 }

```

(a) Go code that uses a function pointer.

```

1 void vuln_fn() {
2     int64_t a[1] = {0}; // C/C++ array
3     // These values are set by a corruptible
4     // source, e.g., user input
5     int64_t array_index = 47;
6     int64_t array_value = get_attack();
7
8     // Arbitrary Write to Rust fptr
9     a[array_index] = array_value;
10 }

```

(b) C/C++ that overwrites a Go function pointer.

Figure 3. Sample code to show how CLA can corrupt a Go function pointer to execute a weird machine and circumvent CFI.

```

1 func go_fn(cb_fptr *func(*int64)) {
2     //Go slices have dynamic bounds
3     slice := []int64{4, 5}
4
5     C.vuln_fn(/*Ptr to slice*/)
6
7     // C++ changed the slice size to 128!
8     slice_fp_addr := slice[55]
9 }

```

(a) Go code that passes a slice to C/C++.

```

1 void vuln_fn(int64_t slice_ptr_addr) {
2     // These values are set by a corruptible
3     // source, e.g., user input
4     int64_t array_index = 2;
5     int64_t array_value = 128;
6
7     int64_t* a = (void *)slice_ptr_addr;
8     a[array_index] = array_value;
9 }

```

(b) C/C++ code with an arbitrary write vulnerability.

Figure 4. Example of C/C++ using an arbitrary write to corrupt size of a Go slice.

```

1 // Uses a function pointer provided by C/C++
2 func go_fn(cb_fptr *func(*int64)) {
3     fp_ptr := C.vuln_cb_fptr()
4     (*fp_ptr)()
5 }

```

(a) Go code that calls C/C++ to receive a callback pointer.

```

1 // Returns a call back function to register
2 int64_t vuln_cb_fptr() {
3     int64_t fp_ptr = get_attack();
4     return fp_ptr;
5 }

```

(b) C/C++ code that corrupts a return value to Go.

Figure 5. Sample code to illustrate how CLA can corrupt even data intended to cross the FFI boundary.

APPENDIX B

Manipulative Interference Attacks (MIA)

In this section, we show an example of a vulnerable system to Manipulative Interference Attacks (MIA). In particular, this code can run on `seL4-MCS` in the Microkit environment, or on `seL4` in the environment created under the DARPA Case Program.


```

1 // System specific headers
2 // includes system specific wrapper functions
3 // for print, send_ipc, recv_ipc, recv_notif, reply
4 #include <system.h>
5 // Server specific functions
6 int counter(int h) {
7     int count = 0;
8     int start = 0;
9     if (height%2==0) {
10        // Example of a triggerable,
11        // but not influenceable cycle
12        for (int i=1; i < 400000000; i++) {
13            count++;
14        }
15    }
16    // influenceable cycle
17    for (int i=1; i<= height; i++) {
18        count++;
19    }
20    // Cycle directly influenceable
21    // by function return value
22    for (int i=1; i<=dir_len(start,height); i++) {
23        count++;
24    }
25    // cycle indirectly influenceable
26    // by function return value
27    for (int i=1; i<=indir_len(start,h); i++) {
28        count++;
29    }
30    return count;
31 }
32 // Return value is directly influenceable
33 int dir_len(int n, int height) {
34     // influenceable cycle
35     // with condition set by an addition
36     for (int i=1; i<=n+height; i++) {
37         n = n+i;
38     }
39     return n;
40 }
41 // Return value is indirectly influenceable
42 int indir_len(int n, int height) {
43     int r = 0;
44     // influenceable cycle
45     // with condition set by an addition
46     for (int i=1; i<=n+height; i++) {
47         r = r+i;
48     }
49     return r;
50 }
51 // Startup Function
52 void init(void) {
53     print("SERVER|INFO:␣initializing...\n");
54     /* Nothing to initialise */
55 }
56 // Function called when a notification is signaled
57 void recv_notif(int sender) {
58     print("SERVER|ERR:␣unexpected␣notification\n");
59 }
60 // Function called when an IPC message is received
61 void recv_ipc(int sender, int msg) {
62     int count = 0;
63     print_int("SERVER|INFO:␣From␣%d\n", sender);
64     count = counter(msg);
65     print_int("SERVER|INFO:␣%d\n", count);
66     return reply(count);
67 }

```

(a) Server application in the toy example.

```

68 // System specific headers
69 // includes system specific wrapper functions
70 // for print, send_ipc, recv_ipc, recv_notif, reply
71 #include <system.h>
72 #define SERVER_CH 0
73 void handle_deadline(int n) {
74     print("BENIGN_CLI|INFO:␣deadline␣handled.\n");
75 }
76 void init(void) {
77     print("BENIGN_CLI|INFO:␣initializing...\n");
78     /* Nothing to initialise */
79 }
80 void recv_notif(int sender) {
81     print("BENIGN_CLI|ERR:␣I/O␣received.\n");
82
83     /* message the server */
84     let resp = send_ipc(SERVER_CH, val);
85
86     handle_deadline(resp);
87 }

```

(a) Benign Client in the toy example.

```

88 // System specific headers
89 // includes system specific wrapper functions
90 // for print, send_ipc, recv_ipc, recv_notif, reply
91 #include <system.h>
92 #define SERVER_CH 0
93 void init(void) {
94     int a[1] = {0};
95     int val = 10;
96     print("M_CLI|INFO:␣init␣function␣running\n");
97     while(1) {
98         // These values are set
99         // by a corruptible source
100         int array_index = 2;
101         int array_value = corrupted();
102
103         // Buffer overflow
104         a[array_index] = array_value;
105
106         /* message the server */
107         send_ipc(SERVER_CH, val);
108     }
109 }
110 void recv_notif(int sender) {
111     print("M_CLI|ERR:␣unexpected␣notification\n");
112 }

```

(a) Malicious Client in the toy example.

REFERENCES CITED

- [1] Go-git. <https://github.com/go-git/go-git>.
- [2] Why go was the right choice for cockroachdb, 2015.
<https://www.cockroachlabs.com/blog/why-go-was-the-right-choice-for-cockroachdb/>.
- [3] Bolt, 2021. <https://github.com/boltdb/bolt>.
- [4] cgo, 2021. <https://pkg.go.dev/cmd/cgo>.
- [5] Docker, 2021. <https://github.com/docker/>.
- [6] Dogear, 2021. <https://github.com/mozilla/dogear>.
- [7] Kubernetes, 2021. <https://github.com/kubernetes>.
- [8] Mesalock linux, 2021.
<https://github.com/mesalock-linux/mesalock-distro>.
- [9] mp4parse-rust, 2021. <https://github.com/mozilla/mp4parse-rust>.
- [10] Neqo, an implementation of quic written in rust, 2021.
<https://github.com/mozilla/neqo>.
- [11] Redox operating system, 2021. <https://www.redox-os.org/>.
- [12] Servo, 2021. <https://github.com/servo>.
- [13] plist.h, 2022.
- [14] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)* 13, 1 (2009), 1–40.
- [15] AGENCY, D. A. R. P. Memory safety.
- [16] AGENCY, N. S. Software memory safety.
- [17] AGTEN, P., VAN ACKER, S., BRONDSEMA, Y., PHUNG, P. H., DESMET, L., AND PIESSENS, F. Jsand: complete client-side sandboxing of third-party javascript without browser modifications. In *Proceedings of the 28th Annual Computer Security Applications Conference* (2012), pp. 1–10.
- [18] ALEXANDER FÆRØY. Rustintor, 2020.
<https://gitlab.torproject.org/legacy/trac/-/wikis/RustInTor>.

- [19] ALRAJEH, D., KRAMER, J., RUSSO, A., AND UCHITEL, S. Learning operational requirements from goal models. In *2009 IEEE 31st International Conference on Software Engineering (2009)*, IEEE, pp. 265–275.
- [20] ALRAJEH, D., KRAMER, J., VAN LAMSWEERDE, A., RUSSO, A., AND UCHITEL, S. Generating obstacle conditions for requirements completeness. In *2012 34th International Conference on Software Engineering (ICSE) (2012)*, IEEE, pp. 705–715.
- [21] ANDERSEN, S. Changes to functionality in microsoft windows xp service pack 2. *Microsoft technical document, August (2004)*.
- [22] ANDERSEN, S., AND ABELLA, V. Data execution prevention. changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies, 2004.
- [23] ANDERSON, J. P. Information security in a multi-user computer environment. In *Advances in Computers*, vol. 12. Elsevier, 1972, pp. 1–36.
- [24] ANDREW PAVERD. Control flow guard for clang/llvm and rust, 2020. <https://msrc-blog.microsoft.com/2020/08/17/control-flow-guard-for-clang-llvm-and-rust/>.
- [25] ASMUSSEN, N., HAAS, S., LACKORZYNSKI, A., AND ROITZSCH, M. Core-local reasoning and predictable cross-core communication with m³.
- [26] ASTRAUSKAS, V., MATHEJA, C., POLI, F., MÜLLER, P., AND SUMMERS, A. J. How do programmers use unsafe rust? *Proceedings of the ACM on Programming Languages 4*, OOPSLA (2020), 1–27.
- [27] BALASUBRAMANIAN, A., BARANOWSKI, M. S., BURTSEV, A., PANDA, A., RAKAMARIĆ, Z., AND RYZHYK, L. System programming in rust: Beyond safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (2017)*, pp. 156–161.
- [28] BARRANTES, E. G., ACKLEY, D. H., FORREST, S., AND STEFANOVIĆ, D. Randomized instruction set emulation. *ACM Transactions on Information and System Security (TISSEC) 8*, 1 (2005), 3–40.
- [29] BELT, J., HATCLIFF, J., SHACKLETON, J., CARCIOFINI, J., CARPENTER, T., MERCER, E., AMUNDSON, I., BABAR, J., COFER, D., HARDIN, D., ET AL. Model-driven development for the sel4 microkernel using the hamr framework. *Journal of Systems Architecture 134* (2023), 102789.
- [30] BENDERSKY, E. Pyelftools, 2012. <https://github.com/eliben/pyelftools>.

- [31] BHATKAR, S., DUVARNEY, D. C., AND SEKAR, R. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *USENIX Security symposium (2003)*, vol. 12, pp. 291–301.
- [32] BIGELOW, D., HOBSON, T., RUDD, R., STREILEIN, W., AND OKHRAVI, H. Timely rerandomization for mitigating memory disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (2015)*, pp. 268–279.
- [33] BIGELOW, D., HOBSON, T., RUDD, R., STREILEIN, W., AND OKHRAVI, H. Timely rerandomization for mitigating memory disclosures. In *Proceedings of the 22nd ACM Computer and Communications Security (CCS'15) (Oct 2015)*.
- [34] BITTAU, A., MARCHENKO, P., HANDLEY, M., AND KARP, B. Wedge: Splitting applications into reduced-privilege compartments. USENIX Association.
- [35] BLACKHAM, B., SHI, Y., CHATTOPADHYAY, S., ROYCHOUDHURY, A., AND HEISER, G. Timing analysis of a protected operating system kernel. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS) (2011)*, IEEE Computer Society, pp. 339–348.
- [36] BOOS, K., LIYANAGE, N., IJAZ, R., AND ZHONG, L. Theseus: an experiment in operating system structure and state management. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20) (2020)*, pp. 1–19.
- [37] BOOS, K., AND ZHONG, L. Theseus: A state spill-free operating system. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems (2017)*, pp. 29–35.
- [38] BRANDENBURG, B. B. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.
- [39] BRANDENBURG, B. B. SchedCAT: The schedulability test collection and toolkit, 2022.
- [40] BRUMLEY, D., AND SONG, D. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium (2004)*, vol. 57.
- [41] BURNS, A., AND DAVIS, R. Mixed criticality systems - a review. Tech. rep., Department of Computer Science, University of York, 2013.

- [42] BUROW, N., CARR, S. A., NASH, J., LARSEN, P., FRANZ, M., BRUNTHALER, S., AND PAYER, M. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)* 50, 1 (2017), 1–33.
- [43] 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).
- [44] BUROW, N., ZHANG, X., AND PAYER, M. Sok: Shining light on shadow stacks. In *2019 IEEE Symposium on Security and Privacy (SP)* (2019), IEEE, pp. 985–999.
- [45] CARLINI, N., BARRESI, A., PAYER, M., WAGNER, D., AND GROSS, T. R. {Control-Flow} bending: On the effectiveness of {Control-Flow} integrity. In *24th USENIX Security Symposium (USENIX Security 15)* (2015), pp. 161–176.
- [46] CARLINI, N., BARRESI, A., PAYER, M., WAGNER, D., AND GROSS, T. R. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium (USENIX Security 15)* (2015), pp. 161–176.
- [47] CASTRO, M., COSTA, M., AND HARRIS, T. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006), pp. 147–160.
- [48] CHEN, G., JIN, H., ZOU, D., ZHOU, B. B., LIANG, Z., ZHENG, W., AND SHI, X. Safestack: Automatically patching stack-based buffer overflow vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* 10, 6 (2013), 368–379.
- [49] CHEN, J., NELISSEN, G., HUANG, W., YANG, M., BRANDENBURG, B. B., BLETSAS, K., LIU, C., RICHARD, P., RIDOUARD, F., AUDSLEY, N. C., RAJKUMAR, R., DE NIZ, D., AND VON DER BRÜGGEN, G. Many suspensions, many problems: A review of self-suspending tasks in real-time systems. *Real-Time Systems* 55, 1 (2019), 144–207.
- [50] CHEN, J. B., AND BERSHAD, B. N. The impact of operating system structure on memory system performance. In *Proceedings of the fourteenth ACM symposium on Operating systems principles* (1993), pp. 120–133.
- [51] CHEN, S., XU, J., SEZER, E. C., GAURIAR, P., AND IYER, R. K. Non-control-data attacks are realistic threats. In *USENIX Security Symposium* (2005), vol. 5.

- [52] 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), IEEE, pp. 56–71.
- [53] CHENEY, D. Cgo is not go.
<https://dave.cheney.net/2016/01/18/cgo-is-not-go>.
- [54] CHENG, Y., ZHOU, Z., MIAO, Y., DING, X., AND DENG, R. H. Ropecker: A generic and practical approach for defending against rop attack.
- [55] CHEUNG, W., AND LOONG, A. H. Exploring issues of operating systems structuring: from microkernel to extensible systems. *ACM SIGOPS Operating Systems Review* 29, 4 (1995), 4–16.
- [56] CHIUH, T.-C., AND HSU, F.-H. Rad: A compile-time solution to buffer overflow attacks. In *Proceedings 21st International Conference on Distributed Computing Systems* (2001), IEEE, pp. 409–417.
- [57] CIMPANU, C. Microsoft: 70 percent of all security bugs are memory safety issues, Feb 2019.
- [58] CLEMENTS, A. A., ALMAKHDHUB, N. S., BAGCHI, S., AND PAYER, M. {ACES}: Automatic compartments for embedded systems. In *27th USENIX Security Symposium (USENIX Security 18)* (2018), pp. 65–82.
- [59] COFER, D. Case overview: Cyber assured systems engineering. *seL4 Summit* (2022).
- [60] COWAN, C., PU, C., MAIER, D., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., ZHANG, Q., AND HINTON, H. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX security symposium* (1998), vol. 98, San Antonio, TX, pp. 63–78.
- [61] CROSIGNANI, M., MACCHIAVELLI, M., AND SILVA, A. F. Pirates without borders: The propagation of cyberattacks through firms’ supply chains. *FRB of New York Staff Report*, 937 (2021).
- [62] CROWDSTRIKE, INC. 2021 global threat report, 2021.
- [63] CYBERSECURITY, U., ET AL. The case for memory safe roadmaps: Why both c-suite executives and technical experts need to take memory safe coding seriously.
- [64] DEGIOVANNI, R., CASTRO, P., ARROYO, M., RUIZ, M., AGUIRRE, N., AND FRIAS, M. Goal-conflict likelihood assessment based on model counting. In *Proceedings of the 40th International Conference on Software Engineering* (2018), pp. 1125–1135.

- [65] DEGIOVANNI, R., RICCI, N., ALRAJEH, D., CASTRO, P., AND AGUIRRE, N. Goal-conflict detection based on temporal satisfiability checking. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (2016), pp. 507–518.
- [66] DENNING, P. J. The science of computing: The internet worm. *American Scientist* 77, 2 (1989), 126–128.
- [67] DENNIS, J. B., AND HORN, E. C. V. Programming semantics for multiprogrammed computations. *Commun. ACM* 26, 1 (1983), 29–35.
- [68] DEVIETTI, J., BLUNDELL, C., MARTIN, M. M., AND ZDANCEWIC, S. Hardbound: architectural support for spatial safety of the c programming language. *ACM SIGOPS Operating Systems Review* 42, 2 (2008), 103–114.
- [69] DEWALD, A., HOLZ, T., AND FREILING, F. C. Adsandbox: Sandboxing javascript to fight malicious websites. In *proceedings of the 2010 ACM Symposium on Applied Computing* (2010), pp. 1859–1864.
- [70] DOBROVITSKI, I. Exploit for cvs double free () for linux pserver, 2003.
- [71] DUTA, V., FREYER, F., PAGANI, F., MUENCH, M., AND GIUFFRIDA, C. Let me unwind that for you: Exceptions to backward-edge protection. In *NDSS* (2023).
- [72] ELPHINSTONE, K., AND HEISER, G. From L3 to seL4 what have we learnt in 20 years of L4 microkernels? In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)* (2013), ACM, pp. 133–150.
- [73] ENGLER, D. R., KAASHOEK, M. F., AND O’TOOLE JR, J. Exokernel: An operating system architecture for application-level resource management. *ACM SIGOPS Operating Systems Review* 29, 5 (1995), 251–266.
- [74] 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* (2006), pp. 75–88.
- [75] EVANS, I., FINGERET, S., GONZALEZ, J., OTGONBAATAR, U., TANG, T., SHROBE, H., SIDIROGLOU-DOUSKOS, S., RINARD, M., AND OKHRAVI, H. Missing the point (er): On the effectiveness of code pointer integrity. In *2015 IEEE Symposium on Security and Privacy* (2015), IEEE, pp. 781–796.

- [76] EVANS, I., LONG, F., OTGONBAATAR, U., SHROBE, H., RINARD, M., OKHRAVI, H., AND SIDIROGLOU-DOUSKOS, S. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), pp. 901–913.
- [77] EVANS, I., LONG, F., OTGONBAATAR, U., SHROBE, H., RINARD, M., OKHRAVI, H., AND SIDIROGLOU-DOUSKOS, S. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), pp. 901–913.
- [78] FARKHANI, R. M., JAFARI, S., ARSHAD, S., ROBERTSON, W., KIRDA, E., AND OKHRAVI, H. On the effectiveness of type-based control flow integrity. In *Proceedings of the 34th Annual Computer Security Applications Conference* (2018), pp. 28–39.
- [79] FARKHANI, R. M., JAFARI, S., ARSHAD, S., ROBERTSON, W., KIRDA, E., AND OKHRAVI, H. On the Effectiveness of Type-based Control Flow Integrity. In *Proceedings of IEEE Annual Computer Security Applications Conference (ACSAC'18)* (Dec 2018).
- [80] FEILER, P. H., GLUCH, D. P., AND HUDAK, J. The architecture analysis & design language (aadl): An introduction.
- [81] FERRAIUOLO, A., ZHAO, M., MYERS, A. C., AND SUH, G. E. HyperFlow: A processor architecture for nonmalleable, timing-safe information flow security. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2018), ACM Press, pp. 1583–1600.
- [82] FORD, B., AND LEPREAU, J. Evolving Mach 3.0 to a migrating thread model. In *Proceedings of the Winter 1994 USENIX Technical Conference* (1994), USENIX Association, pp. 97–114.
- [83] FRASSETTO, T., JAUERNIG, P., KOISSER, D., AND SADEGHI, A.-R. Cfinsight: A comprehensive metric for cfi policies. In *NDSS* (2022).
- [84] GABBER, E., SMALL, C., BRUNO, J. L., BRUSTOLONI, J. C., AND SILBERSCHATZ, A. Pebble: A component-based operating system for embedded applications. In *USENIX Workshop on Embedded Systems* (1999), USENIX Association, pp. 55–65.
- [85] GADEPALLI, P. K., GIFFORD, R., BAIER, L., KELLY, M., AND PARMER, G. Temporal capabilities: Access control for time. In *2017 IEEE Real-Time Systems Symposium (RTSS)* (2017), IEEE Computer Society, pp. 56–67.

- [86] GADEPALLI, P. K., PAN, R., AND PARMER, G. Slite: OS support for near zero-cost, configurable scheduling. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2020), IEEE, pp. 160–173.
- [87] GADEPALLI, P. K., PEACH, G., PARMER, G., ESPY, J., AND DAY, Z. Chaos: A system for criticality-aware, multi-core coordination. In *25th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2019), IEEE, pp. 77–89.
- [88] GHOSN, A., KOGIAS, M., PAYER, M., LARUS, J. R., AND BUGNION, E. Enclosure: language-based restriction of untrusted libraries. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2021), pp. 255–267.
- [89] GIL, R., OKHRAVI, H., AND SHROBE, H. There’s a hole in the bottom of the c: On the effectiveness of allocation protection. In *2018 IEEE Cybersecurity Development (SecDev)* (2018), IEEE, pp. 102–109.
- [90] GITHUB. How much rust in firefox?, 2021.
<https://4e6.github.io/firefox-lang-stats/>.
- [91] GÖKTAŞ, E., ATHANASOPOULOS, E., POLYCHRONAKIS, M., BOS, H., AND PORTOKALIDIS, G. Size does matter: Why using {Gadget-Chain} length to prevent {Code-Reuse} attacks is hard. In *23rd USENIX Security Symposium (USENIX Security 14)* (2014), pp. 417–432.
- [92] GOOGLE. Git repositories on fuchsia, 2021.
<https://fuchsia.googlesource.com/>.
- [93] GOVINDAVAJHALA, S., AND APPEL, A. W. Using memory errors to attack a virtual machine. In *2003 Symposium on Security and Privacy, 2003.* (2003), IEEE, pp. 154–165.
- [94] GUDKA, K., WATSON, R. N., ANDERSON, J., CHISNALL, D., DAVIS, B., LAURIE, B., MARINOS, I., NEUMANN, P. G., AND RICHARDSON, A. Clean application compartmentalization with soaap. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), pp. 1016–1031.
- [95] HALLER, I., JEON, Y., PENG, H., PAYER, M., GIUFFRIDA, C., BOS, H., AND VAN DER KOUWE, E. Typesan: Practical type confusion detection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), pp. 517–528.
- [96] HANSEN, P. B. The nucleus of a multiprogramming system. *Communications of the ACM* 13, 4 (1970), 238–241.

- [97] HARDIN, D. S., AND SLIND, K. L. Formal synthesis of filter components for use in security-enhancing architectural transformations. In *2021 IEEE Security and Privacy Workshops (SPW)* (2021), IEEE, pp. 111–120.
- [98] 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)* (2019), pp. 489–504.
- [99] HEISER, G. How to (and how not to) use seL4 IPC, 2019.
- [100] HISER, J., NGUYEN-TUONG, A., CO, M., HALL, M., AND DAVIDSON, J. W. Ilr: Where’d my gadgets go? In *2012 IEEE Symposium on Security and Privacy* (2012), IEEE, pp. 571–585.
- [101] 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)* (2016), IEEE, pp. 969–986.
- [102] 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)* (2016), IEEE, pp. 969–986.
- [103] HUANG, K., HUANG, Y., PAYER, M., QIAN, Z., SAMPSON, J., TAN, G., AND JAEGER, T. The taming of the stack: Isolating stack data from memory errors. In *NDSS* (2022).
- [104] INTEL. Intel®64 and IA-32 architectures software developer’s manual, 2021.
- [105] INTEL, I. and ia-32 architectures software developer’s manual. *Volume 3A: System Programming Guide, Part 1*, 64 (64), 64.
- [106] ISPOGLOU, K. K., ALBASSAM, B., JAEGER, T., AND PAYER, M. Block oriented programming: Automating data-only attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), pp. 1868–1882.
- [107] JERO, S., FURGALA, J., PAN, R., GADEPALLI, P. K., CLIFFORD, A., YE, B., KHAZAN, R., WARD, B. C., PARMER, G., AND SKOWYRA, R. Practical principle of least privilege for secure embedded systems. In *27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2021), IEEE, pp. 1–13.

- [108] JOAB JACKSON. Microsoft: Rust is the industry’s ‘best chance’ at safe systems programming, 2020. <https://thenewstack.io/microsoft-rust-is-the-industrys-best-chance-at-safe-systems-programming/>.
- [109] JUNG, R., JOURDAN, J.-H., KREBBERS, R., AND DREYER, D. Rustbelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages 2*, POPL (2017), 1–34.
- [110] JUNG, R., JOURDAN, J.-H., KREBBERS, R., AND DREYER, D. Safe systems programming in rust: The promise and the challenge. *Communications of the ACM* (2020).
- [111] KC, G. S., KEROMYTIS, A. D., AND PREVELAKIS, V. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security* (2003), pp. 272–280.
- [112] KIL, C., JUN, J., BOOKHOLT, C., XU, J., AND NING, P. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *2006 22nd Annual Computer Security Applications Conference (ACSAC’06)* (2006), IEEE, pp. 339–348.
- [113] 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* (2022), pp. 132–148.
- [114] KIRZNER, O., AND MORRISON, A. An analysis of speculative type confusion vulnerabilities in the wild. In *30th USENIX Security Symposium (USENIX Security 21)* (2021).
- [115] KISZKA, J. Jailhouse hypervisor, 2022.
- [116] KLABNIK, S., AND NICHOLS, C. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.
- [117] KLABNIK, S., AND NICHOLS, C. *The Rust programming language*. No Starch Press, 2023.
- [118] KLEIN, G., ANDRONICK, J., ELPHINSTONE, K., MURRAY, T., SEWELL, T., KOLANSKI, R., AND HEISER, G. Comprehensive formal verification of an os microkernel. *ACM Transactions on Computer Systems (TOCS)* 32, 1 (2014), 1–70.

- [119] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., ET AL. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), pp. 207–220.
- [120] KUZ, I., LIU, Y., GORTON, I., AND HEISER, G. CAmkES: A component model for secure microkernel-based embedded systems. *Journal of Systems and Software* 80, 5 (2007), 687–699.
- [121] LACKORZYŃSKI, A., WARG, A., VÖLP, M., AND HÄRTIG, H. Flattening hierarchical scheduling. In *Proceedings of the 12th International Conference on Embedded Software (EMSOFT)* (2012), ACM, pp. 93–102.
- [122] LAMOWSKI, B., WEINHOLD, C., LACKORZYNSKI, A., AND HÄRTIG, H. Sanderust: Automatic sandboxing of unsafe components in rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems* (2017), pp. 51–57.
- [123] LAMSWEERDE, A. v. *Requirements engineering: from system goals to UML models to software specifications*. John Wiley & Sons, Ltd, 2009.
- [124] LESLIE, B. GrailOS: A micro-kernel based, multi-server, multi-personality operating system. In *Workshop on Object Systems and Software Architectures (WOSSA 2006)* (2006).
- [125] LESLIE, B., AND HEISER, G. The sel4 core platform. *TS/sel4cp/2011-draft-spec.pdf* (2020).
- [126] LEVY, A., CAMPBELL, B., GHENA, B., GIFFIN, D. B., PANNUTO, P., DUTTA, P., AND LEVIS, P. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), pp. 234–251.
- [127] LI, C., SISU, X., CHENYANG, L., GILL, C. D., AND GUERIN, R. Prioritizing soft real-time network traffic in virtualized hosts based on xen. In *2015 IEEE 21st Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2015), IEEE, pp. 95–107.
- [128] LI, S.-W., LI, X., GU, R., NIEH, J., AND HUI, J. Z. A secure and formally verified linux kvm hypervisor. In *Proceedings of the IEEE Symposium on Security and Privacy* (2021).
- [129] LIEDTKE, J. Improving ipc by kernel design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP)* (1993), pp. 175–188.

- [130] LIEDTKE, J. On μ -kernel construction. *ACM SIGOPS Operating Systems Review* 29, 5 (1995), 237–250.
- [131] LIEDTKE, J., ISLAM, N., AND JAEGER, T. Preventing denial-of-service attacks on a μ -kernel for WebOSes. In *Proceedings of The Sixth Workshop on Hot Topics in Operating Systems (HotOS)* (1997), IEEE Computer Society, pp. 73–79.
- [132] 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* (2020), pp. 234–245.
- [133] LIU, S., TAN, G., AND JAEGER, T. Ptrsplit: Supporting general pointers in automatic program partitioning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), pp. 2359–2371.
- [134] LOONWERKS. Case: Cyber assured systems engineering.
- [135] LOONWERKS. Case-final.
- [136] LU, K. Practical program modularization with type-based dependence analysis. In *2023 IEEE Symposium on Security and Privacy (SP)* (2023), IEEE, pp. 1256–1270.
- [137] LYONS, A., MCLEOD, K., ALMATARY, H., AND HEISER, G. Scheduling-context capabilities: A principled, light-weight operating-system mechanism for managing time. In *Proceedings of the Thirteenth EuroSys Conference* (2018), ACM, pp. 26:1–26:16.
- [138] MAFFEIS, S., AND TALY, A. Language-based isolation of untrusted javascript. In *2009 22nd IEEE Computer Security Foundations Symposium* (2009), IEEE, pp. 77–91.
- [139] MAGEE, J., AND KRAMER, J. *State models and java programs*. wiley Hoboken, 1999.
- [140] MATSAKIS, N. D., AND KLOCK, F. S. The rust language. *ACM SIGAda Ada Letters* 34, 3 (2014), 103–104.
- [141] MCKEE, D. P., GIANNARIS, Y., ORTEGA, C., SHROBE, H. E., PAYER, M., OKHRAVI, H., AND BUROW, N. Preventing kernel hacks with hakcs. In *NDSS* (2022), pp. 1–17.
- [142] MERA, A., CHEN, Y. H., SUN, R., KIRDA, E., AND LU, L. D-box: Dma-enabled compartmentalization for embedded applications. *arXiv preprint arXiv:2201.05199* (2022).

- [143] MERGENDAHL, S., BUROW, N., AND OKHRAVI, H. Cross-language attacks. In *NDSS* (2022).
- [144] MERGENDAHL, S., JERO, S., WARD, B. C., FURGALA, J., PARMER, G., AND SKOWYRA, R. The thundering herd: Amplifying kernel interference to attack response times. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2022), IEEE, pp. 95–107.
- [145] MEYERSON, J. The go programming language. *IEEE software* 31, 5 (2014), 104–104.
- [146] MOZILLA FOUNDATION. Oxidation. <https://wiki.mozilla.org/Oxidation>. Accessed on 2021-05-14.
- [147] MURRAY, T., MATICHUK, D., BRASSIL, M., GAMMIE, P., BOURKE, T., SEEFRIED, S., LEWIS, C., GAO, X., AND KLEIN, G. seL4: From general purpose to a proof of information flow enforcement. In *2013 IEEE Symposium on Security and Privacy* (2013), IEEE, pp. 415–429.
- [148] MYERS, A. C. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1999), pp. 228–241.
- [149] MYLOPOULOS, J., CHUNG, L., NIXON, B., ET AL. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Transactions on software engineering* 18, 6 (1992), 483–497.
- [150] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M., AND ZDANCEWIC, S. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2009), pp. 245–258.
- [151] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M., AND ZDANCEWIC, S. Cets: compiler enforced temporal safety for c. In *Proceedings of the 2010 International Symposium on Memory Management* (2010), pp. 31–40.
- [152] NARAYAN, S., DISSELKOEN, C., GARFINKEL, T., FROYD, N., RAHM, E., LERNER, S., SHACHAM, H., AND STEFAN, D. Retrofitting fine grain isolation in the firefox renderer. In *29th USENIX Security Symposium (USENIX Security 20)* (2020), pp. 699–716.
- [153] NARAYANAN, V., HUANG, T., DETWEILER, D., APPEL, D., LI, Z., ZELLWEGER, G., AND BURTSEV, A. Redleaf: Isolation and communication in a safe operating system. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)* (2020), pp. 21–39.

- [154] NECULA, G. C., CONDIT, J., HARREN, M., MCPeAK, S., AND WEIMER, W. Cured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 27, 3 (2005), 477–526.
- [155] NEEDHAM, R. M., AND WALKER, R. D. The cambridge cap computer and its protection system. *ACM SIGOPS Operating Systems Review* 11, 5 (1977), 1–10.
- [156] OF STANDARDS, N. I., AND TECHNOLOGY. The nist cybersecurity framework (csf) 2.0.
- [157] ONE, A. Smashing the stack for fun and profit. *Phrack magazine* 7, 49 (1996), 14–16.
- [158] PAPAERVIPIDES, M., AND ATHANASOPOULOS, E. Exploiting mixed binaries. *ACM Transactions on Privacy and Security (TOPS)* 24, 2 (2021), 1–29.
- [159] PAPAERVIPIDES, M., AND ATHANASOPOULOS, E. Exploiting mixed binaries. *ACM Transactions on Privacy and Security (TOPS)* 24, 2 (2021), 1–29.
- [160] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Transparent {ROP} exploit mitigation using indirect branch tracing. In *22nd USENIX Security Symposium (USENIX Security 13)* (2013), pp. 447–462.
- [161] 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)* (2019), pp. 241–254.
- [162] PARMER, G. The case for thread migration: Predictable IPC in a customizable and reliable OS. In *Proceedings of the Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPert)* (2010), p. 91.
- [163] PARMER, G., AND WEST, R. Predictable interrupt management and scheduling in the Composite component-based system. In *Proceedings of the 29th IEEE International Real-Time Systems Symposium (RTSS)* (2008), IEEE Computer Society, pp. 232–243.
- [164] PATEL, P., VANGA, M., AND BRANDENBURG, B. B. TimerShield: Protecting high-priority tasks from low-priority timer interference. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2017), IEEE Computer Society, pp. 3–12.
- [165] PAX, T. Pax address space layout randomization (aslr). <http://pax.grsecurity.net/docs/aslr.txt> (2003).

- [166] PETERS, S., DANIS, A., ELPHINSTONE, K., AND HEISER, G. For a microkernel, a big lock is fine. In *Proceedings of the 6th Asia-Pacific Workshop on Systems* (2015), pp. 1–7.
- [167] PIETRASZEK, T., AND BERGHE, C. V. Defending against injection attacks through context-sensitive string evaluation. In *International Workshop on Recent Advances in Intrusion Detection* (2005), Springer, pp. 124–145.
- [168] PNUELI, A. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)* (1977), iee, pp. 46–57.
- [169] PROJECT, L. C. I. llvm-test-suite.
- [170] PROJECT, L. C. I. Llm test-suite guide.
- [171] PROJECTS, T. C. Memory safety.
- [172] REED, E. Patina: A formalization of the Rust programming language. *University of Washington, Department of Computer Science and Engineering, Tech. Rep. UW-CSE-15-03-02* (2015).
- [173] RICK HUDSON. Getting to go: The journey of go’s garbage collector, 2018. <https://blog.golang.org/ismmkeynote>.
- [174] RIVERA, E., MERGENDAHL, S., SHROBE, H., OKHRAVI, H., AND BUROW, N. Keeping safe rust safe with galeed. In *Proceedings of the 37th Annual Computer Security Applications Conference* (2021), pp. 824–836.
- [175] RIVERA, E., MERGENDAHL, S., SHROBE, H., OKHRAVI, H., AND BUROW, N. Keeping safe rust safe with galeed. In *Annual Computer Security Applications Conference* (2021), pp. 824–836.
- [176] ROESSLER, N., ATAYDE, L., PALMER, I., MCKEE, D., PANDEY, J., KEMERLIS, V. P., PAYER, M., BATES, A., SMITH, J. M., DEHON, A., ET AL. μ scope: 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* (2021), pp. 296–311.
- [177] ROESSLER, N., AND DEHON, A. Protecting the stack with metadata policies and tagged hardware. In *2018 IEEE Symposium on Security and Privacy* (May 2018), IEEE, pp. 478–495.
- [178] ROSE, K. Did one guy just stop a huge cyberattack? *The New York Times* (2024).

- [179] RSC. Do not let go pointers end up in c, 2014.
<https://github.com/golang/go/issues/8310>.
- [180] RUANE, L. M. Process synchronization in the UTS kernel. *Computing systems* 3, 3 (1990), 387–421.
- [181] RUDD, R., SKOWYRA, R., BIGELOW, D., DEDHIA, V., HOBSON, T., CRANE, S., LIEBCHEN, C., LARSEN, P., DAVI, L., FRANZ, M., ET AL. Address oblivious code reuse: On the effectiveness of leakage resilient diversity. In *NDSS* (2017).
- [182] RUOCCO, S., ET AL. Real-time programming and l4 microkernels. In *Proceedings of the 2006 Workshop on Operating System Platforms for Embedded Real-Time Applications, Dresden, Germany* (2006).
- [183] 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)* (2020), pp. 1677–1694.
- [184] SCHUSTER, F., TENDYCK, T., LIEBCHEN, C., DAVI, L., SADEGHI, A.-R., AND HOLZ, T. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *2015 IEEE Symposium on Security and Privacy* (2015), IEEE, pp. 745–762.
- [185] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. Q: Exploit hardening made easy. In *USENIX Security Symposium* (2011), vol. 10.
- [186] SEIBERT, J., OKHRAVI, H., AND SÖDERSTRÖM, E. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security* (2014), pp. 54–65.
- [187] SEREBRYANY, K., BRUENING, D., POTAPENKO, A., AND VYUKOV, D. Addresssanitizer: A fast address sanity checker. In *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)* (2012), pp. 309–318.
- [188] SEWELL, T., KAM, F., AND HEISER, G. Complete, high-assurance determination of loop bounds and infeasible paths for WCET analysis. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2016), IEEE, pp. 1–11.
- [189] SEWELL, T., WINWOOD, S., GAMMIE, P., MURRAY, T., ANDRONICK, J., AND KLEIN, G. seL4 enforces integrity. In *International Conference on Interactive Theorem Proving* (2011), Springer, pp. 325–340.

- [190] SEWELL, T. A. L., MYREEN, M. O., AND KLEIN, G. Translation validation for a verified OS kernel. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation* (2013), pp. 471–482.
- [191] SHA, L., RAJKUMAR, R., AND LEHOCZKY, J. P. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on computers* 39, 9 (1990), 1175–1185.
- [192] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security* (2007), pp. 552–561.
- [193] SHACHAM, H., PAGE, M., PFAFF, B., GOH, E.-J., MODADUGU, N., AND BONEH, D. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security* (2004), pp. 298–307.
- [194] SHAPIRO, J., SMITH, J., AND FARBER, D. EROS: A fast capability system. In *17th ACM Symposium on Operating systems principles* (December 1999), ACM, pp. 170–185.
- [195] SHAPIRO, J. S. Vulnerabilities in synchronous IPC designs. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy* (2003), IEEE Computer Society, pp. 251–262.
- [196] SHAPIRO, R., BRATUS, S., AND SMITH, S. W. Weird machines. In *In Proceedings of the 7th USENIX Workshop on Offensive Technologies (WOOT)* (2013), p. 11.
- [197] SHOSHITAISHVILI, Y., WANG, R., SALLS, C., STEPHENS, N., POLINO, M., DUTCHER, A., GROSEN, J., FENG, S., HAUSER, C., KRUEGEL, C., AND VIGNA, G. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)* (2016), IEEE, pp. 985–999.
- [198] SONG, C., MOON, H., ALAM, M., YUN, I., LEE, B., KIM, T., LEE, W., AND PAEK, Y. Hdfi: Hardware-assisted data-flow isolation. In *2016 IEEE Symposium on Security and Privacy (SP)* (2016), IEEE, pp. 1–17.
- [199] SONG, D., LETTNER, J., RAJASEKARAN, P., NA, Y., VOLCKAERT, S., LARSEN, P., AND FRANZ, M. Sok: Sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)* (2019), IEEE, pp. 1275–1295.
- [200] SPRUNT, B., SHA, L., AND LEHOCZKY, J. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems* 1, 1 (1989), 27–60.

- [201] STEINBERG, U., AND KAUER, B. Nova: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems (EuroSys)* (2010), ACM, pp. 209–222.
- [202] STEINBERG, U., WOLTER, J., AND HÄRTIG, H. Fast component interaction for real-time systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS)* (2005), IEEE Computer Society, pp. 89–97.
- [203] STEVANOVIC, M. Linux toolbox. In *Advanced C and C++ Compiling*. Springer, 2014, pp. 243–276.
- [204] SWITZER, J. F. Preventing ipc-facilitated type confusion in rust. Master’s thesis, Massachusetts Institute of Technology, 2020.
- [205] SZEKERES, L., PAYER, M., WEI, T., AND SONG, D. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy* (2013), IEEE, pp. 48–62.
- [206] TAN, G., APPEL, A. W., CHAKRADHAR, S., RAGHUNATHAN, A., RAVI, S., AND WANG, D. Safe java native interface. In *Proceedings of IEEE International Symposium on Secure Software Engineering* (2006), vol. 97, Citeseer, p. 106.
- [207] TICE, C., ROEDER, T., COLLINGBOURNE, P., CHECKOWAY, S., ERLINGSSON, Ú., LOZANO, L., AND PIKE, G. Enforcing {Forward-Edge}{Control-Flow} integrity in {GCC} & {LLVM}. In *23rd USENIX security symposium (USENIX security 14)* (2014), pp. 941–955.
- [208] TINO CAER. How microsoft is adopting rust, 2020. <https://medium.com/@tinocaer/how-microsoft-is-adopting-rust-e0f8816566ba>.
- [209] TRAN, M., ETHERIDGE, M., BLETSCH, T., JIANG, X., FREEH, V., AND NING, P. On the expressiveness of return-into-libc attacks. In *International Workshop on Recent Advances in Intrusion Detection* (2011), Springer, pp. 121–141.
- [210] TRUSTWORTHY SYSTEMS TEAM, DATA61. seL4 reference manual: Version 12.0.0, 2020.
- [211] 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)* (2019), pp. 1221–1238.
- [212] VAN LAMSWEERDE, A., DARIMONT, R., AND LETIER, E. Managing conflicts in goal-driven requirements engineering. *IEEE transactions on Software engineering* 24, 11 (1998), 908–926.

- [213] VANDERLEEST, S. H. The open source, formally-proven sel4 microkernel: considerations for use in avionics. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)* (2016), IEEE, pp. 1–9.
- [214] VESTAL, S. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *28th IEEE International Real-Time Systems Symposium* (2007), IEEE Computer Society, pp. 239–243.
- [215] WAGLE, P., COWAN, C., ET AL. Stackguard: Simple stack smash protection for gcc. In *Proceedings of the GCC Developers Summit* (2003), pp. 243–255.
- [216] 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* (1993), pp. 203–216.
- [217] WANG, H., WANG, P., DING, Y., SUN, M., JING, Y., DUAN, R., LI, L., ZHANG, Y., WEI, T., AND LIN, Z. Towards memory safe enclave programming with rust-sgx. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (2019), pp. 2333–2350.
- [218] WANG, Q., REN, Y., SCAPEROTH, M., AND PARMER, G. SPeCK: A kernel for scalable predictability. In *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2015), IEEE Computer Society, pp. 121–132.
- [219] WANG, Q., SONG, J., AND PARMER, G. Stack management for hard real-time computation in a component-based OS. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS)* (2011), IEEE Computer Society, pp. 78–89.
- [220] WANG, Q., SONG, J., PARMER, G., VENKATARAMANI, G., AND SWEENEY, A. Increasing memory utilization with transient memory scheduling. In *Proceedings of the 33rd IEEE Real-Time Systems Symposium (RTSS)* (2012), IEEE Computer Society, pp. 248–259.
- [221] WARD, B. C., SKOWYRA, R., SPENSKY, C., MARTIN, J., AND OKHRAVI, H. The leakage-resilience dilemma. In *European Symposium on Research in Computer Security* (2019), Springer, pp. 87–106.
- [222] WATSON, R. N., WOODRUFF, J., NEUMANN, P. G., MOORE, S. W., ANDERSON, J., CHISNALL, D., DAVE, N., DAVIS, B., GUDKA, K., LAURIE, B., ET AL. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy* (2015), IEEE, pp. 20–37.

- [223] WEI, N., AND SIM, S. Strengthening memory safety in rust: exploring cheri capabilities for a safe language. In *Master's Thesis: Wolfson College* (2020).
- [224] WEI, T., WANG, T., DUAN, L., AND LUO, J. Secure dynamic code generation against spraying. In *Proceedings of the 17th ACM conference on Computer and communications security* (2010), pp. 738–740.
- [225] WOODRUFF, J., WATSON, R. N., 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 *Proceeding of the 41st Annual International Symposium on Computer Architecture* (Piscataway, NJ, USA, 2014), ISCA '14, IEEE Press, pp. 457–468.
- [226] WOODRUFF, J., WATSON, R. N., 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)* (2014), IEEE, pp. 457–468.
- [227] XI, H., AND PFENNING, F. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation* (1998), pp. 249–257.
- [228] XU, J., DI BARTOLOMEO, L., TOFFALINI, F., MAO, B., AND PAYER, M. Warpattack: bypassing cfi through compiler-introduced double-fetches. In *2023 IEEE Symposium on Security and Privacy (SP)* (2023), IEEE, pp. 1271–1288.
- [229] 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), IEEE, pp. 79–93.
- [230] ZHOU, F., CONDIT, J., ANDERSON, Z., BAGRAK, I., ENNALS, R., HARREN, M., NECULA, G., AND BREWER, E. Safedrive: Safe and recoverable extensions using language-based techniques. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006), pp. 45–60.
- [231] ZHOU, Y., WANG, X., CHEN, Y., AND WANG, Z. Armlock: Hardware-based fault isolation for arm. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security* (2014), pp. 558–569.

- [232] ZINZINDOHOUE, J.-K., BHARGAVAN, K., PROTZENKO, J., AND BEURDOUCHE, B. Hacl*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), pp. 1789–1806.
- [233] ZUEPKE, A., AND KAISER, R. Deterministic futexes: Addressing WCET and bounded interference concerns. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2019), IEEE, pp. 65–76.