MIT Libraries | DSpace@MIT

MIT Open Access Articles

Manipulative Interference Attacks

The MIT Faculty has made this article openly available. ***Please share*** how this access benefits you. Your story matters.

Citation: Mergendahl, Samuel, Fickas, Stephen, Norris, Boyana and Skowyra, Richard. 2024. "Manipulative Interference Attacks."

As Published: https://doi.org/10.1145/3658644.3690246

Publisher: ACM|Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security

Persistent URL: https://hdl.handle.net/1721.1/158086

Version: Final published version: final published article, as it appeared in a journal, conference proceedings, or other formally published context

Terms of use: Creative Commons Attribution

Massachusetts Institute of Technology

Manipulative Interference Attacks

Samuel Mergendahl
University of Oregon
Eugene, OR, USA
smergend@cs.uoregon.edu

Stephen Fickas
University of Oregon
Eugene, OR, USA
fickas@cs.uoregon.edu

Boyana Norris
University of Oregon
Eugene, OR, USA
norris@cs.uoregon.edu

Richard Skowyra
MIT Lincoln Laboratory
Lexington, MA, USA
richard.skowyra@ll.mit.edu

Abstract

A μ-kernel is an operating system (OS) paradigm that facilitates a strong cybersecurity posture for embedded systems. Unlike a monolithic OS such as Linux, a μ-kernel reduces overall system privilege by deploying most OS functionality within isolated, userspace protection domains. Moreover, a μ-kernel ensures confidentiality and integrity between protection domains (*i.e.,* spatial isolation), and offers timing predictability for real-time tasks in mixed-criticality systems (*i.e.,* temporal isolation). One popular μ-kernel is seL4 which offers extensive formal guarantees of implementation correctness and flexible temporal budgeting mechanisms.

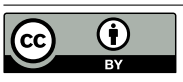
However, we show that an untrusted protection domain on a μ-kernel can abuse service requests to other protection domains in order to corrode system availability. We generalize this denial-of-service (DoS) attack strategy as *Manipulative Interference Attacks (MIAs)* and introduce techniques to efficiently identify instances of MIAs within a configured system. Specifically, we propose a novel hybrid approach that first leverages static analysis to identify software components with *influenceable* execution times, and second, uses an automatically generated model-based analysis to determine which compromised protection domains can *manipulate* the influenceable components and trigger MIAs. We investigate the risk of MIAs in several representative system examples including the seL4 Microkit, as well as a case study of seL4 software artifacts from the DARPA Cyber Assured Systems Engineering (CASE) program. In particular, we demonstrate that our analysis is efficient enough to discover practical instances of MIAs in real-world systems.

CCS Concepts

• **Computer systems organization** → **Embedded software**; **Real-time system specification**; • **Software and its engineering** → **Model checking**; **Automated static analysis**; **State systems**; • **Security and privacy** → **Denial-of-service attacks**.

Keywords

Systems, Denial-of-Service, Model Checking, Static Analysis



This work is licensed under a Creative Commons Attribution International 4.0 License.

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690246

ACM Reference Format:
Samuel Mergendahl, Stephen Fickas, Boyana Norris, and Richard Skowyra. 2024. Manipulative Interference Attacks. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690246

1 Introduction

While embedded systems traditionally must overcome strict size, weight, and power (SWaP) constraints, more recently, these systems have also begun to instate cybersecurity requirements [98] to protect the confidentiality, integrity, and availability (*i.e.,* the CIA triad [7]) of the system. In particular, some embedded systems, such as a Cyber-Physical System (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 [27, 111], with which attackers continue to identify new, advanced code reuse attacks that circumvent modern defenses [39, 130, 139]. For example, popular defenses to code reuse attacks, such as control-flow integrity [1, 131], address space randomization [11, 12, 64], run-time monitoring [23, 102], and stack protection [20, 26, 30, 59], all have been shown to be insufficient against a modern attacker [18, 43–45, 52, 115, 122]. Moreover, recent data-only attacks can achieve arbitrary execution without modifying the control flow of a program [22, 58, 61]. While memory-safe programming languages have the potential to mitigate the heart of code reuse attacks (*i.e.,* memory corruption) [3, 33], 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 [88]. Lastly, software supply chain attacks, such as the SolarWinds [32], NetPetya [31], or the recent backdoor found in the widely used compression utility, xz Utils [113], 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 [53, 87], or even automatically derived [28, 79, 83]. Moreover, these boundaries can be enforced with process isolation [15, 95], additional software checks [42, 94, 137], special hardware primitives [38, 87, 103, 112, 114, 133, 143], or constructs found in the programming language [13, 51, 73, 96, 142]—each with different trade-offs.

However, embedded systems with HRT tasks also require that 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 OS have also been proposed [8, 41].

One popular system design to provide both spatial and temporal isolation is a μ-kernel [76]. 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 [21, 48, 62, 71]. 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 [14, 40, 54, 75, 120, 127]. For example, seL4 [67] is a capability-based μ-kernel [37, 49, 123] that has seen extensive adoption in real-world systems [10, 55, 135] due to its unique and comprehensive guarantees of formal verification [66]. Namely, seL4 has formally proven its functional correctness (*i.e.,*seL4 correctly implements its specification [119]), its integrity and information flow security (*i.e.,*seL4 has no spatial side channels [91, 118]), and offers high assurance timing bounds and modern mixed-criticality scheduling mechanisms [84, 117].

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 [49, 74, 89, 105].

In this paper, we introduce a new type of denial-of-service (DoS) attack that we call **Manipulative Interference Attacks (MIAs)**. While DoS attacks on inter-component messaging are not new [78, 124], MIAs define 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, MIAs can cause devastating, critical system failures.

Because the types of interactions that lead to MIAs are potentially complex and easily overlooked, we present an analysis framework to automatically identify instances of MIAs 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 MIAs. 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 MIAs. 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 online[1].

Throughout this paper, we instantiate our framework on the seL4 μ-kernel, but the problem is fundamental to mixed criticality, co-resident execution contexts. In particular, MIAs—and our analysis framework—are extensible to any other system designs that attempt to temporally isolate co-resident system components.

The remainder of this paper is structured as follows. We first describe code-reuse attacks and their impact on real-time, mixed criticality systems in Section 2. We then propose a cyber-resilient threat model for embedded systems in Section 3. Following this, this paper makes several contributions:

- We introduce a new type of availability attack called Manipulative Interference Attacks (MIAs), that can cause deadline misses in embedded systems in Section 4.
- We create an analysis framework to automatically detect MIAs at compilation time in Section 5.
- We leverage a series of case studies to investigate the practicality of MIAs for real-world systems in Section 6.
- We evaluate our analysis framework and demonstrate that a risk analysis of MIAs is indeed efficient enough for compilation time analysis in Section 7.

2 Background

Memory corruption attacks and research into real-time schedulability have both had extensive amounts of published literature. A comprehensive treatment of both areas is beyond the scope of this paper. However, in this section, we provide an introductory background that focuses on topics required to understand the rest of the paper. For more information, interested readers should refer to surveys and systematization of knowledge papers in these areas

[1] https://github.com/smergendahl/mia

for a more thorough discussion on memory corruption attacks and defenses [17, 125, 126, 130], μ-kernels [24, 40], and mixed-criticality scheduling [14, 16].

2.1 Code Reuse Attacks

Memory corruption attacks have plagued computer systems for decades [36]. Early defense strategies such as Data Execution Prevention (DEP) [6] helped mitigate code injection attacks [100], but attacks evolved into more advanced code reuse techniques such as return-to-libc [132] and return-oriented programming (ROP) [121]. 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 [130]. For example, Control Flow Integrity (CFI) [1, 131] and stack defenses, such as stack canaries [30], shadow stacks [26], and SafeStack [20, 59], 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 [18, 44, 45]. In particular, in order for an enforcement policy to mitigate non-control, data-only attacks [22, 58, 61], it would need to be as fine-grained as Data Flow Isolation (DFI) [19] which is often considered too expensive for real-world deployment. Similarly, randomization techniques [11, 12, 64] are another type of run-time exploit mitigation, but can suffer from various forms of information-leakage attacks that limit their effectiveness [115, 122].

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 [86, 90]. Similar to sanitizers [126], 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 [63, 138]. Unfortunately, legacy code bases require incremental adoption of these languages for practical reasons, but incremental adoption has been shown to be ineffective at preventing code reuse attacks [88].

2.2 Component Isolation

Due to the prevalence of memory corruption [27, 111] and the aforementioned "Eternal War" in memory [130], 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. Often, security research 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 [53, 87], but automatically defined component boundaries have also been proposed [28, 79, 83].

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 [15, 95]. 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 [42, 94, 137]. This allows software components to remain in the same address space, but can sometimes incur prohibitive run-time costs.

However, software can rely on special hardware primitives to perform the needed checks at a lower cost [38]. For example, Intel Memory Protection Keys [60] have been shown to offer an intra-process isolation primitive [103, 133] which is especially helpful to help prevent corruption from spreading in multi-language applications [112]. Similar primitives have been proposed for AArch-32 [143], AArch-64 [87], and RISC-V [114] architectures.

As the lowest cost option, constructs found in the programming language can also act as an isolation primitive [51, 142]. In fact, there have been a number of proposals to pursue this strategy within an operating system [13, 73, 96]. While language-defined boundaries offer strong compile-time guarantees, they cannot prevent the spread of memory corruption at run-time.

2.3 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.

To constrain the temporal interference between components, modern isolation techniques often also build budget awareness into systems with mixed-criticality tasks [136]. For example, sporadic servers can ensure that periodic HRT tasks still safely execute by using deferrable servers that enforce execution budgets over fixed windows of time [127]. In particular, a thread's budget has an initial value that depletes corresponding to a thread's execution, and when a budget is exhausted, the thread suspends and awaits budget replenishment.

However, isolated components may request services from other isolated components through synchronous Inter-Process Communication (IPC) [75] which can complicate budget-management policy. If not provisioned carefully, IPC contention between multiple clients may impact the system's response times [124]. For example, when a server computes on behalf of a client, it can deplete its own budget or inherit budget from the client [84, 129]. The former can lead to budget attacks [78], 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 complicate the execution of the server's functionality. Additionally, IPC should process clients in priority order [84], and servers may leverage priority inheritance [128, 129] from clients. If the system does not adopt priority inheritance, the system designer must carefully assign priorities to a server as the ceiling of each potential client to follow the Immediate Priority Ceiling Protocol (I-PCP) [120]. Similarly, a shared server should never block [56].

A relatively new type of temporal interference has been shown to occur when a system service unexpectedly requires large amounts of processing for 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 [105]. Similarly, previous research identified that even kernel processing of execution budgets [89], system calls from another core [49], or virtualized network traffic [74] 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.

2.4 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 since the 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 [62].

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 [40, 75, 76]. Synchronous IPC mimics a function call such that the client thread blocks until the server thread returns.

One L4 μ-kernel variant that has seen significant adoption is seL4 [67]. Its popularity stems from its comprehensive formal guarantees of correctness [66]. In particular, seL4 has formally proven that its implementation correctly derives its specification [119]. In fact, this proof extends to the binary implementation, which means that seL4 is free of software bugs on multiple different architectures (*i.e.,*AArch-32, AArch-64, x86_64, and RISC-V). Moreover, seL4 has additional proofs for integrity and information flow [91, 118], as well as high assurance timing bounds [117]. 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 [108]). 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 [84]. Namely, IPC servers on seL4-MCS can inherit budget (but not priority) and seL4-MCS priority sorts IPC queues based on client priority.

2.5 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 [34, 35, 69]. *Goals* are prescriptive statements of how the system should behave that can guide the refinement of a specification and generate concrete implementation tasks [4]. However, goals themselves are often initially too ideal and require refinement [5]. In particular, the goal may overly assume the benevolence of a system component [34]. For example, in a μ-kernel with fixed priority scheduling, a system goal to have every thread scheduled within a timeslice may need to assume threads will actually yield their execution to the scheduler. Further, goals need inconsistency management in which the system must identify contradictory low-level requirements [93]. Because defining goals that *never* conflict with each other is difficult, a system may instead check if goals diverge [134]. *Divergent* goals are neither always contradictory nor always consistent, but instead, become inconsistent when certain boundary conditions occur. However, divergent goals may still be satisfactory for a system if the boundary conditions that lead to inconsistency are sufficiently unlikely or can be mitigated (*e.g.,* unlikely environmental factors, such as extreme heat, on a CPS that cause system deficiency).

One way to represent the divergence of goals is to use Linear Temporal Logic (LTL) [109]. LTL extends first-order logic to consider temporal statements. For example, first-order logic can analyze statements such as "if **A** is true, then **B** is true", whereas LTL allows for the analysis of statements such as "if **A** is currently true, then **B** will eventually be true at some point in the future". Notably, LTL includes a number of new operators: $\Box$ represents "always", $\Diamond$ represents "eventually", and $\bigcirc$ represents "in the next state", among others. Given a representative state machine model of the system, a system designer can define goals as LTL statements, and check if the LTL statement holds true in every state of the system model.

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 [34].

3 Threat Model

In this work, we consider an embedded system 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 separates 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

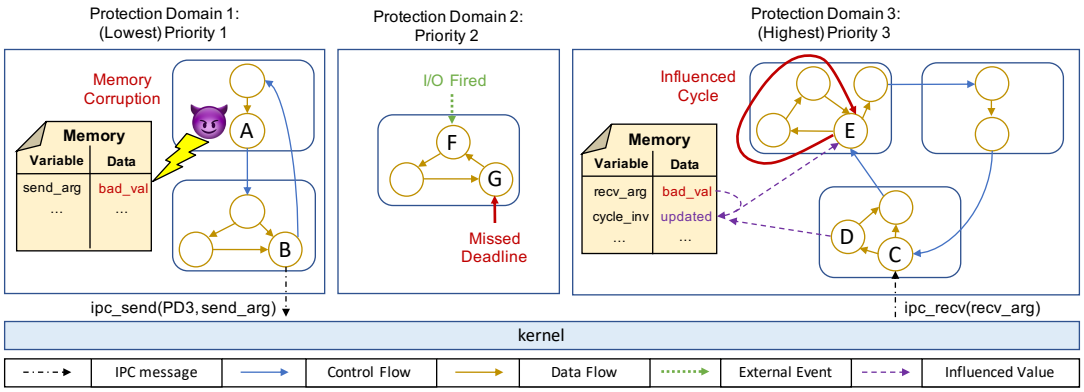


Figure 1: Manipulative Interference Attacks (MIAs) leverage a corrupt, low-criticality component to influence a higher priority component to cause interference on its behalf.

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 breaking system) and some of the application components may be less critical (*e.g.,* a vehicle infotainment system). In fact, such system functionality is often consolidated and co-resident in Cyber-Physical Systems (CPS) 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 [1] and Stack Canaries [30] or other stack protections. While many embedded systems may adopt a less-than-ideal CFI policy for performance reasons [131], we show that MIAs 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 delay 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.

4 Manipulative Interference

In this section, we introduce a new type of attack that we call Manipulative Interference Attacks (MIAs), 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, MIAs occur when a corrupt software component maliciously increases cycle iterations as a means to cause unexpectedly long processing at a high-priority component such that another critical component in the system exhibits long delays.

4.1 Overview of MIAs

We define MIAs 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 cycle invariant in the IPC receiver. Because IPC in a system must follow the Immediate Priority Ceiling Protocol (I-PCP) [56, 120], the priority of the manipulated component may be high enough to delay another critical component.

Figure 1 provides an overview of MIAs. 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**, *cycle_inv* is updated with respect to *bad_val*. Because *cycle_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. Unlike the classical budget drain attack [78], this interference occurs even without a dependency between PD2 and PD3. 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.

4.2 MIA Primitives

In order for MIAs 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 different from the traditional budget drain attack [78]: MIAs do not attempt to *drain* the budget of the manipulated server as a means to specifically prevent other components from accessing the service. Instead, MIAs do not require the victim have a dependency on the manipulated service. If the service has higher priority than the target victim, the service will interfere with the victim on behalf of the attacker. In particular, 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. However, an attacker could leverage a MIA to also drain a server's budget as a means to perform the traditional budget drain attack.

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 with an exit invariant that is dependent on input from an external source.
- **Triggerable Cycle:** This is a cycle which exists on a control or data flow path that directly follows after input from an external source.
- **Unbounded Cycle:** This is a cycle that 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.

5 Identifying MIAs

In this section, we describe a methodology to automatically identify when the primitives required to successfully execute a 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 MIAs, rather than solely relying on manual, technical expertise. Moreover, similar to the triage of software vulnerabilities, we must prioritize any instances of MIAs 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 accel and avoid common pitfalls where the tools may fail. 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 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 *automatically* generate the system model with readily available system build artifacts.

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 MIAs. 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 [47] to prioritize which MIA instances are more likely.

5.1 Cycle Detection

In the first stage of our analysis, as seen in Figure 2, 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 an 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. As a mature technique to track the influence of untrusted data, taint analysis is well-suited for our use-case. We mark the variable that holds the IPC message, and track which other variables this message impacts. When the

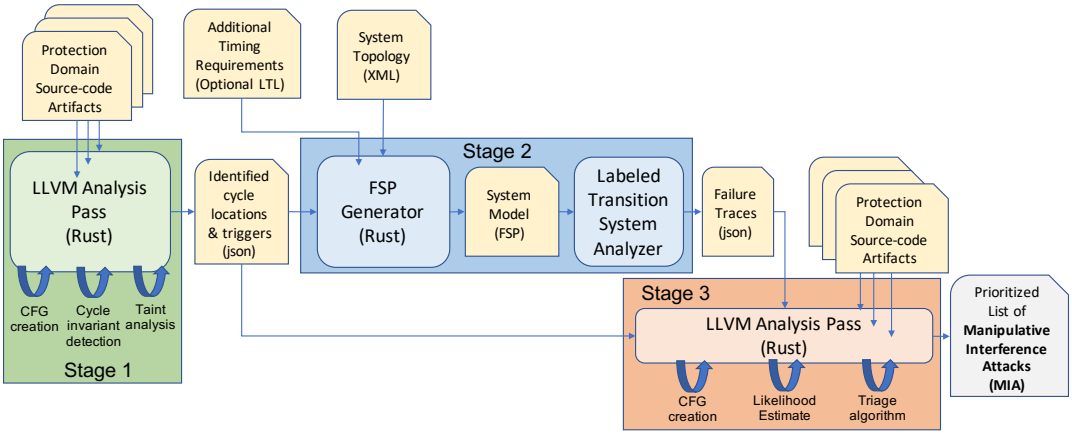


Figure 2: Our analysis first identifies cycles that can be manipulated and influenced, second automatically generates a model to verify LTL properties related to MIAs, and third triages the identified failures by those that are triggered by components with weak code reuse protection.

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.

5.2 Requirements Analysis

In the second stage of our analysis, as seen in Figure 2, 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) [85]. 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) [85]. 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*. FSP fluents track when certain system conditions occur and can be combined with logical operators to create more complex LTL formulas. LTSA then searches the possible execution paths for a conflict in the LTL formulas.

For example, we could define an FSP fluent as:

$$\texttt{fluent}\ pd_{blocked} = < \{wait\_on\_ipc,\ rec\_ipc\_msg\} > \quad (1)$$

This fluent becomes true when the action `wait_on_ipc` is taken, and transitions to false after the action `rec_ipc_msg`. With this fluent, we can further define an LTL formula:

$$\forall pd,\ \Box(pd_{blocked} \implies \Diamond \neg pd_{blocked}) \quad (2)$$

The LTSA tool can then check this LTL formula over all possible system traces. Namely, if this LTL assert holds, we know that for all ($\forall$) protection domains in the system (pd), it is always the case ($\Box$), that if a domain is blocked ($pd_{blocked} \implies$), it will eventually ($\Diamond$) receive a message and become unblocked ($\neg pd_{blocked}$).

Additionally, we *automatically* generate the formal FSP system model from the existing, informal 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 a formal model of the system. In particular, *FSPGen* transpiles the system configuration into a system model written in the FSP language in which the model tracks application state, such as remaining budget, and models system calls to seL4. Our front-end currently supports the XML system configuration used in the seL4 Microkit [72] and the system build artifacts from the DARPA CASE study [10, 55, 135]. Moreover, our back-end currently supports the original seL4 kernel and its seL4-MCS extensions. However, our tool is flexible enough to support additional systems in the future.

5.3 Triage

In the final stage of our analysis, as seen in Figure 2, we compute a stand-in metric for the likelihood of the identified failure found by LTSA. Of course, all identified instances of MIAs are attacks that should be addressed, but we take a hierarchical approach to triage instances of MIAs. 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. Specifically, we leverage the novel metric used to evaluate CFI policies, CFGInsulation, that quantifies how easy it is for an attacker to build a code-reuse exploit under CFI [47]. 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. Since MIA must send a maliciously crafted IPC message (*i.e.,* a system call),

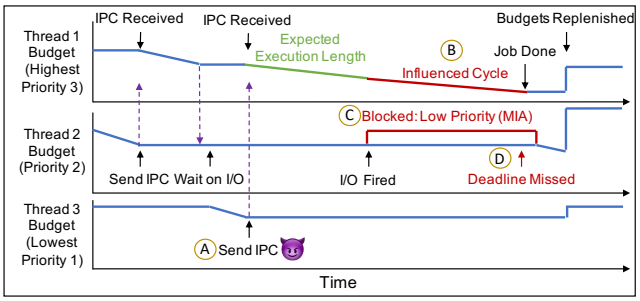


Figure 3: Mixed Criticality Scheduling used on seL4-MCS for Microkit offers better system utilization, but still cannot prevent MIAs.

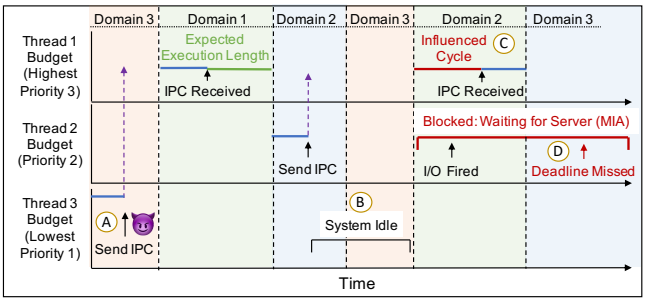


Figure 4: Domain Scheduling used on seL4 for the DARPA CASE program leads to complex IPC issues and ultimately MIAs.

CFGInsulation is a reasonable stand-in for the "likelihood" of an attack; the worse the CFGInsulation score for a component, the more likely the component could act erratically and trigger a MIA.

5.4 Analysis Limitations

In this section, we describe the current limitations of our analysis tool. Importantly, because MIAs are a new type of attack, our analysis prioritizes efficiently finding some instances of MIAs over comprehensively identifying all primitives in a configured system that indicate a possible MIA.

First, our static analysis discussed in Section 5.1 will not identify all cycle primitives in an implementation. For example, because we do not perform inter-procedural static analysis to discover cycles, we may miss cycles triggered across module boundaries. Because inter-procedural static analysis can bloat computational complexity, we choose to only identify cycles within a module boundary. However, as we show in Section 6.3, this is still sufficient to identify MIA primitives in real-world applications.

Similarly, our static analysis only identifies cycles in the data-flow graph within a single function, and does not consider cycles in the control-flow graph. In particular, a manipulated component that continuously traverses its control-flow graph could potentially invoke a similar effect. However, because Cyber-Physical Systems (CPS) have such strict timing requirements, an influenced data-cycle can often cause a deadline miss.

Third, our goal-conflict analysis described in Section 5.2 adopts a similar philosophy to prioritize efficiency over completeness. For example, our LTL analysis does not consider multi-hop communication. Similar to reflection-based distributed denial-of-service (DDoS) attack, a compromised component could *indirectly* manipulate a high-priority component and cause delay though an intermediary component. However, because such an analysis will become expensive, our tool currently will not identify these relationships.

Lastly, the *FSPGen* tool that generates a model for LTL analysis currently only supports generating a model for seL4-based systems. While MIAs may exist in other μ-kernels—or even other separation-based systems—we chose to initially study seL4 due to its popularity.

6 Case Studies

In this section, we introduce two systems built on seL4 and discuss how MIAs 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), so we investigate two popular choices for orchestration on seL4.

First, we describe the seL4 Microkit [72], 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 [10] that facilitates system deployment on the original seL4 kernel.

Each version of the kernel has different benefits, and moreover, offers different ways to provide temporal isolation, so we study the impact of MIAs in both situations. Specifically, the goal of this section is to discuss how MIAs can arise in both system designs, regardless of the temporal isolation strategy used. Correspondingly, we create a toy example system that suffers from MIAs, and discuss how neither system design can inherently prevent the attack, and instead, a system designer must check and account for MIAs in either case. In particular, Figure 3 and Figure 4 show our toy example system under mixed-criticality scheduling and domain scheduling, respectively. Afterwards, in Section 6.3, we investigate an seL4 system for MIAs. The studied toy example system and the automatically generated FSP model for each system can be found online, along with a demonstration of each vulnerable system.

6.1 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 [84], 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 facilitate IPC. Importantly to MIAs, systems built on Microkit deploy within protection domains scheduled with respect to priority and budget.

In Figure 3, 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. 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

```
fn protected(
  &mut self,channel: Channel,msg_info: MessageInfo
 ) -> Result<MessageInfo, Self::Error> {
  Ok(match channel {
   ASSISTANT => match
    msg_info.recv_using_postcard::<Request>() {
      Ok(req) => {
        let draft = {
          let mut buf = vec![0; req.draft_size];
            /* snip  */
        };
        let masterpiece = Masterpiece::complete(
                draft_height, draft_width, &draft);
        /* snip  */
      },
      Err(_) => {
        MessageInfo::send_unspecified_error(),
      }
   },
   _ => unreachable!(),
  })
}
```

(a) Influencable cycle on Artist IPC.

```
fn notified(
  &mut self, channel: Channel
 ) -> Result<(), Self::Error> {
  match channel {
   SERIAL_DRIVER => {
    while let Ok(b) = self.serial_client.read() {
      if let b'\n' | b'\r' = b {
        /* Forward message to Artist */
      } else {
       let c = char::from(b);
       if c.is_ascii() && !c.is_ascii_control() {
         if self.buffer.len() == MAX_SUBJECT_LEN {
          /* Forward message to Artist */
         }
         self.serial_client.write(b).unwrap();
         self.buffer.push(b);
       }
      }
    }
   }
   _ => unreachable!(),
  }
  Ok(())
}
```

(b) Triggerable cycle on Assistant wait.

Figure 5: The Banscii untrusted protection domains can cause MIAs.

higher-priority thread blocks (see **C** at Thread 2 in Figure 3). Following from research into MCS systems, Microkit can achieve high system utilization (*e.g.,* the system is never idle in Figure 3).

However, when a high-priority thread acts as an IPC server (in this case, Thread 2), a compromised thread has an opportunity to trigger MIAs. When Thread 3 sends a maliciously crafted IPC message to Thread 1 (see **A** in Figure 3), the message could influence a cycle at the receiver that triggers unexpectedly long processing time (see **B** in Figure 3). 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 **D** in Figure 3). It is important to note that

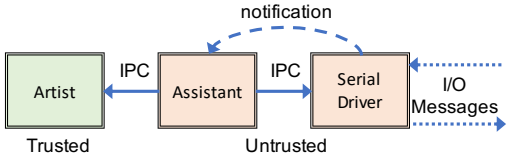


Figure 6: The system configuration for the Banscii application.

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.

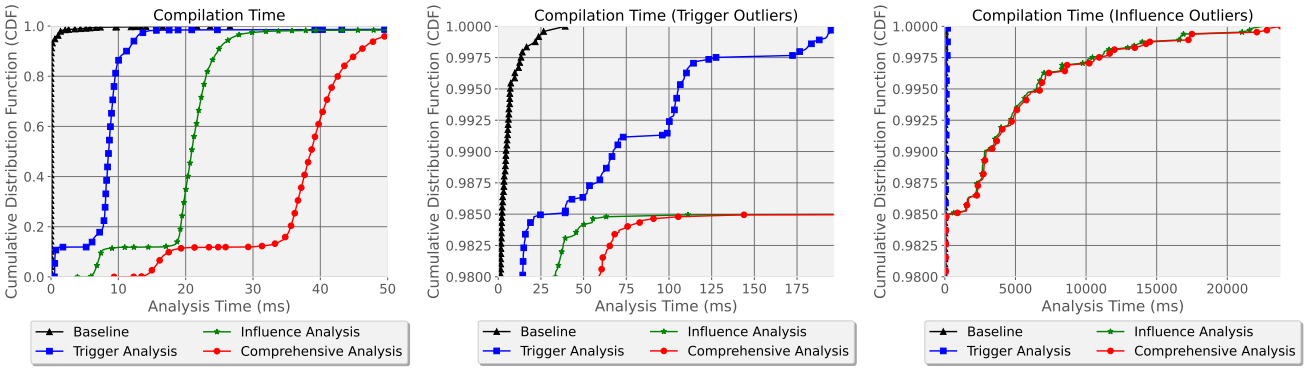
6.2 DARPA CASE

The DARPA CASE program [2] 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 [29, 81, 82] for system assurance. In particular, these tools realize cyber-resiliency requirements based on an initial Architecture Analysis and Design Language (AADL) model [46] in which components are automatically generated from formal specifications with verified system design properties, such as information flow. One of the tools, HAMR [10], automatically stitches the system together for the CAmkES seL4 configuration environment [68].

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 4, 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 which can lead to periods when the entire system is idle (see Figure 4 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 seL4-MCS, system IPC leads to an opportunity for compromised components to trigger MIAs. As before, when Thread 3 sends a maliciously crafted IPC message to Thread 1 (see **A** in Figure 4), its message can influence a cycle at the receiver that triggers an unexpectedly long processing time (see **C** in Figure 4) which ultimately leads to a missed deadline (see **D** in Figure 4). Moreover, under a semi-honest threat model, this system will again operate as desired, and meet all required deadlines.



(a) Full distribution. **(b) Outliers of the baseline LLVM tests.** **(c) Outliers of our MIA analysis.**

Figure 7: Cumulative Distribution Function (CDF) of our MIA LLVM analysis compared to the baseline LLVM benchmark test-suite.

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 [56]. 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.

6.3 Real-World Microkit Example

Next we investigate an real-world example system built on seL4 to illustrate how MIAs can arise in system design beyond our toy example. We study the Banscii application, found in the main seL4 repository [116], which serves as a representative application for using the Rust programming language on seL4. While Banscii is a text-based application, it operates as a canonical use-case for seL4: the system is split into untrusted and trusted components to safely handle the reception of potentially corrupted I/O messages, similar to safety critical applications, such as avionics.

Banscii includes three protection domains: a Serial Driver to consume I/O messages, an Assistant protection domain to batch and forward messages, and a trusted Artist component that cryptographically processes each message before returning the processed message out the serial link. With seL4 protection domains, these cryptographic operations can maintain spatial and temporal isolation from the untrusted serial interface. Figure 5a and Figure 5b contain portions of the Artist and Assistant implementations respectively, and Figure 6 provides an overview for the system architecture. However, using our analysis tool described in Section 5, we can automatically identify that this system design is vulnerable to MIAs. In particular, we next describe two examples of cycles identified in this application as primitives for MIAs.

First, in Figure 5a, we highlight the function `protected` that is called when the trusted Artist component receives an IPC message. Because the Assistant component is untrusted, a memory corruption within the Assistant component could cause corrupted data to be sent to the Artist. After message deserialization (on line 6), the `req` variable holds the received IPC value. Using our taint analysis, we learn that `req` also influences the size of the `buf` variable (see line 9) whose allocation contains a cycle dependent on the size of `buf`. Moreover, our cycle analysis identifies another concealed cycle within the `Masterpiece::complete` function (line 12), also dependent on the size of `buf`. Thus, due to the absence of sanitization on the size of `req.draft_size`, a memory corruption in Assistant can lead to influenceable execution lengths at the Artist. While CPS can sometimes withstand some system delay, these influenceable cycles in Artist can cause multiple seconds of delay, which is often enough to cause mission failure.

Second, Figure 5b highlights the `notified` function called at the Assistant when the Serial Driver receives a new message. Our analysis identified the `while` loop on line 28 as a triggerable cycle, but additionally, this loop may subtly never exit. Namely, if data continues to arrive, but neither of the `if` conditions are met on lines 29 and 33, the Assistant will silently remain in this loop and never forward the processed message back out the serial link, causing potentially missed deadlines. In particular, because the Serial Driver is untrusted, the data received in `self.serial_client.read()` may become corrupted and lead to such a situation. Because an attacker can cause this loop to never exit, this example of MIA is clearly enough to cause mission failure. Interested readers can find a demonstration of these analysis and their system delay online.

7 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 MIAs 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. To inspire others to find novel methodologies to identify new instances of MIAs, or improve our analysis times, we make our tool (and evaluation) available online.

7.1 Static Analysis

There are two places where we leverage static analysis to identify MIAs. First, in Stage 1 of our tool, we leverage the LLVM compiler infrastructure to find influenceable, triggerable, and unbounded

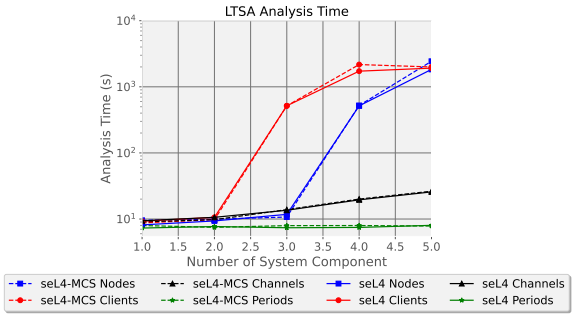


Figure 8: LTL analysis is exponential with respect to the number of protection domains and the number of clients to a IPC server.

cycles. Additionally, unlike the original CFInsulation metric [47], we analyze LLVM IR rather than the binary for our triage analysis.

Figure 7 shows a Cumulative Distribution Function (CDF) of our analysis times compared to the baseline LLVM test-suite used by the compiler infrastructure project [80, 110]. In particular, Figure 7a 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 triggerable analysis occurs under 35*ms*. While compilation times should be fast, we believe 35*ms* is indeed reasonable to identify primitives for MIAs. Moreover, in Figure 7b, we see that even the highly optimized analysis times for vanilla LLVM tests can also reach this length of analysis times. However, in Figure 7c, we see that roughly 1% of the time, our analysis can trigger exceptionally long analysis times: MIA analysis can take up to 25*s*. In particular, for the top 100 test times in Figure 7c, influenceable analysis is the cause of decrease in performance. As such, a system designer could increase analysis speed and only identify triggerable, but not influencable cycles. Notably, if a MIA only invokes a triggerable cycle (but not necessarily an influenceable cycle) missed deadlines can still arise in the system.

7.2 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 MIAs. Namely, we create and benchmark 40 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, length of execution period, and *degree* of each IPC channel (*i.e.,* the average number of clients to a particular IPC server) from 1 to 5, with an analysis timeout of 3600*s*, or one hour. We pause to highlight the success of our *FSPGen* tool. Manually generating 40 different models of 40 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 8, we load each model into LTSA, and benchmark analysis time. First, all tests completed within an hour, and all systems with 3 or less protection domains require roughly 10s 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, but budget and period management also only scales linearly.

8 Discussion and Future Work

In this section, we first discuss several potential extensions to our analysis tool to increase its usability as well as ways to identify additional instances of MIAs. Second, we enumerate a few ways to mitigate MIAs, and finally, we discuss how to leverage MIAs to perform other types of attacks, such as creating a covert channel between two isolated components.

8.1 Potential MIA Analysis Extensions

While our MIA analysis can study systems built on top of either the original seL4 kernel or on the seL4-MCS extensions to the kernel, MIAs are not unique to seL4 systems. Other μ-kernels, such as Composite [49, 104] or NOVA [128], could also suffer from MIAs. Similarly, as shown in Section 6, our tool can ingest both the XML system configuration for Microkit [106] and the tool-chain developed on the DARPA CASE program [10, 55, 135], but our tool is also flexible enough for other operating systems built on top of seL4 to input their unique system configuration (*e.g.,* other systems built on the CAmkES system configuration for seL4 [68]).

However, as mentioned in Section 5, our analysis operates with a philosophy that reduces false positives: if our tool identifies an instance of MIAs, we have confidence that we have indeed found a possible attack. This philosophy necessitates that our tool will not identify some instances of MIAs (and instead, it will exhibit some false negatives). In order to expand the types of MIAs we can identify, there are several possible extensions to our analysis tool. For example, we could perform more complex static analysis during the first stage of our tool to identify other types of potentially influenceable cycles. For example, our tool currently identifies natural loops that are influenceable, but other program-flow cycles, such as cycles in control-flow, could also trigger MIAs. Similarly, at the system configuration level, our tool automatically generates only a few LTL statements indicative of MIAs. While a system designer can manually input additional LTL statements to the second stage of our tool to verify, we could extend our FSP generation to automatically check additional, more expressive goals. For example, we could look for special cases of MIAs when multiple threads are involved, similar to the Thundering Herd Attacks [89].

Lastly, while our analysis tool is likely sufficient for most embedded system deployments (*e.g.,* microcontrollers), there are likely optimizations to our analysis tool. However, as mentioned in Section 5.4, there is a delicate balance between performance scalability and detection completeness. We believe future work should further investigate the trade-off of scalability and attack detection.

8.2 Defensive Techniques to MIAs

MIAs are difficult to prevent because it takes advantage of *approved* communication pathways in the system. The compromised component manipulates a high priority component with which it specifically maintains communication access. This is not an unreasonable situation (*e.g.,* an OS patina on top of a μ-kernel [62]), and simply removing the communication pathway is not always acceptable. In fact, because the attack derives from a high priority server overly trusting the content of its received messages, MIAs share similarities to the well-known **Confused Deputy** attack in which a privileged component is tricked into misusing its authority in a system. For example, CIVSCOPE [25] and the Dereference Under the Influence (DUI) introduced by Hu, *et al.* [57] demonstrate that the confused deputy attack can influence memory operations across component boundaries. Future work should consider if defenses to the confused deputy can apply to preventing MIAs. For example, if the system component deploys proper sanitization on received messages [70] (*e.g.,* checking against a range of approved values), a system designer may be able to mitigate MIAs.

Similarly, a shared system service could eliminate any unbounded, triggerable, or influenceable cycles to prevent MIAs. In particular, Worst-Case Execution Timing (WCET) analysis is a common technique to derive the timing of a system component that can feed into the schedulability analysis of the system. However, much work on mixed-criticality scheduling is exactly motivated by the fact that determining task worst-case execution times is notoriously difficult in dynamic systems, especially on modern complex architectures. Moreover, WCET analysis often annotates loop bounds using the expected behavior of the system rather than under a malicious trust model. Instead, a system could offer a weaker guarantee that simply removes the correlation of cycle lengths from received IPC messages to sufficiently prevent MIAs.

In addition to prevention of MIAs, a CPS should also include detection and response mechanisms [65]. For example, because a CPS often maintains physical inertia with the surrounding environment, cyber-resiliency techniques such as BFT++ [65] or YOLO [97] could allow the CPS to tolerate some execution delay caused by MIAs. Moreover, if a MIA triggers excessive delay and many deadline misses, a CPS can further deploy watchdog timers to catch and respond to task starvation caused by a MIA. These resiliency techniques should be further explored with respect to MIAs.

8.3 Covert Channels

While this work primarily considers MIAs as an attack vector to cause delay at highly critical software components, MIAs could also create a covert channel between two components that should not have communication access. Namely, a low-criticality component could create varying lengths of delay at a second, colluding component with which it shares no communication paths. Since the second component can record the lengths of delay it exhibits, a covert channel would exist between the two components which could degrade the confidentiality of the system. However, we leave the size of such a communication channel to future work.

9 Related Work

The closest related work to MIAs are those works that manually identify cases where a low-criticality component can cause unusually large amounts of processing elsewhere in the system [49, 74, 89, 105]. For example, Patel, *et al.* demonstrate that system services that manage timers should carefully process timer events based on priority; otherwise, processing a timer that corresponds to a low-priority thread can cause priority interference for higher-priority threads [105]. Correspondingly, Mergendahl, *et al.* show that the budget replenishment and IPC priority sorting mechanisms of the seL4-MCS μ-kernel can themselves be used as an attack vector when a large number of malicious threads exist [89]. Moreover, Gadepalli, *et al.* discuss that the big kernel lock used by seL4 also causes μ-kernel processing that can delay high-priority components of the system (specifically, components that reside on another core [49]). Each of these works suggests that high priority processing on behalf of low-priority components can delay other high-priority components. However, in this work, we generalize this phenomenon and suggest a methodology to automatically identify instances of manipulative behavior, rather than rely on complex and manual system technical expertise.

Micro-architectural denial-of-service (DoS) attacks are another type of attack, similar to MIAs, that can delay hard real-time (HRT) tasks [9, 92]. Rather than target shared software components like MIAs, these attacks target shared hardware resources, such as DRAM or system caches to cause delay. Moreover, significant work has been done to mitigate these threats, such as bounding memory access [107, 141] or adding awareness of these DoS attacks to the memory allocator [77, 101, 140]. In fact, the seL4 μ-kernel has worked to prevent these threats [50].

In addition to our work, there have also been several works that leverage Linear Temporal Logic (LTL) [109] to perform goal-conflict analysis [5, 69, 134] and identify divergent or inconsistent goals in a system [93]. For example, previous work has also automatically searched for boundary conditions on goal expressions using LTL [35, 134]. Moreover, rather than manually input the likelihood of conflicts, Degiovanni, *et al.* exploit string model counting techniques to automatically estimate the likelihood of each identified boundary condition [34]. However, to the best of our knowledge, our analysis is the first to apply goal-oriented requirements analysis to automatically identify inconsistencies of goals (and their likelihood) specific to mixed-criticality scheduling (*i.e.,* the budget and priority assignment in a system) under a malicious threat model.

Similar to our work, system configuration written in formal languages, such as AADL [46], has previously been shown to facilitate automatic model-based system analysis [10, 99]. However, to the best of our knowledge, our work is the first to *derive* the formal system configuration required for such analysis from the informal system configuration more commonly found in real-world systems.

10 Conclusion

In this work, we introduced a new type of attack 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. We call these attacks Manipulative Interference Attacks (MIAs), and propose a methodology to automatically identify instances of MIAs given readily available system build artifacts. Our analysis takes a hybrid approach that combines static analysis with goal-oriented, conflict analysis to identify where conflict can arise in the budget and priority assignments of a mixed-criticality system when a low-criticality component becomes 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 MIAs.

References

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)* 13, 1 (2009), 1–40.

[2] Defense Advanced Research Projects Agency. [n.d.]. Memory Safety. https://www.darpa.mil/program/cyber-assured-systems-engineering

[3] National Security Agency. 2023. Software Memory Safety. (2023).

[4] Dalal Alrajeh, Jeff Kramer, Alessandra Russo, and Sebastin Uchitel. 2009. Learning operational requirements from goal models. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 265–275.

[5] Dalal Alrajeh, Jeff Kramer, Axel Van Lamsweerde, Alessandra Russo, and Sebastián Uchitel. 2012. Generating obstacle conditions for requirements completeness. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 705–715.

[6] Starr Andersen. 2004. Changes to functionality in Microsoft Windows XP service pack 2. *Microsoft technical document, August* (2004).

[7] James P Anderson. 1972. Information security in a multi-user computer environment. In *Advances in Computers*. Vol. 12. Elsevier, 1–36.

[8] Nils Asmussen, Sebastian Haas, Adam Lackorzyński, and Michael Roitzsch. 2024. Core-Local Reasoning and Predictable Cross-Core Communication with M 3. In *2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 199–211.

[9] Michael Bechtel and Heechul Yun. 2019. Denial-of-service attacks on shared cache in multicore: Analysis and prevention. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 357–367.

[10] Jason Belt, John Hatcliff, John Shackleton, Jim Carciofini, Todd Carpenter, Eric Mercer, Isaac Amundson, Junaid Babar, Darren Cofer, David Hardin, et al. 2023. Model-driven development for the seL4 microkernel using the HAMR framework. *Journal of Systems Architecture* 134 (2023), 102789.

[11] Sandeep Bhatkar, Daniel C DuVarney, and Ron Sekar. 2003. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits.. In *USENIX Security symposium*, Vol. 12. 291–301.

[12] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. 2015. Timely rerandomization for mitigating memory disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 268–279.

[13] Kevin Boos and Lin Zhong. 2017. Theseus: A state spill-free operating system. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*. 29–35.

[14] Björn B. Brandenburg. 2011. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. Ph.D. Dissertation. The University of North Carolina at Chapel Hill.

[15] David Brumley and Dawn Song. 2004. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium*, Vol. 57.

[16] A. Burns and R. Davis. 2013. *Mixed Criticality Systems - A Review*. Technical Report. Department of Computer Science, University of York.

[17] Nathan Burow, Xinping Zhang, and Mathias Payer. 2019. SoK: Shining light on shadow stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 985–999.

[18] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. 2015. {Control-Flow} bending: On the effectiveness of {Control-Flow} integrity. In *24th USENIX Security Symposium (USENIX Security 15)*. 161–176.

[19] Miguel Castro, Manuel Costa, and Tim Harris. 2006. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th symposium on Operating systems design and implementation*. 147–160.

[20] Gang Chen, Hai Jin, Deqing Zou, Bing Bing Zhou, Zhenkai Liang, Weide Zheng, and Xuanhua Shi. 2013. Safestack: Automatically patching stack-based buffer overflow vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* 10, 6 (2013), 368–379.

[21] J Bradley Chen and Brian N Bershad. 1993. The impact of operating system structure on memory system performance. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*. 120–133.

[22] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. 2005. Non-Control-Data Attacks Are Realistic Threats.. In *USENIX Security Symposium*, Vol. 5.

[23] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Robert H Deng. 2014. ROPecker: A generic and practical approach for defending against ROP attack. (2014).

[24] WH Cheung and Anthony HS Loong. 1995. Exploring issues of operating systems structuring: from microkernel to extensible systems. *ACM SIGOPS Operating Systems Review* 29, 4 (1995), 4–16.

[25] Yi Chien, Vlad-Andrei Bădoiu, Yudi Yang, Yuqian Huo, Kelly Kaoudis, Hugo Lefeuvre, Pierre Olivier, and Nathan Dautenhahn. 2023. CIVSCOPE: Analyzing Potential Memory Corruption Bugs in Compartment Interfaces. In *Proceedings of the 1st Workshop on Kernel Isolation, Safety and Verification*. 33–40.

[26] Tzi-cker Chiueh and Fu-Hau Hsu. 2001. RAD: A compile-time solution to buffer overflow attacks. In *Proceedings 21st International Conference on Distributed Computing Systems*. IEEE, 409–417.

[27] Catalin Cimpanu. 2019. Microsoft: 70 percent of all security bugs are memory safety issues. https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/

[28] Abraham A Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. 2018. {ACES}: Automatic compartments for embedded systems. In *27th USENIX Security Symposium (USENIX Security 18)*. 65–82.

[29] Darren Cofer. 2022. CASE Overview: Cyber Assured Systems Engineering. *seL4 Summit* (2022).

[30] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. 1998. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks.. In *USENIX security symposium*, Vol. 98. San Antonio, TX, 63–78.

[31] Matteo Crosignani, Marco Macchiavelli, and André F Silva. 2021. Pirates without borders: The propagation of cyberattacks through firms' supply chains. *FRB of New York Staff Report* 937 (2021).

[32] CrowdStrike, Inc. 2021. 2021 Global Threat Report. https://go.crowdstrike.com/rs/281-OBQ-266/images/Report2021GTR.pdf

[33] US Cybersecurity et al. 2023. The Case for Memory Safe Roadmaps: Why Both C-Suite Executives and Technical Experts Need to Take Memory Safe Coding Seriously. (2023).

[34] Renzo Degiovanni, Pablo Castro, Marcelo Arroyo, Marcelo Ruiz, Nazareno Aguirre, and Marcelo Frias. 2018. Goal-conflict likelihood assessment based on model counting. In *Proceedings of the 40th International Conference on Software Engineering*. 1125–1135.

[35] Renzo Degiovanni, Nicolás Ricci, Dalal Alrajeh, Pablo Castro, and Nazareno Aguirre. 2016. Goal-conflict detection based on temporal satisfiability checking. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 507–518.

[36] Peter J Denning. 1989. The science of computing: The Internet worm. *American Scientist* 77, 2 (1989), 126–128.

[37] Jack B. Dennis and Earl C. Van Horn. 1983. Programming semantics for multiprogrammed computations. *Commun. ACM* 26, 1 (1983), 29–35. https://doi.org/10.1145/357980.357993

[38] Joe Devietti, Colin Blundell, Milo MK Martin, and Steve Zdancewic. 2008. Hardbound: architectural support for spatial safety of the C programming language. *ACM SIGOPS Operating Systems Review* 42, 2 (2008), 103–114.

[39] Victor Duta, Fabian Freyer, Fabio Pagani, Marius Muench, and Cristiano Giuffrida. 2023. Let Me Unwind That For You: Exceptions to Backward-Edge Protection. In *NDSS*.

[40] Kevin Elphinstone and Gernot Heiser. 2013. 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)*. ACM, 133–150.

[41] Dawson R Engler, M Frans Kaashoek, and James O'Toole Jr. 1995. Exokernel: An operating system architecture for application-level resource management. *ACM SIGOPS Operating Systems Review* 29, 5 (1995), 251–266.

[42] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C Necula. 2006. XFI: Software guards for system address spaces. In *Proceedings of the 7th symposium on Operating systems design and implementation*. 75–88.

[43] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. 2015. Missing the point (er): On the effectiveness of code pointer integrity. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 781–796.

[44] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control jujutsu: On the

weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 901–913.
[45] Reza Mirzazade Farkhani, Saman Jafari, Sajjad Arshad, William Robertson, Engin Kirda, and Hamed Okhravi. 2018. On the effectiveness of type-based control flow integrity. In *Proceedings of the 34th Annual Computer Security Applications Conference*. 28–39.
[46] Peter H Feiler, David P Gluch, and John Hudak. 2006. The architecture analysis & design language (AADL): An introduction. (2006).
[47] Tommaso Frassetto, Patrick Jauernig, David Koisser, and Ahmad-Reza Sadeghi. 2022. CFInsight: A Comprehensive Metric for CFI Policies.. In *NDSS*.
[48] Eran Gabber, Christopher Small, John L. Bruno, José Carlos Brustoloni, and Avi Silberschatz. 1999. Pebble: A component-based operating system for embedded applications. In *USENIX Workshop on Embedded Systems*. USENIX Association, 55–65.
[49] Phani Kishore Gadepalli, Gregor Peach, Gabriel Parmer, Joseph Espy, and Zach Day. 2019. Chaos: A System for Criticality-Aware, Multi-core Coordination. In *25th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 77–89.
[50] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. 2019. Time protection: The missing OS abstraction. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–17.
[51] Adrien Ghosn, Marios Kogias, Mathias Payer, James R Larus, and Edouard Bugnion. 2021. Enclosure: language-based restriction of untrusted libraries. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 255–267.
[52] Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. 2014. Size Does Matter: Why Using {Gadget-Chain} Length to Prevent {Code-Reuse} Attacks is Hard. In *23rd USENIX Security Symposium (USENIX Security 14)*. 417–432.
[53] Khilan Gudka, Robert NM Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G Neumann, and Alex Richardson. 2015. Clean application compartmentalization with SOAAP. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 1016–1031.
[54] Per Brinch Hansen. 1970. The nucleus of a multiprogramming system. *Commun. ACM* 13, 4 (1970), 238–241.
[55] David S Hardin and Konrad L Slind. 2021. Formal synthesis of filter components for use in security-enhancing architectural transformations. In *2021 IEEE Security and Privacy Workshops (SPW)*. IEEE, 111–120.
[56] Gernot Heiser. 2019. How to (and how not to) use seL4 IPC. https://microkerneldude.org/2019/03/07/how-to-and-how-not-to-use-sel4-ipc/
[57] Hong Hu, Zheng Leong Chua, Zhenkai Liang, and Prateek Saxena. 2015. Identifying arbitrary memory access vulnerabilities in privilege-separated software. In *Computer Security–ESORICS 2015: 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II 20*. Springer, 312–331.
[58] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 969–986.
[59] Kaiming Huang, Yongzhe Huang, Mathias Payer, Zhiyun Qian, Jack Sampson, Gang Tan, and Trent Jaeger. 2022. The taming of the stack: Isolating stack data from memory errors. In *NDSS*.
[60] Intel Intel. 64. and IA-32 architectures software developer's manual. *Volume 3A: System Programming Guide, Part* 1, 64 (64), 64.
[61] Kyriakos K Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. 2018. Block oriented programming: Automating data-only attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1868–1882.
[62] Samuel Jero, Juliana Furgala, Runyu Pan, Phani Kishore Gadepalli, Alexandra Clifford, Bite Ye, Roger Khazan, Bryan C Ward, Gabriel Parmer, and Richard Skowyra. 2021. Practical Principle of Least Privilege for Secure Embedded Systems. In *27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 1–13.
[63] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2020. Safe systems programming in Rust: The promise and the challenge. *Commun. ACM* (2020).
[64] Gaurav S Kc, Angelos D Keromytis, and Vassilis Prevelakis. 2003. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*. 272–280.
[65] David R Keppler, M Faraz Karim, Matthew S Mickelson, and J Sukarno Mertoguno. 2024. Experimentation and implementation of BFT++ cyber-attack resilience mechanism for cyber physical systems. *ACM Transactions on Cyber-Physical Systems* (2024).
[66] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems (TOCS)* 32, 1 (2014), 1–70.
[67] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 207–220.
[68] Ihor Kuz, Yan Liu, Ian Gorton, and Gernot Heiser. 2007. CAmkES: A component model for secure microkernel-based embedded systems. *Journal of Systems and Software* 80, 5 (2007), 687–699.
[69] A van Lamsweerde. 2009. *Requirements engineering: from system goals to UML models to software specifications*. John Wiley & Sons, Ltd.
[70] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Yi Chien, Felipe Huici, Nathan Dautenhahn, and Pierre Olivier. 2022. Assessing the impact of interface vulnerabilities in compartmentalized software. *arXiv preprint arXiv:2212.12904* (2022).
[71] Ben Leslie. 2006. GrailOS: A micro-kernel based, multi-server, multi-personality operating system. In *Workshop on Object Systems and Software Architectures (WOSSA 2006)*.
[72] Ben Leslie and Gernot Heiser. 2020. The sel4 core platform. *TS/sel4cp/2011-draft-spec.pdf* (2020).
[73] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 234–251.
[74] Chong Li, Xi Sisu, Lu Chenyang, Christopher D. Gill, and Roch Guerin. 2015. 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)*. IEEE, 95–107.
[75] Jochen Liedtke. 1993. Improving IPC by kernel design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP)*. 175–188.
[76] Jochen Liedtke. 1995. On μ-kernel construction. *ACM SIGOPS Operating Systems Review* 29, 5 (1995), 237–250.
[77] Jochen Liedtke, Hermann Hartig, and Michael Hohmuth. 1997. OS-controlled cache predictability for real-time systems. In *Proceedings Third IEEE Real-Time Technology and Applications Symposium*. IEEE, 213–224.
[78] Jochen Liedtke, Nayeem Islam, and Trent Jaeger. 1997. Preventing denial-of-service attacks on a µ-kernel for WebOSes. In *Proceedings of The Sixth Workshop on Hot Topics in Operating Systems (HotOS)*. IEEE Computer Society, 73–79.
[79] Shen Liu, Gang Tan, and Trent Jaeger. 2017. Ptrsplit: Supporting general pointers in automatic program partitioning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2359–2371.
[80] llvm compiler infrastructure project. [n.d.]. llvm-test-suite. https://github.com/llvm/llvm-test-suite
[81] Loonwerks. [n.d.]. CASE: Cyber Assured Systems Engineering. https://loonwerks.com/projects/case.html
[82] Loonwerks. [n.d.]. CASE-Final. https://github.com/loonwerks/CASE-Final/tree/main
[83] Kangjie Lu. 2023. Practical program modularization with type-based dependence analysis. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1256–1270.
[84] Anna Lyons, Kent McLeod, Hesham Almatary, and Gernot Heiser. 2018. Scheduling-context capabilities: A principled, light-weight operating-system mechanism for managing time. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 26:1–26:16.
[85] Jeff Magee and Jeff Kramer. 1999. *State models and java programs*. wiley Hoboken.
[86] Nicholas D Matsakis and Felix S Klock. 2014. The rust language. *ACM SIGAda Ada Letters* 34, 3 (2014), 103–104.
[87] Derrick Paul McKee, Yianni Giannaris, Carolina Ortega, Howard E Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. 2022. Preventing Kernel Hacks with HAKCs.. In *NDSS*. 1–17.
[88] Samuel Mergendahl, Nathan Burow, and Hamed Okhravi. 2022. Cross-Language Attacks.. In *NDSS*.
[89] Samuel Mergendahl, Samuel Jero, Bryan C Ward, Juliana Furgala, Gabriel Parmer, and Richard Skowyra. 2022. The thundering herd: Amplifying kernel interference to attack response times. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 95–107.
[90] Jeff Meyerson. 2014. The go programming language. *IEEE software* 31, 5 (2014), 104–104.
[91] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. 2013. seL4: From general purpose to a proof of information flow enforcement. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 415–429.
[92] Thomas Moscibroda Onur Mutlu. 2007. Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX security*.
[93] John Mylopoulos, Lawrence Chung, Brian Nixon, et al. 1992. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Transactions on software engineering* 18, 6 (1992), 483–497.
[94] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 245–258.

[95] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Retrofitting fine grain isolation in the Firefox renderer. In *29th USENIX Security Symposium (USENIX Security 20)*. 699–716.

[96] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. 2020. RedLeaf: Isolation and Communication in a Safe Operating System. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 21–39.

[97] Luyao Niu, Dinuka Sahabandu, Andrew Clark, and Radha Poovendran. 2022. Verifying safety for resilient cyber-physical systems via reactive software restart. In *2022 ACM/IEEE 13th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 104–115.

[98] National Institute of Standards and Technology. 2024. The NIST Cybersecurity Framework (CSF) 2.0. (2024).

[99] Peter Csaba Ölveczky, Artur Boronat, and José Meseguer. 2010. Formal semantics and analysis of behavioral AADL models in Real-Time Maude. In *International Conference on Formal Methods for Open Object-Based Distributed Systems*. Springer, 47–62.

[100] Aleph One. 1996. Smashing the stack for fun and profit. *Phrack magazine* 7, 49 (1996), 14–16.

[101] Shrinivas Anand Panchamukhi and Frank Mueller. 2015. Providing task isolation via tlb coloring. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 3–13.

[102] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. 2013. Transparent {ROP} exploit mitigation using indirect branch tracing. In *22nd USENIX Security Symposium (USENIX Security 13)*. 447–462.

[103] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. libmpk: Software abstraction for intel memory protection keys (intel {MPK}). In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 241–254.

[104] Gabriel Parmer. 2010. 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)*. 91.

[105] Pratyush Patel, Manohar Vanga, and Björn B. Brandenburg. 2017. TimerShield: Protecting High-Priority Tasks from Low-Priority Timer Interference. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE Computer Society, 3–12.

[106] Mathieu Paturel, Isitha Subasinghe, and Gernot Heiser. 2023. First steps in verifying the seL4 Core Platform. In *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems*. 9–15.

[107] Rodolfo Pellizzoni and Heechul Yun. 2016. Memory servers for multicore systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 1–12.

[108] Sean Peters, Adrian Danis, Kevin Elphinstone, and Gernot Heiser. 2015. For a microkernel, a big lock is fine. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*. 1–7.

[109] Amir Pnueli. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. ieee, 46–57.

[110] LLVM Compiler Infrastructure Project. [n.d.]. LLVM test-suite Guide. https://llvm.org/docs/TestSuiteGuide.html

[111] The Chromium Projects. [n.d.]. Memory Safety. https://www.chromium.org/Home/chromium-security/memory-safety/

[112] Elijah Rivera, Samuel Mergendahl, Howard Shrobe, Hamed Okhravi, and Nathan Burow. 2021. Keeping safe rust safe with galeed. In *Proceedings of the 37th Annual Computer Security Applications Conference*. 824–836.

[113] Kevin Rose. 2024. Did One Guy Just Stop a Huge Cyberattack? *The New York Times* (2024).

[114] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. 2020. Donky: Domain Keys–Efficient {In-Process} Isolation for {RISC-V} and x86. In *29th USENIX Security Symposium (USENIX Security 20)*. 1677–1694.

[115] Jeff Seibert, Hamed Okhravi, and Eric Söderström. 2014. 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*. 54–65.

[116] seL4 Foundation. [n.d.]. rust-microkit-demo. https://github.com/seL4/rust-microkit-demo

[117] Thomas Sewell, Felix Kam, and Gernot Heiser. 2016. 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)*. IEEE, 1–11.

[118] Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. 2011. seL4 enforces integrity. In *International Conference on Interactive Theorem Proving*. Springer, 325–340.

[119] Thomas Arthur Leck Sewell, Magnus O Myreen, and Gerwin Klein. 2013. Translation validation for a verified OS kernel. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 471–482.

[120] Lui Sha, Ragunathan Rajkumar, and John P Lehoczky. 1990. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on computers* 39, 9 (1990), 1175–1185.

[121] Hovav Shacham. 2007. 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*. 552–561.

[122] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*. 298–307.

[123] Jonathan Shapiro, Jonathan Smith, and David Farber. 1999. EROS: A fast capability system. In *17th ACM Symposium on Operating systems principles*. ACM, 170–185.

[124] Jonathan S. Shapiro. 2003. Vulnerabilities in Synchronous IPC Designs. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 251–262.

[125] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 985–999.

[126] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: Sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1275–1295.

[127] Brinkley Sprunt, Lui Sha, and John Lehoczky. 1989. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems* 1, 1 (1989), 27–60.

[128] Udo Steinberg and Bernhard Kauer. 2010. NOVA: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems (EuroSys)*. ACM, 209–222.

[129] Udo Steinberg, Jean Wolter, and Hermann Härtig. 2005. Fast Component Interaction for Real-Time Systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE Computer Society, 89–97.

[130] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 48–62.

[131] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing {Forward-Edge}{Control-Flow} integrity in {GCC} & {LLVM}. In *23rd USENIX security symposium (USENIX security 14)*. 941–955.

[132] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. 2011. On the expressiveness of return-into-libc attacks. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 121–141.

[133] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. {ERIM}: Secure, efficient in-process isolation with protection keys ({MPK}). In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1221–1238.

[134] Axel Van Lamsweerde, Robert Darimont, and Emmanuel Letier. 1998. Managing conflicts in goal-driven requirements engineering. *IEEE transactions on Software engineering* 24, 11 (1998), 908–926.

[135] Steven H VanderLeest. 2016. The open source, formally-proven seL4 microkernel: considerations for use in avionics. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*. IEEE, 1–9.

[136] Steve Vestal. 2007. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *28th IEEE International Real-Time Systems Symposium*. IEEE Computer Society, 239–243.

[137] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. 1993. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*. 203–216.

[138] Hongwei Xi and Frank Pfenning. 1998. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*. 249–257.

[139] Jianhao Xu, Luca Di Bartolomeo, Flavio Toffalini, Bing Mao, and Mathias Payer. 2023. Warpattack: bypassing cfi through compiler-introduced double-fetches. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1271–1288.

[140] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. 2014. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 155–166.

[141] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. 2013. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 55–64.

[142] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. 2006. SafeDrive: Safe and recoverable extensions using language-based techniques. In *Proceedings of the 7th symposium on Operating systems design and implementation*. 45–60.

[143] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. 2014. Armlock: Hardware-based fault isolation for arm. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*. 558–569.