Static Stack Allocation Analysis for Capability-Based Microkernel in Composite

by Ming-Hsien Wu

B.S. in Industrial Engineering and Industrial Management, June 2016, National Tsing Hua University
M.S. in Electrical Engineering, Jan 2019, National Taiwan University

A Thesis submitted to

The Faculty of
The School of Engineering and Applied Science
of the George Washington University
in partial fulfillment of the requirements
for the degree of Master of Science

May 18, 2025

Thesis directed by

Gabriel Parmer
Professor of Computer Science

© Copyright 2025 by Ming-Hsien Wu
All rights reserved

Acknowledgments

I would like to express my deepest gratitude to my advisor, Professor Gabriel Parmer, and the GWU Systems Lab. Their guidance transformed me from a rookie into someone who truly understands microkernel systems. They provided me with the opportunity to work hands-on with the Composite system, allowing me to engage in real-world research and refine my understanding of system software.

I am also sincerely thankful to Dr. Jie Zhou and my teammates in the Rust project. Their support helped me grasp the principles of static and dynamic analysis, greatly enhancing my knowledge in the field.

A heartfelt thank you goes to my beloved wife, Pei-Chun. Her unwavering support made it possible for me to pursue my studies here. She stood by my side for two years, let me complete my master's thesis. Without her encouragement and assistance, studying abroad would have been impossible.

I would also like to thank my roommate, Brenda, for supporting us as a fresh-off-the-boat couple and helping us navigate the challenges of living in a new country.

Lastly, I am grateful to my classmates and friends— Yao-Tian, Oscar, and Verna —who made my time at GWU an unforgettable experience. Their friendship and support enriched my academic journey and campus life.

Abstract

Static Stack Allocation Analysis for Capability-Based Microkernel in Composite

Efficient memory management is crucial in microkernel-based systems, where components operate in isolated environments. Composite, our microkernel system, currently allocates a large, fixed-size stack for each component and exposes all possible interfaces to its dependent components. While this approach simplifies implementation, it results in inefficient memory usage and assumes implicit trust among components, which can compromise system isolation and security.

To address these issues, we develop a binary code analyzer that detects stack usage and service dependencies within Composite. Our analyzer processes x86-64 binary executables, disassembling them into assembly code and simulating execution to track stack-related operations and function calls. By analyzing stack modifications and control flow instructions, we construct a Control Flow Graph (CFG) to map function relationships and identify inefficiencies such as excessive stack usage, recursion, and redundant service dependencies.

Our approach is built for flexibility, allowing it to support a diverse array of configuration files without requiring custom tuning for specific system environments. It analyzes the object files produced during the compilation of each component and aggregates the results to perform a comprehensive, system-wide dependency analysis. This enables precise identification of true synchronous invocations and the elimination of unnecessary service dependencies, ultimately enhancing resource efficiency and system isolation.

Our work makes the following contributions: A binary analyzer for stack usage in Composite, capable of detecting function dependencies and optimizing memory management. A method to track stack operations and control flow, including recursion detection and indirect call resolution. An integration with the Composer system to infer synchronous invocations and improve security isolation.

Table of Contents

Acknowledgments . . . . . **iv**

Abstract . . . . . **v**

List of Figures . . . . . **vii**

List of Tables . . . . . **viii**

1 Introduction **1**

1.1 Contribution 4

2 Motivation **5**

2.1 Definitions 5
2.2 Fixed Stack Allocation and Memory Overhead 7
2.3 Interface Exposure and Capability Mismanagement 8
2.4 Challenges Arising from These Inefficiencies 8
2.5 Proposed Solution 9

3 Background **11**

3.1 Configuration Analysis and Optimization in Operating Systems 11
3.2 Enhancing Patch Coverage and Kernel Configuration 12
3.3 Microkernel Design and Synchronous Mechanisms 13

4 Design **15**

4.1 Overview of Analysis Process 16
4.2 Instruction-Level Analysis 16
4.3 Implementation 17
4.4 Draw Back 19

5 Evaluation **22**

5.1 System Overview and Workflow 22
5.2 Experiment Setup 23
5.3 Capability Reduction Analysis 23
5.4 Stack Size Analysis 26

6 Discussion **29**

6.1 Recursive Function Analysis 29
6.2 Function Pointers and Dynamic Resolution 29
6.3 Link Time Optimization and Dead Code Elimination 30

7 Conclusions **32**

8 Future Works **33**

Bibliography **34**

List of Figures

1.1 Overview of Compsite build Process 3

2.1 Component-based 9

4.1 Composer Flow Chart 15

4.2 Ping Pong depedency 17

4.3 Steps of analyzer for Synchronous Invocation 18

4.4 Analyzer passes 21

5.1 Column chart for Reduction of synchronous invocations on each components 25

5.2 Thread Migration (adapted from Patina Paper [4]). 27

List of Tables

5.1 Reduction of synchronous invocations 24

5.2 Reduction of synchronous invocations on each components 26

5.3 Thread's Stack Size of each component 28

Chapter 1: Introduction

Modern microkernel-based operating systems decompose the traditional monolithic architecture into modular components, each responsible for specific functionalities. This design allows users to configure the system by including only the components they need, in contrast to monolithic systems, which require installing the entire operating system as a single package, often with numerous unused features. By supporting selective inclusion of components, microkernels provide greater flexibility, making them well-suited for a wide range of applications, from embedded systems to high-performance computing. However, this component-based approach introduces challenges in resource management and isolation, as components must function independently while still interacting efficiently to perform system-wide operations.

Each component runs within its own protection domain, which defines the resources it can access and the operations it is allowed to perform. This isolation ensures that faults or security vulnerabilities within one domain do not compromise others. To facilitate controlled inter-component communication, microkernels use capabilities, which are explicit authorization tokens that grant a protection domain permission to access a resource or service from another component. However, many system tasks require multiple components to cooperate, leading to synchronous invocations, where a consumer component requests services from a provider component via a predefined set of function pairs.

To leverage the benefits of isolation and modularity, researchers have increasingly partitioned system resources across different components, aiming to enhance stability and fault tolerance. In theory, component independence improves system reliability—if one component fails, it should not compromise others. However, in practice, many components require inter-component communication and resource sharing to complete tasks. This synchronous between components complicates resource management and security enforcement.

Since components request service dynamically, determining the exact resource requirements of an individual component in isolation becomes a complex challenge. In particular, the dynamic nature of component interactions makes resource allocation, stack size estimation, and capability management nontrivial.

In our component-based microkernel system, Composite, functionality is divided across independent components. While this modular design offers flexibility and extensibility, it also requires an effective mechanism to manage service access and resource allocation. In a system where all components are inherently trusted, broad access to services and resources may be acceptable. However, under an isolation-based security model, fine-grained control over capabilities becomes essential.

In Composite, we enforce capabilities as explicit sets of authorized operations or interfaces, which regulate synchronous invocations between components. This mechanism enhances security by preventing unauthorized access and limiting interactions to only what is explicitly permitted. Such enforcement aligns with the Principle of Least Privilege (PoLP)—a fundamental security concept stating that each user, process, or system component should be granted only the minimum permissions necessary to perform its intended function.

To facilitate system composition and configuration, we introduce Composer, an orchestration system designed to build and configure Composite based on user-defined specifications. Composer allows users to define custom system configuration files, specifying which components and services should be included. Through Composer, we control synchronous invocations, ensuring that components interact only within predefined capability constraints. By integrating synchronous control into the system composition process, we aim to disable capability of dependencies and improve security and resource efficiency.

To address these challenges, we developed a binary analysis and system-wide analysis tool for Composite to enhance resource management and control inter-component synchronous invocations. Specifically, we propose a system-wide approach to automatically

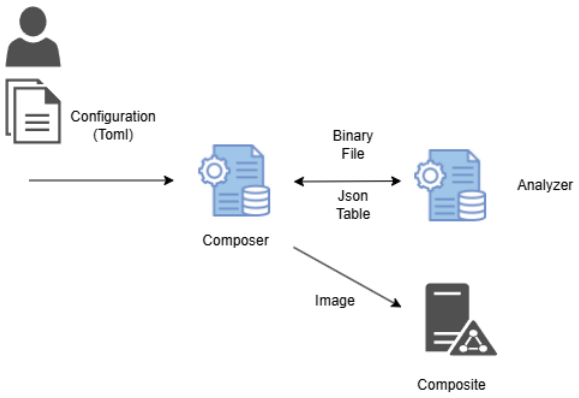


Figure 1.1: Overview of Compsite build Process

determine both the stack size requirements for threads and the capability requirements for inter-component dependencies. By analyzing system configuration files and performing binary-level analysis on each component, our tool aggregates the results to construct a global view of the system. This enables the generation of resource allocation recommendations, allowing users to optimize memory usage and enforce capability constraints with minimal manual effort.

Our approach is designed to be adaptable to a wide range of configuration files, ensuring broad applicability without the need for specialized tuning for specific system setups. The analyzer processes the object files generated during compilation for each component, and combines the analysis results across components to perform a system-wide dependency analysis. This enables us to accurately identify actual synchronous invocations and eliminate redundant service dependencies, leading to more efficient resource usage and improved system isolation.

Figure 1.1 provides a high-level overview of our system's workflow, illustrating the interaction between Composer, our binary analysis tool, and Composite.

1.1 Contribution

The primary contribution of our paper is the development of an analysis tool for Composite that identifies and evaluates the capability of dependencies between components. Our tool performs a system-wide analysis, aggregating results across all components. This enables us to detect unnecessary capabilities of dependencies, improve inter-component isolation, and accurately allocate stack sizes for each thread within each component. Our approach supports principled resource management and strengthens the enforcement of the Principle of Least Privilege in microkernel-based systems.

Chapter 2: Motivation

Resource management is a fundamental challenge in all operating systems, particularly in component-based microkernel systems, where components run in each protection domain but must still coordinate to provide complete system functionality. In Composite, our microkernel system, resource allocation and interface exposure are currently managed through static provisioning and broad service inclusion, leading to inefficient memory utilization, increased security risks, and unnecessary system complexity.

To address these issues, Composer, an orchestration system, processes user-defined configuration files to construct the kernel image for Composite. The configuration files specifies which components to include and defines the dependencies between them. For example, if the user specifies two components—Ping and Pong—and indicates that Ping calls Pong, Composer will include both components and establish a connection between these components in the kernel image. These configuration files is used to generate a tailored instance of Composite with the required services and dependencies.

Our analyzer integrates with Composer by consuming this configuration files to perform a system-wide level analysis on components of the system. It uses the specified component relationships to identify actual resource usage, eliminate unnecessary capabilities of dependencies, and guide capability enforcement based on real interactions rather than static assumptions.

2.1 Definitions

In this section, we define key terms used throughout the paper. These terms are commonly used in both Composite and our analysis, and are essential for understanding the system model and methodology. To ensure clarity and consistency, we summarize these definitions, allowing readers to reference them as needed.

1. Component – A component is the minimal functional unit in Composite. Each component operates within its own protected domain, isolating its resources from other components. This isolation enforces strong boundaries, ensuring that faults or unauthorized access do not propagate beyond the component's scope. Each Component is implemented as a separate binary.

2. Dependency – A dependency occurs when one component requires services from another component. In such cases, the requesting component is said to depend on the service-providing component. Dependencies define the interactions and communication paths between components in the system.

3. Interface – An interface is the communication boundary through which a service-providing component exposes functionality to requesting components. It defines the entry points that other components can call to access specific service-providing component. Interfaces serve as the interaction layer between two dependencies of components and help enforce protection domain, ensuring controlled and secure inter-component communication.

4. Protection Domain – A protection domain refers to the boundary around a component that isolates its resources—such as memory, execution context, and capabilities—from other components. Each component in Composite operates within its own protection domain, ensuring fault isolation and enforcing the Principle of Least Privilege.

5. Synchronous Invocation – A synchronous invocation, also known as a protected procedure call (PPC), occurs when a requesting component calls a service provided by another service-providing component. Two components have capability of dependencies will have synchronous invocation through the interface. This mechanism enables controlled, synchronous communication between components while maintaining protection domain boundaries.

6. Capability – A capability is an authorization token that grants a component the permission to access a specific interface or service provided by another component. When a requesting component attempts to invoke synchronous invocations from a service-providing component, Composite checks its capabilities to determine whether the call is permitted. Capabilities enforce fine-grained access control and are a core mechanism for maintaining security and isolation within the system.

7. Composer – Composer is an orchestration system used to build the final kernel image of Composite. Users define system structure and behavior through configuration files, specifying which components to include and how they interact. Based on this input, Composer generates a customized kernel image tailored to the provided configuration.

8. Configuration File – A configuration file is a user-defined input to Composer that specifies the set of components to include and their dependencies. It outlines how components interact and which services are required. Composer parses this file to generate a customized kernel image.

9. Principle of Least Privilege - The Principle of Least Privilege(PoLP) is a fundamental security concept that states that a user, process, or system component should only have the minimum permissions necessary to perform its function

2.2 Fixed Stack Allocation and Memory Overhead

One of the most pressing inefficiencies in Composite arises from fixed stack allocation. A thread of each component is statically assigned a fixed-sized stack, regardless of its actual memory requirements. While this simplifies implementation, it results in significant memory overhead—Threads in many components do not fully utilize their allocated stack space, while others may require larger stacks, leading to potential stack overflows or system failures. A more adaptive stack allocation strategy is required to reduce memory waste while ensuring system stability.

2.3 Interface Exposure and Capability Mismanagement

Another critical inefficiency in Composite arises from interface exposure and service provisioning. Currently, Composite does not fully leverage Composer to selectively configure inter-component dependencies. Instead, each component exposes all possible interfaces by default, which significantly increases the security risk by introducing unnecessary service dependencies. This broad exposure expands the system's attack surface, weakening the overall isolation and violating the Principle of Least Privilege.

Furthermore, excessive interface exposure undermines the capability-based security model. Granting overly broad authorizations to components allows them to request services from system components—even when those services are not required. This not only increases inter-component coupling, but also expands the attack surface, making privilege escalation more feasible. To address this, a more fine-grained capability management strategy is needed to enforce strict access controls and reduce implicit dependencies.

Figure 2.1 illustrates the concept of protected domains, a key principle in capability-based system design. Both applications and the operating system should be structured as clearly separated components, each providing a well-defined interface for offering services to others. By maintaining strict boundaries between components and carefully managing exposed interfaces, the system can achieve better security, isolation, and resource efficiency.

2.4 Challenges Arising from These Inefficiencies

The inefficiencies introduce several key challenges:

1. Memory Overhead – The static allocation of resources, including stack memory and service dependencies, leads to inefficient utilization of system resources.

2. Security Risks – The inclusion of excessive interfaces increases the attack surface, making privilege escalation. Unnecessary service dependencies introduce implicit trust relationships that could be exploited by adversaries.

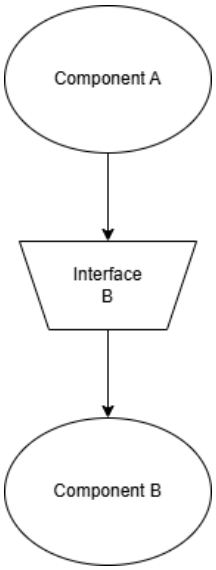


Figure 2.1: Component-based

3. System Complexity – The lack of fine-grained dependency tracking results in increased code complexity, making the system harder to maintain, analyze, and optimize. Excessive inter-component dependencies hinder performance tuning and security hardening efforts.

2.5 Proposed Solution

To address these challenges, we propose a system-wide analysis tool for evaluating memory utilization and inter-component dependencies in Composite. Our method performs direct analysis of ELF binaries for each component to obtain a precise, low-level perspective on stack usage and service dependencies. The tool analyzes the ELF object files generated by Composer, enabling it to reason about the system-wide structure and interactions between components. This analysis allows Composer to identify and disable unnecessary capabilities, thereby reducing the attack surface and improving resource efficiency and security.

This enables us to:

1. Identify stack size allocation by assessing component stack usage and assigning appropriate capabilities to components.

2. Detect service dependencies, minimizing unnecessary interface exposure and improving system security.

3. Enhance system maintainability by providing fine-grained resource dependencies.

By integrating binary-level and system-wide analysis techniques, our approach provides a mechanism for precise resource allocation, specifically determining stack requirements for threads within each component and disabling unnecessary capability dependencies between components. This enhances memory efficiency, system security, and configurability, making Composite more adaptable to a wide range of deployment scenarios with minimal manual intervention.

Chapter 3: Background

3.1 Configuration Analysis and Optimization in Operating Systems

Configuration parsing has been extensively explored in previous research. Reinhard et al. [10] address the challenge of analyzing and validating the complex configurations of the Linux kernel. Their approach slices the large Linux configuration space into smaller, SAT-solvable problems, enabling the determination of an effective configuration for a given Linux system. They developed a tool to detect and analyze defects within kernel configurations, specifically focusing on logical and symbolic inconsistencies between the configuration space and the implementation space. Additionally, they contributed to kernel stability by proposing patches to the Linux community to resolve these issues.

To further tackle the complexity of OS configuration tuning, Wayfinder [5] introduces an automated approach for exploring both configuration and implementation spaces. By iterating through configurations—such as variations in network stack size—Wayfinder executes build and test tasks, measures performance, and identifies optimal system settings. It employs three search strategies: Grid Search, Random Search, and Bayesian Optimization. Demonstrating its effectiveness, Wayfinder successfully optimizes NGINX performance on the Unikraft LibOS, highlighting the significant impact of configuration choices on system performance.

Another notable approach is HermiTux [6], a unikernel that provides binary compatibility with Linux applications, eliminating the need for porting. This unikernel model maintains security, fast boot time, and a low memory footprint, while still allowing native Linux executables to run efficiently. The authors compare HermiTux's performance with Linux, containers, and other unikernels, demonstrating minimal overhead and broad application support.

The EROS microkernel [9] exemplifies a capability-based system that integrates trans-

parent persistence. Contrary to the belief that capability-based systems are inherently slow, EROS achieves performance comparable to conventional systems like Linux in fundamental operations. It employs a single-level storage model, caching techniques, and optimized abstract objects (processes, nodes, pages) to enhance efficiency. The EROS kernel remains minimal and secure, managing only primitive capability types, while higher-level system services are delegated to non-privileged applications. This demonstrates that capability-based security can be both practical and efficient on commodity hardware.

CAmkES (Component Architecture for microkernel-based Embedded Systems) [8] is a framework built on top of the seL4 microkernel that enables component-based system design for embedded environments. It allows developers to define components, their interfaces, and communication channels using a declarative architecture description language (ADL). CAmkES supports strong isolation, inter-component communication, and compile-time system composition, making it well-suited for building secure and modular embedded systems. Like Composite, CAmkES emphasizes static system configuration and capability-based access control, but it also integrates with formal verification tools, reinforcing seL4's mission of providing provable software correctness and security guarantees.

3.2 Enhancing Patch Coverage and Kernel Configuration

To improve patch coverage in highly configurable software systems like the Linux kernel, krepair [11] introduces a novel approach for automating configuration adjustments. Traditional methods, such as maximal configurations or random configurations, often suffer from long build times or incomplete patch coverage. krepair addresses this issue by automatically repairing configuration files to ensure patch coverage without excessive compilation overhead. The system analyzes patch lines, gathers constraints, and iteratively modifies the configuration file using an automated theorem prover. Experiments on Linux kernel patches demonstrate that krepair achieves high patch coverage with minimal changes, resulting in faster build times compared to maximal configurations. This innovation enhances the

efficiency of continuous integration testing in large-scale software projects.

FreeRTOS [3] provides a configurable build system that allows developers to select only the necessary components at compile time, optimizing memory usage and performance for embedded applications. The configuration is managed through header files (FreeRTOSConfig.h), where users enable or disable features such as task scheduling policies, inter-task communication mechanisms, and memory management options. This compile-time customization ensures that only required functionalities are included, reducing system overhead. Similarly, Composite follows a user-defined configuration approach, where Composer orchestrates component selection before compilation, allowing fine-grained capability enforcement and resource management.

3.3 Microkernel Design and Synchronous Mechanisms

Ford and Lepreau [2] investigate the transition from static to migrating threads in the Mach 3.0 microkernel. They introduce a model where threads migrate between tasks during remote procedure calls (RPCs), improving efficiency and simplifying kernel design. Their work introduces a decoupled thread abstraction, distinguishing between execution contexts and schedulable threads. Key modifications include kernel-based threads and upcalls, which enhance RPC performance and simplify thread manipulation semantics, all while maintaining backward compatibility. Their approach is validated through a functional Unix single server and client implementation.

In the pursuit of secure and efficient embedded systems, Jero et al. [4] introduced "Patina," a novel RTOS API designed to bridge the gap between the security of minimal µ-kernels and the functionality of feature-rich operating systems. Patina aims to provide essential services like communication channels, timers, event management, and synchronization while adhering to the Principle of Least Privilege. The authors implemented Patina on two different µ-kernels, Composite and seL4, demonstrating its adaptability and exploring the trade-offs between security and performance. Their evaluation showed that

Patina achieved comparable or superior performance to Linux in many scenarios, while offering enhanced isolation and predictability. This work highlights the feasibility and benefits of building PoLP-based RTOS abstractions, crucial for securing increasingly complex embedded systems.

The paper [7] argues for the migrating thread model over synchronous IPC for predictable inter-process communication in configurable operating systems. It compares both models, focusing on execution accounting, communication end-points, and system customizability. The paper uses L4 and Composite as examples, highlighting the benefits of thread migration for resource management and accounting. It also addresses potential limitations and proposes enhancements to synchronous IPC inspired by the migrating thread model. Ultimately, it advocates for thread migration as a strong foundation for predictable and reliable systems.

Finally, Dennis and Van Horn [1] present foundational principles for concurrent processing in computer systems. They introduce meta-instructions for parallel execution, computation protection, and resource sharing, highlighting the importance of a robust supervisor for resource allocation and scheduling. Key concepts include segments for structured data storage, capabilities for access control, and processes as execution units. Their work significantly influences modern operating system security models, particularly in multiprogramming environments.

Chapter 4: Design

As discussed earlier, Composite, our microkernel-based operating system, utilizes Composer 4.1, an orchestration system, to configure and assemble the system based on user-defined configuration files. To enhance this process, we introduce a system-wide analysis tool that integrates with Composer, allowing for comprehensive, automated analysis of component interactions and resource usage. Our tool is incorporated as an additional pass within Composer, as shown in Figure 4.4.

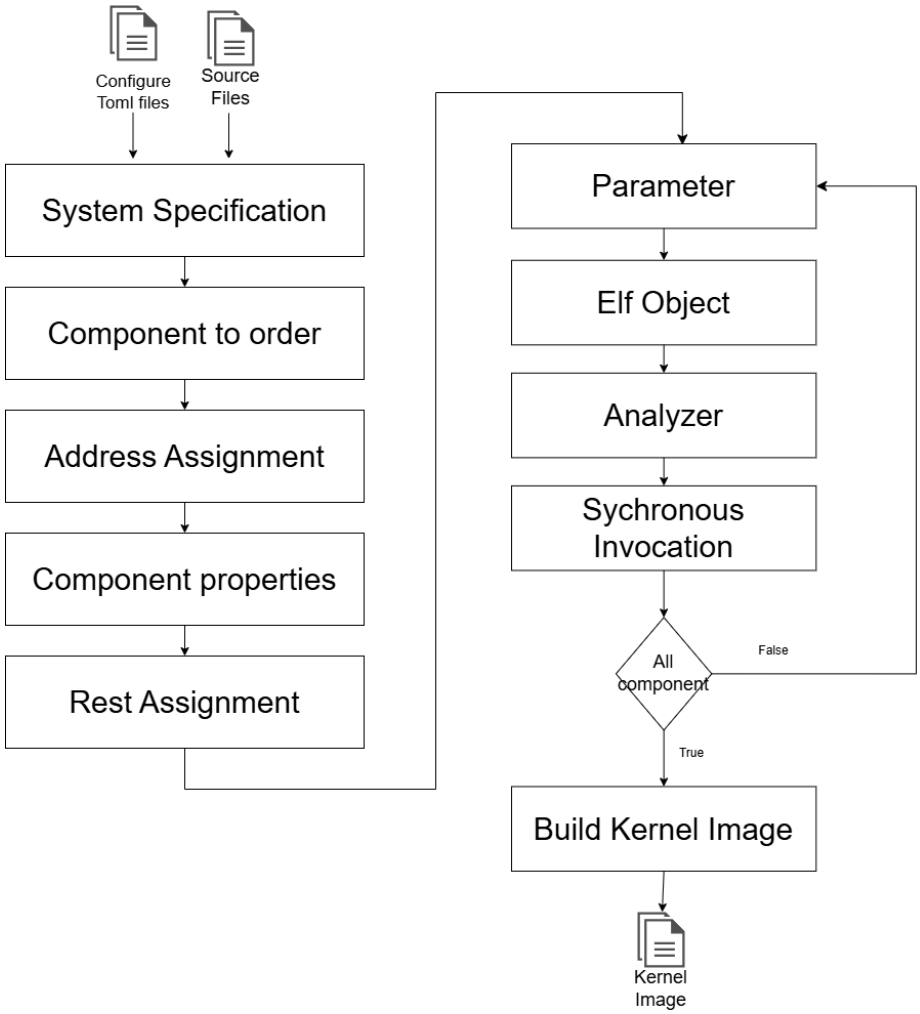


Figure 4.1: Composer Flow Chart

4.1 Overview of Analysis Process

Initially, each component is compiled within Composer using default values, generating ELF binary object files that are subsequently passed into our analyzer. These default values are typically conservative, encompassing all possible synchronous invocation combinations to ensure comprehensive coverage of potential interactions.

Given a component's binary file compiled by Composer and its entry function, the analysis begins at the "entry" function and proceeds in reverse order relative to the original compilation sequence. This reverse processing order is essential because it allows us to start from leaf components and trace synchronous entry points before constructing the capability set.

For example, Figure 4.2 illustrates a simple Composite system where the Ping component makes a synchronous invocation to the Pong component, while the Pong component provides the corresponding service. During compilation, the system is processed in the order: Timer $\rightarrow$ Pong $\rightarrow$ Ping. However, during analysis, we reverse this order, analyzing components in the sequence: Ping $\rightarrow$ Pong $\rightarrow$ Timer. This approach allows us to first identify synchronous invocations in the leaf nodes, ensuring an accurate representation of inter-component dependencies.

To facilitate this, we maintain a dependency table that is utilized in a subsequent Composer pass to construct the appropriate capability of dependencies table. If certain synchronous invocations are unreachable under a given configuration files, it indicates that these operations are unnecessary, and the corresponding capability of dependencies can be disabled, reducing system overhead and potential security risks.

4.2 Instruction-Level Analysis

For each component, we trigger our analyzer with the corresponding ELF binary file and process it instruction by instruction. Starting from the given entry function address, we:

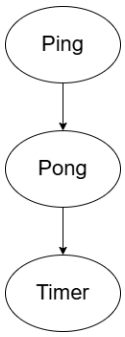


Figure 4.2: Ping Pong depedency

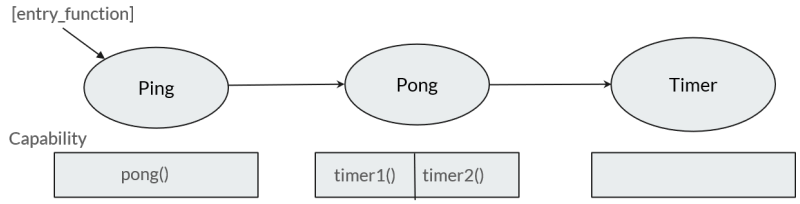
Fetch the next instruction and execute instruction to track control flow and stack usage. Analyze synchronous invocations, identifying dependencies between components. Update the capability of dependencies table for future passes in Composer. Once a component has been fully analyzed, the capability of dependencies table is updated to reflect its service dependencies. In the next iteration, if a consumer component invokes a service provided by another component, we incorporate the relevant synchronous functions into the entry function for further analysis.

By iteratively analyzing components in reverse compilation order, our tool construct precise capability of dependencies table, ensuring that only necessary operations are authorized while optimizing resource allocation and minimizing unnecessary inter-component dependencies.

Figure 4.3 illustrates the steps of this analysis process. We iterate through each component, collecting and aggregating capability of dependencies to build a comprehensive table. This enables Composer to reason about which synchronous invocations are unnecessary, allowing it to prune unused service calls and enforce Principle of Least Privilege more effectively.

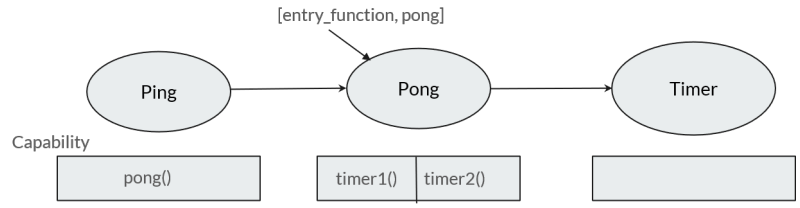
4.3 Implementation

Our analyzer is implemented in Python, leveraging its extensive library ecosystem to avoid unnecessary reinvention of existing functionalities. Specifically, we utilize Capstone, a



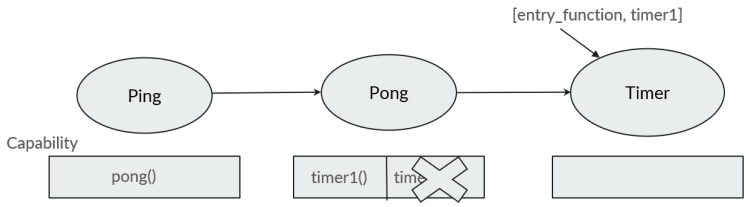
Iteration 1

(a) Iteration1



Iteration 2

(b) Iteration2



Iteration 3

(c) Iteration3

Figure 4.3: Steps of analyzer for Synchronous Invocation

lightweight disassembly framework, to facilitate instruction-level analysis. Capstone enables us to parse ELF binaries, extract relevant data from text and data sections, and disassemble instructions for further analysis.

We propose the algorithm in Algorithm 1, which follows a structured approach:

1. Fetch each instruction from the disassembled binary.
2. Decode the instruction to determine its type and operands.
3. Execute the instruction based on its impact on stack size and control flow.

Although we refer to this step as "execution," we do not fully emulate instruction behavior. Instead, we focus on stack size modifications and branch instructions, which are critical for understanding function calls and synchronous invocations.

1. If an instruction increases or decreases the stack size, we record the change to track memory usage dynamically.
2. If the instruction is a branch operation, we log it to identify function calls and synchronous invocations.

When a synchronous invocation is detected, we store the corresponding component capability of dependencies in a dictionary or map, mapping the invoking component to its target service provider. This data is used to construct a capability of dependencies table of inter-component dependencies, which is then incorporated into subsequent Composer passes for capability enforcement and resource optimization.

4.4 Draw Back

In our algorithm, handling function pointers presents a challenge. Since function pointers are dynamically determined at runtime, their targets cannot be resolved at compile time. However, because Composite is user-configurable, we can control which components are

Algorithm 1 Analyzer algorithm

Require: Elf binary and Entry Functions.

```
save_pc = stack()
pc = entry_fucntion.address
while (pc ≠ exit_pc) do
    fetch the instruction with pc
    decode the instruction
    execute the instruction
    if instruction is a call or jump instruction then
        save_pc.push(++pc)
        pc = jumporcalltargetdst
        continue
    else if instruction is return or out of bound then
        pc = save_pc.pop()
        continue
    end if
    pc = pc++
end while
```

included, allowing us to eliminate certain function pointer dependencies. Additionally, for function pointers that cannot be avoided, we hardcode a function pointer table within the analyzer to resolve known targets, ensuring accurate analysis.

We will further discuss this issue, along with its implications and possible solutions, in Sections 5 and 6.

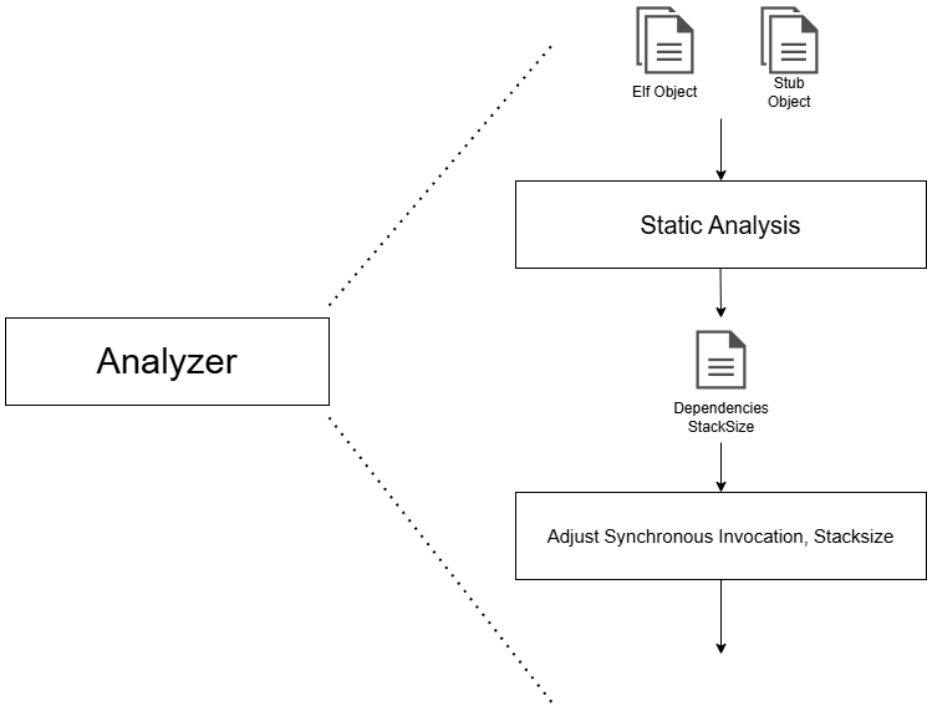


Figure 4.4: Analyzer passes

Chapter 5: Evaluation

Our analyzer enables us to verify the correctness of stack size estimation and synchronous invocation tracking in the Composite microkernel system. To evaluate its effectiveness, we conducted experiments on multiple Patina configuration files to assess both stack allocation accuracy and the extent to which unnecessary capabilities of dependencies can be eliminated. Patina provides an abstraction layer over microkernels and is designed to enforce the Principle of Least Privilege, making it an ideal test foundation for our analysis.

We evaluated six distinct Patina configuration files, each consisting of three to seven components, with a total of eight unique components studied throughout our experiments.

5.1 System Overview and Workflow

Figure 4.4 shows our work flow, which consist of the following steps:

1. Component Compilation - Each component is compiled into an ELF binary object file using Composer, starting with default conservative values for stack size and synchronous capabilities.

2. Binary Analysis - Our analyzer processes each component's binary, disassembles instructions with capstone, and track stack instruction and function calls.

3. Dependency Resolution - The analyzer constructs a capability of dependency table by searching synchronous invocations.

4. Capability Enforcement - If a synchronous function in non-reachable, it is disabled from the capability, reducing, unnecessary permissions.

5. Stack Size Refinement - The analyzer update stack allocations based on actual usage, replacing the default 8 KB allocation with a calculated optimal value.

Our method ensures that only necessary synchronous invocations and capabilities remain.

5.2 Experiment Setup

To evaluate our analyzer, we tested it on six different Patina configuration files in Composite, each containing three to seven components. We ran the analyzer within Composer, using its default configuration settings, and measured:

1. The reduction in synchronous invocations (i.e., unnecessary service functions).

2. The reduction in stack size allocations for each component.

We compiled each configuration using GCC 11.4.0 on an Intel i3-8100 CPU (4 GB RAM) running Ubuntu 22.04 LTS. Each component was compiled into an ELF binary and analyzed using our Python-based static analysis tool, which leverages the Capstone disassembly framework. We wrote our analyzer in Python (version 3.12.3) and used the Capstone module (version 5.0.3). Our evaluation focuses on the Patina test suite. Patina is a prototypical RTOS API designed to provide essential OS services typically found in full-featured systems like Linux, but often absent in minimal and more secure microkernel-based systems. It enforces the Principle of Least Privilege (PoLP) to reduce the attack surface and improve system security. The Patina tests serve as a meaningful benchmark for our analyzer, as they demonstrate its ability to support PoLP by identifying and eliminating unnecessary synchronous invocations and limiting capability exposure.

5.3 Capability Reduction Analysis

We first evaluated the capability enforcement mechanism to determine how many synchronous invocations our analyzer could eliminate. By running the analyzer on each configuration, starting from the entry function (cosrt_upcall_entry), we measured the number of service functions that could be deactivated under different system setups.

Configuration	Unnecessary Capability of Dependencies	Total Capability of Dependencies
patina_event	113	171
patina_timer	110	171
patina_mutex	43	60
patina_sem	43	60
patina_chan_bench	111	171
patina_chan_bench_inter	141	217

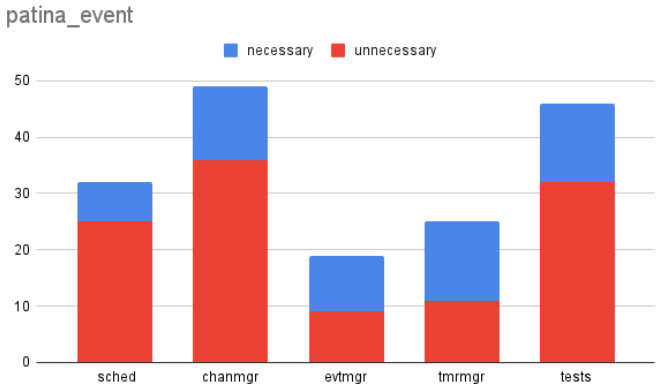
Table 5.1: Reduction of synchronous invocations

The results, presented in Table 5.1 and Table 5.2, indicate that our analyzer successfully eliminates approximately two-thirds of the total function calls associated with unnecessary service functions. This means that two-thirds of the capabilities of dependencies in the system can be disabled, significantly reducing the attack surface and limiting unnecessary privileges for components.

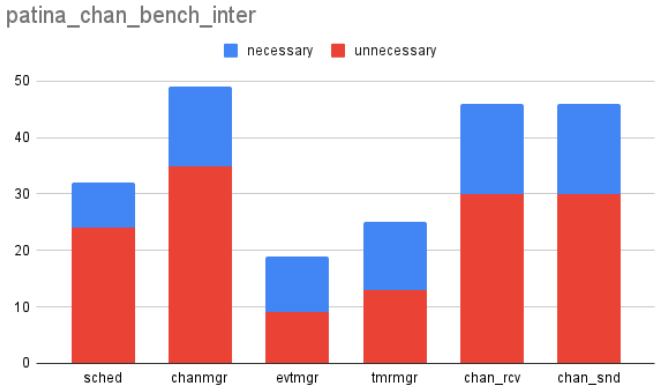
For example, in Table 5.1, the patina_chan_bench_inter configuration file shows that our analysis identified 141 out of 217 synchronous invocations as unnecessary, allowing us to disable 65% of capabilities of dependencies. By removing unused dependencies, our approach enhances component isolation, minimizes the risk of unauthorized access, and mitigates potential security vulnerabilities.

In Table 5.2, we further examine these configuration files at the component level, where each row represents a component within the configuration. Our analysis shows that each component can eliminate approximately 60% to 65% of its capabilities, aligning with the overall results observed in Table 5.1.

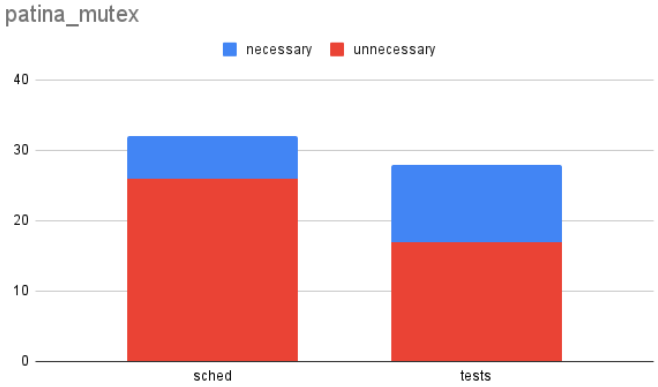
In Section 4, we previously discussed the challenge of handling function pointers. Since we have prior knowledge of these specific configuration files, we hardcoded 57 function pointer entries into a lookup table for Patina tests. If the analyzer encounters an undetermined function pointer address, it searches this table to resolve the function target, ensuring more accurate analysis and preventing unresolved function calls from impacting the final capability enforcement results.



(a) patina_event



(b) patina_chan_bench_inter



(c) patina_mutex

Figure 5.1: Column chart for Reduction of synchronous invocations on each components

patina_event	Unnecessary Capability of Dependencies	Total Capability of Dependencies
sched	25	32
chanmgr	36	49
evtmgr	9	19
tmrmgr	11	25
tests	32	46

patina_chan_bench_inter	Unnecessary Capability of Dependencies	Total Capability of Dependencies
sched	24	32
chanmgr	35	49
evtmgr	9	19
tmrmgr	13	25
chan_rcv	30	46
chan_snd	30	46

patina_mutex	Unnecessary Capability of Dependencies	Total Capability of Dependencies
sched	26	32
tests	17	28

Table 5.2: Reduction of synchronous invocations on each components

5.4 Stack Size Analysis

Next, we evaluated the accuracy of stack size estimation for each thread in each component. By analyzing execution paths and tracking stack usage, our approach ensures that stack allocations accurately reflect actual requirements, avoiding conservative over-provisioning and unnecessary memory consumption. The results in Table 5.3 confirm the correctness of our stack size calculations, demonstrating that our method effectively optimizes memory usage.

In Composite, client components invoke protected procedure calls to access services from server components. To maintain strong isolation, when a thread requests a service through a synchronous invocation, it migrates to the server component, where a dedicated stack is allocated. The picture 5.2 from the paper [4] could illustrate the protected procedure

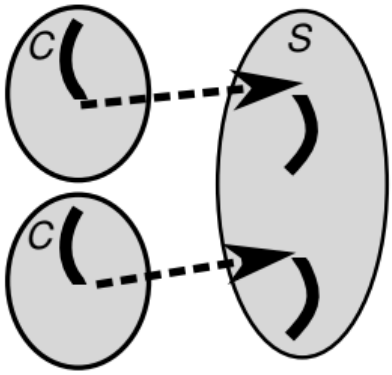


Figure 5.2: Thread Migration (adapted from Patina Paper [4]).

call. The default stack size allocation is 8 KB per thread. However, our analysis reveals that most thread of components require significantly less stack space, with many reducing their allocation by nearly 60%. For example, the Booter component's stack size was reduced from 8 KB to 3224 bytes, achieving a 60% reduction without compromising functionality.

One notable observation is that including functions such as printf or vfprintf significantly increases stack size requirements. These functions introduce additional stack overhead, leading to larger-than-necessary allocations. Understanding such behavior allows developers to make more informed decisions about memory optimization when designing system components.

These results highlight the effectiveness of our analyzer in improving resource management and enhancing security enforcement in Composite. By precisely identifying required stack sizes and eliminating redundant capabilities of dependencies, our approach not only optimizes memory usage but also enhances system security by minimizing potential attack surfaces in the microkernel environment.

Component	Stacksize of thread with analysis	Default thread Stacksize
booter	3224	8192
capmgr	1616	8192
sched	1176	8192
chanmgr	1008	8192
evtmgr	400	8192
tmrmgr	992	8192
chan_rcv	1056	8192
chan_snd	1152	8192

Table 5.3: Thread's Stack Size of each component

Chapter 6: Discussion

While our analyzer effectively analyzes capabilities and stack size, it has certain limitations, particularly in handling recursion and function pointers. These challenges stem from the inherent difficulty of statically resolving dynamic execution behavior.

6.1 Recursive Function Analysis

One limitation is analyzing recursive functions. Since recursion involves potentially unbounded function calls, determining the exact number of iterations a recursive function may execute at static compile time is challenging. This uncertainty can lead to imprecise stack size estimation, as the depth of recursion remains unknown. Consequently, our analyzer cannot precisely infer stack usage in such cases, leading to potential inaccuracies. However, since the Misra C and JPL/NASA coding guidelines discourage recursive in embedded system, and we have in-depth knowledge of the Composite system, we mitigate this issue by modifying the source code to explicitly define recursion bounds, and making system-specific assumptions within the analyzer. If neither approach is feasible, our analyzer flags recursion as an unresolved case and outputs an error, indicating that the recursion depth is unknown.

6.2 Function Pointers and Dynamic Resolution

Another challenge arises with function pointers, which introduce runtime-dependent behavior, making static analysis difficult. Since function pointers resolve their target addresses at runtime, our analyzer cannot directly determine their exact destinations through static analysis alone.

To address this, we leverage our understanding of the Composite system to impose assumptions on function pointer resolution. We construct a set of all possible function pointer targets. Upon encountering a function pointer, our analyzer searches this set to

determine, the maximum stack size required, and the possible capabilities needed. This approximation ensures that our analyzer remains functional, even though it does not achieve perfect precision in all cases.

In the Misra C and JPL/NASA coding guidelines discourage the dynamic assignment of function pointers, favoring strict rules for function pointer usage. This restriction benefits our analyzer, as it provides structured information to determine function pointer addresses more reliably. As a result, we could make an assumption withing our analyzer and reduce the need for hardcoded function pointer tables, improving the efficiency of our analysis.

Despite these limitations, our approach remains highly effective in most scenarios, providing valuable insights into resource allocation and capability enforcement. Future work will focus on enhancing function pointer resolution and integrating dynamic profiling techniques to further refine stack size estimation and synchronous invocation analysis.

6.3 Link Time Optimization and Dead Code Elimination

Link Time Optimization (LTO) is an optimization technique in which the compiler defers certain optimizations to the linking phase rather than performing them during individual compilation units. This enables the compiler to analyze the entire program across multiple object files, allowing for more aggressive optimizations that rely on global visibility. In Composite, applying traditional LTO is challenging because Composer analyzes components individually, without a whole-program view of the object files. To address this limitation, our work introduces an analysis framework that performs inter-object dependency analysis, similar in spirit to LTO. Our analyzer links information across components, enabling Composer to reason about system-wide synchronization behavior and make informed decisions based on actual inter-component interactions.

Dead Code Elimination (DCE) is another optimization that removes code that is never executed—such as unused functions, unreachable branches, or unreferenced variables. While DCE is not yet fully implemented in Composer, our analyzer lays the groundwork

by identifying unnecessary synchronous invocations. Currently, we use this information to disable capabilities associated with unreachable services. As a future step, we aim to extend this mechanism to eliminate the corresponding dead code entirely, enabling whole-program optimization and minimization across the Composite system.

Chapter 7: Conclusions

In Composite, the default system configuration relies on fixed stack size allocations and permits all possible synchronous invocations, leading to inefficiencies in resource management and increased security risks. This approach not only wastes memory but also weakens system isolation, exposing components to unnecessary service dependencies and potential vulnerabilities.

Our analyzer introduces a systematic approach to reducing capability exposure and accurately determining stack size allocation. Through binary-level and system-wide analysis, it enables users to precisely identify the necessary stack size for threads and the required capabilities of dependencies for each component. Our methodology is system-wide within the Composite system, as it operates solely on ELF binaries, without requiring access to high-level source code.

The analyzer performs a system-wide analysis by processing the object files generated by Composer, combines the analysis results across components to construct a global view of the system. This allows us to perform a comprehensive dependency analysis, through which we can accurately identify synchronous invocations and eliminate unnecessary inter-component dependencies.

Experimental results demonstrate that our analyzer reduces the total number of synchronous invocations by 60%, effectively minimizing unnecessary component interactions and enhancing system isolation. Additionally, it optimizes stack memory usage, reducing stack allocations by nearly 50%, thereby improving overall efficiency and security.

This work highlights the importance of system-wide, automated static analysis in fine-grained resource management for microkernel-based systems. Future work includes expanding our analysis capabilities to support more complex execution patterns and further refining our capability enforcement mechanisms to strengthen system security.

Chapter 8: Future Works

In this paper, we propose a system-wide method for analyzing Composite, focusing primarily on the initialization phase of the system. However, our current approach does not consider temporal execution aspects, where certain functions or synchronous invocations may become irrelevant after initialization.

As future work, we aim to extend our analyzer with temporal analysis, enabling us to track how capabilities and synchronous invocations evolve over time. Some functions or synchronous invocation may only be executed once during system startup, after which they remain unused for the rest of the execution. By identifying these transient invocations, we can further refine capability enforcement, ensuring that once initialization is complete, unnecessary permissions can be revoked, thereby strengthening system security and isolation.

Additionally, we plan to improve function pointer resolution in Composite. Function pointers introduce uncertainty in stack size estimation and synchronous invocation tracking, reducing the precision of our analysis. We aim to incorporate dynamic analysis techniques to infer function pointer targets, allowing us to more accurately determine function calls and synchronous dependencies. This would lead to more efficient memory utilization and better component isolation, enhancing both security and performance.

By integrating these enhancements, our approach will further improve the precision of capability management, reduce attack surfaces, and enable more accurate resource allocation beyond initial system configuration. Future iterations of our analyzer will incorporate dynamic analysis techniques to capture runtime behavior, further improving efficiency, security, and adaptability in Composite.

Bibliography

[1] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 9(3):143–155, March 1966.

[2] Bryan Ford and Jay Lepreau. Evolving mach 3.0 to a migrating thread model. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94, page 9, USA, 1994. USENIX Association.

[3] FreeRTOS. FreeRTOS - Market-leading RTOS (Real-Time Operating System), 2024. Accessed: Mar. 6, 2025.

[4] Samuel Jero, Juliana Furgala, Runyu Pan, Phani Kishore Gadepalli, Alexandra Clifford, Bite Ye, Roger Khazan, Bryan C. Ward, Gabriel Parmer, and Richard Skowyra. Practical principle of least privilege for secure embedded systems. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–13, 2021.

[5] Alexander Jung, Hugo Lefeuvre, Charalampos Rotsos, Pierre Olivier, Daniel Oñoro Rubio, Felipe Huici, and Mathias Niepert. Wayfinder: towards automatically deriving optimal os configurations. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '21, page 115–122, New York, NY, USA, 2021. Association for Computing Machinery.

[6] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. A binary-compatible unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2019, page 59–73, New York, NY, USA, 2019. Association for Computing Machinery.

[7] Gabriel Parmer. The case for thread migration : Predictable ipc in a customizable and reliable os. 2010.

[8] seL4 Project. Camkes documentation. `https://docs.sel4.systems/projects/camkes/`, 2025. Accessed: Mar. 6, 2025.

[9] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. Eros: a fast capability system. *SIGOPS Oper. Syst. Rev.*, 33(5):170–185, December 1999.

[10] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature consistency in compile-time-configurable system software: facing the linux 10,000 feature problem. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, page 47–60, New York, NY, USA, 2011. Association for Computing Machinery.

[11] Necip Fazıl Yıldıran, Jeho Oh, Julia Lawall, and Paul Gazzillo. Maximizing patch coverage for testing of highly-configurable software without exploding build times. *Proc. ACM Softw. Eng.*, 1(FSE), July 2024.