The Pennsylvania State University
The J.Jeffrey and Ann Marie Fox Graduate School

EXPLOITING LANGUAGE AND SEMANTICS SPECIFIC FEATURES FOR PROGRAM ANALYSIS

A Dissertation in
Informatics
by
Tianrou Xia

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2025

The dissertation of Tianrou Xia was reviewed and approved by the following:

Dinghao Wu
Professor in the College of Information Sciences and Technology
Co-Chair of Committee
Dissertation Co-Advisor

Taegyu Kim
Assistant Professor in the College of Information Sciences and Technology
Co-Chair of Committee
Dissertation Co-Advisor

Hong Hu
Assistant Professor in the College of Information Sciences and Technology
Major Field Member

Sencun Zhu
Associate Professor in the Department of Computer Science and Engineering
Outside Unit & Field Member

Carleen Maitland
Professor in the College of Information Sciences and Technology
Program Head

Abstract

Program analysis plays an essential role in software and system security, supporting a wide spectrum of techniques. However, a fundamental challenge persists: highly precise analyses often incur prohibitive cost while efficient analyses often sacrifice accuracy. This dissertation proposes an insight of addressing this challenge through two projects respectively targeting indirect call resolution and memory safety enforcement.

The first project focuses on resolving indirect call targets. Existing solutions are either imprecise (using type-based analysis) or inefficient (using data-tracking analysis), while Multi-Layer Type Analysis (MLTA) provides a balance by leveraging the data structure hierarchies commonly adopted in C/C++. However, it fails to accurately track the data flow among multi-layer types, introducing false positives. To overcome this limitation, we propose Strong Multi-Layer Type Analysis (SMLTA) and implement it in DeepType. It addresses the challenges in multi-layer type matching, avoiding type information loss and further improving precision without losing efficiency. Evaluation on the Linux kernel, 5 web servers, and 14 user applications shows that DeepType refines indirect call targets by 43.11% on average compared to MLTA's prototype TypeDive, while reducing runtime overhead by 5.45%–72.95%, achieving improvements in both precision and scalability.

The second project enforces memory safety in Rust. Although Rust provides strong safety guarantees, it allows unsafe code to bypass compiler checks, which reintroduces memory vulnerabilities. The widely used AddressSanitizer (ASan) and its Rust-specific successors, ERASan and RustSan, suffer from high overheads and limited bug coverage due to inherent limitations of ASan's red zone and shadow memory mechanisms. We present LiteRSan, a novel memory safety sanitizer that addresses the limitations of prior approaches. By aligning with Rust's ownership model, LiteRSan performs Rust-specific static analysis aware of pointer lifetimes to identify risky pointers. It then selectively instruments them to enforce only the necessary spatial or temporal checks. LiteRSan introduces significantly lower runtime overhead (18.84%) and negligible memory overhead (0.81%) compared with existing ASan-based sanitizers while detecting memory safety bugs that prior techniques missed.

Altogether, DeepType and LiteRSan illustrate a common insight: language and semantics specific characteristics enable program analysis to reduce overhead while improving precision, achieving a balance that general language-agnostic approaches fail to reach. This insight motivates future advances in program analysis to further strengthen the security of modern software systems.

Table of Contents

List of Figures

List of Tables

Acknowledgments

At the beginning of my academic journey, pursuing a Ph.D. was never part of the plan. By a twist of fate, however, I embarked on this path. Looking back now, I am deeply grateful for that decision. It has shaped who I am today. Through this journey I have learned to think calmly, to accept the unexpected with composure, to remain curious about new ideas, and to focus not only on results but also on the process itself. The road has been long, often winding, but illuminated by stars. I was never truly alone. I owe much to the people who have supported and guided me, in my work, in my life, and in the quiet shaping of my spirit.

First and foremost, I would like to express my heartfelt gratitude to my advisor Dr. Dinghao Wu. It was his recognition and confidence in me that made me believe I could become a researcher. He is a scholar of the old-school, with deep expertise in program analysis and formal verification. He has always been able to pierce through complexity and uncover the essence of a problem, pointing out flaws in my work with precision and clarity. Though I am now near the end of my Ph.D., his professionalism still reminds me of how much more there is to learn and how far I can still grow. Dinghao's style of mentorship has been trust and freedom. He allowed me the space to explore, to make mistakes, and to choose the research directions that truly interested me. He seldom intervened unless I sought his help. This approach helped me grow into an independent researcher, one capable of making difficult decisions in critical situations. I am deeply grateful for his trust, for bearing the costs of my trial and error, and for providing the space in which I could find my own path.

I am equally grateful to my co-advisor Dr. Taegyu Kim, who guided me during a period of confusion and helped explore new research directions. He always respected my choices, while offering advice that was carefully tailored to my background. When we had different opinions, he always listened to me thoughtfully and provided constructive feedback that improved my work. From him, I learned the importance of listening, which I previously neglected but is significant in collaboration. Beyond research, Teagyu also paid attention to my career development. He shared his experiences, provided practical guidance, and supported me in concrete ways, such as analyzing reviewer' feedback together, inviting me to guest lecture in his class, and taking detailed notes during my presentations and later suggesting improvements. I am truly grateful for his support.

I would also like to thank Dr. Hong Hu, my committee member and collaborator on the DeepType project. When I was new to LLVM, he generously shared technical

knowledge and practical tips that greatly accelerated my learning. He organized a weekly reading group, which provided me with a valuable platform for learning and exchange. Through those meetings I not only learned the latest advances in the field, but also improved my presentation skills and developed sharper critical thinking through lively discussion. His active and creative mind has always made our conversations stimulating and rewarding.

My gratitude also goes to Dr. Sencun Zhu, my outside committee member and former master's advisor. He guided me through my very first research project on malware detection, teaching me how to identify problems and devise solutions, which opened the door to research for me. He has always been warm, kind, and generous with his help. For my dissertation, his insights reached beyond the scope of the projects, offering high-level perspectives that encouraged me to think more globally and critically.

I would also like to thank my collaborators and lab mates. Although our research directions were not always closely aligned, their expertise broadened my horizons and the discussions often provided fresh perspectives. The Ph.D. journey is long and filled with struggles, their empathy and support were a source of comfort during difficult times.

Finally, and most importantly, I want to thank my family and friends for their infinite encouragement and love. Although living overseas has meant long stretches without seeing one another, their care and greetings have never ceased. They have always been attentive to my well-being, asking about my health, worrying about my stress, and offering words of comfort and encouragement. Their constant support has been a source of strength and warmth throughout my Ph.D. journey.

To my parents, Mr. Yifeng Xia and Ms. Jianfang Xu, who have selflessly supported every decision I have made. I know their hope was for a daughter close at hand, someone who could share meals and conversations at home. Yet my path took me far away, across an ocean and time zones, leaving us with hurried phone calls instead of everyday companionship. Still, they put aside their own needs to support my dreams. When I was almost giving up, they reassured me of my abilities and encouraged me to keep going, yet also reminded me that choosing another path would never diminish their love. Their love has always been a safe harbor, allowing me to set sail without fear.

To my husband, Dr. Kaiming Huang, without whom I can hardly imagine completing this Ph.D. He was the serendipity at the very beginning of this story. If my parents have been the harbor, then he has been the captain. Over the eleven years since we met, he has been guiding the ship with me through storms and tides. He has felt my worries more deeply, my sorrows more painfully, and my joys more brightly, than I have. His love and support have given me the courage to overcome every obstacle.

In the end, though hardship and struggle are the constants of life, and though I am but an ordinary figure in this vast world, I am profoundly grateful for everything I have been given. I believe there are things in this world that will never change. May the flowers ever bloom, and may the good dreams remain.

Chapter 1 Introduction

Software and systems permeate every aspect of modern life. Their influence will deepen as society becomes increasingly interconnected. Along with the enhanced productivity and convenience, vulnerabilities in software and system security raise public concerns, as their failures can lead to widespread and devastating consequences. For example, the global IT outage in 2024 [158], as a result of a flawed update to CrowdStrike's Falcon platform, caused widespread "blue screen of death" errors across millions of Windows machines. This single event disrupted daily life, halted businesses and government operations worldwide, and led to financial losses of at least $10 billion U.S. dollars [155]. Recognizing these risks, governments and institutions have been paying closer attention to software and system security. The U.S. Executive Order 14028 [2], for instance, mandates cybersecurity and software security enhancement, calling for rigorous detection, protection, and security event logging to mitigate vulnerabilities and threats.

1.1 Program Analysis in Software and System Security

Program analysis plays an essential role in advancing software and system security. By analyzing program structure, values, and potential behaviors during execution, it derives critical properties of the program and enables a broad spectrum of security mechanisms, such as vulnerability detection [28, 70, 154], program hardening against exploitation [43, 52, 53, 151], and fault localization for timely patches [30, 168, 169], collectively enforcing security at different stages of the software lifecycle. Beyond these applications that directly target security, program analysis is also widely applied in the broader domain of software quality assurance, such as optimization techniques [77, 164] and formal verification [45, 109], where it improves computational efficiency performance and establishes strong correctness and reliability guarantees. Altogether, these techniques

built upon program analysis not only strengthen the reliability and resilience of modern software and systems, but also enhance their efficiency, correctness, and trustworthiness.

To support these applications, different approaches in program analysis are adopted either individually or in hybrid forms. Each approach inspects a distinct kind of information or attribute for specific purposes, yet they are interrelated and often complement one another.

Type-based analysis examines the declared or inferred types of program variables, expressions, and functions. It can detect simple errors such as type mismatches and illegal casts [25,38,132,148], or violations of typing rules [4,5,84]. Beyond error detection, type information serves as a lightweight yet effective abstraction for program reasoning. For example, type-based methods are frequently applied to resolve indirect calls and construct call graphs, where they approximate possible call targets by matching function pointer types with function signatures [12,128]. Compared to more precise analyses, type-based analysis is adopted due to its scalability and low overhead, making it a practical choice for large codebases. However, this efficiency comes at the cost of precision, since type information alone abstracts away detailed control-flow and data-flow relationships. Consequently, type-based approaches are often combined with other analyses, such as points-to analyses, to improve accuracy while maintaining scalability [78, 165].

Data-flow analysis tracks how values propagate across program variables and statements, enabling reasoning about program behaviors beyond simple type information. Data-flow analysis often relies on control-flow graphs (CFGs) [6], which capture the possible execution paths through a program. CFGs support a broad spectrum of applications, ranging from compiler optimizations such as liveness analysis and reaching definitions to advanced software security mechanisms, including fuzzing [16, 40, 79] for bug discovery, control-flow integrity (CFI) [1, 99, 100, 149] for preventing control-flow hijacking, and isolation techniques [75, 94] for runtime protection.

An essential instance of data-flow analysis is points-to analysis, also known as alias analysis, which determines the set of memory locations a pointer or reference may refer to and captures relationships between pointers. It is crucial for reasoning about heap objects, enforcing memory safety, and enabling optimizations. Classical techniques include Andersen's inclusion-based analysis [8] and Steensgaard's unification-based analysis [129], with widely used implementations such as SVF (Static Value-Flow) [130] and LLVM's pointer analysis passes. These analyses face the long-standing challenge of balancing precision and efficiency, which has led to a range of specialized variants tailored to different purposes and requirements.

From the perspective of sensitivity to program semantics, data-flow analysis can be distinguished among flow-sensitivity, path-sensitivity, and context-sensitivity. Flow-sensitive analyses account for the order of statements and maintain different facts at different program points, yielding greater accuracy, but at the cost of higher complexity compared to flow-insensitive approaches. Path-sensitive analyses refine this further by distinguishing between different execution paths (e.g., branch conditions), enabling precise reasoning about conditional behaviors, but they often suffer from path explosion. Context-sensitive analyses differentiate among calling contexts in inter-procedural analysis, avoiding rough merges across function calls and enabling more precise data tracking, but increase computational cost and memory usage relative to context-insensitive analyses.

From the perspective of scope, data-flow analysis can be intra-procedural, restricted to reasoning within a single function, or inter-procedural, extending across function boundaries to capture global program behaviors. High-precision analyses often combine flow-sensitivity, path-sensitivity, and context-sensitivity with inter-procedural reasoning, but these combinations are computationally expensive. Scalable analyses reduce the cost by sacrificing some forms of sensitivity, which improves efficiency but inevitably leads to a loss of accuracy.

Complementing static approaches, dynamic analysis gathers information from program's actual program executions, including concrete paths, memory accesses, and thread interactions, etc. Tools such as Valgrind [95] and AddressSanitizer [122] leverage dynamic analysis to detect memory errors at runtime for runtime profiling and validation. While dynamic analysis provides precision with respect to the actual behaviors during execution, it suffers from inherent limitations: coverage is restricted to the inputs specified, and runtime monitoring may introduce significant overhead.

1.2 Precision and Efficiency

Despite the broad range of approaches, program analysis faces the critical challenge of balancing precision against efficiency, as described in Chapter 1.1. Analyses that capture rich semantic details, such as context-sensitive, flow-sensitive and path-sensitive points-to analysis, can achieve high precision, but they often incur prohibitive computational cost and struggle to scale to large, real-world programs. Conversely, lightweight analyses that prioritize efficiency, such as type-based methods or coarse-grained data-flow analyses, can process large codebases, but they may be too imprecise to detect certain vulnerabilities, or not applicable to practical enforcement. This trade-off has been troubling the design

of program analysis tools, where developers must decide between depth of reasoning and practical scalability.

This dissertation exposes applicable solutions to address the challenge of bridging precision and efficiency for program analysis. Through two works respectively focusing on two research fields, one on resolving indirect call targets (Chapter 3) and the other on enforcing memory safety in Rust (Chapter 4), this dissertation demonstrates how semantically tailored, domain-specific techniques can achieve both accuracy and performance improvements compared to general program analysis approaches.

To be specific, the first work DeepType leverages the hierarchical data structures commonly found in C/C++ programs, integrating type-based and data-flow reasoning into a multi-layered type analysis. This hybrid approach enables the precise resolution of indirect call targets while maintaining scalability to large codebases. The second work LiteRSan respects Rust's ownership and lifetime semantics to design a Rust-specific static analysis that improves precision and reduces redundant instrumentation of sanitizer checks. Combined with a lightweight metadata-based runtime mechanism, LiteRSan enforces memory safety efficiently with subtle compilation, runtime and memory overhead.

Altogether, these two projects illustrate how language and semantics specific features can reconcile precise reasoning with practical performance in program analysis for security enforcement. They highlight a broader insight: instead of seeking one-size-fits-all solutions, advancing program analysis requires instantiated techniques tailored to the semantics of specific languages or domains, thereby achieving the optimal balance between precision and efficiency.

1.3 Research Problems

To illustrate how language and semantics specific characteristics advance the balance between precision and efficiency in program analysis, I will present two concrete projects that leverage this insight. Each project addresses individual research problem that is fundamental yet crucial in software and system security.

The first project, DeepType (see Chapter 3), addresses a long-standing challenge in program analysis: the resolution of indirect call targets in C/C++ programs. Indirect calls are pervasive in real-world software, arising from function pointers, virtual methods, and callbacks. Accurately resolving their targets is essential for constructing precise control-flow graphs (CFGs), which in turn plays an important role in a wide range of compiler optimizations, security mechanisms (e.g., control-flow integrity), and program verification

tasks. However, this problem is notoriously difficult. As low-level programming languages, C and C++ allow complex pointer manipulation and flexible use of data structures, making indirect call resolution highly challenging. Existing approaches either sacrifice precision, leading to excessive false positives, or incur prohibitive computational cost, limiting scalability.

DeepType tackles this challenge by advancing the state of type-based analysis, multi-layer type analysis (MLTA) [78]. Building on the intuition of MLTA, DeepType leverages the common characteristic of C/C++ programs: the pervasive use of composite data structure hierarchies, which can be exploited to narrow down call targets. While TypeDive pioneered the idea of multi-layer type analysis, it produces false positives by design because it does not explicitly model the data flow between multi-layer types, which weakens the restrictions provided by multi-layer types. DeepType improves imprecision by tracking data-flow relationships across multi-layer types, thereby respecting the hierarchical type system. This refinement takes full advantage of the extra type information and restrictions to differentiate more precisely among potential call targets, reducing false positives while maintaining efficiency.

The second project, LiteRSan (see Chapter 4), develops a comprehensive yet lightweight sanitizer to enforce memory safety in Rust. Although Rust is a memory-safe language, its support for unsafe code reintroduces memory safety vulnerabilities. More importantly, these vulnerabilities can propagate from unsafe blocks to safe code, making it challenging to uncover all memory safety bugs. Detecting such bugs is critical as memory safety remains a foundational requirement for building reliable and secure systems.

Sanitization has become the widely used technique for detecting memory safety violations, with tools such as AddressSanitizer (ASan) integrated into the Rust compiler toolchain. However, existing sanitizers face two fundamental limitations in the Rust context. First, they are not designed to align with Rust's type system and ownership semantics, resulting in redundant checks. Second, ASan and its successors rely on coarse-grained shadow memory and red zone mechanisms, which can fail to capture certain bugs while simultaneously incurring significant runtime and memory overhead. LiteRSan addresses these limitations with two key innovations. First, it introduces a Rust-specific static analysis that respects Rust's ownership rules to identify *risky pointers*, which may violate spatial or temporal safety. By instrumenting checks only for these pointers, LiteRSan eliminates the redundant instrumentation deployed by generic sanitizers, thereby reducing unnecessary overhead. Second, it replaces shadow memory and red zones with a fine-grained, metadata-based runtime validation mechanism. This

mechanism maintains lightweight and compact metadata to track pointer and object states precisely at runtime, ensuring complete coverage of memory safety violations with significantly lower cost. These techniques enable LiteRSan to achieve a balance that existing approaches fail to reach: comprehensive bug detection coupled with minimized compilation, runtime, and memory overhead. LiteRSan demonstrates how language-specific insights can fundamentally advance sanitizer design, providing both precision and efficiency in enforcing memory safety for Rust.

Despite targeting different languages and problem domains, both projects follow the same philosophy: leverage language and semantics specific features to tailor analysis strategies, which can simultaneously achieve high accuracy and practical performance, advancing the reliability and security of modern software systems.

1.4 Dissertation Structure

The remainder of this dissertation is organized as follows. Chapter 2 reviews the background and previous work related to both research problems. Chapter 3 details the design, implementation, and evaluation of DeepType. The result is published in USENIX Security Symposium 2024 [165]. Chapter 4 presents the design, implementation, and evaluation of LiteRSan. A manuscript on LiteRSan is under review [166]. Finally, Chapter 5 concludes the dissertation and discusses potential directions for future research.

Chapter 2
Background and Related Work

This chapter reviews the background and related work of the two projects: (1) indirect call target resolution, presented in Section 2.1, and (2) Rust memory safety enforcement, presented in Section 2.2.

2.1 Indirect Call Target Resolution

This section introduces the background knowledge of indirect call target resolution, including traditional analyses and advanced multi-layer type analysis in Section 2.1.1, and reviews related work in this domain in Section 2.1.2.

2.1.1 Background

To resolve indirect call targets, existing approaches can generally be categorized into type-based analysis and data-flow analysis. Type-based analysis checks the types of call sites and functions (i.e., their signatures) to determine whether an address-taken function is a valid target. As discussed in Section 1.1, this approach is efficient and scalable. However, it often lacks precision in large and complex programs, where many functions share identical signatures and thus cannot be distinguished using type information alone. In contrast, data-flow analysis achieves higher precision by explicitly tracking how function pointers propagate through variables and memory. By following the actual values of function pointers used to make indirect calls, data-flow analysis can determine the precise callees. While this approach provides strong accuracy guarantees, it suffers from poor scalability due to its computational cost. For example, SVF (Static Value-Flow Analysis) [130], which applies data-flow techniques to construct control-flow graphs, may require several days to complete its analysis on a large codebase such as the Linux kernel.

Multi-Layer Type Analysis (MLTA) [78] was introduced as a compromise between these two extremes, upgrading the scalable type-based methods with precision gains from data-flow analysis. MLTA builds on a common characteristic of C/C++ programs: function pointers are frequently embedded within composite data structures, often organized hierarchically. To leverage this property, MLTA extends traditional types into *multi-layer types* that capture not only the type of the function pointer itself but also the surrounding layers of composite structures that encapsulate it.

Illustrative Example. Consider a function pointer `b.a.ptr` of type `void(int)*`, where `ptr` is a field of `struct A`, and `a` (an instance of `struct A`) is itself a field of `struct B`. In this case, MLTA extends the traditional type

`void(int)*`

into the multi-layer type

`void(int)*` | `struct A` | `struct B`,

where "`|`" separates different layers.

Following the working principles of type-based analysis, MLTA then compares the multi-layer types of functions and indirect call sites to determine valid targets.

Although MLTA improves indirect call resolution by refining granularity beyond function signatures alone and achieves better precision than purely type-based methods while remaining more scalable than full data-flow analysis, it still exhibits important limitations. Specifically, MLTA's handling of data-flow and type matching does not fully respect the semantics of multi-layer types.

While MLTA defines multi-layer types, it still tracks data flow in the same manner as traditional type systems: by recording flows between variables and their traditional types. Similarly, its type matching strategy splits a multi-layer type into separate traditional types and then records potential targets for each layer independently. For example, if the function pointer `b.a.ptr` takes the address of function `foo`, MLTA splits its multi-layer type `void(int)*` | `struct A` | `struct B` into three independent layer types: `void(int)*`, `struct A`, and `struct B`. It then records `foo` as a valid target for each of these types individually. When analyzing an indirect call site, MLTA performs the same splitting, identifies potential targets for each layer separately, and calculates their intersection.

This splitting strategy makes MLTA traditional type-level reasoning, weakening the additional restrictions that multi-layer types are intended to enforce.

As a result, MLTA may fail to distinguish function pointers with different multi-layer types and introduce false positives, undermining its intended precision. These false positives will be illustrated in detail with an example in Section 3.1.1.

2.1.2 Related Work

Prior work on resolving indirect call targets spans a spectrum of program analysis techniques, each trading off precision against computational cost. At one end are conservative methods that impose minimal analysis and computation but tend to yield highly over-approximated target sets. Next are type-based approaches, which improves precision yet still limited. Then there are works based on data-flow analyses, which achieve higher precision by tracing pointer propagation but at significantly higher cost. Finally, dynamic methods offer the greatest accuracy by relying on runtime information. This section reviews and compares these approaches.

Conservative Solutions

Conservative approaches to resolving indirect call targets rely on minimal static analysis and are easy to implement with strong soundness guarantee. These approaches are lightweight and straightforward to deploy, particularly in settings where source code is unavailable. Moreover, they ensure that no legitimate target is missed, providing soundness at the cost of precision.

One of the earliest and most influential efforts in this field is the work of Abadi et al. on Control-Flow Integrity (CFI) [1], which enforces that every computed control transfer must target a location permitted by a statically constructed control-flow graph (CFG). In practice, this means that all address-taken functions identified in the binary are included as valid indirect call targets. Zhang et al. proposed CCFIR [175], which further develops this idea by deriving a white-list of valid targets from the relocation tables provided by ASLR. Indirect control transfers are then forced to pass through a controlled "springboard" section, where the enforcement mechanism ensures that they only reach entries on the white-list. This design makes CCFIR efficient, compatible with commodity binaries, and deployable without requiring access to source code. Similarly, Zhang and Sekar introduced COTS [177], a system that targets commercial off-the-shelf binaries. Under their model, every valid function in the binary is conservatively treated

as a potential target for indirect calls. This approach again prioritizes simplicity and soundness, relying only on relocation information to enumerate possible targets.

Despite their advantages, conservative methods suffer from a fundamental limitation: lack of precision. Because the target sets they compute are large, which often include all address-taken functions or even all valid functions, these solutions introduce a significant number of false positives. This over-approximation impairs the effectiveness of control-flow integrity enforcement, since an attacker may still redirect control flow to a wide range of allowed but malicious destinations. Furthermore, for downstream program analyses or optimizations, the size of the target sets reduces their utility, as reasoning about such imprecise call graphs becomes both inefficient and less meaningful. Consequently, while conservative solutions remain valuable as a foundation, they provide limited precision in practice and motivate the development of more refined techniques.

Type-Based Analysis

Type-based analysis offers a more precise way to restrict indirect call targets when type information is available. It is efficient by leveraging type signatures of functions and function pointers used at indirect call sites to eliminate large numbers of invalid targets without requiring heavy-weight static analysis. This makes type-based approaches practical for large programs, as they are lightweight, easy to integrate into compiler toolchains, and scale well to real-world applications. Furthermore, type-based analysis is less conservative than approaches that simply include all functions, since type compatibility rules provide restrictions that reduce the size of target sets.

Prior works targeting scalability adopted type-based analysis. Tice et al. introduced two influential techniques, VTV and IFCC [145]. VTV provides forward-edge CFI protection for GCC by validating the correctness of vtable pointers at virtual call sites. Specifically, it checks that the vtable pointer used for a call corresponds to either the static type of the object or one of its subclasses, ensuring type-safe dispatch. IFCC, implemented in LLVM, generalizes this idea by partitioning function pointers into disjoint sets based on type signatures and enforcing that each indirect call can only target functions within its assigned set. This design provides efficient protection with low runtime overhead, although the granularity of type information limits precision.

Building on these ideas, Niu and Tan developed MCFI [99], which broadens the allowable targets to include all address-taken functions whose types are structurally equivalent to the function pointer's type. This structural matching provides a flexible yet principled way of determining compatibility and yields higher coverage, but it can still

over-approximate targets in programs with widespread use of generic or weakly-typed interfaces. Victor et al. proposed TypeArmor [150], which further refines type-based restrictions by incorporating both function signatures and the number of arguments into the target resolution process. By enforcing compatibility on these dimensions, TypeArmor significantly reduces the number of spurious targets compared to earlier type-based schemes.

More recently, Multi-Layer Type Analysis (MLTA) [78] extended the type-based paradigm by leveraging the observation that function pointers in C/C++ programs are frequently embedded in composite data structures. MLTA augments traditional function pointer types with the types of the surrounding data structures, thereby forming multi-layer types that more precisely capture program semantics. By comparing multi-layer types of call sites and functions, MLTA can significantly refine the set of valid targets beyond what is achievable using flat type signatures alone.

Another prominent line of work focuses on C++ virtual functions, where the class hierarchy provides additional type context. Systems such as ShrinkWrap [46], SafeDispatch [58], VFGuard [107], IFCC [145], and VTrust [174] all use variations of class hierarchy analysis combined with object type information to identify valid targets for virtual calls. These approaches exploit the inheritance relationships in C++ to ensure that only compatible vtables or object types are permitted at call sites. While effective, these systems still operate within a single-layer type framework, limiting their ability to distinguish more nuanced cases involving nested data structures.

Despite their advantages, type-based analyses also face notable limitations. Their reliance on type compatibility alone means that they remain less precise than data-flow or hybrid approaches, particularly in programs with pervasive type reuse, inheritance, or casting. In such contexts, multiple semantically distinct functions may share equivalent type signatures, leading to overly large target sets. Furthermore, these analyses are inherently flow-insensitive, since they do not consider the actual propagation of function pointers at runtime. As a result, while type-based analysis strikes a useful balance between scalability and precision, it cannot eliminate false positives effectively, and its precision is fundamentally bounded by the type system itself.

Data-Flow Analysis

Data-flow analysis methods refine indirect call target resolution by explicitly tracking how function pointer values are produced, propagated, stored, and used. Compared to purely type-based schemes, these approaches generally achieve higher precision because

they reason about actual data flow among pointers rather than relying solely on function signature and type information. They are also language and representation agnostic, operating at the source, intermediate representation, or binary level. This flexibility makes them particularly attractive for analyzing stripped binaries or kernels where type information is incomplete or unavailable. Moreover, by deriving targets from concrete function pointer values, data-flow analysis can compute target sets that are often significantly smaller than those produced by conservative or type-based techniques, improving both the security of CFI enforcement and the precision of downstream analyses such as call graph construction.

Ge et al. developed a fine-grained CFI framework for kernels that applies taint analysis to identify indirect call targets [42]. Their system begins by tainting function pointers initialized with a given function or with previously tainted pointers. If a function pointer resides in a structure, the taint is propagated to the corresponding field of all objects of that structure type. This policy allows the system to leverage data-structure information in addition to the function pointer itself, producing more precise target sets than pure type-based matching. Evaluated on FreeBSD, MINIX, and BitVisor kernels, their approach eliminated over 70% of indirect targets compared to prior state-of-the-art techniques, while incurring macrobenchmark overheads of less than 2%. These results demonstrate that data-tracking reasoning can substantially improve forward-edge CFI in kernel environments without prohibitive runtime cost.

Kim et al. proposed Block-based Pointer Analysis (BPA), which performs data-flow analysis directly on stripped binaries [62]. BPA introduces a block memory model, where heap, stack, and global memory regions are partitioned into abstract blocks. Pointer analysis is then performed within these blocks to infer aliasing and resolve indirect call targets. This approach provides a scalable balance between precision and efficiency for binary-level target identification. Building on this foundation, they later introduced BinPointer [63], which extends BPA with offset sensitivity. By tracking offsets within each block, BinPointer can distinguish between different fields within the same abstract block, yielding finer-grained aliasing information and improved precision, particularly in programs that rely heavily on pointer arithmetic. However, while effective for C-style programs, these block-based analyses face difficulties in handling aggressive type casting and lack support for the richer class semantics of C++, leading to missed targets.

Another representative system is CFGAccurate, built on the angr binary analysis platform [124]. CFGAccurate interleaves four techniques, forced execution, backward slicing, symbolic execution, and value-set analysis, to iteratively refine control-flow graphs

until no new indirect call targets emerge. This combination enables more complete and precise target recovery than single-technique approaches, especially in complex binaries with indirect branches. Nevertheless, the reliance on symbolic execution and repeated refinement introduces high computational overhead, making CFGAccurate less practical for large-scale programs.

In summary, data-flow analyses provide target sets based on concrete pointer propagation and offer precision advantages over conservative or type-based methods. They are especially valuable for binary analysis, where type information is limited. However, these benefits come with notable drawbacks. Precise flow-sensitive, path-sensitive, and context-sensitive analyses incur significant computational costs. Abstractions such as block models struggle with pervasive type casting, and symbolic execution suffers from path explosion and solver overhead. These limitations motivate the exploration of hybrid techniques that combine type structure with selective data-flow reasoning to balance scalability and precision.

Dynamic Analysis

Dynamic approaches represent another important class of techniques for resolving indirect call targets. Unlike static methods, which approximate possible targets before execution, dynamic systems leverage runtime information to enforce that each indirect call targets a valid destination. The primary advantage of this strategy is precision during execution. Because checks are applied to the actual runtime state, dynamic analysis can eliminate many of the over-approximations inherent in static analysis. Moreover, dynamic methods are often more robust to complex program features such as aggressive casting, inline assembly, or dynamically generated code, which are difficult for static analysis to reason about. By validating control transfers at runtime, these approaches can provide strong security guarantees.

Several representative systems illustrate the design space of dynamic enforcement. PittyPat [37] applies runtime profiling to monitor control transfers and enforces that indirect calls only target legitimate functions observed during profiling. While this reduces the attack surface compared to conservative policies, its coverage is inherently tied to the quality and completeness of training inputs. Griffin [41] introduces a hybrid design by combining both hardware and software techniques. It instruments indirect branches and consults shadow stacks and metadata at runtime, achieving low overhead while providing strong forward- and backward-edge protection. uCFI [50] further extends this domain by proposing a unified framework that enforces fine-grained control-flow

integrity dynamically with reduced instrumentation cost, making runtime validation more efficient and practical. CCFI [149] protects control data by using cryptographic message authentication codes (MACs) to verify the integrity of code pointers at runtime. This approach provides strong tamper-resistance guarantees but comes with the cost of maintaining and verifying MACs on each pointer use. Finally, PiCFI [100] combines static analysis with lightweight runtime checking: it leverages static pointer analysis to narrow target sets and then instruments dynamic checks to validate runtime targets, striking a balance between coverage and performance.

Despite their strengths, dynamic analyses face fundamental limitations. Their reliance on runtime checks inevitably introduces performance overhead, which may be prohibitive in performance-critical systems. Coverage is another challenge: unless the entire execution space is exercised, dynamic systems can only validate targets encountered at runtime, leaving unexecuted paths unchecked. Additionally, instrumentation and metadata maintenance can increase complexity and memory usage, limiting deployment in resource-constrained settings such as embedded systems or kernels.

2.2 Rust Memory Safety Enforcement

This section provides background and reviews prior work on Rust memory safety enforcement. It first introduces Rust's built-in memory safety model and the primary causes of memory safety violations, and then introduces the traditional pointer analyses [8,129,130] and AddressSanitizer (ASan) [122], which form the foundation of many existing Rust memory safety tools, as discussed in Section 2.2.1. Finally, prior work on Rust memory safety enforcement is discussed in Section 2.2.2.

2.2.1 Background

Rust's Memory Safety Guarantees

Spatial memory safety refers to the prevention of out-of-bounds memory accesses, which are among the most common sources of vulnerabilities in low-level languages such as C and C++. Rust enforces spatial safety by design. Unlike C/C++, where pointers can be arithmetically operated or arbitrarily cast, Rust prohibits explicit pointer arithmetic on references [65]. A reference in Rust (&T or &mut T) is always tied to a well-defined memory region, and its validity is guaranteed by the compiler. For data structures such as arrays, slices, and vectors, Rust maintains internal metadata (e.g., length and capacity)

that is automatically checked at access points. At compile time, the compiler statically verifies bounds for indexing operations. For dynamically determined indices, Rust inserts runtime bounds checks to ensure no access exceeds the container's limits. This design ensures that in safe Rust, a dereference can never reach beyond allocated memory region.

Nevertheless, Rust also acknowledges the need for low-level control in systems programming and thus provides an *unsafe* mode. Unsafe blocks allow developers to perform operations such as raw pointer arithmetic, unchecked array indexing, casting between pointer types, or calling external functions (FFI). These unsafe constructs bypass the compiler's spatial checks and can lead to memory safety violations, making them the root cause of out-of-bounds errors [9, 88, 108]. While necessary for writing performance-critical code (e.g., implementing memory allocators, system calls, or custom data structures), unsafe code reintroduces the possibility of spatial errors such as buffer overflows or out-of-bound memory dereferences.

Temporal memory safety ensures that memory is only accessed while it is still valid, thereby preventing errors such as use-after-free, double free, and dangling references. Rust enforces temporal safety through its ownership and borrowing model. Each object in Rust has a unique owner, and when the owner goes out of scope, the Rust compiler automatically inserts the deallocation code to free the memory. This eliminates the possibility of memory leaks caused by forgotten deallocations and prevents double deallocations of the same resource. Borrowing rules complement ownership by managing the lifetimes of references. A reference can only live as long as its owner, and the compiler's borrow checker enforces this property statically [65]. This guarantees that no reference outlives the object it points to, thus preventing dangling pointers. Furthermore, the aliasing rules enforce that multiple immutable references may co-exist, but only exactly one mutable reference may exist. This rule ensures not only memory safety but also no data-race in multi-threaded programs, making Rust unique among system programming languages in providing memory and thread safety.

As with spatial safety, unsafe code provides privileges that bypass these temporal safety guarantees. In unsafe blocks, developers can create and manipulate raw pointers, cast lifetimes manually, or interface with foreign code, none of which are verified by the borrow checker. While powerful, these capabilities open the door to temporal safety violations: a raw pointer may outlive the object it points to, memory may be freed multiple times, or stale references may be dereferenced after deallocation [9, 108]. Thus, even though safe Rust guarantees temporal safety at compile time, unsafe constructs remain the primary source of temporal vulnerabilities.

Pointer Analyses in Rust

Pointer or alias analysis is a long-standing problem in program analysis, aimed at determining whether two pointers may refer to the same memory location at a given program point [48]. Classical pointer analysis is often formulated in terms of three relations: must-alias, may-alias, and no-alias. If the analysis cannot prove that two pointers always refer to distinct memory locations, it conservatively labels them as may-alias. This conservative strategy ensures soundness but typically yields significant over-approximation.

In the context of C and C++, such over-approximation is common and inevitable because of the languages' permissive pointer semantics. Arbitrary casts, unchecked pointer arithmetic and memory accesses make it difficult for static analyses to rule out aliasing relationships precisely [33, 49]. As a result, alias analysis often produces large sets of may-alias relationships, which increases false positives in downstream applications such as bug detection, sanitizer instrumentation, or optimization.

When applied to Rust, classical pointer analyses become even less effective. Rust introduces a unique ownership and borrowing model in which each object has a single owner, and references must follow strict lifetime and mutability rules. The Rust compiler enforces these rules at compile time through the borrow checker, guaranteeing memory safety in safe code. However, general-purpose alias analyses are unaware of these semantics. They conservatively assume that pointers can alias in ways that Rust's type system and ownership discipline explicitly forbid. Consequently, these analyses not only inherit the over-approximation problems seen in C/C++ but amplify them in Rust, producing unnecessary false positives.

A prominent example is SVF (Static Value-Flow Analysis) [131], a widely used state-of-the-art pointer and alias analysis framework. SVF supports advanced inter-procedural, flow-sensitive, and context-sensitive analyses and has been applied to numerous domains, including security and compiler optimization. Notably, both ERASan [88] and RustSan [27], the state-of-the-art Rust memory safety sanitizers, adopt SVF in their underlying static analysis to identify potentially unsafe pointers before applying sanitizer instrumentation. While powerful in theory, SVF exhibits two fundamental drawbacks in the Rust context.

First, SVF does not align with Rust's ownership semantics. Since it does not model ownership and borrowing rules, SVF conservatively reports aliasing relationships that cannot occur in Rust programs. This leads to imprecise identification of unsafe pointers, producing excessive and redundant instrumentation when integrated into sanitizers.

Second, SVF incurs substantial analysis cost. The combination of context-sensitivity, flow-sensitivity, and inter-procedural reasoning introduces high computational complexity, resulting in excessive analysis times on medium- to large-scale codebases [64, 69]. These scalability issues make SVF impractical for use in large-scale or production Rust analysis pipelines, where timely feedback and low overhead are essential.

Address Sanitizer

A widely adopted approach to enforcing memory safety at runtime is *memory safety sanitizers*. These tools instrument program code to insert runtime checks that validate the safety of memory operations, with the goal of detecting both spatial errors (e.g., buffer overflows, out-of-bounds accesses) and temporal errors (e.g., use-after-free, double free). Unlike static analyses, which analyzes code features without execution, sanitizers monitor actual program behaviors, thereby leveraging precise program states of concrete execution paths to detect memory safety violations. Their general applicability have led to integration into mainstream compilers and adoption in large-scale software projects across multiple languages, particularly C and C++.

Among these tools, AddressSanitizer (ASan) [122] stands out as the most widely deployed memory safety sanitizer due to practicality and effectiveness. It has been integrated into both GCC and Clang/LLVM toolchains, and is frequently used in production-scale projects written in C, C++, and, more recently, Rust. ASan works by instrumenting every memory access instruction (i.e., loads, stores, and stack or heap accesses) with runtime checks that validate whether the access is legitimate. This design makes ASan both comprehensive and practical, enabling it to uncover numerous previously unknown memory safety bugs in real-world software.

ASan's detection capability is built on three core design mechanisms: *red zones*, *shadow memory*, and *quarantine*.

Red zones. ASan surrounds allocated memory objects with red zones, which are memory regions acting as buffers between valid objects. Any access that falls into a red zone triggers an error, thereby catching common buffer overflows and out-of-bounds accesses.

Shadow memory. To efficiently check temporal memory safety, ASan maintains a shadow memory space that encodes the validity status of program's memory regions. For each byte of program memory, a corresponding entry in the shadow memory indicates whether it is allocated or not. Instrumented memory accesses consult the shadow memory to validate whether the memory address is valid, thereby catching use-after-free and double-free bugs.

Quarantine. To strengthen shadow memory mechanism and mitigate temporal errors, ASan employs a quarantine mechanism. When memory is freed, it is not immediately returned to the allocator. Instead, it is placed into a quarantine pool for a limited time, ensuring that dangling pointers to the freed region are less likely to be reused immediately. Once the quarantine expires, the memory is recycled and reallocated for new objects.

While ASan's combination of red zones, shadow memory, and quarantine provides a practical defense against many memory errors, the approach suffers from two fundamental limitations: limited bug detection capability and and high runtime and memory overhead.

ASan does not guarantee complete detection of spatial and temporal errors. For spatial safety, its red zones are fixed in size. Large buffer overflows that extend beyond the red zone can access adjacent valid objects without being detected, particularly in heap or global data regions. For temporal safety, shadow memory is updated once a memory region is reallocated, erasing evidence of prior dangling pointers. Although the quarantine mechanism delays reuse, it only provides short-lived protection. If a dangling pointer remains in scope after the quarantine expires and the region is reallocated, the error will go undetected. As a result, ASan may miss use-after-free bugs or overflows that exceed red zones.

ASan's design also introduces significant overhead, which limits its applicability in performance-critical settings. Instrumentation of every memory access typically slows down execution by at least 2–3×, depending on workload characteristics. In addition, the use of red zones and shadow memory imposes large memory costs: red zones increase fragmentation, while shadow memory and metadata can double or even triple total memory usage in some applications. This overhead makes ASan impractical for deployment in resource-constrained environments or for long-running production systems.

These limitations highlight the need for more precise and lightweight sanitization mechanisms. Ideally, a sanitizer should retain ASan's comprehensive bug detection

capability but eliminate reliance on coarse-grained shadow memory and red zones, instead adopting more fine-grained, semantics-aware mechanisms that align with Rust's ownership and borrowing semantics.

2.2.2 Related Work

Memory Sanitizing

The most widely adopted sanitization tools are AddressSanitizer (ASan) [122] and its successors. Among them, ERASan [88] and RustSan [27] are particularly designed for Rust, both of which aim to reduce ASan's prohibitive performance and memory costs while preserving its ability to detect spatial and temporal memory violations.

ERASan [88] introduces the idea of selective sanitization for Rust programs. Instead of instrumenting all memory accesses, ERASan leverages SVF (Static Value-Flow Analysis) [131] to identify program locations where raw pointers or their aliases may be used and thus violate memory safety. ASan checks are then retained only for these potentially unsafe accesses. This strategy significantly reduces the instrumentation overhead compared to ASan's whole-program instrumentation, while still providing equivalent memory safety guarantees. However, the reliance on whole-program pointer analysis makes ERASan computationally expensive during compilation, as SVF computes exhaustive alias information across the entire program. This results in long compile times and poor scalability for large codebases.

RustSan [27] adopts a similar approach but extends selective sanitization to cover a broader range of unsafe behaviors. By analyzing unsafe constructs beyond raw pointer usage, RustSan targets those program points where Rust's ownership and borrowing guarantees no longer apply. Like ERASan, it employs SVF to statically identify relevant pointers, and then preserves ASan checks only for those operations. While this reduces runtime overhead compared to full ASan, RustSan still inherits the scalability limitations of SVF-based pointer analysis and thus suffers from high compile-time cost.

In contrast, our solution LiteRSan achieves lightweight yet comprehensive memory safety enforcement through improvements in two critical dimensions. First, LiteRSan provides more complete safety guarantees by precisely identifying unsafe pointers through a Rust-specific static analysis that aligns with ownership and lifetime semantics, rather than relying on whole-program alias information. This precision allows LiteRSan to eliminate redundant checks without missing subtle temporal or spatial violations. Second, LiteRSan deploys a lightweight metadata-based runtime validation mechanism

in place of ASan's coarse-grained red zone and shadow memory mechanisms, achieving significantly lower runtime and memory overhead. Finally, because LiteRSan avoids SVF's heavyweight alias analysis, it achieves significantly lower compile-time overhead than ERASan and RustSan, making it practical for large-scale Rust programs.

Memory Isolation

Another major line of defense against unsafe code, which includes both unsafe Rust code and external C/C++ libraries, is memory isolation, which aims to confine the effects of memory errors by separating trusted and untrusted regions. Rather than detecting errors directly, isolation approaches focus on ensuring that memory used exclusively by safe Rust code cannot be corrupted by unsafe components.

XRust [74] was one of the earliest attempts to protect Rust programs from unsafe code through isolation. XRust builds on SVF [131] to identify memory objects and pointer dereferences that may be influenced by unsafe code. Once these objects are identified, XRust instruments bounds checks around pointer dereferences to ensure they remain within valid object boundaries. This design prevents unsafe Rust code from corrupting safe Rust memory, but at the cost of runtime overhead from added bounds checking. Moreover, the reliance on SVF introduces scalability challenges when analyzing large Rust codebases.

Trust [14] takes a different approach by leveraging hardware-assisted memory protection. Specifically, Trust employs Intel Memory Protection Keys (MPK) [55] to enforce isolation between memory that is exclusively used by safe Rust code and memory that may be accessed by unsafe code. Similar to XRust, it uses SVF to identify unsafe accesses, but instead of inserting software-based checks, Trust configures memory protection keys to prevent unsafe code from directly manipulating safe memory regions. This approach significantly reduces runtime overhead compared to pure software-based checking, while still protecting the integrity of safe Rust memory.

MetaSafe [61] focuses on protecting critical metadata associated with Rust's safe abstractions, such as the length fields of `String` or `Vec`. MetaSafe moves this metadata into a dedicated, isolated memory region, which is protected using Intel MPK. This ensures that even if unsafe code manipulates raw pointers to data buffers, the associated metadata—which governs bounds and ownership semantics—remains protected. By safeguarding metadata integrity, MetaSafe prevents unsafe code from undermining Rust's compiler-enforced safety guarantees.

PKRU-Safe [64] addresses a key limitation of the above approaches: the heavy compile-time cost of SVF-based analysis. Instead of relying on static alias analysis, PKRU-Safe employs lightweight dynamic monitoring to identify unsafe memory accesses at runtime. Once identified, these accesses are isolated using Intel MPK. This hybrid approach reduces compile-time overhead while still leveraging hardware isolation to protect safe memory. However, it inherits the limitations of MPK, including limited number of keys and potential key management complexity in multi-threaded contexts.

Sandcrust [66] focuses on protecting Rust applications that depend on untrusted external C libraries. Rather than attempting to isolate unsafe memory within a single process, Sandcrust enforces process-level separation: unsafe library code is executed in a separate sandboxed process, with communication handled through inter-process communication (IPC). This strong isolation boundary prevents unsafe C code from corrupting Rust's safe memory space, albeit with higher overhead from context switches and IPC operations.

Fidelius Charm [7] addresses similar challenges in the context of untrusted C libraries. Instead of sandboxing the entire library, it selectively protects sensitive data by migrating it to and from protected memory pages when crossing library boundaries. Before calling into an untrusted C library, sensitive data is copied out of protected memory, and once the library returns, the data is restored. This approach reduces the performance overhead of full sandboxing but requires careful data movement to prevent leaks.

While these isolation-based techniques provide strong protection for safe Rust memory, they share common limitations. By design, they allow memory errors within unsafe code to occur, only ensuring that the damage does not propagate into safe Rust regions. As a result, vulnerabilities such as use-after-free or buffer overflows can still manifest inside unsafe code itself, potentially leading to denial-of-service or exploitable states if the unsafe code interacts with external components.

In contrast, LiteRSan goes beyond isolation by actively detecting memory safety bugs both within unsafe code and in "safe" code that is indirectly affected by unsafe constructs. For example, temporal violations on smart pointers, caused by misuse in unsafe code, can propagate into safe abstractions. LiteRSan directly detects such bugs through lightweight runtime checks, whereas isolation-based approaches would allow them to occur silently. That said, like other sanitization tools, LiteRSan provides more comprehensive bug detection coverage than isolation-based techniques.

Static Bug Detection

Static analyses for Rust aim to detect memory safety and correctness bugs without executing programs, typically by reasoning over Rust's mid-level IRs (e.g., MIR) or over crate source code. Compared with dynamic sanitization, static tools can achieve broad code coverage and catch bugs early in the development cycle. However, they often trade precision for scalability and thus can suffer from high false-positive rates.

Rudra [13] targets ecosystem-scale analysis of unsafe Rust and focuses on three high-impact bug families: potential overflow, uninitialized memory, and panic safety issues. It builds pattern-based static analyses that scan crates on `crates.io`, combining syntactic heuristics with targeted data-flow reasoning to surface likely memory-safety vulnerabilities in unsafe code. Rudra demonstrated large-scale practicality by analyzing the Rust package ecosystem and reporting numerous previously unknown issues, but its pattern-driven approach can over-approximate in ambiguous contexts, contributing to notable false-positive rates.

MirChecker [72] performs fully automated bug detection directly on Rust's Mid-level Intermediate Representation (MIR). It introduces abstract-interpretation domains tailored to MIR to reason about numeric properties, pointer states, and control flow, enabling checks for memory-safety violations and runtime crashes that elude purely syntactic scans. By operating at MIR, MirChecker benefits from compiler desugaring while retaining enough structure to analyze ownership-relevant operations; nevertheless, like many abstract-interpretation systems, its precision depends on domain choices and widening or narrowing strategies, which can lead to conservative alarms.

Rupair [51] focuses on buffer overflows in Rust and goes beyond detection to automatic rectification. It uses static analysis to locate overflow-prone code patterns and then generates patches to fix the issues, aiming to reduce developer effort in triaging and remediation. This detection-and-repair pipeline shows that actionable fixes can be synthesized for a class of memory errors in Rust, though its specialization to buffer overflows limits coverage of other bug families.

SafeDrop [29] studies invalid memory deallocation in Rust, such as use-after-free and double-free arising from ownership-based resource management. It implements a path-sensitive, field-sensitive static data-flow analysis over crate APIs, using a modified Tarjan-style algorithm for scalable path sensitivity and a cache-based strategy for efficient interprocedural reasoning. Evaluations indicate high recall on known CVEs with moderate analysis overhead compared to baseline compilation time.

FFIChecker [71] concentrates on cross-language memory-management issues at Rust's Foreign Function Interface (FFI) boundaries. It statically reasons about ownership mismatches and lifetime violations that occur when Rust interacts with manual memory management in C/C++. By modeling how objects cross FFI calls and how their lifetimes are (mis)managed across languages, FFIChecker detects errors that conventional Rust-only analyses miss.

SyRust [135] complements static bug finders by automatically generating well-typed Rust programs to test library APIs. It introduces a program-synthesis technique that encodes typing and ownership constraints as logical formulas so that generated tests respect Rust's borrowing rules and API chaining constraints. This semantic-aware generation improves effectiveness for libraries while ensuring synthesized programs compile and execute, broadening dynamic coverage in areas that static analyses flag as suspicious.

These tools demonstrate strong coverage within their targeted scopes: ecosystem-wide unsafe code scanning (Rudra), MIR-level numeric/semantic checks (MirChecker), overflow detection and repair (Rupair), deallocation errors (SafeDrop), FFI lifetime/ownership mismatches (FFIChecker), and semantic test generation (SyRust). However, they also illustrate common challenges of static detection in Rust: precision limits from over-approximation (e.g., pattern matches or conservative abstract domains), high false-positive rates in complex ownership/aliasing scenarios, and non-trivial analysis time at scale (especially for interprocedural, path-sensitive data-flow). For instance, Rudra achieves broad detection but reports a substantial fraction of benign findings under large-scale scans, reflecting the inherent conservatism of pattern-based static analysis.

In contrast, LiteRSan focuses on sound bug manifestation at runtime with selective, semantics-aware instrumentation. Rather than classifying potential bugs purely statically, it aligns with Rust's ownership and lifetime rules to precisely identify unsafe pointers and then performs lightweight dynamic validation. This hybrid stance avoids false positives (every reported violation corresponds to an actual illegal access) while keeping compile-time costs and runtime overhead low, complementing static tools that provide coverage but may require substantial analyses.

2.3 Takeaways

Prior work on indirect call resolution spans a wide spectrum of techniques. Conservative solutions guarantee soundness but are excessively imprecise, leading to large target sets that undermine their downstream utility. Type-based analyses improve scalability and

are straightforward to implement, but they lack the precision needed to handle complex programs where many functions share similar type signatures. Data-based analyses, by contrast, achieve far greater accuracy by tracking pointer propagation through program states, yet this precision comes at the expense of prohibitive compile-time cost and scalability limitations. Dynamic approaches offer runtime precision but at the cost of runtime overhead and incomplete coverage. Collectively, these approaches illustrate the long-standing trade-off between efficiency and precision in resolving indirect calls.

Efforts to enforce memory safety in Rust have likewise followed multiple directions. Sanitization tools, such as ASan and its Rust-specific derivatives (ERASan, RustSan), provide broad bug-detection coverage but incur high runtime and memory overhead, limiting their practicality. Memory isolation approaches use static or dynamic analyses combined with hardware mechanisms (e.g., MPK) to protect safe Rust memory, but they cannot prevent bugs within unsafe code itself. Static bug detection tools analyze ownership, lifetimes, and data flows to detect potential vulnerabilities, but they often suffer from high false-positive rates and long analysis times when scaled to large codebases. These lines of work underscore the challenge of achieving strong safety guarantees in Rust without sacrificing performance.

Across both domains, from indirect call target identification to Rust memory safety enforcement, the central lesson is consistent: *achieving high precision often comes at a prohibitive efficiency cost.* Conservative or lightweight analyses scale well but lack of accuracy, while precise analyses offer accuracy at the cost of scalability and efficiency. The following chapters present two approaches, DeepType and LiteRSan, which are designed to bridge this gap in respective research domains. By leveraging targeted, semantics and domain specific static analyses, these systems demonstrate that it is possible to achieve both precision and efficiency, addressing the long-standing barrier in program analysis research.

Chapter 3
DeepType: Indirect Call Target Resolution

Indirect call, used commonly to determine the functions to be called at runtime, is a fundamental feature of C/C++ for achieving dynamic program characteristics. Production software (e.g., nginx) and operating systems (e.g., Linux) intensively utilize indirect calls to dynamically adapt program behaviors according to runtime environments and demands, through loading and linking the desired shared libraries.

Precise identification of indirect call targets is of paramount importance, as the control-flow transitions between indirect calls and their respective targets play an essential role in the construction of a global control-flow graph (CFG), which is extensively adopted in various security-related fields. Static analysis tools rely on CFG for bug detection and program hardening [52, 57, 60, 73, 120, 170, 172, 173, 176, 178], program partitioning and privilege separation [19, 23, 54, 76, 80, 113], pruning redundant paths in symbolic execution [18, 21, 26, 147, 160], and guiding directed fuzzing for specific objectives [17, 24, 96, 102, 105, 106, 125]. Additionally, control-flow Integrity (CFI) defenses [1, 20, 42, 99, 100, 145, 175, 177] have been proposed to mitigate control-flow hijacking attacks. However, the strength of CFI depends on the precision of CFG. Imprecise CFG construction results in advanced attacks that bypass CFI [56, 82, 112, 121, 123, 163].

The key challenge in construction of an accurate CFG is identifying the targets of indirect calls. Modern compilers like GCC and Clang cannot determine these targets without additional analysis and instrumentation. Conservative approaches [175, 177] consider all functions or those with address taken as potential targets for each indirect call, producing a considerable number of false positive edges within CFG, which impair the functionality of the applications built upon CFG and impose unnecessary cost. Data tracking analysis [42, 62, 124, 130, 178] tracks value flow, which pursues accuracy at the cost

of high performance overhead while the accuracy depends on the precision of the taint analysis and points-to analysis techniques they employ. Type-based analysis [99, 145, 150] checks function signatures to identify functions whose types match with an indirect call. While this approach is efficient, it is susceptible to false positive targets if many functions share the same type as the actual target.

Multi-layer type analysis (MLTA) [78] was proposed for the purpose of improving the accuracy of type-based analysis. Given the fact that function pointers can be members of composite data structures, the type of a function pointer along with the composite types holding it compose a multi-layer type. For example, if function pointer `ptr` with type `void (int)*` is a member of object `a` with type `struct.A`, the variable `a.ptr` has multi-layer type `void (int)* | struct.A`. A function f also has this multi-layer type if it is assigned to `a.ptr`. MLTA matches multi-layer types of functions and indirect calls to refine indirect call targets.

However, multi-layer types introduce challenges in type matching because address-taken functions may be propagated between different multi-layer types through information flow, making it hard to collect all targets for an indirect call (see Section 3.1.2). MLTA bypasses the challenges by splitting a multi-layer type into several two-layer types, and adopts each layer's type as the basic unit for information storage and type matching (see Section 3.1.1), which avoids missing targets while producing false positive targets. It relinquishes the type information between spitted layers and weakens the restrictions provided by multi-layer types, thereby negatively affecting accuracy.

This paper proposes an advanced approach, Strong Multi-Layer Type Analysis (SMLTA), to mitigate the false positive targets produced by MLTA. It adheres to the strong restriction that identifies only those functions as targets whose entire multi-layer types match with the indirect calls. SMLTA addresses the challenges in multi-layer type matching by resolving the relationships between multi-layer types based on the directions of information flow, and utilizes an adapted breadth-first search (BFS) algorithm [157] to discover all multi-layer types engaged in the propagation of target functions. It also employs a conservative strategy to deal with ambiguous type information due to information flow.

We implemented SMLTA within a prototype, DeepType,[1] which contains two working phases: 1) information collection and 2) target identification. In the first phase, DeepType establishes and maintains the mappings between multi-layer types and their associated functions (Section 3.3.1). The collected multi-layer types are organized in a

[1] DeepType is available at https://github.com/s3team/DeepType.git.

hierarchical manner for the purpose of quick retrieval (Section 3.3.2). It also preserves the relationships between multi-layer types as indicated by the directions of information flow, facilitating the tracing of multi-layer types engaged in function propagation (Section 3.3.3). In the second phase, DeepType determines the multi-layer type of each indirect call and discovers all other multi-layer types engaged in target propagation (Section 3.4.1). Then, it verifies whether the engaged multi-layer types match with the indirect call to precisely identify associated functions as targets (Section 3.4.2). Additionally, DeepType handles diverse instructions and code patterns in real-world programs to reduce the inaccuracy caused by corner cases (Section 3.5).

We evaluated DeepType on Linux kernel, 5 server programs, and 14 user-level applications (Section 3.6). We compared it with TypeDive (i.e., the prototype of MLTA), as MLTA is the-state-of-the-art approach in type-based analysis. The results indicate that DeepType outperforms TypeDive in the precision of indirect call target identification, reducing the average number of indirect call targets by 43.11% on average across most benchmarks. In terms of performance, DeepType decreases the runtime overhead by 5.45% to 72.95% and achieves lower memory consumption in all evaluated benchmarks. Additionally, a case study on a real-world CVE shows that SMLTA is more powerful than MLTA in reducing attack surface and preventing exploits.

In summary, this work makes the following contributions.

- Propose a novel approach called Strong Multi-Layer Type Analysis (SMLTA) that employs strong restrictions provided by multi-layer types to refine indirect call targets.
- Develop a prototype, DeepType, which overcomes challenges in multi-layer type matching and utilizes SMLTA to precisely and efficiently identify indirect call targets.
- Evaluate DeepType with 20 benchmarks and compare it with TypeDive, exhibiting its capability in further refining indirect call targets and reducing both runtime overhead and memory consumption. Additionally, we demonstrate that SMLTA offers a higher level of security.

3.1 Motivation and Challenges

This section clarifies the challenges in indirect call target identification using multi-layer types, describes how MLTA and SMLTA address these challenges, and highlights the accuracy improvement achieved by SMLTA through an example.

3.1.1 Motivating Example

Listing 3.1 shows a code snippet with buffer overflow vulnerability. The `strcpy` at line 24 can overwrite the memory location adjacent to `buf` if the length of `msg` is larger than `MAX_LEN`. The adjacent memory location belongs to `w_op`, which is afterwards assigned to `u->uw` at line 25. So, the function pointer `u->uw->low_priv` can point to an address manipulated by the attacker after buffer overflow occurs. At line 28, this function pointer is used to make an indirect call, resulting in arbitrary execution if a lenient CFG is deployed on this program. To prevent such control-flow hijacking attack and other unexpected bugs caused by imprecise CFG, precisely identifying indirect call targets is crucially required.

```
1  typedef void (*fp)(char*);
2  struct Write {fp low_priv; fp high_priv;};
3  struct User {struct Write *uw; ...};
4  struct Kernel {struct Write *kw; ...};
5
6  void func_init(struct Write *w_op, struct Kernel *k) {
7      w_op->low_priv = &write_to_shared_mem;
8      w_op->high_priv = &write_to_protected_mem;
9      k->kw->low_priv = &write_to_protected_mem;
10     k->kw->high_priv = &write_to_kernel_mem;
11 }
12
13 void user_priv_write(fp icall_ptr, char *buf) {
14     ...
15     (*icall_ptr)(buf);
16 }
17
18 void write_to_mem (char *msg) {
19     struct Kernel *k;
20     struct User *u;
21     struct Write *w_op;
22     char buf[MAX_LEN];
23     func_init(w_op, k);
24     strcpy(buf, msg);   // buffer overflow
25     u->uw = w_op;
26     ...
27     if (user_mode()) {
28         if (low_priv()) (*u->uw->low_priv)(buf);
29         else user_priv_write(u->uw->high_priv, buf);
30     }
31 }
```

Listing 3.1: A program vulnerable to control-flow hijacking attack through indirect call. The `strcpy` at line 24 has buffer overflow vulnerability, which allows attackers to rewrite the function pointer `u->uw->low_priv` and redirect control-flow through the indirect call at line 28.

Traditional Type-Based Analysis

Traditional type-based analysis examines the signatures of address-taken functions and the types of indirect calls. If these types match, the function is identified as a potential target. This approach is not impacted by the challenges described in Section 3.1.2 because it solely relies on the types of function pointers, ignoring composite data structures holding them. In Listing 3.1, the indirect call at line 28 has type `void (char*)*`, which matches with functions `write_to_shared_mem`, `write_to_protected_mem`, and `write_to_kernel_mem`. As a result, all of them are identified as targets though only `write_to_shared_mem` is the real target. Similarly, the indirect call at line 15 also has the three targets while only function `write_to_protected_mem` is the real target.

Multi-Layer Type Analysis (MLTA)

MLTA utilizes the extra type information extracted from composite data structures and supports field-sensitivity considering that a composite data structure may have multiple members holding different functions. If one function is assigned to a pointer, MLTA records the mappings between the multi-layer type and the function, which is called "type-func confinement". It splits multi-layer types and uses each layer type along with an index as key, as Table 3.1 shows. Similarly, the original and transformed types in type assignment and casting operations are logged in a split manner as well.

In Listing 3.1, the indirect call at line 28 has multi-layer type `void (char*)* | struct.Write | struct.User`. MLTA identifies potential targets by collecting associated functions for each layer and calculating the intersection to find common functions. For the first layer `void (char*)*` and the second layer `struct.Write` with index 0, both have no original type. Thus, MLTA retrieves the associated functions from Table 3.1, resulting in set {`write_to_shared_mem`, `write_to_protected_mem`, `write_to_kernel_mem`} for the first layer and set {`write_to_shared_mem`, `write_to_protected_mem`} for the second layer. The third layer `struct.User` with index 0 has an original type `struct.Write` (see line 25), thus MLTA gathers the associated functions for both types, generating a set {`write_to_shared_mem`, `write_to_protected_mem`, `write_to_kernel_mem`}. Finally, it computes the intersection of these three sets. The common functions in resulting set {`write_to_shared_mem`, `write_to_protected_mem`} are identified as targets. The indirect call at line 15 with type `void (char*)*` has no original type. So, MLTA directly generates the target set {`write_to_shared_mem`, `write_to_protected_mem`, `write_to_kernel_mem`}, according to Table 3.1. In contrast with tradition type-based

Type	Index	Functions

`void (char*)*`	-	`write_to_shared_mem(7)`
		`write_to_protected_mem(8,9)`
		`write_to_kernel_mem(10)`

`struct.Write`	0	`write_to_shared_mem(7)` `write_to_protected_mem(9)`
	1	`write_to_protected_mem(8)` `write_to_kernel_mem(10)`
`struct.User`	0	-
	...	-
`struct.Kernel`	0	`write_to_protected_mem (9)` `write_to_kernel_mem(10)`
	...	-

Table 3.1: Mappings between types and functions in MLTA for Listing 3.1. MLTA splits multi-layer types into two-layer types and maintains mappings between these types and associated functions. The two-layer types are presented by a composite type along with an index, which indicates the member type at a specific position. For example, `struct.Write` with index 0 represents the two-layer type `void (char*)* | struct.Write`. The numbers in parenthesis indicate the line numbers where the functions are confined to the types.

analysis, MLTA narrows down the target set for the indirect call at line 28, but there is still a visible gap between MLTA result and ground truth.

Strong Multi-Layer Type Analysis (SMLTA)

SMLTA employs entire multi-layer types as keys for information storage, as Table 3.2 shows, to circumvent the false positive targets caused by splitting multi-layer types. The relationships between multi-layer types are also recorded by entire multi-layer types. We call multi-layer type `T2` a "`friend type`" relative to multi-layer type `T1` if information flows from `T2` to `T1` (i.e., a function may be propagated from `T2` to `T1`). To identify targets for an indirect call, SMLTA exhaustively searches for its friend types and gathers all associated functions as targets.

In Listing 3.1, the indirect call at line 28 has multi-layer type `void (char*)* | struct.Write#0 | struct.User#0` where "`#0`" indicates the index. SMLTA exhaustively discovers all friend types that may have information flowing to this multi-layer type, only `void (char*)* | struct.Write#0` in this example, and then gathers associated functions of these multi-layer types. The resulting set {`write_to_shared_mem`} contains the identified target, which is exactly the real target. The indirect call at line 15 has type `void (char*)*`. Because the function pointer `icall_ptr` is a parameter, SMLTA adds a

Type	Functions
`void (char*)* | s.Write#0`	`write_to_shared_mem(7)`
`void (char*)* | s.Write#1`	`write_to_protected_mem(8)`
`void (char*)* | s.Write#0 | s.Kernel#0`	`write_to_protected_mem(9)`
`void (char*)* | s.Write#1 | s.Kernel#0`	`write_to_kernel_mem(10)`

Table 3.2: Mappings between types and functions in SMLTA for Listing 3.1 program. SMLTA treats each multi-layer type as a whole and uses the entire multi-layer type as a basic unit in storage-purposed data structures. In this table, the structs are abbreviated as "s". The index of each member in a composite type is denoted as "`#N`" where `N` is a number. For instance, `s.Write#0` represents `struct.Write` with index 0.

"`fuzzy type`" as its outer layer, which matches with any type. Thus, this indirect call can match with all address-taken functions whose first layer is `void (char*)*`, resulting in the target set {`write_to_shared_mem`, `write_to_protected_mem`, `write_to_kernel_mem`} without missing potential targets.

3.1.2 Challenges

The use of multi-layer types presents challenges in matching indirect calls with all potential targets, particularly when functions are propagated across multi-layer types due to information flow. We categorize the challenges into three classes consistent to three forms of information flow.

First, type assignment and casting operations can transform one multi-layer type into another, resulting in information flow from the original to the transformed type, as well as affecting members of composite types. For example, the type assignment operation (line 25) indicates information flow from `w_op` to `u->uw`, as well as `w_op->low_priv` to `u->uw->low_priv`. In this context, the function `write_to_shared_mem` with multi-layer type `void (char*)* | struct.Write` (line 7), should be propagated to `void (char*)* | struct.Write | struct.User`. Because information flow affects individual layers as well as overall multi-layer types, it is challenging to track various multi-layer types involved in function propagation and identify all potential targets.

Second, a function pointer performing as an actual parameter can sometimes discard outer layers during parameter passing, leading to discrepancies of type information at different positions. For instance, the actual parameter `u->uw->high_priv` (line 29) has multi-layer types `void (char*)* | struct.Write` and `void (char*)* | struct.Write | struct.User`, while the corresponding formal parameter `icall_ptr` (line 13) has type `void (char*)*`. The obscured outer layers can result in mismatch of

multi-layer types, making it challenging to identify all potential targets for an indirect call that uses a formal parameter.

Third, some mechanisms, such as virtual tables [159] used by compilers, can introduce information flow between composite types (e.g., `struct.Write`) and general pointer types (e.g., `char*`). These general pointer types can hinder function propagation since they do not have associated functions, resulting in missing potential targets.

MLTA bypasses these challenges by splitting each multi-layer type, thus simplifying the complex information flow between them to straightforward information flow between traditional types, which is easier to track. It conservatively confines a function to each layer of its multi-layer type, ensuring no missing target even if the outer layers are discarded. Specifically, to overcome the third challenge, MLTA marks all general pointer types and composite types that interact with them as "escaping types". It skips the layers of escaping types when calculating intersection, which removes the impact of general pointer types.

In SMLTA, the first challenge is addressed by maintaining the relationships between multi-layer types and the exhaustive search of all friend types engaged in function propagation. The second challenge is addressed by fuzzy type, which conservatively admits that all types could possibly match with the discarded outer layer types, ensuring no missing targets. The third challenge is also solved by conducting an exhaustive search of friend types. This search treats general pointer types as bridges between composite types, unblocking function propagation stuck on general pointer types.

3.2 Overview and Workflow

The prototype of SMLTA is called DeepType, which contains two phases: *Information Collection* and *Target Identification*. The workflow is shown in Figure 3.1.

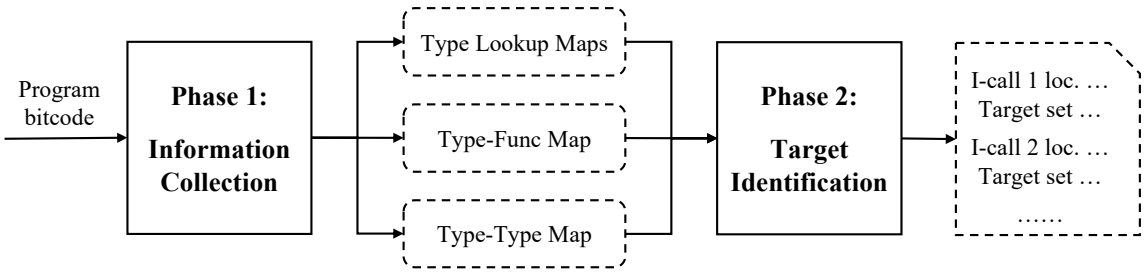

Figure 3.1: Workflow of DeepType. DeepType contains two working phases. In phase 1, it collects and records type information in three data structures. In phase 2, it refers to the recorded type information to discover friend types, gather associated functions, and identify targets.

Given bitcode as input, **phase 1** analyzes initialization instructions and type propagation instructions to collect information. An initialization instruction assigns a function to a function pointer. We say, it *confines* the function to the corresponding multi-layer type. The mappings between the entire multi-layer types and associated functions are stored in *Type-Func Map*. For quick access and retrieval purpose, DeepType archives these multi-layer types in *Type Lookup Maps* in a hierarchical manner using multi-layer mappings. Type propagation instructions include type assignment and casting instructions that possibly propagate functions from one multi-layer type to another through information flow. DeepType records the mappings between destination and source multi-layer types of information flow in *Type-Type Map* using entire multi-layer types as keys and values. The sources are *friend types* relative to destinations.

In **phase 2**, DeepType analyzes each indirect call instruction to figure out its multi-layer type. If the function pointer is a formal parameter, *fuzzy type* is added to represent potentially discarded outer layers. Then, DeepType exhaustively searches for friend types from *Type-Type Map*, retrieves multi-layer types that match with the indirect call or its friend types from *Type Lookup Maps*, and obtains associated functions from *Type-Func Map*. The union of all associated functions are potential targets for the indirect call. Finally, DeepType outputs a list of indirect calls in the analyzed program along with their locations and respective targets.

3.3 Design of Phase 1: Information Collection

This section presents how SMLTA collects information from initialization and type propagation instructions, and how it organizes multi-layer types hierarchically.

3.3.1 Type-Function Confinements

A function could be considered as a potential target of an indirect call only if it is utilized to initialize local and/or global variables. We confine the function to the multi-layer type of the initialized function pointer by establishing a mapping between them in *Type-Func Map*. It is straightforward to determine the multi-layer types of local variables, by extracting the types layer by layer as they are loaded. However, the multi-layer types of formal parameters and nested global variables may not be fully ascertainable using the same method.

The multi-layer type of a formal parameter is uncertain because it may have extra outer layers that are lost during parameter passing. To complete its multi-layer, we introduce *fuzzy type*, as defined in Definition 1, to cover for missing layers. The fuzzy type can be matched with any type. For instance, `void (int)* | fuzzy type` matches with any multi-layer type whose first layer is `void (int)*`, including `void (int)*`, `void (int)* | struct.A#0`, and `void (int)* | struct.A#0 | struct.X#0`, etc. Similarly, when an index is uncertain,[2] we employ *fuzzy index*, which matches with any index.

Definition 1. ***Fuzzy type*** marks the type of an uncertain layer. The existence of this layer is uncertain, and the type of this layer is uncertain. In type verification, a fuzzy type can match with any type, even if the corresponding layer does not exist.

```
1  typedef struct bfd_target
2  {
3      ...
4      void* (*_bfd_read_ar_hdr_fn) (bfd *);
5      ...
6  } bfd_target;
7
8  extern const bfd_target x86_64_elf32_vec;
9      ...
10 static const bfd_target * const _bfd_target_vector[] =
11 {
12     ...
13     &x86_64_elf32_vec,  // nested global variable
14     ...
15 };
```

Listing 3.2: Nested global variable. `x86_64_elf32_vec` is a nested global variable because it serves as a member in another global variable `_bfd_target_vecor`. This example is from targets.c from binutils-2.35.

A nested global variable is one that initializes members of other variables, thus may have extra layers involved in other initialization instructions. Its multi-layer type is uncertain when analyzing the instruction initializing it. To gather complete type information, we track all variables that hold this global variable iteratively. For instance, Listing 3.2 shows a nested global variable `x86_64_elf32_vec` (line 8) with type `struct.bfd_target`, which initializes a member of another global variable `_bfd_target_vector` (line 13). Thus, the function pointer `_bfd_read_ar_hdr_fn` (line 4), as a member of the global

[2]In LLVM IR, index is typically derived from GetElementPtrInst when the corresponding operand is a constant. However, index may be non-constant operand such as a phi instruction, indicating an uncertain index until runtime.

variable, should be confined to both `void (bfd*)* | struct.bfd_target#53` and `void (bfd*)* | struct.bfd_target#53 | vector.bfd_target#237`, ensuring that a potential target could be identified through both multi-layer types.

3.3.2 Multi-Layer Type Organization

For efficient access and retrieval, we organize the collected multi-layer types hierarchically using multi-layer mappings and store them in *Type Lookup Maps*. We prepare N maps for a program, in which the multi-layer types have up to N+1 layers. To archive a multi-layer type `T`, the first n layers of `T` is used as a key in the n-th map (where $1 \leq n \leq N$), and the n-th layer of `T` is stored as the corresponding value. If `T` consists of M layers, where M is less than N, only the first M maps are utilized for storage.

Figure 3.2 shows an outline of *Type Lookup Maps* and exemplifies the utilization of this data structure via two multi-layer types: ① `void (int)* | struct.A | struct.X` and ② `void (int)* | struct.B | struct.Y | struct.P`.[3] *The First Map* stores the mapping between the common first-layer type `void (int)*` and respective second-layer types `struct.A`, for ①, and `struct.B`, for ②. *The Second Map* stores mappings between first-two-layer types and corresponding third-layer types, among which the first entry is for ① and the second entry is for ②. *The Third Map* works in a similar way to store the rest part of ②. If a multi-layer type contains more than four layers, we will establish more maps based the number of layers, to record the rest parts.

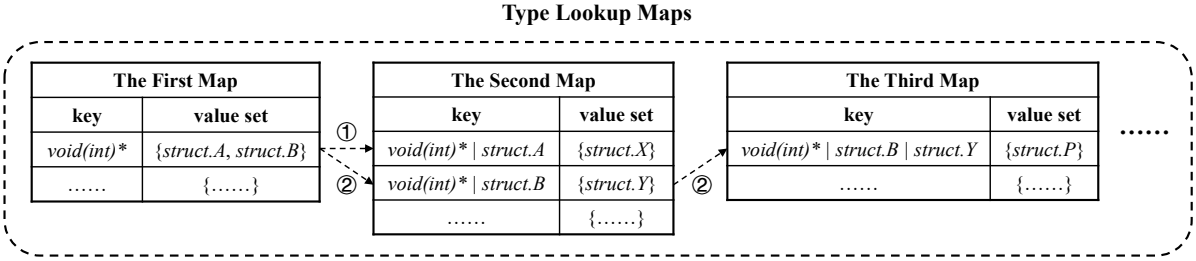

Figure 3.2: Outline of Type Lookup Maps with two example types stored. Type Lookup Maps use multi-layer mappings to archive multi-layer types. The First Map records the first-layer types and corresponding second-layer types; The Second Map records the first-two-layer types and corresponding third-layer types; So on so forth. ① `void (int)* | struct.A | struct.X` and ② `void (int)* | struct.B | struct.Y | struct.P` are archived as examples.

The question is: *how many mappings are sufficient in Type Lookup Maps?* In other word, what is the optimal value of N? This number should neither be too large to

[3]The indexes are omitted to simplify the example and emphasize the key point, which is how to hierarchically archive multi-layer types.

guarantee the performance of DeepType nor too small to hold the multi-layer types. Our solution is to decide the number of mappings conforming to the multi-layer types in the analyzed programs. Take the benchmarks (see Section 3.6.1) in this work as an example, Table 3.3 shows the number of multi-layer types with different layer counts.

Programs	Total	<=8 layers	> 8 Layers
binutils	311*	311*	0
sqlite	220	220	0
nginx	101	101	0
httpd	88	88	0
openvpn	43	43	0
proftpd	88	88	0
sshd	46	46	0
linux	5,447	5,447	0

Table 3.3: Number of multi-layer types with different layer counts. 311* presents the average of binutils programs.

Across all benchmarks, the multi-layer types in them contain at most 8 layers, which means 7-layer mappings are sufficient to record these multi-layer types. Thus, we adopt 7-layer mappings in our implementation. If multi-layer types with more than 8 layers are common in other programs, more mappings can be deployed in DeepType to supply extra precision in practice.

3.3.3 Type Relationship Resolving

A function can be propagated from one multi-layer type to another via type propagation instructions. To collect all involved multi-layer types, we analyze *type assignment* and *casting* instructions and introduce *friend type*, as defined in Definition 2, to describe their relationships.

Definition 2. `A` is a ***friend type*** relative to `B` if either of the following holds:

1. There exists information flow from `A` to `B`.
2. There exists information flow from `A` to `C`, and `C` to `B`.

Due to the information flow, the associated functions of `A` are propagated to `B`. We say, `A` *shares* functions with `B`.

The second condition manifests that one multi-layer type is a friend type relative to another if there exists a chain of information flow following the first condition. This chain

can be as long as the entire program. We record the mapping between one multi-layer type and its *direct* friend types, satisfying the first condition, in *Type-Type Map*. The *indirect* friend types, satisfying the second condition, can be inferred through the direct relationships recorded in *Type-Type Map*.

Type Assignment

A type assignment instruction assigns the value of one variable to another. These two variables have different multi-layer types, but share the same first-layer type (e.g, `void(int)*` and `void(int)*|struct.A`). Such instructions create a one-way relationship: the source type of information flow is a friend type relative to the destination type. For example, Listing 3.3 shows a type assignment operation (line 10) where the return value of function `bfd_get_section_by_name` is assigned to `pupdate->section`, resulting in information flow from the source type `asection` to the destination type `asection | struct.section_add#5`, which allows `asection` to share its associated functions with `asection | struct.section_add#5`.

```
1  struct section_add {...; asection *section;};
2
3  asection *bfd_get_section_by_name (bfd *abfd, const char *name);
4
5  copy_object (bfd *ibfd, bfd *obfd, const bfd_arch_info_type *input_arch)
6  {
7      ...
8      struct section_add *pupdate;
9      ...
10     pupdate->section = bfd_get_section_by_name (ibfd, pupdate->name);
11     ...
12 }
```

Listing 3.3: Type assignment. A variable with multi-layer type `asection` is assigned to another variable with multi-layer type `asection|struct.section_add#5` in line 10. This example is from objcopy.c in binutils-2.35.

Type Casting

A type casting instruction transforms one type into another, resulting in distinct multi-layer types. In Listing 3.4, the original type of `entry` is `struct.bfd_hash_entry*` (line 2). It is transformed to `struct.string_hash_entry` at line 4. The type casting at line 6 transforms `struct.string_hash_entry*` to `struct.bfd_hash_entry*`. Considering

the flow-insensitivity feature of our static analysis, [4] it is uncertain whether a type casting instruction executes before or after an indirect call, making it uncertain whether the multi-layer type of the indirect call is transformed from another multi-layer type. To address this ambiguity, we conservatively establish a bidirectional relationship between the source and destination types, allowing them to share associated functions with each other so that indirect calls do not miss any potential target.

```
1 static struct bfd_hash_entry *
2 string_hash_newfunc (struct bfd_hash_entry *entry, struct bfd_hash_table *table,
      const char *string)
3 {
4     struct string_hash_entry *ret = (struct string_hash_entry *) entry;
5     ...
6     return (struct bfd_hash_entry *) ret;
7 }
```

Listing 3.4: Type casting. `struct.bfd_hash_entry` is casted to `struct.string_hash_entry` in line 4. `struct.string_hash_entry` is casted to `struct.bfd_hash_entry` in line 6. This example is from ecofflink.c in binutils-2.35.

3.4 Design of Phase 2: Target Identification

This section presents how SMLTA identifies indirect call targets utilizing the type information stored in *Type-Func Map*, *Type-Type Map* and *Type Lookup Maps*.

3.4.1 Friend Type Discovery

For each indirect call, SMLTA examines whether the function pointer is a formal parameter to decide if fuzzy type should be added to complete its multi-layer type. Once the multi-layer type of an indirect call is ascertained, we discover all of its friend types to ensure that the associated functions shared by the friend types will be identified as potential targets. Given that a multi-layer type can be partially transformed from another type, we extract all possible *fragments*, as defined in Definition 3, to support the exhaustive search of friend types relative to the entire multi-layer type of the indirect call.

Definition 3. A ***fragment*** of multi-layer type `T` is one or multiple continuous layers in type `T`.

[4] SMLTA is flow-insensitive for efficiency. Tracking the execution sequence of instructions is beyond our scope.

A multi-layer type with N layers has $\frac{(1+N)\times N}{2}$ possible fragments, as exemplified in Table 3.4. The **Fragment** column lists all possible fragments while the layers before and after each fragment are respectively listed in **Before-Part** and **After-Part** columns. We can substitute each fragment with its friend types and concatenate these types with corresponding Before-Part and After-Part to generate friend types relative to the entire multi-layer type, thereby transforming the task of searching for friend types relative to the multi-layer type into discovering friend types for individual fragments.

Before-Part	Fragment	After-Part
	A	|B|C
A|	B	|C
A|B|	C	
	A|B	|C
A|	B|C	
	A|B|C	

Table 3.4: A table recording the fragments of multi-layer type A|B|C. A multi-layer type A|B|C with 3 layers has 6 possible fragments, including single-layer fragments, two-layer fragments, and three-layer fragments.

For each fragment, the search of direct friend types is straightforward, which is achieved by querying *Type-Type Map*. However, the exhaustive search of indirect friend types can be challenging because these types may be buried in long, cyclic chains of information flow. To address this issue, SMLTA employs an exhaustive search algorithm adapted from Breadth-First Search, which monitors the state of discovered friend types to bypass cycles in chains. This algorithm is detailed in Algorithm 1, where Frag represents a fragment and $\mathsf{Frag}_{\mathsf{ft}}$ represents its friend types.

We prepare three sets, $S_{checked}$, $S_{checking}$, and $S_{unchecked}$, to manage friend types in different states and prevent cyclic search. A friend type currently being used to query the *Type-Type Map* is held in $S_{checking}$ and is moved to $S_{checked}$ post-query. $S_{unchecked}$ holds newly discovered friend types for the next round of search. Initially, the current Frag is placed into $S_{checking}$. In each search iteration, we look for friend types for elements in $S_{checking}$ by querying *Type-Type Map* and place newly discovered friend types to $S_{unchecked}$. After each iteration, update the three sets for the following round of search. Repeat this process until $S_{checking}$ is empty and S_{cheked} is no more updated, indicating that all $\mathsf{Frag}_{\mathsf{ft}}$ have been discovered.

Given friend types of each fragment, we generate the friend types relative to the entire multi-layer type of an indirect call by concatenation operations, and gather them in a set S along with the multi-layer type itself for type verification.

Algorithm 1: Exhaustive search of $\mathsf{Frag}_{\mathsf{ft}}$

Input: *Type-Type-Map*, Frag
Output: All $\mathsf{Frag}_{\mathsf{ft}}$ are placed in S_{checked}

```
1  S_checked ← {};
2  S_checking ← {Frag};
3  S_unchecked ← {};
4  S_checked_origSize ← 0;
5  S_checked_newSize ← 0;
6  while S_checking is not empty do
7      foreach e ∈ S_checking do
8          S_unchecked ← SetMerge(S_unchecked, Type-Type-Map[e]);
9      S_checked_origSize ← S_checked.size();
10     S_checked ← SetMerge(S_checked, S_checking);
11     S_checked_newSize ← S_checked.size();
12     S_checking ← ∅;
13     S_checking ← S_unchecked;
14     S_unchecked ← ∅;
15     if S_checked_origSize == S_checked_newSize then
16         return S_checked;
```

3.4.2 Type Verification

The elements in S, are not exactly the multi-layer types that confine target functions of this indirect call due to three kinds of mismatches: 1) Some friend types generated through fragments do not exist among the recorded multi-layer types; 2) Some elements in S, containing fuzzy type and index, apparently differ from but actually match with the recorded multi-layer types; 3) Some recorded multi-layer types, containing fuzzy type and index, apparently differ from but actually match with the elements in S. Hence, we query *Type Lookup Maps* for type verification and collecting *verified types*, which is defined in Definition 4.

Definition 4. A ***verified type*** is a multi-layer type that is recorded in *Type-Func Map* and *Type Lookup Maps*, and matches with an element in S.

To perform type verification, we check each element in S to find verified types in *Type Lookup Maps*. Given a multi-layer type `T` in S, first of all, we pass the first-layer type of `T` to *The First Map* to achieve a scope of second-layer types. Then, check whether the elements in this scope match with the second-layer type of `T`. Concatenate the first-layer type with every matched second-layer type to generate a set of first-two-layer types. Lookup *The Second Map* to achieve a scope of third-layer types and find those who match with the third-layer type of `T`. Following this working pattern to check the

remaining layers until all verified types are retrieved from *Type Lookup Maps*. Finally, query *Type-Func Map* with verified types to find associated functions. The union of such functions are identified as potential targets of the indirect call.

3.5 Implementation

DeepType is built on LLVM 15.0 in 2.8k lines of C++ code. It supports C and C++ programs, providing particular advantages for programs that frequently employ composite data structures. This section delineates the enhancements in accuracy and performance from implementation perspective.

3.5.1 Special Handling

In addition to the novel approach SMLTA, DeepType enhances accuracy from implementation aspect by addressing diverse instructions and rare code patterns. We identified four typical corners cases and applied special handlings to each, aiming to mitigate inaccuracy resulting from the imprecise processing of these corner cases.

Composite instructions

A composite instruction encapsulates one or more embedded instructions as operands. When analyzing composite instructions, DeepType cannot acquire complete type information due to the obscured types within the embedded instructions. Listing 3.5 demonstrates a composite instruction. It is a `store` instruction that encapsulates a `bitcast` as its second operand, which itself is also a composite instruction embedding a `getelementptr` instruction as its first operand.

```
1 store <2 x i64> %12, <2 x i64>* bitcast (i32 ()** getelementptr inbounds (%struct.
    Sqlite3Config, %struct.Sqlite3Config* @sqlite3Config, i64 0, i32 13, i32 0) to
     <2 x i64>*), align 8, !dbg !56905, !tbaa !11590
```

Listing 3.5: A composite instruction. The `store` instruction in sqlite is composite, incorporating an embedded `bitcast` instruction, which in turn is composite and contains an embedded `getelementptr` instruction.

To gather complete type information, we employ ordered trees to represent hierarchical structure of composite instructions and their embedded instructions, and conduct a systematic analysis of instructions within ordered trees from the lowest to the highest levels, ensuring that the types hidden in embedded instructions can be utilized to generate

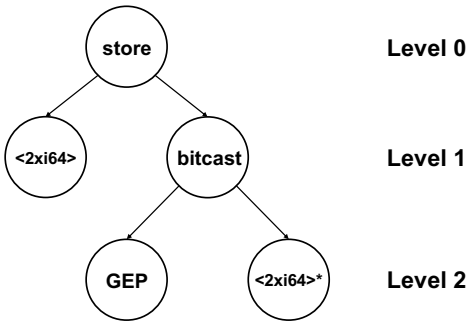

Figure 3.3: Ordered tree of the instructions in Listing 3.5. The root node represents the composite instruction `store`. Its child nodes at level 1 denote the instruction's operands. The second operand is another composite instruction `bitcast`. The child nodes at level 2 correspond to the operands of `bitcast`, among which the first one is a `getelementptr` instruction.

complete multi-layer types when analyzing their parent instructions. For example, we construct a three-level tree to represent the instructions in Listing 3.5. The `store` instruction occupies the root node in level 0, the `bitcast` instruction is situated in the second child node at level 1, and the `getelementptr` instruction resides in the first child node of the `bitcast` at level 2, as Figure 3.3 depicts. The `getelementptr` instruction returns a three-layer type `i32()* | struct.sqlite3_mutex_methods#0 | struct.Sqlite3Config#13`, which serves as the source type for the `bitcast`. So, we record it and the destination type `void (int)* | struct.A` as friend types each other in *Type-Type Map*. This relationship contributes to accuracy, but it could be missed without the special handling to composite instructions.

Anonymous structures

In LLVM bitcode, structs are sometimes anonymous when the specific names are not necessary or when they help to optimize the internal representation. This anonymity helps reduce IR size, but can lead to type mismatches, resulting in imprecise target identification.

We address this by assigning a unique identifier to each anonymous struct based on the sequence of member types. [5] This identifier assists in linking anonymous structs to named equivalents elsewhere in the bitcode, thus mitigating inaccuracy caused by mismatches of anonymous structs. If no named equivalent is found, they are named as `struct.anon`, which conservatively matches with any struct, preventing missing targets resulted from anonymous structs.

[5] As of October 2023, the latest version of TypeDive uses similar method to identify anonymous structs. However, it ignores those anonymous structs whose names never appear in the bitcode, which may lead to missing targets.

Dead functions

The iterative updating and patching of applications often results in the presence of dead functions, which are declared but not invoked, thus will not be executed during runtime. The analysis of instructions within these dead functions generates redundant type information, leading to false positive targets and unnecessary performance overhead. To enhance both accuracy and efficiency, our analysis omits dead functions.

Empty Type

In LLVM IR, the notation "{}" is employed to denote an anonymous struct type when the specific details of the data structure are unnecessary and subsequently omitted. We name it as *empty type*. Contrary to the anonymous structs discussed in Section 3.5.1, which merely lack names, the empty type is anonymous and contains no fields. However, what it represents can often be inferred from `bitcast` and other corresponding instructions in proximity. We categorize the code patterns involving empty types into two distinct classes, addressing each class with tailored solutions. Otherwise, the presence of empty type can impact the precision of our analysis as itself does not match with any struct.

When the empty type serves as the destination type in a `bitcast`, we observed that any subsequent utilization of the empty type in IR corresponds to the usage of the source type in source code. Accordingly, we record the source type as a friend type relative to the empty type, mitigating missing targets that may result from type mismatch. Conversely, when the empty type is the source type in a `bitcast`, we observed that its outer layer types in IR are the outer layer types of the destination type in source code. Thus, we record the empty type's outer layer types as those of the destination type's, ensuring that complete multi-layer types are gathered for type matching and target identification.

3.5.2 Caches

To diminish the performance overhead, we deploy two caches aiming at reducing runtime cost without affecting accuracy. The first one is used to store the verified types of a multi-layer type so that the exhaustive search algorithm and type verification process only run once for each multi-layer type. The second one is used to store identified targets of an indirect call so that another indirect call with the same multi-layer type can be quickly resolved by accessing the cache.

3.6 Evaluation

This sections presents the evaluation results of DeepType. Section 3.6.1 details experiment setup. Section 3.6.2 and Section 3.6.3 demonstrates the effectiveness and overhead, respectively, by comparing DeepType with TypeDive (commit acb8f4c) since MLTA is the-state-of-the-art approach in type-based analysis. Additionally, we evaluate the contribution of SMLTA in accuracy in Section 3.6.4 and present its security impact through a case study in Section 3.6.5.

3.6.1 Experiment Setup

Experiments are conducted on Ubuntu 20.04 with 8-core Intel Core i9-9880H CPU @ 2.30GHz and 16GB DDR4 RAM. Benchmarks include linux kernel, 5 web servers and 14 user applications. The GNU Binutils-2.35 is a collection of binary tools. We selected 13 among 15 programs as the discarded ones barely use indirect calls.[6] SQLite-3.45.1 is a database engine which contains numerous multi-layer types. 5 server programs are nginx, httpd, openVPN, proftpd and sshd. We use Linux-5.1 as the benchmark to show the scalability of DeepType.

These benchmarks are compiled by WLLVM [156] with LLVM-15. We use -g -O0 [7] flags to ensure that the generated bitcode contains debug information and type information, and that the instructions DeepType analyzes are not optimized out. Another flag -Xclang -no-opaque-pointers is used to disable opaque pointers so that pointers' types are sustained. To compare with TypeDive, we execute both tools on LLVM-15 and apply dead function elimination on them to make sure the benchmark bitcode analyzed by DeepType and TypeDive are exactly the same. In the experiments, we deploy 7-layer mappings in *Type Lookup Maps*, which are sufficient to archive the multi-layer types in our benchmarks, as presented in Table 3.3.

3.6.2 Effectiveness of DeepType

The effectiveness of DeepType is demonstrated by its ability to narrow down the scope of indirect call targets. We use Average Number of Targets (ANT) as metric to

[6]The program sysinfo does not contain any indirect call; The program elfedit only has 53 indirect calls without complex multi-layer types, showing exactly the same result for TypeDive and DeepType.

[7]This optimization level compiles the fastest and generates the most debuggable IR code, with which we can determine whether a function pointer is a local variable or formal parameter.

quantitatively measure effectiveness, which is defined as:

$$ANT = \frac{Num(T)}{Num(IC)},$$

where $Num(T)$ represents the total number of identified targets, $Num(IC)$ represents the total number of indirect calls that have targets. While the metric in TypeDive paper [78] is also average number, it only takes into account the indirect calls whose multi-layer types have at least two layers. Their metric neglects the fact that single-layer types can also benefit from MLTA, and that some indirect calls have no target identified because they are not initialized or because DeepType and TypeDive miss targets due to inevitable obstacles in implementation, as elaborated in Section 3.7.1. Therefore, we define ANT to precisely evaluate the effectiveness. To clarify *false positive (FP)* and *false negative (FN)* subsequently used in this paper, we define them in Definition 5.

Definition 5. Given an indirect call, a ***false positive (FP)*** is a function erroneously included in the target set contrary to the ground truth. A ***false negative (FN)*** is a function erroneously excluded from the target set contrary to the ground truth.

Table 3.5 presents the ANT for the benchmarks tested by DeepType and TypeDive, and the reduction rate in ANT achieved by DeepType compated to TypeDive. Given that the binutils programs share numerous library functions, leading to analogous ANT value, we only list their average ANT. The details are available in Table 3.6. The data in Table 3.5 indicates that DeepType reduces the ANT by 43.11% on average across most benchmarks, including binutils, httpd and linux. However, DeepType does not manage to decrease the ANT for nginx, openvpn and proftpd.

Program	DeepType	TypeDive	Reduction Rate
binutils	2.47	10.98	77.50%
sqlite	6.24	8.32	25.00%
nginx	6.38	5.60	-13.93%
httpd	6.23	12.27	49.23%
openvpn	2.35	1.62	-45.06%
proftpd	3.10	2.96	-4.73%
sshd	5.43	5.57	2.51%
linux	9.74	25.17	61.30%

Table 3.5: Average number of inidrect call targets. This table shows the average number of indirect call targets, and the reduction rate produced by DeepType over TypeDive. binutils shows the average of the 13 programs in binutils.

Program	DeepType	DT-weak	TypeDive
addr2line	2.38	2.62	8.60
ar	2.45	2.69	12.73
bfdtest1	2.38	2.62	12.89
bfdtest2	2.38	2.63	12.89
cxxfilt	2.37	2.62	8.64
nm-new	2.48	2.72	13.01
objcopy	2.63	2.87	13.08
objdump	2.95	3.19	11.26
ranlib	2.45	2.69	12.73
readelf	2.29	2.29	2.30
size	2.39	2.64	12.95
strings	2.37	2.62	8.64
strip-new	2.63	2.87	13.08

Table 3.6: Average number of indirect call targets for binutils. Detailed results for the programs in binutils collection.

The reduction in ANT can be attributed to SMLTA and special handlings in DeepType. SMLTA follows the strong restriction that checks the entire multi-layer type of an indirect call to identify targets that match with it. In contrast, TypeDive employs MLTA which separately resolves each layer of a multi-layer type and calculates intersection to determine the target set, potentially leading to FPs. The special handlings in DeepType are tailored to address corner cases where type information may be obscured, which enable DeepType to extract more accurate type information, thereby reducing FPs.

To validate the reasons for ANT reduction, we conducted a manual analysis on `objcopy`, as it contains substantial yet manageable number of indirect calls. By manually examining the indirect calls for which DeepType collects fewer targets than TypeDive, we confirmed the contributions of SMLTA and special handlings, and additionally unveiled another factor contributing to the reduction of ANT.

Field-sensitivity deployed in DeepType enables the differentiation of distinct members within in a composite data structure, even if they share the same type, which further refines indirect call targets. Despite the assertion of field-sensitivity in TypeDive paper, our manual analysis discovered FPs due to its field-insensitive. For instance, in a simplified scenario, a function is confined to `i64(i8*)* | struct.bfd_target#13`, whereas an indirect call has multi-layer type `i64(i8*)* | struct.bfd_target#23`, without any type propagation involved. TypeDive identifies this function as a target because its first and second layers respectively match `i64(i8*)*` and `struct.bfd_target`. By contrast, field-sensitive DeepType mitigates such FPs. Thus, field-sensitivity is another reason for ANT reduction.

Table 3.5 also shows that DeepType does not consistently reduce ANT, particularly in the cases of nginx and openvpn. There are two reasons for the increasing ANT. First, the adoption of fuzzy type leads to the conservative inclusion of all potentially matching types. It decreases FNs meanwhile inevitably brings in FPs, consequently raising the ANT value. Second, the special handlings enable DeepType to extract more precise type information, diminishing both FPs and FNs. In situations where the reduction in FNs surpasses that in FPs within a program, the ANT value increases. Given that the rising ANT is attributed to the reduction in FNs, it is deduced that DeepType is theoretically and practically effective in refining indirect call targets.

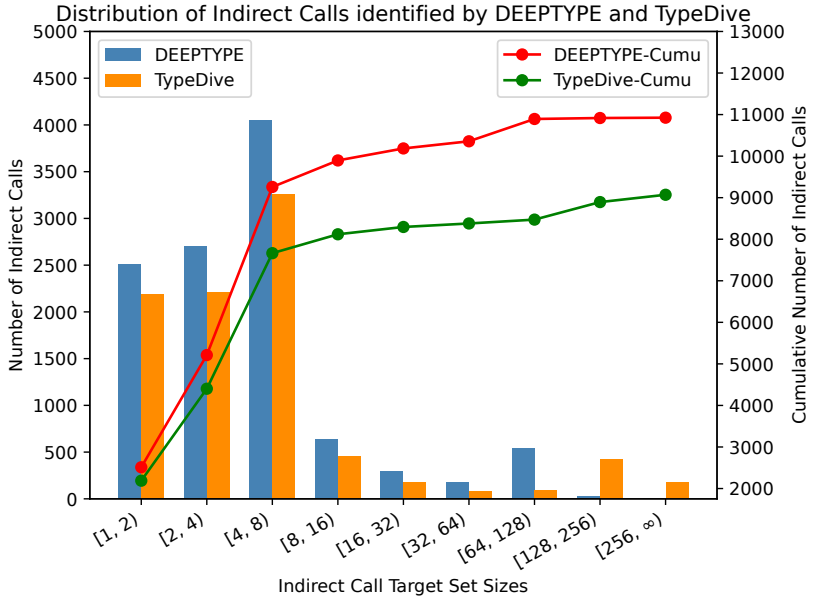

Figure 3.4: Distribution of indirect calls with different sizes of target sets in linux. The y-axis on the left shows number of indirect calls. The y-axis on the right shows cumulative number of indirect calls. The x-axis shows indirect call target sets' sizes ranging from 1 to infinite divided into 9 intervals. DeepType and TypeDive respectively represents the number of indirect calls reported by DeepType and TypeDive. DeepType-Cumu and TypeDive-Cumu respectively represents the cumulative number of indirect calls reported by two tools.

Figure 3.4 shows the distribution of indirect calls with different number of targets in linux kernel. We choose linux kernel because it is a complicated program that can demonstrate the distribution patterns as comprehensive as possible. In the experiment, although a number of indirect calls have no target according to either DeepType or TypeDive or both, we observe that DeepType is capable of finding more valid targets for more indirect calls, thus DeepType and DeepType-Cumu have more indirect calls than TypeDive and TypeDive-Cumu in most intervals. The main difference between DeepType and TypeDive falls in intervals [2,4) and [4,8), the number of targets

represented by these two intervals is much smaller than the average number of indirect call targets reported by TypeDive. As the number of indirect call targets increases, the difference between DeepType and TypeDive gradually decreases and becomes trivial except for interval [64,128), where DeepType has more indirect calls, and interval [128,256), where TypeDive has more indirect calls.

This trend indicates that the ratio of indirect calls with small target sets is lower in DeepType. We define `small` by different values of threshold in Table 3.7. A target set is considered as small when its size is less than the threshold. We observe that when the threshold is 2 or 4, DeepType has lower ratio of indirect calls with small target sets. However, this does not imply that TypeDive outperforms DeepType for these indirect calls. At lower thresholds (i.e., 2 or 4), a single FP or FN can significantly impact whether an indirect call is categorized as with a small or large target set. Conversely, at higher thresholds, the influence of false reports on categorizing diminishes. Therefore, the general trend is more indicative. In most cases, DeepType demonstrates a higher ratio of indirect calls with small target sets, underscoring its effectiveness in refining indirect call targets.

Threshold	DeepType	TypeDive
2	23.0%	24.1%
4	47.7%	48.5%
8	84.7%	84.5%
16	90.6%	89.5%
32	93.2%	91.5%
64	94.8%	92.4%
128	99.7%	93.4%
256	99.9%	98.1%

Table 3.7: Ratio of indirect calls with *small* target sets. The definition of *small* depends on the threshold. If the indirect call's target set size is smaller than the threshold value, this indirect call is considered as with *small* target sets. The last two columns respectively show the ratios of indirect calls with *small* target sets in DeepType and TypeDive.

Additionally, we compiled the benchmarks with various optimization levels to assess the effectiveness of DeepType across these levels, which is reported in Table 3.8. In general, DeepType has higher ANT when the benchmarks are compiled with more aggressive optimization settings for the majority of the programs examined. The increasing ANT can be attributed to the fact that higher optimization levels tend to optimize away certain instructions that DeepType relies upon for analysis, leading to incomplete type information being gathered. Consequently, DeepType produces an increased number of FPs and FNs. When there is a predominance of FPs over FNs, the ANT increases.

Program	O0	O1	O2	O3
binutils	2.47	3.20	3.20	3.13
sqlite	6.24	6.46	6.48	6.56
nginx	6.38	8.00	8.02	7.99
httpd	6.23	6.23	6.23	6.23
openvpn	2.35	2.80	2.45	2.45
proftpd	3.10	3.10	3.10	3.10
sshd	5.43	5.43	5.43	5.43
linux	9.74	9.74	9.74	9.74

Table 3.8: The effectiveness of DeepType on different optimization levels. This table shows the ANT reported by DeepType when analyzing benchmarks respectively compiled with optimization level O0, O1, O2 and O3.

The stability of ANT values across different optimization levels for programs such as http, proftpd, sshd, and linux can be explained by two factors. First, the FPs and FNs caused by the optimized-out instructions achieve a balance, neutralizing their impacts on the ANT value. Second, the minimal number of FPs and FNs does not significantly alter the ANT value.

Despite the reduced efficacy of DeepType on higher optimization levels compared to the O0 level, it nevertheless outperforms TypeDive in those benchmarks where DeepType with O0 optimization surpasses TypeDive. This observation demonstrates that while there is a marginal decrease in effectiveness with higher optimization levels, DeepType retains its comparative advantage over TypeDive.

3.6.3 Performance of DeepType

DeepType employs caches to obviate redundant analysis and improve performance. To evaluate the runtime overhead comprehensively, we disabled the caches in DeepType, resulting in a variant denoted as DT-nocache. We executed DeepType, DT-nocache, and TypeDive on each benchmark for three times to obtain average execution time, which yields more reliable statistics as it helps mitigate the impact of hardware conditions and operating system states through averaging.

Figure 3.5 presents the execution time for each benchmark. DeepType significantly outperforms TypeDive, showing a reduction in overhead ranging from 5.45% to 72.95%, with an average reduction of 37.02%. DT-nocache also demonstrates reduced overhead compared to TypeDive, despite TypeDive is equipped with caches.

To deduce the primary source of runtime overhead and reveal the reason for DeepType's efficiency, we separately measured the execution time of information collection

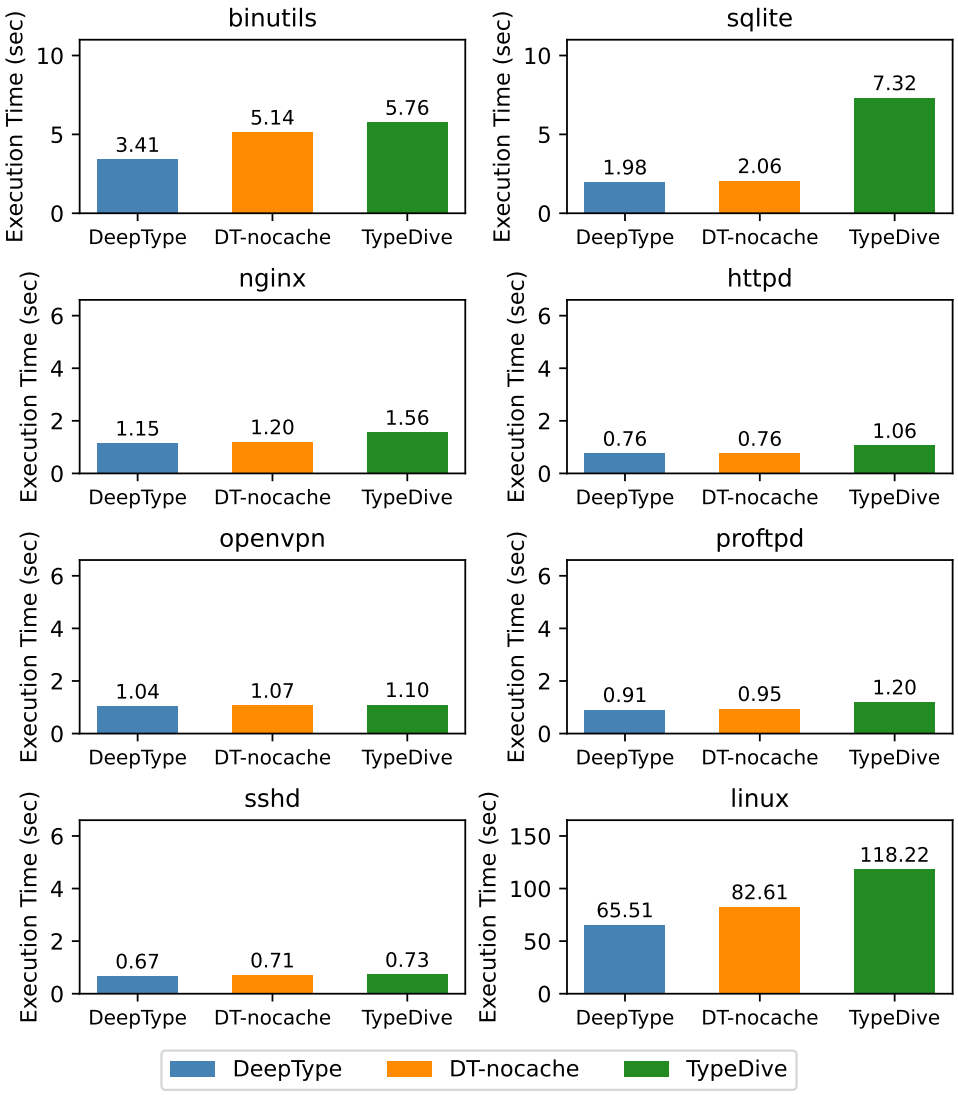

Figure 3.5: Execution time of DeepType, DT-nocache and TypeDive. DT-nocache represents DeepType without caches deployed. For each benchmark, we plot a bar chart to depict the execution times of DeepType, DT-nocache, and TypeDive, and the y-axis scale of which is adjusted to encompass the full data range without excessive magnification, allowing for clear differentiation in execution times. Specifically, the binutils chart illustrates the average execution times across all assessed binutils programs.

and target identification phases, considering the shared general workflow of all tools. The experiments were conducted on linux benchmark, which manifests noticeable differences among three tools. According to Figure 3.6, the percentages of runtime overhead incurred during target identification exhibit a progressive increase among DeepType, DT-nocache, and TyepDive, standing at 20.6%, 44.2%, and 65.0%, respectively. Thus, the lower overhead of DeepType should owe to the target identification phase, wherein DeepType straightforwardly deals with the entire multi-layer type of each indirect call.

In contrast, TypeDive addresses each two-layer type within the multi-layer type and calculates intersections, incurring additional runtime overhead. The fact that DeepType refines indirect call targets also indicates that TypeDive consumes extra computational resources on FPs.

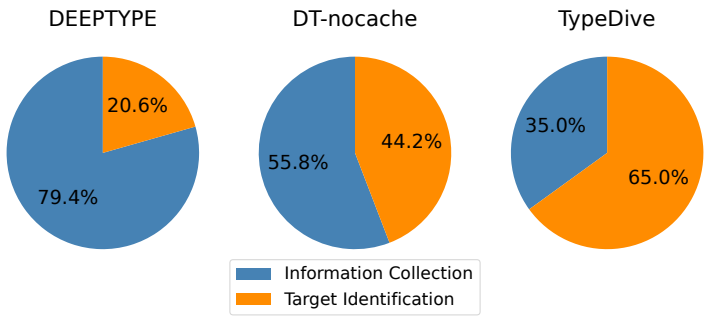

Figure 3.6: Runtime overhead distribution of DeepType, DT-nocache and TypeDive. DeepType and TypeDive follow the same general workflow that contains two phases: 1) collect information and record it in data structures, 2) analyze indirect call sites and recorded information to identify targets.

An additional observation reveals that the distinction in runtime overhead is evident in binutils, sqlite and linux, whereas it is negligible in the five server programs. The reason for this distinction is the higher prevalence of multi-layer types with more layers in binutils, sqlite and linux as illustrated in Figure 3.7. The efficiency advantage of DeepType over TypeDive is primarily attributed to DeepType's one-time resolution on the entire multi-layer type, in contrast to the laborious resolution in TypeDive for each two-layer type. It is noteworthy that a multi-layer type may consist of multiple two-layer types. Consequently, the benchmarks containing more multi-layer types with more layers amplify the runtime overhead discrepancy between DeepType and TypeDive.

Typically, memory overhead is not a main concern in static analysis, as tools that offer efficient performance without compromising precision are often favored. We still measure the memory overhead of DeepType using Massif tool in Valgrind [95] tool suite with `-pages-as-heap=yes` option enabled, to measure all the memory used, and compare it with TypeDive for a thorough performance assessment.

Figure 3.8 shows the memory overhead of DeepType and TypeDive. Regarding user applications and server programs in the benchmarks, both tools exhibit memory overhead below 150 MB. However, in the context of the Linux kernel, both DeepType and TypeDive demonstrate higher memory overheads ranging from 4.2 GB to 4.3 GB. This discrepancy is attributed to the larger size of kernel, which involves more multi-layer

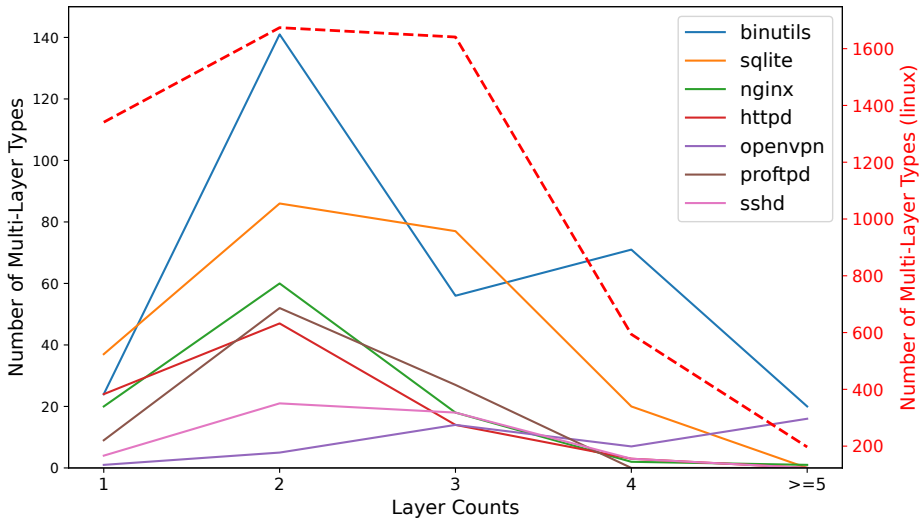

Figure 3.7: Number of multi-layer types with different layer counts. The y-axis on the right shows the number of multi-layer types in linux while the y-axis on the left shows the number of multi-layer types in other benchmakrs.

types. As a result, both tools record a greater volume of type information, leading to increased sizes of data structures and consequently consuming more memory spaces.

DeepType shows lower memory overhead than TypeDive, while the subtle difference between two tools remains consistent across all benchmarks. This consistency is attributed to the comparable memory space occupied by the data structures in two tools. Although DeepType records entire multi-layer types, which have larger sizes than two-layer types, it allocates fewer entries in maps for storage. The difference is due to the additional memory space utilized in TypeDive for recording escaping types, which is designed to overcome the third challenge, as elaborated in Section 3.1.2. In contrast, DeepType adopts the exhaustive search algorithm which does not consume so much memory space as escaping types.

3.6.4 Contribution of SMLTA

As described in Section 3.6.2, DeepType is capable to narrow down the scope of indirect call targets. This capability owes to SMLTA and special handlings to corner cases. To further investigate the impact of SMLTA on effectiveness, we disabled special handlings in a variant denoted as DT-noSH. Different ANT values of DeepType and DT-noSH exhibit the impact of special handlings, revealing the contribution of SMLTA.

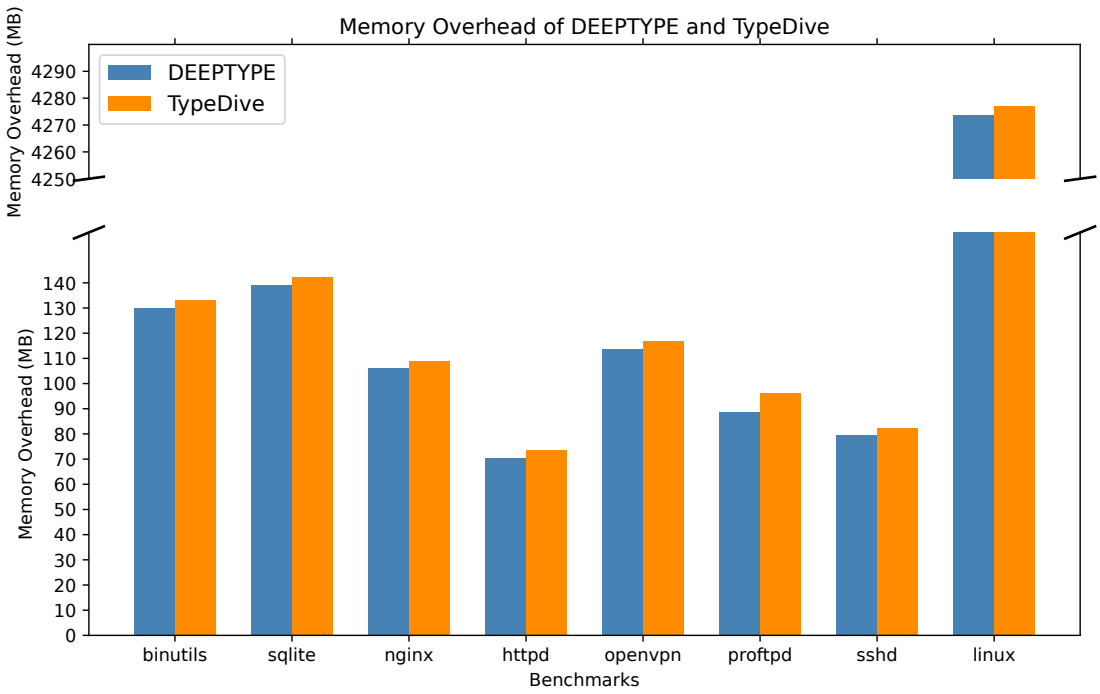

Figure 3.8: Memory overhead of DeepType and TypeDive. The scales from 150 to 4250 on y-axis are cut out because there is a gap between the memory overheads of linux and other benchmarks. DeepType always has lower memory overhead than TypeDive while the difference between two tools is consistently subtle.

Table 3.9 presents the ANT reported by DeepType and DT-noSH. The ANT of DT-noSH is close to that of DeepType, indicating that disabling the special handlings has minimal impact on the effectiveness of DeepType, which reveals the primary role of SMLTA in accuracy improvement. The rationale behind the statistics is that the special handlings are specifically implemented to address corner cases. Without them, DeepType generates slightly more FPs and FNs. Given the definition of ANT, minor fluctuations in the total number of indirect call targets do not significantly alter ANT value when the volume of indirect calls is relatively high. Nonetheless, these special handlings remain crucial, as they play a vital role in mitigating the FPs and FNs that are orthogonal to SMLTA.

Recall that there are four special handlings. First, the systematic analysis of composite instructions facilitates the generation of complete multi-layer types. Among all the benchmarks, nginx and linux exhibit the most significant improvement owing to this special handling, given their relatively higher prevalence of composite instructions. Second, linking anonymous structs with named equivalents enables DeepType to accurately recognize and match multi-layer types. We observe that openvpn and linux obtain the

Program	DeepType	DT-noSH	DT-weak
binutils	2.47	2.48	2.70
sqlite	6.24	6.33	6.97
nginx	6.38	8.62	12.99
httpd	6.23	6.23	7.66
openvpn	2.35	2.39	2.35
proftpd	3.10	3.13	4.22
sshd	5.43	5.42	5.43
linux	9.74	9.72	13.09

Table 3.9: ANT values of DeepType, DT-noSH and DT-weak. DT-noSH exhibits the contribution of SMLTA. DT-weak shows the impact of storing entire multi-layer types in *Type-Func Map*.

most benefit from this special handling compared to other benchmarks. Third, DeepType eliminates dead functions to discard redundant information, thereby mitigating FPs. All benchmarks, except for httpd and openvpn, benefit from this special handling. Last, DeepType handles empty types accordance to specific code patterns. Due to the infrequent occurrence of the empty type, its impact on ANT is negligible.

Although the comparison between DeepType and DT-noSH reveals the significant contribution of SMLTA to effectiveness, we also implemented a weak version of DeepType, denoted as DT-weak, which stores two-layer types in *Type-Func Map*, to help examine the impact of recording entire multi-layer types. As depicted in Table 3.9, DT-weak demonstrates a higher ANT than DeepType across most benchmarks, indicating that recording entire multi-layer types, rather than two-layer types, effectively refines indirect call targets. The difference between two tools is particularly evident on nginx and linux, which contain relatively more complex multi-layer types than other benchmarks. Note that, DT-weak still benefits from SMLTA because only the method of recording multi-layer types in *Type-Func Map* has been modified, while other SMLTA designs, such as multi-layer mappings, continue to play a role in filtering out FPs.

3.6.5 Case Study

While SMLTA is a fundamental tool applicable across various security-related fields (e.g., static bug detection, symbolic execution, fuzzing, and etc.), we demonstrate its security impact through the example of CFI enforcement.

CVE-2023-43641 [31] is an out-of-bounds access that enables arbitrary write in libcue. As detailed in Listing 3.6 at line 3, both the value of index `i` and `ind` can be controlled by attacker. By setting the index `i` negative, an attacker can achieve arbitrary write through preparing the value of `ind`. The exploit [11] utilized this vulnerability to corrupt

a function pointer in glib (cross-referenced by libcue) to achieve arbitrary code execution. To be specific, the exploit overwrites the heap object `info` (line 7), which is allocated in a function invoked by `track_set_index` (line 1). This overwrite enables the corruption of function pointer `pre_parse_func` (line 15) in glib, and the corrupted function pointer is subsequently used to call the `initable_init` function, which behaves similarly to the `system` function, enabling arbitrary code execution.

```
1  void track_set_index (...) {
2      ...
3      track->index[i] = ind;
4  }
5
6  static gboolean get_file_metadata (...) {
7      TrackerExtractInfo *info;
8      ...
9      tracker_extract_info_unref(info);
10     ...
11 }
12
13 gboolean g_option_context_parse (...) {
14     ...
15     if (!(* group->pre_parse_func) (context, group, group->user_data, error))
16     ...
17 }
```

Listing 3.6: CVE-2023-43642 vulnerable code. Line 3 is vulnerable to out-of-bounds access. An exploit for this vulnerablity can overwrite a function pointer in glib to gain code execution.

Similar to many typical CVE exploits, this exploit assumes Control Flow Integrity (CFI) is not deployed. The function pointer `pre_parse_func` has a type mismatch with the target function `initable_init`, making the exploit preventable by both MLTA and SMLTA. However, this does not imply that the vulnerability can be completely mitigated by either MLTA or SMLTA, as attackers may still conduct exploits by corrupting alternative function pointers.

We further examined glib and revealed 5 function pointers that can bypass MLTA, see Table 3.10. MLTA fails to prevent the exploits that corrupting these function pointers because it identifies the function `initable_init` as a valid target for these function pointers, though it is a FP in fact. This FP can be attributed to MLTA's approach of splitting multi-layer types, which confines the function `initable_init` respectively to `gboolean (Ginitable*, GCancellable*, GError**)*` and `struct._GInitableIface` with index 1, weakening the restriction of multi-layer type matching.

For example, `traverse_func` in Table 3.10 has type `gboolean (gpointer*, gpointer*, gpointer*)*`, matching with `gboolean (Ginitable*, GCancellable*,`

Function Pointers	MLTA	SMLTA
callback	✗	✓
callback*	✗	✓
traverse_func	✗	✓
func	✗	✓
predicate	✗	✓

Table 3.10: The capability of MLTA and SMLTA in preventing exploits. The listed function pointers, located in glib, can be corrupted through the vulnerability. MLTA fails to prevent the exploits through the 5 function pointers while SMLTA can prevent these exploits. To differentiate two function pointers named "callback" in separate functions, one is denoted as "callback*".

`GError**)*` due to information flow in glib. Thus, MLTA identifies `initable_init` as a potential target, enabling attackers to rewrite the function pointer with the address of `initable_init` and achieving arbitrary code execution.

In contrary, SMLTA is able to prevent these exploits because it strictly confines `initable_init` to its entire multi-layer type `gboolean (Ginitable*, GCancellable*, GError**)* | struct._GInitableIface#1`, which does not match with any function pointer in Table 3.10. Consequently, SMLTA effectively reduces the attack surface of control-flow hijacking attacks utilizing this vulnerablity. By limiting the attacker's ability to corrupt function pointers, SMLTA offers a higher level of security compared to MLTA in CFI implementation.

3.7 Discussion

This section discusses the soundness of SMLTA (Section 3.7.1), the limitations of this work (Section 3.7.2) and future extensions (Section 3.7.3).

3.7.1 Soundness

This section discusses the soundness of SMLTA. In theory, SMLTA belongs to type-based analysis, the soundness of which has already been proved [87, 97, 103, 153, 162]. In general, the static analysis in DeepType is flow-insensitive, indicating that it does not track the sequences of instructions or data-flow between basic blocks, but purely collects type information, which does not yield any FN.

Specifically, SMLTA contains 4 novel designs, none of which produces FNs. First, fuzzy type and fuzzy index stand for uncertain layers and indexes. They match with any type and index to ensure that no potential targets can be missed. Second, multi-layer

mappings are used to archive the collected multi-layer types without any omission. It does not produce any FN because all recorded multi-layer types can be retrieved through querying the mappings. Third, the relationships among multi-layer types involved in an information flow chain is conservatively recorded. Any potential information flow between two multi-layer types is considered when identifying targets. Finally, the exhaustive search algorithm discovers all friend types that can share associated functions with the multi-layer type of the indirect call, ensuring all potential targets being identified.

3.7.2 Limitations

The implementation of DeepType does not robustly address all corner cases in real-world programs. Besides the ones addressed in Section 3.5.1, some corner cases are out of our scope. For example, LLVM IR only contains type information but omits instantiated values of the members in global variables with composite type. If a function pointer as a member of a global variable is initialized at the global scope, the function assigned to it is apparent in source code but does not appear in the corresponding IR, thus does not appear in bitcode.[8] Given this fact, DeepType is unable to record the mappings between multi-layer types and functions for such initialization instructions, resulting in missing targets for involved indirect calls. Take binutils programs as another example, some indirect call targets are functions in the GNU linker, `LD`, while we use Clang to compile the programs which uses `LLD` as linker.

Albeit the accuracy improvement contributed by SMLTA and special handlings, DeepType still exhibits deficiencies in terms of accuracy. For small programs that contain few complicated multi-layer types, DeepType is limited in reducing FPs compared to TypeDive meanwhile possibly yielding extra FPs instead due to the conservative design and implementation choices for soundness purpose. In addition, if several function pointers have the same multi-layer type but points to different functions, SMLTA produces FPs in this scenario.

In evaluation aspect, there is no standard scale available to calculate the statistics such as FP rate, FN rate and accuracy, due to the absence of ground truth. Although "pseudo ground truth" [63] is a feasible solution, it highly depends on the precision of dynamic analysis adopted. If the dynamic analysis result is not proved to be extremely close to the ground truth, the numerical data of soundness and precision relative to the pseudo ground truth is not convincing subsequently. We use ANT as metric to reflect

[8]Bitcode is a binary encoding of LLVM IR.

the accuracy improvement. But it is convincing only in the context that both SMLTA and MLTA pertain to type-based analysis and the novel designs in DeepType does not produce FNs.

3.7.3 Future Work

SMLTA generates FPs when multiple functions share the same multi-layer type with an indirect call. This is a limitation intrinsic to type-based analysis that relies solely on type verification. To enhance accuracy, we intend to integrate data-flow analysis with SMLTA, assessing both value and multi-layer type to further constrain potential targets, thereby reducing the FPs inherent to SMLTA. To maintain a balance between accuracy and performance, this data-flow analysis will be designed to be lightweight, focusing on intra-procedural analysis.

Motivated by the absence of a standard and reliable metric for comparing various approaches, we also plan to develop a benchmark that includes a comprehensive ground truth. This ground truth will encompass, but is not limited to, indirect call targets, alias pointers, and value sets. This will enable a wide range of basic tools used in both static and dynamic analysis to evaluate their effectiveness, identify the fundamental reasons for any inaccuracies, and enhance their design and implementation accordingly.

3.8 Conclusion

In this paper, we have introduced strong multi-layer type analysis (SMLTA), a novel approach in refining indirect call targets, that thoroughly utilizes type information provided by multi-layer types. It treats the entire multi-layer type as a basic unit for information storage and type verification to improve accuracy. SMLTA resolves relationships between multi-layer types, exhaustively discovers friend types for indirect calls, and employs fuzzy type to overcome the challenges in indirect call target identification using multi-layer types. Additionally, multi-layer mappings are deployed to hierarchically archive multi-layer types for quick access. We implemented SMLTA in DeepType, which is equipped with special handlings to address diverse code patterns and corner cases. DeepType is scalable to large applications with superior effectiveness as well as performance. The experiment results showed that DeepType narrows down the scope of indirect call targets by 43.11% on average across most benchmarks, reduces runtime overhead by 37.02% on average and consumes less memory compared to TypeDive. A case study in

CVE exploit demonstrated that SMLTA is more powerful than MLTA in reducing attack surface and preventing exploits. However, the intrinsic limitation of type-based analysis can still produce false positive targets. We leave it as future work to further improve accuracy through lightweight data-flow analysis.

Chapter 4
LiteRSan: Rust Memory Safety Enforcement

Memory-safe programming languages have emerged as a promising approach [144] to mitigate prevalent memory safety vulnerabilities, which account for 70%–80% of all software vulnerabilities [85, 101, 136]. Among these languages, Rust [65] stands out by enforcing strong compile-time safety guarantees. Its advanced type system detects security issues early and helps confine additional costs to protect safety-critical operations. Studies show that, aside from these checks, Rust's performance can closely match that of C/C++ [179]. Consequently, Rust has rapidly gained adoption in security-critical and performance-sensitive domains [32, 86, 143].

Despite its robust safety guarantees, Rust's type system is not flawless. It can be too restrictive, preventing the expressiveness required for low-level systems programming, or it may introduce prohibitive runtime overhead in performance-critical code paths. Consequently, Rust permits *unsafe* code, such as raw pointer dereferences or calling external C library functions [9, 39, 108], enabling developers to bypass Rust's memory safety checks. Nevertheless, the use of unsafe Rust code reintroduces memory safety vulnerabilities, such as buffer overflows and Use-After-Free (UAF) bugs, undermining Rust's foundational memory-safety benefits [89, 90, 108, 167].

Various detection and mitigation mechanisms have been proposed to address memory safety challenges introduced by unsafe Rust. Static analysis tools, such as Rudra [13], MirChecker [72], and SafeDrop [29], have successfully identified many real-world vulnerabilities in Rust programs. However, these tools typically suffer from high false positives (e.g., Rudra reports approximately 89% false positives [13]), and have limited capability in detecting diverse bug types [27, 88]. Memory isolation techniques, such as XRust [74], TRust [14] and PKRUSafe [64], provide runtime protection by restricting unsafe code's

access to memory objects exclusively used by safe code. Nonetheless, these approaches target only subsets of memory objects and primarily focus on spatial memory errors (e.g., buffer overflows) while neglecting temporal memory errors such as UAF. Rust fuzzing frameworks [3, 22, 35, 171] have also emerged to detect memory safety vulnerabilities. However, it is well-known that the probabilistic nature of the fuzzing approach results in challenges of systematical detection of memory errors [15].

Researchers have also developed tools based on Address Sanitizer (ASan) [122], a compiler-based memory error detector, to reveal memory safety vulnerabilities in Rust. Compared to static analysis (limited bug detection capability), memory isolation (partial protection), and fuzzing (probabilistic by nature), ASan-based approaches provide deterministic dynamic validation of *every* memory access. Notably, ERASan [88] and RustSan [27] have advanced this area by optimizing away redundant checks for memory accesses already instrumented by the Rust compiler, thereby significantly reducing ASan's runtime overhead by 71.4% and 62.3%, respectively.

Despite these advances, existing ASan-based tools still do not fully align with Rust's native safety guarantees. Although ERASan and RustSan remove certain checks already enforced by Rust's type system, their reliance on traditional C/C++ pointer analyses (i.e., SVF [131]) leads to significant over-approximation of unsafe pointers, as such analyses are not integrated with Rust's ownership and borrowing semantics [67]. As a result, both tools introduce superfluous checks for memory accesses that are already guaranteed to be safe, imposing unnecessary runtime overhead. In addition, the static analysis time of ERASan and RustSan is prohibitively high, increasing compilation time by 1,635.35% and 1,193.31% per our measurements, due to their reliance on SVF, which is particularly expensive for large programs [64]. Furthermore, ASan suffers from inherent limitations in bug detection. Its red zones can be bypassed by overflows that exceed the boundaries, and its shadow memory mechanism may fail to detect UAF bugs when freed memory is reallocated post-quarantine, which makes dangling pointers to the original object undetectable. These gaps cause ASan-based tools to provide incomplete bug coverage despite significant overhead.

To address these limitations, our goal is to design a memory error detection mechanism tailored to Rust's inherent safety guarantees while addressing the loopholes introduced by unsafe code and the weaknesses of existing detection frameworks. Specifically, we strive to (1) identify pointers that truly pose spatial or temporal risks by incorporating a Rust-specific static analysis, eliminating extraneous checks on pointers that are either statically-proven safe or protected with compiler-inserted checks, (2) maintain complete

coverage and precision in detecting ***all*** classes of memory errors, including spatial and temporal errors without using heavyweight ASan-based approaches, and (3) minimize overhead by integrating Rust's ownership and borrowing rules into both static analysis and enforcing selective instrumentation for lightweight runtime checks.

Achieving these three objectives requires addressing three major challenges: (1) Rust's allowance of raw pointers within otherwise safe code complicates standard pointer analysis, as many may-alias inferences valid in the C/C++ context break under Rust's stricter ownership model, (2) bridging static checks and runtime validation demands a lightweight metadata design that captures Rust memory safety model, and (3) avoiding expensive and coarse-grained ASan-based runtime checks. To address these challenges, we developed LiteRSan (**Lite-R**ust-**San**itizer), which deploys a Rust-specific static analysis to pinpoint *truly risky* pointers and selectively instrument them with minimal metadata to detect both spatial and temporal memory errors at runtime. This synergy of compile-time insights and targeted runtime checks enables comprehensive and accurate memory error detection with minimal overhead across 28 widely used Rust benchmarks: only 18.84% runtime, 0.81% memory and 97.21% compile-time overhead. In contrast, ERASan incurs 152.05% runtime, 739.27% memory, and 1,635.35% compile-time overhead, while RustSan incurs 183.50%, 861.98%, and 1,193.31%.

In summary, we make the following contributions:

- **Rust-specific Taint Analysis:** We introduce a Rust-specific static analysis scheme that identifies risky pointers by integrating Rust's ownership and borrowing semantics rather than defaulting to generic pointer analysis.
- **Lightweight Metadata Inference and Runtime Checks:** We design a compact metadata mechanism for runtime validation of spatial and temporal safety, removing the heavyweight components (e.g., red zones and shadow memory) of classic sanitizers.
- **Comprehensive and Efficient Bug Detection:** Our approach, LiteRSan, systematically detects spatial and temporal memory errors in Rust. Compared to prior Rust sanitizers, LiteRSan offers complete coverage and higher accuracy in detecting bugs while minimizing compile-time, runtime, and memory overhead compared with existing ASan-based tools.

4.1 Threat Model

We assume that memory errors, including both spatial (e.g., out-of-bound read/write) and temporal (e.g., UAF and double-free) errors, are possible in Rust programs. Our goal is to detect all such memory errors. While directly-linked C/C++ libraries may also contain memory errors, we focus on Rust source code and neither analyze nor harden such external libraries. We also assume that no extra memory safety defenses are deployed beyond Rust's built-in safety support. Memory leaks are out of scope, as they are generally not classified as memory safety violations [92, 93, 122, 134]. Figure 4.1 shows the complete set of the bug patterns that LiteRSan covers, as ASan-based tools [27, 88] do in Rust programs.

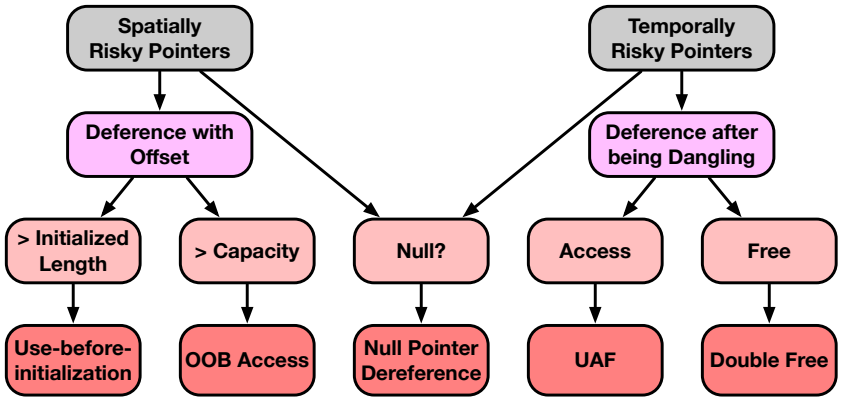

Figure 4.1: Memory safety bug patterns. Memory safety bugs within scope include spatial errors (i.e., use-before-initialization, out-of-bound accesses), null pointer dereferences, and temporal errors (i.e., use-after-free, double free).

Non-memory-safety errors, such as concurrency bugs and logic errors, are outside the protection scope of LiteRSan. In addition, LiteRSan is not designed to detect type conversion bugs. Notably, Rust's `std::mem::transmute()` [115] allows converting the type of an object to any other type. LiteRSan does not address errors caused by misusing this dangerous API. However, LiteRSan can detect type confusion bugs that arise from temporal errors, such as UAF. As mentioned above, LiteRSan does not target external C/C++ libraries. Therefore, cross-language attacks [83] that propagate exploitation from components written in unsafe languages (e.g., C/C++) are out of scope and can be addressed by existing works [111].

4.2 Motivation and Challenges

This section outlines the motivation and challenges behind this work. Chapter 4.2.1 presents an illustrative example showing how ASan-based tools often introduce redundant runtime checks, motivating the need for a more precise static analysis capable of accurately identifying instrumentation sites. Chapter 4.2.2 then discusses the key challenges to be addressed to overcome the limitations of existing memory safety sanitizers.

4.2.1 Motivating Example

While Rust enforces memory safety for most memory accesses (Section 2.2.1), severe errors (e.g., buffer overflows and UAF) can still occur when unsafe code is used. Listing 4.1 shows an example of a common scenario in web applications (e.g., Servo [143]).

```
1  struct Cache {
2      ptr: Option<*mut u8>,
3  }
4
5  impl Cache {
6      fn save(&mut self, ptr: *mut u8) {
7          self.ptr = Some(ptr);}
8
9      fn load(&self) -> Box<String> {
10         unsafe {
11             Box::from_raw(self.ptr.unwrap())}}
12 }
13
14 fn main() {
15     let mut cache = Cache { ptr: None };
16     let token = Box::from("session-token");
17     println!("Session␣token:␣{}", token);
18     {
19         let local_token = token;
20         cache.save(local_token.as_ptr() as *mut u8);
21         // local_token goes out of scope here.
22     }
23     let stale_token = cache.load(); // Dangling pointer
24     println!("Stale␣session␣token:␣{}", stale_token); // UAF
25 }
```

Listing 4.1: Use-after-free by caching a raw pointer after ownership transfer.

In Listing 4.1, the string `"session-token"` is a heap object allocated at line 16. A smart pointer, `token`, points to and owns this object. At line 19, ownership is transferred: a new smart pointer, `local_token`, takes the ownership of the heap object. Lines 20

and 7 define a raw pointer, `self.ptr`, derived from `local_token`. This raw pointer does not take the ownership of the string, so the owner remains `local_token`. `local_token` goes out of scope at the end of line 21, causing the object it owns to be deallocated. The raw pointer `self.ptr` then becomes dangling. At line 23, a new smart pointer, `stale_token`, is created from the dangling raw pointer, and it also becomes dangling. Then, all subsequent dereferences of the two dangling pointers are UAF.

ASan instruments all memory accesses, which can be redundant, incurring high performance and memory overhead without guaranteeing comprehensive memory safety.[1] For the example in Listing 4.1, no spatial memory safety check is necessary. For temporal memory safety, the dereference of `token` (line 17) does not require safety instrumentation, as Rust's ownership model ensures its validity.

To address this deficiency, prior work (i.e., ERASan [88] and RustSan [27]) improves ASan's performance by selectively instrumenting only raw pointers, or pointers in unsafe code, and their aliases. However, for the example discussed here, conventional alias analysis would identify all pointers in Listing 4.1 as aliases to the raw pointer `self.ptr` in unsafe code, resulting in redundant checks inserted to the dereference site of a safe pointer (line 17). This redundancy stems from insufficient consideration of Rust's memory safety guarantees, leading to over-approximating and instrumenting safe pointer dereferences.

Ideally, a Rust memory safety sanitizer should: (1) leverage Rust's memory safety model to precisely differentiate safe pointers (e.g., `token`) from unsafe ones (e.g., `self.ptr`); (2) selectively instrument only unsafe pointers, avoiding redundant checks on Rust-guaranteed safe pointers; and (3) provide comprehensive, accurate, and efficient detection for all memory error classes. Such an approach narrows checks to only unsafe operations, incurring minimal overhead while ensuring comprehensive detection coverage.

4.2.2 Challenges

As discussed in Section 2.2.1, existing memory error detection approaches are both incomplete and inefficient. Static analyzers [13, 72] often produce a high number of false positives, while ASan-based techniques [27, 88] incur significant performance overhead and still fail to detect many bugs. We observed that the shortcomings of ASan-based tools largely stem from analyzing Rust in LLVM IR [68]—a language-independent, low-level compiler intermediate representation, using generic pointer analysis [131] without

[1] As evaluated in MSET [152], ASan failed to detect around 50% of C/C++ memory errors in their constructed benchmark. We believe the rationale would be similar for Rust, as ASan is not aware of Rust's memory safety model.

accounting for Rust's unique memory safety guarantees. This insight guides us in develop LiteRSan. To propose our approach, we first introduce the key concept of *risky pointer*, which will be used throughout the rest of this paper.

Definition 6. A ***risky pointer*** is a pointer whose dereferences may violate memory safety. Such a pointer is *spatially risky* if it bypasses Rust's spatial enforcements, or *temporally risky* if it may outlive its referenced object.

Detailed explanations of spatially and temporally risky pointers are presented in Section 4.4.2. Note that (1) a pointer may be both spatially and temporally risky; (2) A raw pointer becomes risky *only when it is exposed in unsafe code* (Definition 7); and (3) Rust's native smart pointers may also be risky. For example, constructing multiple smart pointers from a raw pointer may violate Rust's ownership rules, rendering these smart pointers risky and potentially causing UAF bugs that elude compiler checks.

Definition 7. An ***exposed raw pointer*** is a raw pointer *directly used in unsafe code*, bypassing Rust's safety guarantees.

We identify three key challenges in building an efficient and comprehensive memory safety sanitizer tailored to Rust.

- **C1: Leveraging Rust's unique type system to precisely identify risky pointers.** Program analysis for Rust in prior work [14, 27, 74, 88] does not utilize Rust's ownership and borrowing semantics, significantly over-approximating risky pointers. A refined approach should integrate Rust's intrinsic memory safety model to more precisely identify risky pointers.
- **C2: Managing lightweight safety metadata for runtime checks.** Relying on a coarse-grained protection scheme like ASan's shadow memory and red zones [122] is expensive and imprecise. Tailoring compact yet fine-grained metadata that incorporates Rust's memory safety guarantees enables more efficient and accurate runtime error detection. Additionally, because raw pointers lack spatial metadata (i.e., bounds information), LiteRSan must infer and maintain their metadata to enable runtime validation.
- **C3: Minimizing overhead while ensuring coverage.** As Rust's memory safety model already protects a substantial amount of memory accesses, additional checks are only needed for those involving risky pointers. The challenge is to minimize cost

while maintaining accuracy and comprehensiveness by (1) selectively instrumenting only the truly risky pointers based on their specific risk types and (2) enforcing an efficient runtime check mechanism rather than incomplete and inefficient ASan-style checks.

By addressing these challenges, LiteRSan complements Rust's inherent memory safety guarantees with precise instrumentation to achieve comprehensive and low-overhead runtime safety checks, closing the gap left by prior work [27, 88].

4.3 LiteRSan Overview and Workflow

To address the three key challenges described in Section 4.2.2, we propose a Rust-specific static analysis to identify risky pointers. Coupling it with a metadata-based runtime checking mechanism, we develop our prototype system, LiteRSan. Figure 4.2 illustrates the main components and the overall workflow of LiteRSan.

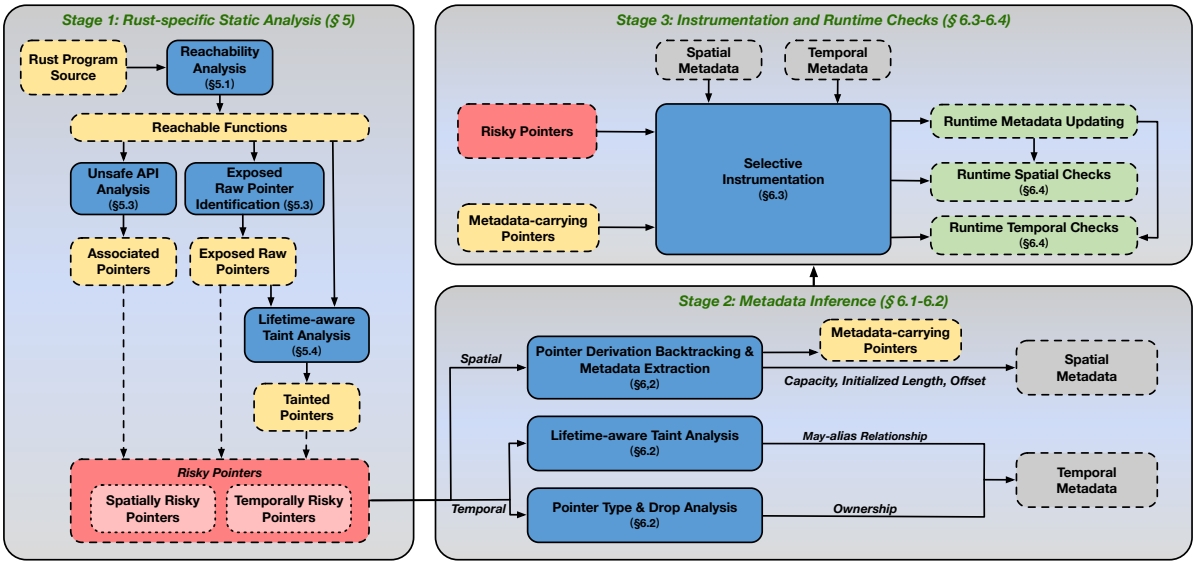

Figure 4.2: LiteRSan overview. LiteRSan consists of three stages. Each addresses one of the primary challenges in enabling efficient and comprehensive sanitizer checks. The output of each stage serves as the input to the next.

Stage 1 conducts Rust-specific static analysis to addresses **C1**. LiteRSan first pre-processes the target Rust program using reachability analysis to narrow the analysis scope to potentially reachable functions (Section 4.4.1). Within this scope, LiteRSan identifies both *Spatially Risky Pointers* (Section 4.4.3) and *Temporally Risky Pointers* (Section 4.4.4). Since the misuse of raw pointers is the primary cause of memory safety

violations in Rust, LiteRSan begins by identifying them. It annotates the instructions involving raw pointers during Mid-level IR (MIR) [114] to LLVM IR code generation, and analyzes the definitions and uses of these pointers in annotated instructions to identify raw pointers. Raw pointers are classified as both spatially and temporally risky because they are exempt from Rust's compile-time safety enforcement. To identify additional temporally risky pointers, LiteRSan performs *lifetime-aware* taint analysis starting from instructions that use raw pointers. This is necessary because raw pointers can propagate temporal risks to other pointers referencing the same memory object. In contrast, spatial risks do not propagate if each raw pointer arithmetic and dereference is instrumented with bounds checking. Additionally, LiteRSan identifies risky pointers used in unsafe APIs that may cause memory safety violations.

Stage 2 constructs lightweight spatial and temporal metadata to address **C2** (Section 4.5.1 Section 4.5.2), enabling efficient runtime validation. For spatially risky pointers, LiteRSan maintains three pieces of metadata: capacity, initialized length and offset. When spatial metadata is unavailable at a pointer's definition site (e.g., a raw pointer derived from another raw pointer), LiteRSan backtracks pointer derivations to extract metadata from the object's allocation site. This process also identifies *metadata-carrying pointers*, which are responsible for transmitting spatial metadata at runtime.

For temporally risky pointers, the risk arises from shared access to the same memory object. Once the object is deallocated by its owner or via an unsafe API, all referencing pointers that remain in scope become dangling. To address this, LiteRSan maintains may-alias relationships and ownership information as temporal metadata. May-aliases are established during Stage 1 via taint analysis, as pointers tainted by the same raw pointer must reference the same object. Among these, LiteRSan analyzes pointer types and `Drop` implementations [36] to identify owners.

Stage 3 addresses **C3** by selectively instrumenting identified risky pointers and metadata-carrying pointers (Section 4.5.3), thereby minimizing runtime overhead while preserving the comprehensive coverage of runtime checks. Leveraging spatial and temporal metadata collected in Stage 2 and updating it during execution, LiteRSan performs accurate and comprehensive detection of both spatial and temporal memory errors (Section 4.5.4). The complete set of memory safety bugs detectable LiteRSan is summarized in Figure 4.1.

4.4 Rust-Specific Static Analysis

In this section, we present LiteRSan's Rust-specific static analysis, which addresses the challenge of identifying risky pointers (C1) discussed in Section 4.2.2. We begin by defining the scope of the analysis in Section 4.4.1 and introducing the definition of risky pointers in Section 4.4.2. We then describe our approach to identifying spatially and temporally risky pointers in Section 4.4.3 and Section 4.4.4, respectively, and conclude this section by discussing soundness and precision in Section 4.4.5.

4.4.1 Static Analysis Scope Restriction

LiteRSan restricts its static analysis to *reachable functions*, motivated by the structure of Rust programs, which often include deeply nested library code, much of which is dead code (i.e., unreachable from the program entry point[2]). To exclude such dead code from LiteRSan's analysis, LiteRSan performs *reachability analysis* to conservatively identify and analyze only potentially reachable functions at runtime.

Specifically, starting from the program's entry point, LiteRSan identifies and enqueues both directly called functions and address-taken functions (i.e., potential indirect call targets [78, 165]) for analysis. For each function in the queue, LiteRSan recursively discovers and further enqueues the function's callees and address-taken functions, thereby restricting its analysis scope to functions potentially reachable during execution. By limiting analysis to reachable functions, LiteRSan focuses on identifying risky pointers within this scope that may lead to memory safety violations.

4.4.2 Risky Pointer Definition

Within LiteRSan's restricted analysis scope, most pointers are safe thanks to Rust's native safety guarantees for safe code, as discussed in Section 2.2.1. However, a subset of pointers remains unprotected and may still violate memory safety. We refer to these as *risky pointers* (Definition 6), and LiteRSan focuses its safety checks exclusively on them. For fine-grained analysis and instrumentation, we further classify risky pointers into *spatially risky* and *temporally risky* categories, corresponding to potential violations of spatial and temporal memory safety, respectively.

[2]The entry point is typically the `main` function. For library crates compiled with built-in benchmarks, each function in the benchmark is treated as an entry point.

Spatially risky pointers include (1) raw pointers and (2) smart pointers used in certain unsafe APIs. Raw pointers are spatially unsafe because arbitrary pointer arithmetic is permitted on them, which may result in invalid pointers whose dereferences are not checked. Moreover, Rust's standard libraries (e.g. `std`) provide unsafe APIs that may subvert bounds checking if misused [10, 61]. When a pointer is used in conjunction with such unsafe APIs, it is considered spatially risky.

Temporally risky pointers include (1) raw pointers and (2) any *valid* (i.e., in-scope according to Rust's scoping rules [140]) pointers that reference the same memory object as a raw pointer. Raw pointers are temporally unsafe because they are exempt from Rust's ownership rules; once the referenced object is deallocated, the raw pointer becomes dangling. Furthermore, as illustrated in Section 4.2.1, if a smart pointer is constructed from a raw pointer and takes ownership of an object that already has an owner, multiple owners will coexist. Deallocating the object through one owner leaves the others dangling. Invalid pointers whose lifetimes have ended (e.g., `token` in Listing 4.1) are excluded from temporally risky pointers, since any use of them is prevented by Rust compiler.

4.4.3 Spatially Risky Pointer Identification

Raw Pointers

The misuse of raw pointers is a primary cause of memory safety bugs in Rust programs [88]. To identify them, LiteRSan tracks raw pointers via LLVM IR metadata annotation during the MIR-to-LLVM IR lowering phase, as outlined in Section 4.3.

Based on ERASan [88]'s approach, LiteRSan attaches custom LLVM metadata [138] to IR instructions by modifying the `codegen-ssa` and `codegen-llvm` components of the `rustc` compiler. To determine the locations of annotations, LiteRSan performs a type-matching analysis during the MIR-to-LLVM IR lowering phase. Specifically, it analyzes the types of program variables and expressions in the MIR (i.e., Rust's mid-level representation) to identify those involving raw pointers (e.g., `*const T`). If a value is of raw pointer type, the corresponding LLVM instruction is tagged with `!rawptr`. Additionally, instructions originating from unsafe code blocks are marked with `!unsafe`. This analysis allows LiteRSan to propagate type information from Rust's MIR and identify the instructions relevant to raw pointers in the resulting LLVM IR.

After annotating the LLVM IR, LiteRSan analyzes instructions tagged with `!rawptr` to determine whether they define or use raw pointers. For each definition, it further checks whether the uses occur within unsafe code by examining the presence of the

`!unsafe` metadata. If the instruction uses a raw pointer, LiteRSan checks whether it appears in unsafe code before tracing its operand back to the corresponding definition site. This step is important because raw pointers can be encapsulated within safe abstractions (e.g., smart pointer creation), where they are not dereferenced and therefore do not pose risks. Through this analysis, LiteRSan accurately identifies only standalone raw pointers that are directly used in unsafe code and may cause memory safety bugs. These raw pointers are considered as *risky pointers*, both spatially risky and temporally risky (discussed in Section 4.4.4).

Unsafe APIs

As Rust's standard libraries (e.g., `std`) provide unsafe APIs that may cause spatial safety violations [10, 61], LiteRSan analyzes these APIs to extend its protection. In general, an API may be unsafe because it directly uses raw pointers or subverts Rust's bounds checks when misused. LiteRSan handles type (1) APIs by identifying underlying raw pointers using the method discussed above and marking them as spatially risky.

Type (2) APIs are more challenging to address. A notable example is `vec::set_len()` [116], which can alter a vector's length to an arbitrary value, potentially resulting in out-of-bounds accesses that bypass the compiler's spatial safety checks. Automatically and comprehensively identifying such APIs would requires analyzing and *understanding* all library code, which is an undecidable problem [110]. Therefore, we manually examined Rust's standard libraries and identified nine APIs that may circumvent bounds checks[3]. LiteRSan marks the pointers involved in these APIs as spatially risky, updates their metadata, and inserts runtime checks accordingly. For example, to detect out-of-bounds accesses potentially caused by `vec::set_len()`, LiteRSan retrieves the capacity of the vector at its definition site and inserts a spatial check at the API's call site to verify whether the new length (i.e., the argument to `vec::set_len()`) exceeds the legal capacity.

4.4.4 Temporally Risky Pointer Identification

To detect temporal memory safety violations, LiteRSan must go beyond merely identifying raw pointers (as discussed in Section 4.4.3). It must also detect any valid pointer that might reference the same memory object as a raw pointer.

[3]Unsafe APIs that may violate memory safety, within our scope, without involving raw pointers include `unchecked_add/sub/mul/neg/shl/shr`, `forward/backward_unchecked` and `set_len`. LiteRSan handles them by inserting bounds or validity checking at their call sites.

This may sound similar to finding all may-alias pointers to this memory object; however, the key difference is that any may-alias smart pointers that have been *moved* or gone out of scope should be excluded (e.g., `token` after line 19 in Listing 4.1).

One approach to identifying temporally risky pointers is to perform alias analysis on each raw pointer, find the complete set of pointers that refer to the same memory object, and then filter out aliased smart pointers that are invalid. However, existing alias analysis frameworks for LLVM (e.g. SVF [131]) do not account for Rust's ownership and borrowing semantics when computing aliases. As a result, they tend to produce largely over-approximated alias sets, since many smart pointers deemed aliases may no longer be valid when a raw pointer references the same memory object. Furthermore, whole-program alias analysis is generally highly expensive for large programs [64]. Therefore, we develop a new *Rust-specific, inter-procedural, flow-sensitive, lifetime-aware taint analysis* to **directly** identify temporally risky pointers without relying on traditional alias analysis.

Illustrative Example

We reuse the example in Listing 4.1 to briefly illustrate the workflow of LiteRSan's taint analysis. In this example, a raw pointer, `self.ptr` is defined at line 20 through an existing smart pointer (`local_token`), which owns a heap object. Here, the raw pointer `self.ptr` serves as a *taint source*. LiteRSan's taint analysis performs two key operations to identify temporally risky pointers to the object pointed by `self.ptr`:

- **Backward propagation** traces the ownership transfer preceding the definition of the raw pointer. This includes identifying the smart pointer (`local_token`) whose ownership is transferred from `token`. Since `token` goes out of scope before `self.ptr` is defined, the analysis terminates backward propagation at its definition site (line 19) without tainting `token`.
- **Forward propagation** tracks how taint from the raw pointer is propagated to pointers derived from it. In this example, a new pointer `stale_token` is derived from `self.ptr` (line 23), making `stale_token` temporally risky.

In short, to capture all temporal safety violations, the taint analysis must consider both the preceding ownership history of any object referenced by a raw pointer (i.e., backward) and all subsequent pointers derived from that raw pointer (i.e., forward).

Exposed Raw Pointer Classification

To distinguish the exposed raw pointers (taint sources) that require different taint analysis propagation directions, we classify them as follow.

- **Type 1 (T1) raw pointers:** exposed raw pointers created by referencing an object already owned by an existing smart pointer (e.g., via `Vec::as_ptr()` [117]). As shown in Listing 4.1, `self.ptr` is derived from a valid smart pointer `local_token`. The creation of such raw pointers implies existing ownership. Consequently, T1 raw pointers require both backward taint analysis (to trace the ownership history of the referenced object) and forward taint analysis (to track subsequent pointer derivations).
- **Type 2 (T2) raw pointers:** exposed raw pointers created to reference a newly allocated memory object. Since there is no preexisting ownership chain to consider, T2 raw pointers require only forward taint analysis.

After locating these definitions, LiteRSan performs an inter-procedural taint analysis starting from the definition site of each exposed raw pointer (i.e., taint source) and propagating on only forward or both directions according to the class of raw pointers.

Lifetime-aware Taint Analysis

LiteRSan performs a combination of backward and forward taint analysis, both of which track pointer derivation instructions (Definition 8), augmented with lifetime-aware propagation that respects Rust's ownership rules, to identify temporally risky pointers.

Definition 8. A ***pointer derivation instruction*** is any operation that produces a new pointer value from an existing one, by one of the following:

- Assignment (direct copy of a pointer),
- Computation (arithmetic or type conversion),
- Memory propagation (store/load through objects), or
- Inter-procedural transfer (via function calls or returns).

- **Backward taint analysis:** Starting from each taint source (i.e., the definition site of a T1 pointer), the analysis traces backward along the derivation chain to the pointers from which T1 is derived. These pointers share the same referenced object with T1, and their definition sites are marked as taint sinks. This propagation halts if ownership is transferred, as any further use of the original owners is disallowed by the

Rust compiler, ensuring that the original owners are free from temporal memory safety violations. In particular, if a pointer is invalidated before the exposed raw pointer is defined—through function return for stack pointers or through explicit `drop` for heap pointers—it is considered safe and excluded from tainted pointers, as it can no longer contribute to temporal safety violations.

- **Forward taint analysis:** Starting from each taint source (i.e., the definition site of a T1 or T2 pointer), the analysis propagates taint to all pointers derived from it. Any pointer derived from a tainted pointer is likewise marked as tainted. This forward propagation continues until no further pointer derivations exist.

Inter-procedural Analysis

Taint propagates inter-procedurally through function calls and returns. For direct calls, forward propagation flows from actual arguments at call site to corresponding formal parameters of the callee, and from callee's return value to the variable receiving it in the caller. Backward propagation flows in the reverse direction and terminates at invalidated pointers, such as those that are out of scope or have transferred ownership.

For indirect calls, LiteRSan conservatively resolves potential call targets using a type-based analysis [99, 146]. This approach matches function signatures (i.e., function prototypes) at indirect call sites with those of address-taken functions. Although more advanced multi-layer type analysis techniques [78, 165] can improve precision, they introduce additional static analysis overhead, while the precision gain in Rust is limited. This is because Rust programs typically rely less on dynamic dispatch [44, 81, 91] than programs in other languages, and the multi-layered structural patterns common in C/C++ are less prevalent in Rust. Once potential callees are identified, taint is propagated in the same manner as for direct calls.

LiteRSan adopts a worklist-based algorithm [98] adapted for inter-procedural taint analysis, as presented in Algorithm 2. First, it caches pointer derivations whose sources originate from other functions, either directly (e.g., formal parameters) or indirectly (e.g., intermediate variables derived from formal parameters), and are therefore *unresolved* within the current function context (lines 4–10). After completing the initial pass over all functions, it performs a depth-first search over the cached derivations to exhaustively propagate taint (lines 11–25). This two-step process ensures that all transitive taint relationships are resolved and that all potential temporally risky pointers are identified. In addition, LiteRSan also addresses temporally risky pointers involved with unsafe APIs, similar to the approach described in Section 4.4.3.

Algorithm 2: Inter Procedural Taint Propagation

Input: F — set of functions; R — set of initially tainted raw pointers
Output: *TaintedSets* — mapping from each taint source to its tainted pointers

```
Chapter etKwBlockBeginfunctionend function begin InterProcTaintPropagation(F, R)
    Initialize TaintedSets to map each r ∈ R to {r}
    Initialize WorkList ← ∅
    for each unresolved pointer derivation D in F do
        (src, dst) ← ExtractSourceAndDestination(D)
        Add D to WorkList
        if src is tainted then
            Add dst to the same tainted set as src
        else if dst is tainted and no ownership transfer in D then
            Add src to the same tainted set as dst
    for each tainted pointer t in TaintedSets do
        Initialize Visited ← ∅
        Initialize Stack ← {t}
        while Stack is not empty do
            p ← pop an element from Stack
            if p ∉ Visited then
                Add p to Visited
                for each unresolved pointer derivation D in WorkList do
                    (src, dst) ← ExtractSourceAndDestination(D)
                    if src = p then
                        Add dst to the same tainted set as t
                        Push dst onto Stack
                    else if dst = p and no ownership transfer in D then
                        Add src to the same tainted set as t
                        Push src onto Stack
    return TaintedSets
```

4.4.5 Soundness and Precision

Source Language Definition

We begin by defining the *source language* on which our Rust-specific static analysis operates. The source language is Rust, restricted to the subset of constructs relevant to potential memory safety violations. That is, `unsafe` operations and the portions of safe code that interact with them through pointer derivations. Rust's ownership model ensures that each value has a unique owner, and its borrow checker enforces strict rules on references (`&T`, `&mut T`). In particular, the compiler guarantees that references never outlive their owners (temporal safety) and that container accesses remain within bounds (spatial safety), inserting runtime checks where necessary. Together, these mechanisms enforce memory safety in Rust code.

However, Rust also permits the use of `unsafe` code blocks, where raw pointers (`*const T`, `*mut T`) can be directly accessed and arbitrarily used. These exposed raw pointers, along with unsafe APIs, are exempt from the lifetime and bounds checks enforced in safe Rust. As a result, they constitute the root cause of spatial and temporal memory safety violations. To formally reason about soundness, we therefore formalize a Rust-inspired source language which abstracts away most of Rust's safe constructs and captures exactly the portion of Rust where the compiler's guarantees no longer apply and where static analysis must conservatively track pointer derivations to enforce memory safety.

Types. Types distinguish among owners, safe references and raw pointers:

$$\tau \ ::= \ T \ \mid \ \mathit{Owner}\langle T\rangle \ \mid \ \&T \ \mid \ \&\mathit{mut}\ T \ \mid \ *\mathit{const}\ T \ \mid \ *\mathit{mut}\ T$$

where T ranges over base types. $\mathit{Owner}\langle T\rangle$ abstracts over Rust's smart pointers (e.g., `Box<T>`, `Rc<T>`, `Arc<T>`) that manage memory allocation and deallocation. References $\&T$ and $\&\mathit{mut}\ T$ are governed by Rust's borrow checker and guaranteed to be spatially and temporally safe. Raw pointers $*\mathit{const}\ T$ and $*\mathit{mut}\ T$ are unrestricted and thus the root cause of potential memory safety violations.

Objects and Pointers. Both spatial and temporal memory safety focuses on the status of memory objects and pointers. Let Loc be a countable set of abstract locations. Identifiers $id \in \mathsf{Loc}$ denote allocated memory regions.

$$\begin{aligned} o &::= \mathtt{obj}(id, \tau) && \text{(memory object of type } \tau \text{ at abstract location } id\text{)} \\ p &::= \mathtt{ptr}(o, \tau) && \text{(pointer of type } \tau \text{ referencing object } o\text{)} \end{aligned}$$

Here, o denotes an allocated memory region (either on stack or heap). A pointer p references an object and carries its type, which may be an owner, a safe reference, or a raw pointer. Owners and references comply with Rust's safety guarantees while raw pointers are free from memory safety rules.

Operations. Operations that are involved in our Rust-specific static analysis.

$$
\begin{array}{rcll}
op & ::= & \text{ptr\_from\_obj}(p, o) & \text{(define pointer } p \text{ from object } o\text{)} \\
& | & \text{ptr\_from\_ptr}(p, p_s, \kappa) & \text{(define pointer } p \text{ from source pointer } p_s \text{ via } \kappa\text{)} \\
& | & *p & \text{(dereference)} \\
& | & p \oplus k & \text{(pointer arithmetic by offset } k\text{)} \\
& | & \text{mod}(o, \mu) & \text{(container modifier that modifies } o\text{)} \\
& | & \text{dealloc}(o, p) & \text{(deallocation of } o \text{ via p)}
\end{array}
$$

Here κ represents a pointer derivation instruction as defined in Definition 8, and μ represents a container modifier (e.g., `push()` or `pop()`) that modifies the memory object's capacity or initialized length. The operations capture exactly what are potentially involved in a spatial or temporal memory safety violation in unsafe Rust: pointer creation from objects ptr_from_obj, pointer derivation from existing pointers ptr_from_ptr, pointer dereference $*p$, pointer arithmetic $p \oplus k$, container modifiers $\text{mod}(o, \mu)$ that change the spatial status (see Section 4.5.1) of underlying memory objects, and deallocation $\text{dealloc}(o)$ that changes temporal status (i.e., also see Section 4.5.1). Pure computations on non pointer values, control flow, and safe reference primitives are omitted because the Rust compiler already enforces their safety or they do not alter aliasing or lifetime relations that expose risks.

Soundness Argument

LiteRSan's Rust-specific static analysis is sound for identifying both spatially and temporally risky pointers in the source language defined above.

Scope (Section 4.4.1). First, the conservative reachability analysis includes every function that may execute at runtime, ensuring that any operation in the set

$$
\{\ \text{ptr\_from\_obj}(\cdot,\cdot),\ \text{ptr\_from\_ptr}(\cdot,\cdot,\kappa),\ *(\cdot),\ (\cdot)\oplus(\cdot),\ \text{mod}(\cdot,\mu),\ \text{dealloc}(\cdot,\cdot)\ \}
$$

that can occur at runtime is analyzed statically. Thus, all risky pointers that can trigger memory errors at runtime are within the analysis scope.

Spatially risky pointer identification (Section 4.4.3). By definition of the type grammar, only raw pointers lack compiler-enforced bounds checks. That says, if a spatial error occurs at an operation either $*p$ or $p \oplus k$, then p must be an exposed raw pointer in this operation, since safe references are checked by the Rust semantics and owners

expose only safe references in safe code. LiteRSan therefore begins by conservatively annotating all instructions involving raw pointers, ensuring that all raw pointers are initially captured. The subsequent analysis prunes false positives by leveraging the fact that exposed raw pointers that bypass Rust's safety guarantees can only be used within unsafe code. Consequently, LiteRSan excludes raw pointers that are encapsulated inside $Owner\langle T\rangle$ and cannot be directly accessed, since any access must go through the owner interface and is protected by memory safety rules. This pruning preserves soundness while improving precision. Additionally, LiteRSan identifies and handles each unsafe API performing any defined operation individually, based on a thorough review of the Rust standard library without sacrificing coverage.

Temporally risky pointer identification (Section 4.4.4). LiteRSan employs a lifetime-aware taint analysis over the pointer creation and derivation operations of the source language. Concretely, taint advances at ptr_from_obj(p, o) and ptr_from_ptr(p, p_s, κ) with κ from Definition 8. Taint sources are exposed raw pointers. A pointer q is reported as temporally risky if there exists a tainted pointer p such that q and p may reference the same object and their lifetimes overlap, where lifetimes are induced by pointer definition (ptr_from_obj(p, o) and ptr_from_ptr(p, p_s, κ)) and deallocation (dealloc(o, p)).

Coverage of alias discovery. The taint propagation tracks only pointer derivation instructions, namely ptr_from_ptr(p, p_s, κ) with κ in Definition 8. This is sufficient because Rust's ownership and borrowing rules ensure that aliasing can only occur through explicit and syntactically visible pointer derivations [59]. Therefore, every alias of an exposed raw pointer is reachable by taint propagation along the pointer derivation chains. Specifically, backward taint propagation stops at ownership transfer sites that create new owners. By Rust's lifetime rules, the original owners become invalid after ownership transfer and any use of such invalid pointers are prohibited by the compiler. Consequently, the original owners cannot alias an exposed raw pointer created later, since its lifetime ends before the raw pointer is introduced. Terminating the backward taint propagation at ownership transfer sites is therefore sound.

Loops. Although loops often require iterative data-flow analysis to reach a fixed point, one pass suffices in this analysis, which reasons only about static derivation relationships, instead of mutable program states (e.g., ranges, offsets, sizes). Since LLVM IR contains a fixed and finite set of pointer derivation instructions and the loop body consists of a finite number of instances of κ, visiting each derivation once reaches the same set of aliased pointers as any iterative fixed point, thus all aliases are conservatively identified.

Inter-procedural propagation. Soundness requires that every inter-procedural derivation be captured. LiteRSan resolves indirect calls with a type-based analysis [99, 146] that is sound as it collects all potential callees. Therefore, every inter-procedural pointer derivation is represented by the operation ptr_from_ptr(p, p_s, κ), where κ is param(f, j) when the j-th formal argument p of a potential callee f receives the actual parameter p_s; κ is ret(f) when the pointer p in caller receives the return value p_s of f. Both param(f, j) and ret(f) are instances of κ included in Definition 8, ensuring that taint propagates inter-procedurally to all potential aliases, which establishes inter-procedural soundness.

Given the arguments above, the lifetime-aware taint analysis is sound for identifying temporally risky pointers in the defined source language. Because aliasing in Rust occurs only through explicit pointer derivations, taint need only propagate along these derivations; ownership transfer provides a safe stopping point; loops contain a finite number of derivations; and inter-procedural flows are conservatively included. Consequently, any pointer whose lifetime overlaps with that of an exposed raw pointer to the same object is tainted, ensuring that all temporally risky pointers are identified.

Precision Discussion

While the analysis is sound by design, potential false negatives may arise in practice due to implementation limitations, such as compiler optimizations or missing IR from dynamically linked code. The approach may also introduce some over-approximation, for example, by analyzing derivations that never occur during actual execution. However, this imprecision is significantly reduced compared to prior work [27, 88] relying on traditional points-to analysis, which is unaware of pointer lifetimes. In contrast, LiteRSan excludes pointers that are invalid at the time of exposed raw pointer creation or not derived from tainted sources, as these are protected by Rust's compile-time safety guarantees and are not susceptible to temporal memory safety violations.

4.5 Lightweight Runtime Checks

In this section, we present the lightweight runtime checks of LiteRSan. LiteRSan uses compact memory safety metadata in place of red zones and shadow memory to address Challenge C2 and adopts a selective instrumentation strategy to address Challenge C3 (see Section 4.2.2). We describe the metadata structures in Section 4.5.1 and the metadata inference approach in Section 4.5.2. Section 4.5.3 details our selective instrumentation

strategy, and Section 4.5.4 explains how the instrumented checks uses memory safety metadata to detect memory safety violations at runtime.

4.5.1 Metadata Structure

For each risky pointer, LiteRSan maintains *spatial metadata* for spatially risky pointers and *temporal metadata* for temporally risky pointers. This metadata is inferred during static analysis, propagated at runtime through instrumentation, and stored in dedicated data structures rather than embedded directly in the pointer representation (e.g., fat pointers). At runtime, the metadata is dynamically updated and used to detect memory safety violations.

Spatial Metadata

For spatially risky pointers, LiteRSan tracks three key attributes that are necessary and sufficient to enforce spatial memory safety in Rust:

- **Capacity:** the maximum number of elements allowed in a referenced memory object. For pointers referencing scalar-type objects (e.g., integers), the capacity is set to 1.
- **Initialized length:** the number of elements that have been initialized within a referenced memory region.
- **Offset:** the index within an object that the pointer references.

LiteRSan maintains a map from each spatially risky pointer to its spatial metadata, which consists of the attributes listed above; this metadata map is stored separately from the pointers themselves.

Temporal Metadata

For temporally risky pointers, LiteRSan tracks the following information as temporal metadata:

- **May-alias relationships:** pointers that may reference the same memory object are grouped to represent their potential aliasing.
- **Ownership:** the owner(s) of the referenced objects.

LiteRSan uses **taint source raw pointers** to associate temporally risky pointers with their temporal metadata via two maps: a *reverse map* that links each temporally risky pointer back to its originating taint source, and a *forward map* that links each taint

source to the corresponding metadata which it shares with its tainted pointers. This dual-mapping design avoids redundant metadata storage for multiple pointers derived from the same source while accurately associating each temporally risky pointer with its temporal metadata.

To record may-alias relationships, LiteRSan groups all temporally risky pointers that may reference the same memory object into a *pointer set*. This pointer set integrates with LiteRSan 's taint analysis, as may-alias pointers identified by LiteRSan share a common taint source and can be grouped during taint analysis without additional effort. When a raw pointer is derived from another, their pointer sets are merged to ensure a complete representation of may-alias relationships.

Ownership is a critical component of temporal metadata, as the owners are the pointers responsible for deallocating the referenced objects. As a result, owners must be tracked at runtime to update the temporal state (i.e., dangling or valid) of all pointers in the same pointer set. To record the owners, LiteRSan maintains a dedicated *owner set*, which is a subset of the pointer set.

4.5.2 Metadata Inference

Metadata inference is performed during static analysis. LiteRSan infers each risky pointer's memory safety metadata at its definition site and instruments the code to receive and maintain the inferred metadata, enabling runtime validation.

Spatial Metadata Inference

To infer spatial metadata for raw pointers, LiteRSan analyzes each raw pointer's definition site and, if necessary, traces pointer derivations back to the allocation site of the memory object, where the spatial metadata is defined. Specifically, LiteRSan backtracks along the pointer-derivation chain to locate the corresponding *root* pointer, which is the first pointer that references the memory object. LiteRSan then extracts spatial metadata from the root pointer's definition site, where the memory allocation appears as an operand at. Depending on how the memory region is referenced, metadata extraction falls into two distinct cases:

- In the **direct case**, the root pointer references a memory object whose spatial metadata can be directly extracted. This occurs when the referenced object is a basic container provided by the Rust standard library, such as vectors (`Vec<T>`) and arrays (`[T; N]`), which manage contiguous memory regions. In this case, LiteRSan directly obtains

spatial metadata from the container's fields. For example, the `length` and `capacity` fields in `Vec<T>` directly provide *Initialized Length* and *Capacity* (Section 4.5.1).

- In the **indirect case**, the root pointer refers to a memory object indirectly via abstractions such as `Box<T>` or `Rc<T>`, which do not explicitly carry spatial metadata. In this case, LiteRSan backtracks to the definition site of the underlying `T`-typed object to infer and extract spatial metadata.

Additionally, when the definition site of any raw pointer involves pointer arithmetic, LiteRSan computes the resultant *Offset*. Another category of spatially risky pointers, smart pointers associated with unsafe APIs, is relatively uncommon. Thus, LiteRSan handles these cases individually, applying sanitizer checks based on the semantics of each unsafe API, as discussed in Section 4.4.3.

Once spatial metadata is extracted from a root pointer, it is transmitted to the spatially risky pointers along the pointer-derivation chain. The root pointer, along with the intermediate pointers on this chain, is referred to as *metadata-carrying pointers*. Note that metadata-carrying pointers are not necessarily risky themselves but are tracked to enable accurate metadata propagation.

To enable runtime checking, LiteRSan propagates statically inferred spatial metadata to spatially risky pointers through inserted instrumentation. This process involves two cases. First, if the risky pointer is a root pointer, the metadata is available at its definition site; therefore, instrumenting its definition site suffices. Second, if the risky pointer is a derived pointer, metadata must be passed along the derivation chain. In this case, LiteRSan instruments the definition sites of all metadata-carrying pointers in the chain to ensure proper metadata propagation to the derived pointer.

Temporal Metadata Inference

For temporal metadata, the may-alias relationships and the taint-source raw pointers are inferred during the identification of temporally risky pointers, through lifetime-aware taint analysis, as described in Section 4.5.1. The owner(s) of the referenced memory objects are inferred by analyzing the definition site of each tainted pointer based on Rust's ownership model. Specifically, owners are smart pointer types (e.g., `Box` and `Rc`) that manage the lifetime of a memory object and are responsible for its deallocation. LiteRSan detects owners by analyzing pointer types and determining whether they are associated with memory deallocation, typically indicated by their `Drop` implementation [36].

Robustness and Completeness Guarantee

LiteRSan adopts a hybrid strategy to infer spatial and temporal metadata, ensuring that metadata is accurately and reliably extracted.

Spatial metadata is inferred for (i) raw pointers and (ii) smart pointers associated with unsafe APIs. For raw pointers, LiteRSan extracts initial metadata from referenced objects via static analysis and passes it to runtime functions through instrumentation. This method is **robust** because Rust's ownership and borrowing rules guarantee aliasing occurs only through explicit, visible derivations [59], making metadata fully traceable. It is also **complete** since static analysis extracts only initial metadata, while runtime instrumentation ensures precise, immediate updates. During execution, root pointers are initialized before derived pointers, enabling accurate metadata transmission and preventing stale values. For smart pointers associated with unsafe APIs, LiteRSan addresses each case individually based on its semantics. The small number of such APIs, combined with tailored handling, ensures **robustness** and **completeness**.

Temporal metadata consists of (i) may-alias relationships and (ii) object owners. May-alias relationships are inferred through lifetime-aware taint analysis, proven sound in Section 4.4.5. Owners are identified based on pointer types and `Drop` usage, following Rust's ownership rules. As both types and `Drop` usage are statically deterministic, this approach guarantees both **completeness** and **robustness**.

4.5.3 Selective Instrumentation

LiteRSan performs selective instrumentation by applying only the runtime checks necessary to each identified risky pointer. To support these checks, it also instruments the program to propagate statically inferred metadata and manage it at runtime. This section first introduces the five classes of instrumentation used in LiteRSan and how they are selectively applied according to the risky pointer type and program context.

Instrumentation Types

LiteRSan defines five instrumentation classes (**I1**–**I5**) to initialize and update metadata during execution, and perform runtime memory safety checks based on the metadata.

- **I1**: Pointer activation (metadata initialization).
- **I2**: Spatial metadata update.
- **I3**: Pointer deactivation (temporal metadata update).

- **I4**: Spatial safety checks.
- **I5**: Temporal safety checks.

I1 type. I1 instrumentation is inserted at the definition sites of root pointers, metadata-carrying pointers, and identified risky pointers. It propagates statically inferred metadata to the runtime function, and marks each pointer as active by registering it with its associated spatial and temporal metadata.

For *spatially risky pointers* and their derivation sources (i.e., root and metadata-carrying pointers), I1 establishes a runtime mapping between each pointer and its spatial metadata. Root pointers are initialized with metadata directly from static analysis, while derived pointers inherit metadata from their source pointer. For *temporally risky pointers*, I1 maintains a mapping from each taint-source raw pointer to associated temporal metadata and a reverse mapping from each tainted pointer to its taint source, as discussed in Section 4.5.1. Depending on whether the definition site corresponds to a taint source or a tainted pointer, I1 creates or updates these mappings.

I2 type. LiteRSan employs **I2** to update spatial metadata at runtime in two scenarios: (1) when the offset of a spatially risky pointer or metadata-carrying pointer is modified (e.g., through pointer arithmetic), and (2) when the underlying memory object is modified (e.g., through container operations, such as `push()` and `pop()`).

For (1), I2 updates the *Offset* field in its associated spatial metadata. For (2), I2 updates and synchronizes the *Initialized length* and/or *Capacity* field(s) for both the pointer performing the modification and all preceding pointers in the derivation chain. This is because those pointers all reference the same memory object.

I3 type. I3 instrumentation is inserted at deallocation sites, including function returns for stack-allocated objects and explicit `drop` operations for heap-allocated objects, to deactivate invalidated pointers. If a non-owner pointer is deallocated, LiteRSan only marks this pointer as dangling, as the referenced object is not deallocated. If an owner deallocates the memory object, or if the last owner is invalidated (e.g., via `mem::forget()`), I3 queries the reverse map to identify the taint-source raw pointer and marks it, along with all pointers in its pointer set, as dangling.

I4 and I5 types. LiteRSan applies **I4** at pointer arithmetic and dereference sites of spatially risky pointers, and **I5** at dereference and deallocation sites of temporally risky pointers, to detect spatial and temporal memory safety violations, respectively. The detection mechanisms for I4 and I5 are detailed in Section 4.5.4.

Instrumentation Strategy

LiteRSan employs a selective instrumentation strategy that inserts only the necessary code at each instrumentation site. This approach ensures that metadata remains up-to-date and that memory safety violations are detected efficiently. Table 4.1 summarizes the instrumentation strategy.

Pointer Type	Definition	Dereference	Pointer Arithmetic	Container Modifier	Deallocation
Spatially Risky Pointers	**I1**	**I4**	**I2**, **I4**	**I2**	-
Temporally Risky Pointers	**I1**	**I5**	-	-	**I5**, **I3**
Metadata-Carrying Pointers	**I1**	-	**I2**	**I2**	-

Table 4.1: Selective instrumentation strategy. Each class of instrumentation is applied based on the type of pointer and the type of operation, ensuring that only the necessary code is inserted at each instrumentation site. At the pointer arithmetic of a spatially risky pointer, I2 is before I4. At the deallocation site of a temporally risky pointer, I5 is before I3.

For *all three pointer types*, LiteRSan inserts **I1** at their definition sites to register the pointers along with their associated spatial or temporal metadata at runtime, making them activated. This metadata is later used to initialize derived pointers or to validate memory safety at runtime.

For *spatially risky pointers*, which may cause spatial memory safety violations, LiteRSan selectively inserts **I2** and **I4**. **I2** is placed at pointer arithmetic operations (e.g., `add()`, `offset()`) and at container modifier operations (e.g., unsafe API `set_len()`) to update spatial metadata instantly at runtime, enabling I4 to precisely perform spatial memory safety validation. **I4** is inserted at pointer arithmetic and dereference sites to detect spatial violations using the maintained spatial metadata. Importantly, I2 is placed before I4 at pointer arithmetic sites, ensuring that any invalid pointer arithmetic is detected immediately using the up-to-date metadata.

For *temporally risky pointers*, which can result in temporal memory safety violations, LiteRSan selectively applies **I3** and **I5**. **I3** is inserted at deallocation sites (e.g., `drop()`) to update temporal metadata, ensuring that the temporal validity state of each pointer is accurately maintained. **I5** is applied at dereference and deallocation sites to detect temporal errors, such as use-after-free and double-free. At deallocation sites, I5 is placed before I3 to prevent I3 from prematurely marking the pointer as invalid and causing erroneous double-free reports.

For *metadata-carrying pointers*, which serve only to transmit spatial metadata (see Section 4.5.2), LiteRSan selectively applies **I2** at pointer arithmetics and container

modifiers (e.g., `Vec::push()`), ensuring that spatial metadata is instantly updated and accurately propagated to spatially risky pointers.

4.5.4 Runtime Check Mechanism

LiteRSan maintains and updates memory safety metadata through I1–I3, and leverages this metadata at runtime to detect spatial and temporal violations via I4 and I5, respectively. Figure 4.1 (see Section 4.1) summarizes the complete set of memory errors that LiteRSan detects and illustrates how I4 and I5 perform runtime checks.

For all risky pointers, LiteRSan first performs null checks at dereferences. For spatially risky pointers, I4 compares the pointer's *Offset* against its *Initialized Length* and *Capacity*. An access is alarmed as a use-before-initialization if the offset exceeds the initialized length, or as an out-of-bounds access if it exceeds the capacity. For temporally risky pointers, I5 consults the temporal metadata maintained by I3 to determine whether the pointer is dangling. A dereference of a dangling pointer triggers a use-after-free alarm, while a deallocation of a dangling pointer raises a double-free alarm.

Benefits of Our Strategy

The benefits of LiteRSan are to impose lower runtime and memory overhead while providing more comprehensive detection coverage in comparison with ASan-based approaches [27, 88]. Specifically, LiteRSan selectively instruments only the pointers that may potentially violate memory safety and inserts only necessary checks for them. These pointers are only a subset of the pointers that existing ASan-based techniques [27, 88] check. Moreover, LiteRSan can detect memory safety bugs that existing ASan-based approaches may miss by maintaining the fine-grained spatial and temporal metadata. This metadata is compact and lightweight, contributing further to runtime efficiency.

4.6 Implementation

We implement LiteRSan on top of LLVM-14. It takes the program's LLVM bitcode as an input, performs static analysis, applies selective instrumentation, and generates instrumented LLVM bitcode.

The input bitcode is generated from Rust programs using a customized version of `rustc-1.64-nightly`. This compiler is extended to support metadata annotation during the MIR-to-LLVM IR lowering phase. Following ERASan's [88] annotation mechanism,

we modify the `codegen-llvm` and `codegen-ssa` to insert LLVM metadata on instructions involving raw pointers. These annotations enable LiteRSan to identify raw pointers during static analysis, as described in Section 4.4.3. On the other hand, the output bitcode includes inserted calls to runtime functions (i.e., I1-I5 in Section 4.5.3). At runtime, the instrumented code is invoked to update metadata instantly and detect memory safety violations as the program executes.

4.7 Evaluation

We elaborate the experiment setup (Section 4.7.1) and evaluate LiteRSan in comparison with the two state-of-the-art Rust sanitizers, ERASan [88] and RustSan [27], as follows: runtime overhead (Section 4.7.2), memory overhead (Section 4.7.3), compilation overhead (Section 4.7.4), and bug detection capability (Section 4.7.5).

4.7.1 Experiment Setup

All experiments were conducted on a server with an Intel Xeon Gold 6230 CPU, 80 cores, and 754 GB RAM, running Ubuntu 24.04.

Benchmarks

We evaluated LiteRSan on 28 benchmarks: 26 most frequently downloaded Rust crates from `crates.io`[4] and two real-world applications (`servo` and `ripgrep`). For each benchmark, we compile and execute both the baseline versions (without instrumentation) and the instrumented versions produced by each sanitizer 20 times. The average compilation time, execution time, and memory usage are used to compute the respective overheads. For the benchmarks shared with ERASan, we use its experiment setup [34]. Thus, we use the same test cases to ensure a fair comparison. For the remaining benchmarks, we use their native test suites.

Compilation Process

To ensure both accurate static analysis and evaluation on realistic production situation, we adopt a staged compilation process. For each benchmark, the compiler first emits LLVM IR with inlining and LLVM prepopulate passes disabled so that MIR-derived

[4] `crates.io` is Rust's official package registry. Each crate is implemented entirely in Rust and compiled as an independent unit, functioning as either a library or an executable with benchmark input.

annotations are preserved in the IR. At this stage, LiteRSan and the comparison tools (ERASan and RustSan) perform static analysis and insert their respective instrumentation. After instrumentation, compilation resumes with the standard optimization pipeline to produce optimized (`i.e., -O3`) executables.

This approach is essential rather than a shortcut. Running LLVM optimizations before analysis can replace or eliminate original instructions and drop critical metadata, leading to missed identification of risky operations. This metadata-preservation challenge is not unique to LiteRSan but equally affects ERASan and RustSan; analyzing directly on optimized IR would cause all these tools to miss protecting unsafe operations. By applying analysis on pre-optimized IR, we preserve full semantic information and ensure that risky pointers are precisely identified.

At the same time, compiling instrumented IR under `-O3` guarantees that our evaluation reflects realistic deployment conditions. Most LLVM optimizations transform instructions in place rather than reordering them, so checks remain associated with the correct memory operations. Furthermore, because LiteRSan implements its checks as runtime calls with observable actual effects, subsequent LLVM optimizations do not remove them. Our experiments demonstrate this property in practice, as LiteRSan achieves 100% bug detection even when executing fully optimized binaries.

In summary, this staging is a necessary and fair methodology: it preserves metadata for accurate analysis, ensures that checks are preserved under aggressive optimization, and yields performance results correspond to practical compilation, runtime, and memory costs under realistic deployment. Future improvements in preserving Rust-specific metadata across optimization could streamline this process, but the present design is the only viable way to ensure correctness across Rust memory safety sanitizers.

Ablation Study

To decompose LiteRSan's overhead, we developed a variant Semi-LiteRSan, which uses LiteRSan 's static analysis to identify risky pointers and selectively instruments runtime checks, but employs ASan's runtime validation mechanisms in place of LiteRSan 's metadata-based approach. Comparing Semi-LiteRSan with LiteRSan isolates the benefit of lightweight metadata while comparing Semi-LiteRSan with RustSan and ERASan highlights the benefit of our precise risky pointer identification, as RustSan and ERASan employ ASan's runtime validation mechanisms.

4.7.2 Runtime Overhead

Runtime overhead refers to the additional execution time introduced by sanitizer checks compared to the baseline execution time without instrumentation. Table 4.2 shows the runtime overhead of LiteRSan, ERASan, and RustSan. Across all benchmarks, LiteRSan consistently achieves the lowest overhead. By geometric mean, LiteRSan incurs only 18.84% runtime overhead, significantly lower than ERASan's 152.05% and RustSan's 183.50%, presenting reductions of 87.61% and 89.73%, respectively.

Benchmark	LOC	Pointer Count			Runtime Overhead (%)		
		Raw	Risky	Aliased	LiteRSan	ERASan	RustSan
base64	7,025	1,787	14,320	131,242	35.21	-	431.28
byteorder	3,411	95	355	5,391	1.72	53.36	76.37
bytes(buf)	5,867	88	376	2,904	28.39	137.90	154.33
bytes(bytes)	5,867	91	411	2,089	25.57	166.97	169.62
bytes(mut)	5,867	102	484	2,267	27.82	157.28	165.48
indexmap	8,693	386	2,214	32,132	23.06	287.14	293.65
itoa	613	9	32	291	20.34	116.11	131.05
memchr	1,139	50	185	3,133	13.18	212.39	217.74
num-integer	2,383	570	3,095	8,449	1.17	5.59	8.34
ryu	3,443	17	82	2,247	17.71	63.80	70.89
semver	2,483	24	81	839	4.83	317.21	388.52
smallvec	2,912	59	278	982	13.34	134.53	152.33
strsim-rs	1,102	109	431	1,015	1.06	380.51	389.72
uuid(format)	4,971	15	50	62,149	40.62	362.41	411.04
uuid(parse)	4,971	15	50	62,091	37.31	338.06	402.53
bat	53,517	2,567	25,546	138,726	321.11	894.97	931.36
crossbeam-utils	31,246	64	290	2,227	1.28	116.09	136.17
hashbrown	10,384	51	383	6,596	9.32	58.65	69.45
hyper	20,952	1,824	15,201	114,269	43.57	278.16	297.23
rand(generators)	15,220	27	78	2,619	31.85	26.49	31.64
rand(misc)	15,220	66	258	2,527	7.94	10.12	23.47
regex	65,417	294	3,896	6,383	38.54	831.87	867.34
ripgrep	33,226	1,864	18,734	-	304.03	-	-
syn	58,884	1,088	23,034	186,730	72.29	583.25	618.92
tokio	69,875	1,482	19,375	74,697	53.35	563.24	593.22
unicode	172,875	95	363	1,054	8.89	47.96	62.37
url	40,595	353	2,266	34,368	21.06	612.75	686.41
servo	11.26 M	1.27 M	14.63 M	-	86.58	-	-
GeoMean	-	-	-	-	**18.84**	152.05	183.50

Table 4.2: Runtime overhead comparison. Benchmarks are grouped by scale. Pointer count reports exposed raw pointers (Raw) and risky pointers (Risky) identified by LiteRSan, along with raw pointers plus aliases (Aliased) identified by traditional points-to analysis. Overheads are shown for LiteRSan, ERASan, and RustSan. Nonapplicable results are listed as -. For some benchmarks (`base64`, `ripgrep`, and `servo`), ERASan and/or RustSan do not have runtime overhead because the benchmarks cannot successfully be executed with the sanitizers employed.

The superior efficiency of LiteRSan arises from two key factors. First, it applies lifetime-aware taint analysis to precisely identify risky pointers, greatly reducing false positives that would otherwise result in unnecessary instrumentation. In contrast, ERASan and RustSan rely on traditional points-to analysis in SVF [131], conservatively treating all aliases of raw pointers (ERASan) or pointers in unsafe code (RustSan) as risky, leading to redundant instrumentation for pointers whose safety is already enforced by the Rust compiler. As shown in the *Pointer Count* columns of Table 4.2, LiteRSan identifies far fewer risky pointers than the aliased counts across all benchmarks, with reductions ranging from 38.96% to 99.92%. Second, LiteRSan employs the lightweight metadata-based runtime mechanism in place of the heavyweight red zone and shadow memory mechanisms deployed in ASan-based tools.

To quantify the contributions of these two components discussed above, we conduct an ablation study using Semi-LiteRSan, a variant of LiteRSan introduced earlier in this section. As presented in Table 4.3, the runtime overhead of Semi-LiteRSan is 70.04% by geometric mean and is consistently higher than LiteRSan across all benchmarks but still lower than ERASan and RustSan. Comparing Semi-LiteRSan with LiteRSan isolates the impact of the lightweight metadata-based runtime mechanism, as both instrument the same sites but differ in runtime validation mechanisms. The results demonstrate that the metadata-based runtime checking in LiteRSan reduces overhead by 73.10%. Comparing Semi-LiteRSan against ERASan and RustSan, which use the same runtime validation mechanisms but different static analyses, highlights the benefit of our Rust-specific analysis, achieving reductions of 53.94% and 61.83%, respectively.

In addition, Table 4.3 also compares LiteRSan with ASan by reporting the number of risky pointers identified by LiteRSan (*Risky* column), the total pointers guarded by ASan (*ASan-guarded* column), and their runtime overheads. Across all benchmarks, LiteRSan identifies greatly fewer risky pointers than ASan-guarded pointers, indicating that most pointers in Rust are already guaranteed safe and therefore ASan checks are excessively redundant. By leveraging precise Rust-specific static analysis, LiteRSan substantially reduces the number of instrumented pointers and achieves significant runtime performance gains. Specifically, ASan incurs 359.90% runtime overhead by geometric mean, while Semi-LiteRSan incurs 70.04%, demonstrating that precise risky pointer identification and selective instrumentation reduce overhead by 80.54%. LiteRSan further lowers the overhead to 18.94%, showing that replacing ASan's heavyweight shadow memory and red zones with our lightweight metadata-based mechanism yields an additional 73.10% reduction.

Benchmark	Pointer Count		Runtime Overhead (%)		
	Risky	ASan-guarded	LiteRSan	Semi-LiteRSan	ASan
base64	14,320	1,075,072	35.21	89.27	624.71
byteorder	355	24,275	1.72	17.75	131.23
bytes(buf)	376	37,783	28.39	78.27	289.20
bytes(bytes)	411	18,354	25.57	79.32	292.98
bytes(mut)	484	26,796	27.82	79.64	295.37
indexmap	2,214	378,711	23.06	87.86	419.93
itoa	32	5,195	20.34	88.56	241.11
memchr	185	16,633	13.18	53.17	342.19
num-integer	3,095	75,990	1.17	5.77	35.12
ryu	82	9,769	17.71	52.62	101.45
semver	81	4,787	4.83	32.91	536.83
smallvec	278	13,736	13.34	53.71	284.08
strsim-rs	431	4,038	1.06	26.63	522.02
uuid(format)	50	730,713	40.62	75.66	481.02
uuid(parse)	50	731,856	37.31	96.46	467.23
bat	25,546	1,859,420	321.11	542.13	1,187.60
crossbeam-utils	290	5,695	1.28	16.19	187.15
hashbrown	383	67,106	9.32	35.51	124.61
hyper	15,201	737,542	43.57	84.70	323.03
rand(generators)	78	17,807	31.85	55.77	151.86
rand(misc)	258	9,632	7.94	30.49	131.42
regex	3,896	49,383	38.54	243.58	1,584.07
ripgrep	18,734	962,161	304.03	524.54	1,210.16
syn	23,034	1,770,205	72.29	278.64	1,390.21
tokio	19,375	728,064	53.35	108.46	914.47
unicode	363	7,715	8.89	42.59	157.94
url	2,266	255,893	21.06	85.74	937.53
servo	14.63 M	16,994,787	86.58	218.05	1,281.96
GeoMean	-	-	**18.84**	70.04	359.90

Table 4.3: Ablation study of runtime overhead. The table shows pointer counts (risky and ASan-guarded), and runtime overhead comparison of LiteRSan, Semi-LiteRSan, and ASan across benchmarks.

4.7.3 Memory Overhead

Memory overhead refers to the additional memory consumed by sanitizer checks over the baseline memory usage. We measured memory overhead using the Linux `time` command [137], which reports the peak resident set size (max RSS) of the process. This metric captures the maximum amount of memory consumption during execution, which is widely adopted as a practical measure of memory overhead.

As shown in Table 4.4, LiteRSan demonstrates a substantial advantage in memory efficiency over both ERASan and RustSan. Across all benchmarks, LiteRSan incurs only trivial overhead, with a geometric mean of only 0.81%. In contrast, ERASan and RustSan

Benchmark	LOC	Pointer Count			Memory Overhead (%)		
		Expo-raw	Risky	Aliased	LiteRSan	ERASan	RustSan
base64	7,025	1,787	14,320	131,242	3.56	-	5,271.84
byteorder	3,411	95	355	5,391	0.03	357.73	406.27
bytes(buf)	5,867	88	376	2,904	2.68	86.27	98.58
bytes(bytes)	5,867	91	411	2,089	2.15	2,218.86	2,143.63
bytes(mut)	5,867	102	484	2,267	2.19	5,376.09	5,339.28
indexmap	8,693	386	2,214	32,132	1.78	1,754.14	2,090.46
itoa	613	9	32	291	1.43	65.71	69.83
memchr	1,139	50	185	3,133	1.96	49.61	61.56
num-integer	2,383	570	3,095	8,449	0.02	524.07	674.62
ryu	3,443	17	82	2,247	0.28	81.69	85.22
semver	2,483	24	81	839	1.88	6,832.92	7,683.53
smallvec	2,912	59	278	982	1.14	4,370.14	4,518.99
strsim-rs	1,102	109	431	1,015	1.36	5,568.87	5,729.60
uuid(format)	4,971	15	50	62,149	0.15	875.22	879.71
uuid(parse)	4,971	15	50	62,091	0.15	1,065.03	1,094.13
bat	53,517	2,567	25,546	138,726	4.96	4,619.51	5,017.39
crossbeam-utils	31,246	64	290	2,227	0.05	57.85	58.97
hashbrown	10,384	51	383	6,596	0.37	6,613.42	6,814.56
hyper	20,952	1,824	15,201	114,269	3.69	2,681.30	2,966.32
rand(generators)	15,220	27	78	2,619	0.26	85.18	88.64
rand(misc)	15,220	66	258	2,527	0.22	1,457.08	1,654.97
regex	65,417	294	3,896	6,383	1.83	8,191.68	8,574.25
ripgrep	33,226	1,864	18,734	-	4.77	-	-
syn	58,884	1,088	23,034	186,730	1.65	327.11	343.74
tokio	69,875	1,482	19,375	74,697	1.33	1,343.17	1,504.36
unicode	172,875	95	363	1,054	0.86	56.83	63.38
url	40,595	353	2,266	34,368	1.37	312.35	356.88
servo	11.26 M	1.27 M	14.63 M	-	2.82	-	-
GeoMean	-	-	-	-	**0.81**	739.27	861.98

Table 4.4: Memory overhead comparison. Benchmarks are grouped by scale. Pointer counts report exposed raw pointers (Expo-raw) and risky pointers identified by LiteRSan, along with raw pointers plus aliases (Aliased) identified by traditional points-to analysis. Memory overheads are shown for LiteRSan, ERASan, and RustSan. Nonapplicable results are listed as -.

impose significantly higher memory costs, reaching 739.27% and 861.98%, respectively. These results highlight the contribution of LiteRSan 's lightweight metadata design, which maintains only the essential spatial and temporal information for runtime validation, eliminating the substantial memory footprint associated with shadow memory and red zone mechanisms in ASan-based tools.

The contribution of metadata-based runtime mechanism is evident when comparing LiteRSan with Semi-LiteRSan. As shown in Table 4.5, Semi-LiteRSan incurs 443.90% memory overhead, whereas LiteRSan reduces this by 99.82%. Additionally, reduced instrumentation also plays a key role. Semi-LiteRSan achieves 39.95% and

Benchmark	Pointer Count		Memory Overhead (%)		
	Risky	ASan-guarded	LiteRSan	Semi-LiteRSan	ASan
base64	14,320	1,075,072	3.56	3,015.06	10,723.03
byteorder	355	24,275	0.03	248.62	1,065.10
bytes(buf)	376	37,783	2.68	37.63	411.25
bytes(bytes)	411	18,354	2.15	1,044.84	21,368.49
bytes(mut)	484	26,796	2.19	3,433.34	64,467.47
indexmap	2,214	378,711	1.78	1,233.98	4,620.00
itoa	32	5,195	1.43	28.21	123.68
memchr	185	16,633	1.96	39.85	81.62
num-integer	3,095	75,990	0.02	416.44	769.33
ryu	82	9,769	0.28	66.26	92.20
semver	81	4,787	1.88	4,686.86	13,347.98
smallvec	278	13,736	1.14	2,863.90	78,376.98
strsim-rs	431	4,038	1.36	3,943.39	9,869.13
uuid(format)	50	730,713	0.15	157.72	1,008.17
uuid(parse)	50	731,856	0.15	166.83	1,239.21
bat	25,546	1,859,420	4.96	1,817.64	36,539.25
crossbeam-utils	290	5,695	0.05	41.91	548.96
hashbrown	383	67,106	0.37	4,067.88	28,530.65
hyper	15,201	737,542	3.69	1,752.87	35,747.78
rand(generators)	78	17,807	0.26	69.77	378.25
rand(misc)	258	9,632	0.22	1,050.56	8,025.48
regex	3,896	49,383	1.83	6,739.62	37,082.47
ripgrep	18,734	962,161	4.77	51.02	28,728.19
syn	23,034	1,770,205	1.65	33.68	609.24
tokio	19,375	728,064	1.33	769.19	1,843.51
unicode	363	7,715	0.86	41.17	333.48
url	2,266	255,893	1.37	206.59	791.42
servo	14.63 M	16,994,787	2.82	8,174.46	52,329.43
GeoMean	-	-	**0.81**	443.90	3,282.12

Table 4.5: Ablation study of memory overhead. The table shows pointer counts (risky and ASan-guarded), and memory overhead comparison of LiteRSan, Semi-LiteRSan, and ASan across benchmarks.

48.50% lower overhead than ERASan and RustSan, respectively, purely due to the reduced instrumentation sites identified by precise Rust-specific static analysis.

Moreover, when compared with ASan, which introduces 3,282.12% overhead, Semi-LiteRSan lowers memory overhead by 86.48% through precise static analysis and selective instrumentation, while LiteRSan further cuts overhead by 99.06%, again demonstrating the benefit of the lightweight metadata-based runtime checks.

4.7.4 Compilation Overhead

Compilation overhead refers to the additional compilation time introduced by a sanitizer's static analysis and instrumentation compared to the baseline build. As shown

in Table 4.6, LiteRSan consistently incurs substantially lower overhead. By geometric mean, LiteRSan produces 97.21% overhead, compared to 1,635.35% for ERASan and 1,193.31% for RustSan, presenting reductions of 94.06% and 91.85%, respectively. Moreover, all benchmarks are successfully compiled with LiteRSan deployed, while both comparison tools fail to complete compilation for `servo` within a 24-hour timeout and and crash with segmentation faults when analyzing `ripgrep` due to SVF errors. These results highlight the scalability of LiteRSan for large, complex, real-world applications.

Benchmark	**LOC**	**Pointer Count**			**Compilation Overhead (%)**		
		Expo-raw	**Risky**	**Aliased**	**LiteRSan**	**ERASan**	**RustSan**
base64	7,025	1,787	14,320	131,242	174.17	SE	6,260.03
byteorder	3,411	95	355	5,391	66.52	614.35	674.93
bytes(buf)	5,867	88	376	2,904	47.94	636.77	748.59
bytes(bytes)	5,867	91	411	2,089	45.25	625.32	682.33
bytes(mut)	5,867	102	484	2,267	46.19	627.09	755.26
indexmap	8,693	386	2,214	32,132	103.59	5,807.79	2,310.17
itoa	613	9	32	291	47.76	334.31	414.24
memchr	1,139	50	185	3,133	86.14	514.86	560.28
num-integer	2,383	570	3,095	8,449	186.57	1,359.64	1,512.04
ryu	3,443	17	82	2,247	64.03	868.87	931.11
semver	2,483	24	81	839	42.34	390.24	451.72
smallvec	2,912	59	278	982	59.21	422.05	489.63
strsim-rs	1,102	109	431	1,015	58.25	452.52	528.97
uuid(format)	4,971	15	50	62,149	207.82	9,230.63	2,669.96
uuid(parse)	4,971	15	50	62,091	202.32	9,038.49	2,684.37
bat	53,517	2,567	25,546	138,726	167.91	25,020.17	4,826.05
crossbeam-utils	31,246	64	290	2,227	104.06	586.35	639.52
hashbrown	10,384	51	383	6,596	72.16	1,227.80	1,182.08
hyper	20,952	1,824	15,201	114,269	184.98	20,920.41	5,127.40
rand(generators)	15,220	27	78	2,619	84.27	776.41	835.54
rand(misc)	15,220	66	258	2,527	95.93	1,079.36	866.17
regex	65,417	294	3,896	6,383	128.70	1,463.51	923.09
ripgrep	33,226	1,864	18,734	-	142.07	SEGV	SEGV
syn	58,884	1,088	23,034	186,730	123.84	25,397.17	1,499.46
tokio	69,875	1,482	19,375	74,697	149.02	18,607.29	4,965.73
unicode	172,875	95	363	1,054	54.95	549.73	677.47
url	40,595	353	2,266	34,368	191.89	1,618.63	1,521.94
servo	11.26 M	1.27 M	14.63 M	-	186.13	TO	TO
GeoMean	-	-	-	-	**97.21**	1,635.35	1,193.31

Table 4.6: Compilation overhead comparison. Benchmarks are grouped by scale. Pointer counts report exposed raw pointers (Expo-raw) and risky pointers identified by LiteRSan, along with raw pointers plus aliases (Aliased) identified by traditional points-to analysis. Compilation vverheads are shown for LiteRSan, ERASan, and RustSan. Nonapplicable results are listed as -. SE indicates silent exit, SEGV indicates segmentation fault, and TO indicates a compilation timeout.

The efficiency of LiteRSan arises from its lightweight, Rust-specific static analysis, in contrast to the SVF-based approaches of ERASan and RustSan. As discussed in

Section 2.2.1, SVF is a heavyweight points-to analysis framework, which substantially increases compilation cost. LiteRSan instead restricts analysis to reachable functions and examines only instructions relevant to risky pointer identification, performing this in a single pass. Despite its efficiency, this analysis remains sufficient to identify all spatially and temporally risky pointers, as formally proved in Section 4.4.5, enabling precise selective instrumentation with modest compilation overhead.

4.7.5 Security Evaluation

In addition to performance improvement, LiteRSan provides a more comprehensive memory error detection coverage than ERASan and RustSan, both of which share the same capability as ASan. Therefore, we show the bug detection capability of LiteRSan and compare it only with ASan (whose detailed approach and limitations are discussed in Section 2.2.1).

RUSTSEC ID	Type	Class	ASan	LiteRSan
RUSTSEC-2023-0021	NPD	Null-pointer deref	✓	✓
RUSTSEC-2023-0024	NPD	Null-pointer deref	✓	✓
RUSTSEC-2023-0038	OOB	Spatial	✓	✓
RUSTSEC-2023-0039	OOB	Spatial	✓	✓
RUSTSEC-2023-0056	OOB	Spatial	✗	✓
RUSTSEC-2024-0002	OOB	Spatial	✗	✓
RUSTSEC-2025-0003	OOB	Spatial	✓	✓
RUSTSEC-2025-0005	OOB	Spatial	✓	✓
RUSTSEC-2025-0018	OOB	Spatial	✓	✓
RUSTSEC-2023-0045	UBI	Spatial	✓	✓
RUSTSEC-2023-0087	UBI	Spatial	✗	✓
RUSTSEC-2024-0018	UBI	Spatial	✓	✓
RUSTSEC-2024-0374	UBI	Spatial	✓	✓
RUSTSEC-2024-0400	UBI	Spatial	✓	✓
RUSTSEC-2023-0010	DF	Temporal	✓	✓
RUSTSEC-2023-0078	UAF	Temporal	✗	✓
RUSTSEC-2024-0007	UAF	Temporal	✓	✓
RUSTSEC-2024-0017	UAF	Temporal	✓	✓
RUSTSEC-2025-0016	UAF	Temporal	✓	✓
RUSTSEC-2025-0022	UAF	Temporal	✓	✓

Table 4.7: Bug detection capability of ASan and LiteRSan. Listed are the 20 most recent memory safety vulnerabilities in RustSec, grouped by bug class.

We analyzed bugs reported by RustSec, the Rust Security Advisory Database [161], over the past two years, focusing on the cases where bug root causes (i.e., PoCs) are publicly available for validation. We list memory safety bugs in our scope (discussed in Section 4.1) in Table 4.7. LiteRSan successfully detects all of 20 bugs, whereas ASan

fails to identify two out-of-bounds access bugs, one use-before-initialization bug, and one use-after-free bug. These cases occur in Rust-specific contexts (illustrated as case studies). Because ASan was originally designed for C/C++, it effectively detects conventional memory safety violations but lacks the ability to handle Rust-specific memory safety rules and check per-pointer spatial and temporal memory safety. In contrast, LiteRSan incorporates Rust's memory safety rules in its static analysis and enforces per-pointer spatial and temporal memory safety checks, enabling the detection of such missing bugs.

RUSTSEC ID	Type	Class	ASan	LiteRSan
RUSTSEC-2020-0061	NPD	Null-pointer deref	✓	✓
RUSTSEC-2023-0013	NPD	Null-pointer deref	✓	✓
RUSTSEC-2020-0039	OOB	Spatial	✓	✓
RUSTSEC-2020-0167	OOB	Spatial	✓	✓
RUSTSEC-2021-0003	OOB	Spatial	✓	✓
RUSTSEC-2021-0048	OOB	Spatial	✓	✓
RUSTSEC-2021-0094	OOB	Spatial	✓	✓
RUSTSEC-2023-0015	OOB	Spatial	✓	✓
RUSTSEC-2023-0016	OOB	Spatial	✓	✓
RUSTSEC-2023-0030	OOB	Spatial	✓	✓
RUSTSEC-2023-0032	OOB	Spatial	✓	✓
RUSTSEC-2019-0023	UAF	Temporal	✓	✓
RUSTSEC-2020-0005	UAF	Temporal	✓	✓
RUSTSEC-2020-0060	UAF	Temporal	✓	✓
RUSTSEC-2020-0091	UAF	Temporal	✓	✓
RUSTSEC-2020-0097	UAF	Temporal	✓	✓
RUSTSEC-2022-0070	UAF	Temporal	✓	✓
RUSTSEC-2022-0078	UAF	Temporal	✓	✓
RUSTSEC-2023-0005	UAF	Temporal	✓	✓
RUSTSEC-2023-0009	UAF	Temporal	✓	✓
RUSTSEC-2021-0031	UAF	Temporal	✓	✓
RUSTSEC-2021-0128	UAF	Temporal	✓	✓
RUSTSEC-2021-0130	UAF	Temporal	✓	✓
RUSTSEC-2019-0009	DF	Temporal	✓	✓
RUSTSEC-2019-0034	DF	Temporal	✓	✓
RUSTSEC-2020-0038	DF	Temporal	✓	✓
RUSTSEC-2021-0018	DF	Temporal	✓	✓
RUSTSEC-2021-0028	DF	Temporal	✓	✓
RUSTSEC-2021-0033	DF	Temporal	✓	✓
RUSTSEC-2021-0039	DF	Temporal	✓	✓
RUSTSEC-2021-0042	DF	Temporal	✓	✓
RUSTSEC-2021-0047	DF	Temporal	✓	✓
RUSTSEC-2021-0053	DF	Temporal	✓	✓

Table 4.8: Detection capability of ASan and LiteRSan on memory safety vulnerabilities. The listed vulnerabilities are grouped by bug class. They are memory safety vulnerabilities discovered and registered in RustSec earlier than the 20 memory safety vulnerabilities in Table 4.7.

As a complement of Table 4.7, Table 4.8 reports the detection results of LiteRSan and ASan on earlier RustSec vulnerabilities. Together, these tables cover all publicly disclosed memory safety bugs reported in the RustSec Advisory Database to date. In total, the combined dataset includes 55 vulnerabilities: 21 use-after-free (UAF), 3 double-free (DF), 21 out-of-bounds accesses (OOB), 7 use-before-initialization (UBI), and 3 null-pointer dereference (NPD). They also cover all the vulnerabilities experimented by ERASan. As a result, LiteRSan successfully detects all the listed bugs identified by ASan, demonstrating full coverage of ASan's detection capabilities on Rust memory safety bugs, while detecting additional Rust-specific bugs, including UBI and certain safe-code out-of-bounds violations. We illustrate one spatial memory safety bug in Case Study 1 and one temporal memory safety bug in Case Study 2.

Case Study 1

RUSTSEC-2023-0056 [118] is an out-of-bounds access vulnerability in the `vm-memory` crate [142]. In this crate, `get_slice` is a trait method intended to return a smart pointer-like abstraction, `VolatileSlice`, over a slice, but it lacks a default implementation. If a user implements this method incorrectly, for example, by miscomputing the `offset` or `count`, the internal pointer in the returned `VolatileSlice` may reference memory outside the intended region, potentially leading to out-of-bounds access.

Several methods in `VolatileMemory` trait, such as `get_ref` and `get_array_ref`, invoke `get_slice` without proper bounds checking, thereby raising potential memory safety violations. Listing 4.2 illustrates this issue using `get_atomic_ref` as an example. In line 6, `get_slice` is invoked to wrap an allocated memory region with a requested size of `size_of::<T>()` bytes. In line 9, the internal pointer of the returned `VolatileSlice` (i.e., `slice.addr`) is cast and dereferenced without verifying whether the underlying memory actually aligns with the requested bounds. If `get_slice` returns a region smaller than the requested region, any dereference beyond the actual region results in an out-of-bounds access.

According to our experiment, ASan cannot detect this bug because it only places red zones around memory objects. However, in this case, the pointer returned by `get_slice` may point to a valid memory object, but beyond the actual valid bound, which is within this object. As a result, *invalid* accesses beyond the actual bound but within the larger allocated object remain undetected by ASan, since no red zones are placed at the logical boundary returned by `get_slice`. In contrast, LiteRSan tracks memory safety metadata for each pointer at its definition site. This allows LiteRSan to precisely

```
fn get_slice(&self, offset: usize, count: usize)
                    -> Result<VolatileSlice<BS<Self::B>>>;

fn get_atomic_ref<T: AtomicInteger>(&self, offset: usize)
                                          -> Result<&T> {
    let slice = self.get_slice(offset, size_of::<T>())?;
    slice.check_alignment(align_of::<T>())?;

    unsafe { Ok(&*(slice.addr as *const T)) }
}
```

Listing 4.2: Potential out-of-bounds access in `get_atomic_ref`.

extract the actual bound of `slice.addr` and perform spatial memory safety checks, detecting potential out-of-bounds access.

Case Study 2

RUSTSEC-2023-0078 [119] is a use-after-free vulnerability reported in the `tracing` crate [141]. As shown in Listing 4.3, the vulnerability originates from the improper use of `mem::forget` in line 4, where the exclusive owner of the underlying memory object is forgotten. While `mem::forget` prevents the object's destructor from being called, the Rust compiler considers the object to be logically invalid after its owner is forgotten. The memory region may subsequently be reused by the compiler, making any future access to the original object via existing pointers a use-after-free violation.

```
pub fn into_inner(self) -> T {
    let span: *const Span = &self.span;
    let inner: *const ManuallyDrop<T> = &self.inner;
    mem::forget(self);

    let _span = unsafe { span.read() };
    let inner = unsafe { inner.read() };
    ManuallyDrop::into_inner(inner)
}
```

Listing 4.3: Potential use-after-free in `Instrumented::into_inner`.

This vulnerability stems from a violation of Rust's ownership model rather than traditional heap misuse found in C/C++. Because the memory is never explicitly freed, ASan does not update its shadow memory to mark the region as invalid, thus fails to detect the temporal safety violations. Covering this type of vulnerability in ASan is fundamentally challenging as ASan is unaware of Rust's ownership semantics. To detect such bugs, ASan would need to determine whether an object still has a valid owner at every program point, which requires a significant change in the underlying

design of ASan. In contrast, LiteRSan is designed with ownership awareness. It tracks ownership and marks the pointers referencing the same object as dangling when the last owner is dropped. Any subsequent dereferences of the dangling pointers are flagged as use-after-free. This allows LiteRSan to detect ownership-related memory safety violations that lie beyond ASan's capabilities.

4.8 Discussion

This section discusses the limitations of LiteRSan and explores how complementary safety mechanisms can be combined to provide more comprehensive coverage of Rust memory safety.

4.8.1 Limitations

Although LiteRSan significantly advances memory safety enforcement in Rust and surpasses the capabilities of ASan-based tools, it shares some of their inherent limitations. Specifically, type conversion bugs and cross-language vulnerabilities remain out of scope.

Type Conversion Bugs

We consider type conversion bugs, such as those introduced via unsafe APIs like `transmute()` [139], out of scope, as it is widely accepted as orthogonal to spatial and temporal memory safety. The same view is shared by many prior works [25,47,104]. Type conversion bugs stem from reinterpreting one type as another, which can break safety invariants without violating spatial bounds or temporal validity. As a result, LiteRSan may not be able to detect them if they do not violate spatial bounds or temporal validity. State-of-the-art ASan-based tools [27, 88] also share the same problem [127]. One way to address this problem is to integrate type confusion bug detection techniques [25]. But it is worth noting that LiteRSan is able to detect such type confusion bugs if they stem from memory errors such as UAF.

Cross-language Attacks

LiteRSan leverages Rust's ownership and borrowing semantics to infer memory safety metadata and enforce spatial and temporal safety. As a result, it does not guarantee the detection of memory safety violations originating from external code written in languages without such semantics, such as C/C++ libraries interfaced via FFI. Similar to ERASan

and RustSan, LiteRSan does not cover cross-language memory safety violations, which are considered out of scope.

To address cross-language attacks, one potential direction is to integrate LiteRSan with existing isolation or sandboxing techniques [126, 133], to mitigate memory errors originating from external code. Another direction is to extend the scope of LiteRSan to external libraries and enforce runtime checks at FFI boundaries. However, this requires a deep understanding of the semantics of each external API, which is difficult to generalize and automate. It also requires static analysis on C/C++ code, which lacks Rust's safety guarantees, making Rust-specific analysis inapplicable. Despite these challenges, cross-language memory safety is a promising direction for future work.

4.8.2 Toward Full Coverage of Rust Memory Safety

Achieving full coverage of Rust memory safety requires combining different classes of defenses, as each has inherent limitations and addresses different aspects of the problem.

Static Analysis

Static analysis tools such as Rudra [13], SafeDrop [29], and MIRChecker [72] aim to detect potential vulnerabilities at compile time without execution overhead. They are designed to catch certain classes of errors systematically but often suffer from false positives and limited precision in analyzing complex unsafe code. While static analysis cannot enforce safety at runtime, it provides an essential first line of defense and complements dynamic techniques like sanitizers.

Isolation and Sandboxing

Isolation techniques, including XRust [74], TRust [14], and PKRUSafe [64], enforce strong protection boundaries to eliminate memory safety violations. Unlike sanitizers, which aim to detect violations, isolation focuses on preventing exploitation. These approaches can complement LiteRSan by mitigating risks from external code (e.g., C libraries via FFI) and by providing fallback protection when runtime detection is incomplete.

Fuzzing Frameworks

Fuzzing frameworks such as AFL++ [40], Cargo-fuzz [22], and Honggfuzz [35] explore program behaviors dynamically by generating diverse inputs. While fuzzing can expose previously unknown bugs, it does not provide soundness guarantee and is fundamentally

coverage-driven. When combined with sanitizers like LiteRSan, fuzzing becomes significantly more effective, as sanitizer checks increase the likelihood of detecting subtle memory safety violations during fuzzing campaigns.

Memory Safety Sanitizers

Sanitizers such as ASan, ERASan, RustSan, and LiteRSan provide runtime detection of memory safety violations with varying trade-offs in precision and performance. LiteRSan advances this line of work by introducing Rust-specific static analysis to reduce overhead while maintaining soundness. However, as discussed earlier, sanitizers alone cannot address all classes of bugs, such as type confusion or FFI-related vulnerabilities.

Integration for Comprehensive Memory Safety

Though various classes of defenses and advanced techniques in each class, no single approach provides complete coverage of Rust memory safety. Static analysis is efficient but incomplete; fuzzing is effective in practice but probabilistic; isolation is strong but coarse-grained; and sanitizers are precise but introduce runtime overhead. A promising path forward is to integrate these complementary mechanisms in a layered fashion. For example, static analysis can prune potential errors early, memory safety sanitizers can selectively enforce runtime safety checks, fuzzing can explore residual cases, and isolation can restrict violations from unsafe external code. By combining these complementary approaches, the community can further approach more practical and comprehensive memory safety for Rust programs.

4.9 Conclusion

Rust provides strong memory safety through its ownership semantics and type system. However, these guarantees can be undermined by the use of unsafe code, which reintroduces memory safety vulnerabilities. To detect such bugs, ASan-based tools are commonly used. Yet, even state-of-the-art sanitizers like ERASan and RustSan incur substantial performance and memory overhead, and still fail to catch certain memory safety violations.

Therefore, we propose a novel Rust memory sanitizer, LiteRSan, with lower overhead and more comprehensive and accurate memory error detection than ERASan and RustSan. We achieve this goal by precisely identifying risky pointers and selectively instrumenting

those risky pointers to minimize overhead while ensuring higher detection coverage than ERASan and RustSan. As a result, LiteRSan imposes 18.84% runtime overhead, 97.21% compilation overhead, and 0.81% memory overhead, with geometric mean, while ERASan and RustSan, respectively, incur 152.05% and 183.50% runtime overhead, 1635.35% and 1193.31% compilation overhead, and 739.27% and 861.98% memory overhead. Furthermore, LiteRSan detects 55 memory safety vulnerabilities with 100% accuracy, unlike ASan-based approaches that miss four of them.

Chapter 5
Conclusion and Future Work

This dissertation has explored how exploiting language and semantics specific features can advance program analysis by improving both precision and efficiency. Traditional program analyses often face an inherent trade-off: fine-grained approaches offer higher precision at significant computational cost, while coarse-grained techniques achieve efficiency at the expense of accuracy. This work demonstrates that leveraging domain knowledge embedded in programming languages and problem contexts provides a practical path to reconciling this trade-off.

Through two concrete projects, DeepType and LiteRSan, the dissertation shows that semantics-aware insights enable program analysis to advance in both accuracy and efficiency. The first project, DeepType, introduced strong multi-layer type analysis (SMLTA) for C/C++, which leverages the types of composite data structures and explicitly models data-flow across their type hierarchies to refine indirect call target resolution. The second project, LiteRSan, developed a Rust-specific memory safety sanitizer that incorporates Rust's ownership and borrowing semantics to precisely identify risky pointers and enforce memory safety with low compilation and runtime overheads. Altogether, these projects demonstrate that exploiting language and semantics is not merely an optimization, but a foundational design principle for building precise, efficient, and practical program analyses.

5.1 Key Insights

A key insight that emerges from both projects is that language and semantics specific features provide a guiding principle for advancing program analysis, particularly in achieving a more effective balance between precision and efficiency.

In the DeepType project, the fundamental intuition behind multi-layer type analysis is that function pointers in C/C++ are frequently embedded within composite data structures. By exploiting this structural characteristic and explicitly modeling the data-flow across multiple layers of these types, DeepType maintains multi-layer type information and performs multi-layer type matching. This approach enables precise indirect call target resolution without incurring the prohibitive cost of whole-program, flow-sensitive pointer analysis. Semantic knowledge of C/C++ type system restricts the analysis search space, yielding high precision at modest computational cost.

In LiteRSan project, the key observation is that Rust already enforces strong safety guarantees through its ownership and borrowing system. Instead of treating all pointers uniformly as in traditional points-to analysis, LiteRSan explicitly models Rust's semantics, excluding compiler-guaranteed safe pointers and focusing exclusively on risky pointers that may violate memory safety at runtime. The precise identification of risky pointers, coupled with lightweight metadata-based runtime checks, delivered significant precision and efficiency improvements over language-agnostic sanitizers such as ASan and its variants.

In the LiteRSan project, the insight lies in recognizing that Rust enforces strong safety guarantees through its ownership and borrowing model. Rather than treating all pointers uniformly, as in traditional points-to analysis, LiteRSan explicitly incorporates Rust's semantics to exclude compiler-guaranteed safe pointers and focuses exclusively on risky pointers that may lead to runtime memory safety violations. This precise identification, combined with lightweight metadata-based runtime checks, achieves significant improvements in both precision and efficiency over language-agnostic sanitizers such as ASan and its successors.

Altogether, these projects illustrate that semantic awareness enables analyses that are both precise and lightweight, in contrast to generic, language-agnostic approaches. More broadly, they demonstrate that leveraging program semantics can fundamentally reshape the cost–precision landscape of program analysis.

5.2 Extensions and Future Directions

While this dissertation has shown the benefits of exploiting language and semantics, it also points to several promising directions for extending these ideas.

Building upon DeepType, one extension is to generalize strong multi-layer type analysis (SMLTA) beyond function pointers. By modeling the multi-layer type flows of

general pointers, SMLTA could be used to identify the memory objects they reference or to capture higher-level abstractions such as sockets, file descriptors, or I/O management structures. Such an extension would broaden the utility of SMLTA beyond call graph construction to enforcing memory safety, strengthening RAII-style resource management, and supporting analyses of complex systems-level abstractions.

LiteRSan demonstrates the power of Rust-specific static analysis for memory safety enforcement, but this analyses approach can be adapted for broader domains, such as constructing fine-grained data- and control-dependency graphs for Rust programs. These graphs could enable analyses that isolate provably safe regions of code from unsafe code or from safe code that depends on unsafe blocks. This direction would provide stronger compartmentalization guarantees for Rust applications and open pathways for formally verifying subsets of real-world Rust systems.

The principle of exploiting semantics and domain specific properties to balance precision and efficiency extends beyond the two research problems studied in this dissertation. Other security-critical challenges, such as concurrency bug detection, side-channel defense, or software isolation and sandboxing, could benefit from analyses that are tailored to the unique abstractions of a language or research domain. For example, concurrency analyses could exploit language-level synchronization constructs, while side-channel defenses could exploit compiler-level representations of timing-sensitive operations.

Finally, the rise of large language models (LLMs) and AI-driven systems introduces new domains where semantics-aware program analysis may play a role. These systems are still programs, albeit with unique structures such as computational graphs, dynamic control flows, and data propagation across neural layers. Borrowing from the lessons of this dissertation, program analysis techniques could be adapted to exploit these semantics to detect, mitigate, and verify security vulnerabilities in AI systems. Such efforts would extend program analysis into emerging domains while retaining the core principle of precision–efficiency balance through semantic awareness.

Bibliography

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, page 340–353, New York, NY, USA, 2005. Association for Computing Machinery.

[2] U.S. General Services Administration. Improving the nation's cybersecurity, 2025.

[3] The afl.rs Project Developers. RUST Fuzzing : afl.rs. `https://github.com/rust-fuzz/afl.rs`, 2025.

[4] Arushi Aggarwal. *Hybrid Static/Dynamic Type Safety for C/C++ Programs*. PhD thesis, University of Illinois at Urbana-Champaign, 2011.

[5] Alexander Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35(2-3):79–111, 1999.

[6] Frances E Allen. Control flow analysis. *ACM Sigplan Notices*, 5(7):1–19, 1970.

[7] Hussain M. J. Almohri and David Evans. Fidelius Charm: Isolating Unsafe Rust Code. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, CODASPY '18, page 248–255, New York, NY, USA, 2018. Association for Computing Machinery.

[8] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, Citeseer, 1994.

[9] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. How Do Programmers Use Unsafe Rust? *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[10] AWS. Verify the safety of the rust standard library. `https://aws.amazon.com/blogs/opensource/verify-the-safety-of-the-rust-standard-library/`, 2025.

[11] Kevin Backhouse. Cueing up a calculator: an introduction to exploit development on linux. `https://github.blog/2023-12-06-cueing-up-a-calculator-an-introduction-to-exploit-development-on-linux/`, 2023.

[12] David F Bacon and Peter F Sweeney. Fast static analysis of c++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 324–341, 1996.

[13] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. Rudra: Finding memory safety bugs in rust at the ecosystem scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 84–99, New York, NY, USA, 2021. Association for Computing Machinery.

[14] Inyoung Bang, Martin Kayondo, HyunGon Moon, and Yunheung Paek. TRust: A compilation framework for in-process isolation to protect safe rust against untrusted code. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6947–6964, Anaheim, CA, August 2023. USENIX Association.

[15] Marcel Böhme, Danushka Liyanage, and Valentin Wüstholz. Estimating residual risk in greybox fuzzing. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 230–241, New York, NY, USA, 2021. Association for Computing Machinery.

[16] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 2329–2344, 2017.

[17] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2329–2344, New York, NY, USA, 2017. Association for Computing Machinery.

[18] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. Rwset: Attacking path explosion in constraint-based test generation. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 351–366, Berlin, Heidelberg, 2008. Springer-Verlag.

[19] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *13th USENIX Security Symposium (USENIX Security 04)*, San Diego, CA, August 2004. USENIX Association.

[20] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Comput. Surv.*, 50(1), apr 2017.

[21] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association.

[22] The cargo-fuzz Project Developers. RUST Fuzzing : cargo fuzz. `https://github.com/rust-fuzz/cargo-fuzz`, 2025.

[23] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 45–58, New York, NY, USA, 2009. Association for Computing Machinery.

[24] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2095–2108, New York, NY, USA, 2018. Association for Computing Machinery.

[25] Hung-Mao Chen, Xu He, Shu Wang, Xiaokuan Zhang, and Kun Sun. Typepulse: detecting type confusion bugs in rust programs. In *Proceedings of the 34th USENIX Conference on Security Symposium*, SEC '25, USA, 2025. USENIX Association.

[26] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, page 265–278, New York, NY, USA, 2011. Association for Computing Machinery.

[27] Kyuwon Cho, Jongyoon Kim, Kha Dinh Duy, Hajeong Lim, and Hojoon Lee. RustSan: Retrofitting AddressSanitizer for efficient sanitization of rust. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 3729–3746, Philadelphia, PA, August 2024. USENIX Association.

[28] Cristina Cifuentes, François Gauthier, Behnaz Hassanshahi, Padmanabhan Krishnan, and Davin McCall. The role of program analysis in security vulnerability detection: Then and now. *Computers & Security*, 135:103463, 2023.

[29] Mohan Cui, Chengjun Chen, Hui Xu, and Yangfan Zhou. Safedrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis. *ACM Trans. Softw. Eng. Methodol.*, 32(4), May 2023.

[30] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P Kemerlis. Retracer: Triaging crashes by reverse execution from partial memory dumps. In *Proceedings of the 38th International Conference on Software Engineering*, pages 820–831, 2016.

[31] CVE-2023-43642. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-43641`, 2023.

[32] Jansens Dana. Supporting the use of Rust in the Chromium project. https://security.googleblog.com/2021/09/supporting-use-of-rust-in-chromium.html, 2021.

[33] Alain Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. *SIGPLAN Not.*, 29(6):230–241, June 1994.

[34] The ERASan Developers. Github : ERASan. https://github.com/S2-Lab/ERASan, 2025.

[35] The Honggfuzz Project Developers. RUST Fuzzing : honggfuzz-rs. https://github.com/rust-fuzz/honggfuzz-rs, 2025.

[36] The Rust Project Developers. Trait drop, 2025.

[37] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. Efficient protection of Path-Sensitive control security. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 131–148, Vancouver, BC, August 2017. USENIX Association.

[38] Gregory J Duck and Roland HC Yap. Effectivesan: type and memory error detection using dynamically typed c/c++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–195, 2018.

[39] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. Is Rust Used Safely by Software Developers? In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 246–257, New York, NY, USA, 2020. Association for Computing Machinery.

[40] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. {AFL++}: Combining incremental steps of fuzzing research. In *14th USENIX workshop on offensive technologies (WOOT 20)*, 2020.

[41] Xinyang Ge, Weidong Cui, and Trent Jaeger. Griffin: Guarding control flows using intel processor trace. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 585–598, New York, NY, USA, 2017. Association for Computing Machinery.

[42] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. Fine-grained control-flow integrity for kernel software. In *2016 IEEE European Symposium on Security and Privacy*, pages 179–194, 2016.

[43] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. K-miner: Uncovering memory corruption in linux. In *Network and Distributed System Security (NDSS) Symposium*, 2018.

[44] Krzysztof Grajek. Rust static vs. dynamic dispatch, 2024.

[45] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. The seahorn verification framework. In *International Conference on Computer Aided Verification*, pages 343–361. Springer, 2015.

[46] Istvan Haller, Enes Göktaş, Elias Athanasopoulos, Georgios Portokalidis, and Herbert Bos. Shrinkwrap: Vtable protection without loose ends. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC '15, page 341–350, New York, NY, USA, 2015. Association for Computing Machinery.

[47] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. Typesan: Practical type confusion detection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 517–528, New York, NY, USA, 2016. Association for Computing Machinery.

[48] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, page 54–61, New York, NY, USA, 2001. Association for Computing Machinery.

[49] Susan Horwitz. Precise flow-insensitive may-alias analysis is np-hard. *ACM Trans. Program. Lang. Syst.*, 19(1):1–6, January 1997.

[50] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing unique code target property for control-flow integrity. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1470–1486, New York, NY, USA, 2018. Association for Computing Machinery.

[51] Baojian Hua, Wanrong Ouyang, Chengman Jiang, Qiliang Fan, and Zhizhong Pan. Rupair: Towards automatic buffer overflow detection and rectification for rust. In *Proceedings of the 37th Annual Computer Security Applications Conference*, ACSAC '21, page 812–823, New York, NY, USA, 2021. Association for Computing Machinery.

[52] Kaiming Huang, Yongzhe Huang, Mathias Payer, Zhiyun Qian, Jack Sampson, Gang Tan, and Trent Jaeger. The taming of the stack: Isolating stack data from memory errors. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, 2022. The Internet Society.

[53] Kaiming Huang, Mathias Payer, Zhiyun Qian, Jack Sampson, Gang Tan, and Trent Jaeger. Top of the heap: Efficient memory error protection of safe heap objects. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 1330–1344, 2024.

[54] Yongzhe Huang, Vikram Narayanan, David Detweiler, Kaiming Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. KSplit: Automating device driver isolation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 613–631. USENIX Association, July 2022.

[55] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, June 2021. Order Number: 253665-075US.

[56] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block oriented programming: Automating data-only attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1868–1882, New York, NY, USA, 2018. Association for Computing Machinery.

[57] Suman Jana, Yuan Jochen Kang, Samuel Roth, and Baishakhi Ray. Automatically detecting error handling bugs using error specifications. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 345–362, Austin, TX, August 2016. USENIX Association.

[58] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Safedispatch: Securing c++ virtual calls from memory corruption attacks. In *Network and Distributed System Security Symposium*, 2014.

[59] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

[60] Yuan Kang, Baishakhi Ray, and Suman Jana. Apex: Automated inference of error specifications for c apis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, page 472–482, New York, NY, USA, 2016. Association for Computing Machinery.

[61] Martin Kayondo, Inyoung Bang, Yeongjun Kwak, HyunGon Moon, and Yunheung Paek. MetaSafe: Compiling for protecting smart pointer metadata to ensure safe rust integrity. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 3711–3728, Philadelphia, PA, August 2024. USENIX Association.

[62] Sun Hyoung Kim, Cong Sun, Dongrui Zeng, and Gang Tan. Refining indirect call targets at the binary level. In *Network and Distributed System Security Symposium (NDSS)*, 2021.

[63] Sun Hyoung Kim, Dongrui Zeng, Cong Sun, and Gang Tan. Binpointer: Towards precise, sound, and scalable binary-level pointer analysis. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, CC 2022, page 169–180, New York, NY, USA, 2022. Association for Computing Machinery.

[64] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. Pkru-safe: Automatically locking down the heap between safe and unsafe languages. In *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022.

[65] Steve Klabnik and Carol Nichols. The rust programming language. `https://doc.rust-lang.org/stable/book/`, 2022.

[66] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. Sandcrust: Automatic sandboxing of unsafe components in rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, PLOS'17, page 51–57, New York, NY, USA, 2017. Association for Computing Machinery.

[67] The Rust Programming Language. Understanding ownership. `https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html`, 2024.

[68] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2nd International Symposium on Code Generation and Optimization*, CGO '04, Palo Alto, CA, 2004. IEEE Computer Society.

[69] Guoren Li, Hang Zhang, Jinmeng Zhou, Wenbo Shen, Yulei Sui, and Zhiyun Qian. A hybrid alias analysis and its application to global variable protection in the linux kernel. In *32nd USENIX Security Symposium*, 2023.

[70] Hongzhe Li, Taebeom Kim, Munkhbayar Bat-Erdene, and Heejo Lee. Software vulnerability detection using backward trace analysis and symbolic execution. In *2013 International Conference on Availability, Reliability and Security*, pages 446–454, 2013.

[71] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C. S. Lui. Detecting cross-language memory management issues in rust. In *Computer Security – ESORICS 2022: 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26–30, 2022, Proceedings, Part III*, page 680–700, Berlin, Heidelberg, 2022. Springer-Verlag.

[72] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C.S. Lui. Mirchecker: Detecting bugs in rust programs via static analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 2183–2196, New York, NY, USA, 2021. Association for Computing Machinery.

[73] Zhenpeng Lin, Yueqi Chen, Yuhang Wu, Dongliang Mu, Chensheng Yu, Xinyu Xing, and Kang Li. Grebe: Unveiling exploitation potential for linux kernel bugs. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2078–2095, 2022.

[74] Peiming Liu, Gang Zhao, and Jeff Huang. Securing unsafe rust programs with xrust. In *Proceedings of the ACM/IEEE 42nd International Conference on Software*

Engineering, ICSE '20, page 234–245, New York, NY, USA, 2020. Association for Computing Machinery.

[75] Shen Liu, Gang Tan, and Trent Jaeger. Ptrsplit: Supporting general pointers in automatic program partitioning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2359–2371, 2017.

[76] Shen Liu, Gang Tan, and Trent Jaeger. Ptrsplit: Supporting general pointers in automatic program partitioning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2359–2371, New York, NY, USA, 2017. Association for Computing Machinery.

[77] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2: bounded translation validation for llvm. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 65–79, New York, NY, USA, 2021. Association for Computing Machinery.

[78] Kangjie Lu and Hong Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1867–1881, New York, NY, USA, 2019. Association for Computing Machinery.

[79] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2019.

[80] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Software fault isolation with api integrity and multi-principal modules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 115–128, New York, NY, USA, 2011. Association for Computing Machinery.

[81] Giorgio Martinez. Unlocking performance: Optimizing rust's dynamic dispatch. `https://medium.com/@giorgio.martinez1926/unlocking-performance-optimizing-rusts-dynamic-dispatch-600b57f78f99`, 2023.

[82] Samuel Mergendahl, Nathan Burow, and Nathan Burow. Cross-language attacks. In *Network and Distributed System Security Symposium (NDSS)*, 2022.

[83] Samuel Mergendahl, Nathan Burow, and Hamed Okhravi. Cross-language attacks. In *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, 01 2022. The Internet Society.

[84] Benjamin Cisneros Merino. *Jiapi: A Type Checker Generator for Statically Typed Languages*. PhD thesis, University of Nevada, Las Vegas, 2022. UNLV Theses, Dissertations, Professional Papers, and Capstones No. 4579.

[85] Microsoft. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape, 2019.

[86] Shane Miller and Carl Lerche. Sustainability with Rust. `https://aws.amazon.com/blogs/opensource/sustainability-with-rust/`, 2022.

[87] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[88] Jiun Min, Dongyeon Yu, Seongyun Jeong, Dokyung Song, and Yuseok Jeon. Erasan: Efficient rust address sanitizer. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4053–4068, 2024.

[89] MITRE Corporation. Cve-2018-1000810. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1000810`, 2018.

[90] MITRE Corporation. Cve-2019-16760. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-16760`, 2019.

[91] Amit Nadiger. Dynamic & static dispatch in rust. `https://www.linkedin.com/pulse/dynamic-static-dispatch-rust-amit-nadiger/`, 2023.

[92] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Softbound: highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 245–258, New York, NY, USA, 2009. Association for Computing Machinery.

[93] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. CETS: Compiler-Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, pages 31–40. ACM, 2010.

[94] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. Lightweight kernel isolation with virtualization and vm functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS international conference on virtual execution environments*, pages 157–171, 2020.

[95] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, jun 2007.

[96] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. Binary-level directed fuzzing for Use-After-Free vulnerabilities. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 47–62, San Sebastian, October 2020. USENIX Association.

[97] Flemming Nielson. The typed lambda-calculus with first-class processes. In *Proceedings of the Parallel Architectures and Languages Europe, Volume II: Parallel Languages*, PARLE '89, page 357–373, Berlin, Heidelberg, 1989. Springer-Verlag.

[98] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 2005.

[99] Ben Niu and Gang Tan. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 577–587, New York, NY, USA, 2014. Association for Computing Machinery.

[100] Ben Niu and Gang Tan. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 914–926, New York, NY, USA, 2015. Association for Computing Machinery.

[101] NSA-CSS. Nsa releases guidance on how to protect against software memory safety issues, 2022.

[102] Shankara Pailoor, Andrew Aday, and Suman Jana. MoonShine: Optimizing OS fuzzer seed selection with trace distillation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 729–743, Baltimore, MD, August 2018. USENIX Association.

[103] Jens Palsberg. Type-based analysis and applications. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 20–27, 2001.

[104] Chengbin Pang, Yunlan Du, Bing Mao, and Shanqing Guo. Mapping to bits: Efficiently detecting type confusion errors. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, page 518–528, 2018.

[105] Eric Pauley, Gang Tan, Danfeng Zhang, and Patrick McDaniel. Performant binary fuzzing without source code using static instrumentation. In *2022 IEEE Conference on Communications and Network Security (CNS)*, pages 226–235, 2022.

[106] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: Fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710, 2018.

[107] Aravind Prakash, Xunchao Hu, and Heng Yin. vfguard: Strict protection for virtual function calls in cots c++ binaries. In *Network and Distributed System Security Symposium*, 2015.

[108] Boqin Qin, Yilun Chen, Haopeng Liu, Hua Zhang, Qiaoyan Wen, Linhai Song, and Yiying Zhang. Understanding and Detecting Real-World Safety Issues in Rust. *IEEE Trans. Softw. Eng.*, 50(6):1306–1324, March 2024.

[109] Zvonimir Rakamarić and Michael Emmi. Smack: Decoupling source language details from verifier implementations. In *International Conference on Computer Aided Verification*, pages 106–113. Springer, 2014.

[110] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[111] Elijah Rivera, Samuel Mergendahl, Howard Shrobe, Hamed Okhravi, and Nathan Burow. Keeping safe rust safe with galeed. In *Annual Computer Security Applications Conference*, ACSAC '21, page 824–836, New York, NY, USA, 2021. Association for Computing Machinery.

[112] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1), mar 2012.

[113] Konstantin Rubinov, Lucia Rosculete, Tulika Mitra, and Abhik Roychoudhury. Automated partitioning of android applications for trusted execution environments. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 923–934, New York, NY, USA, 2016. Association for Computing Machinery.

[114] Rust Project Developers. Rust compiler development guide, 2025.

[115] Rust Project Developers. The rust standard library, 2025.

[116] Rust Project Developers. The rust standard library, 2025.

[117] Rust Project Developers. The rust standard library, 2025.

[118] RustSec Advisory Database. Rustsec-2023-0056. urlhttps://rustsec.org/advisories/RUSTSEC-2023-0056.html, 2023.

[119] RustSec Advisory Database. Rustsec-2023-0078. `https://rustsec.org/advisories/RUSTSEC-2023-0078.html`, 2023.

[120] Suman Saha, Jean-Pierre Lozi, Gaël Thomas, Julia L. Lawall, and Gilles Muller. Hector: Detecting resource-release omission faults in error-handling code for systems software. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12, 2013.

[121] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *2015 IEEE Symposium on Security and Privacy*, pages 745–762, 2015.

[122] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference*, pages 309–318, Boston, MA, June 2012. USENIX Association.

[123] Hovav Shacham. 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*, CCS '07, page 552–561, New York, NY, USA, 2007. Association for Computing Machinery.

[124] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[125] Mathias Payer Sirus Shahini, Mu Zhang and Robert Ricci. Arvin: Greybox fuzzing using approximate dynamic cfg analysis. In *The 18th ACM ASIA Conference on Computer and Communications Security*, 2023.

[126] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijin Volckaert, Per Larsen, and Michael Franz. Sok: Sanitizing for security. In *2019 IEEE Symposium on Security and Privacy*, Los Alamitos, CA, USA, may 2019.

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

[128] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for java. *ACM SIGPLAN Notices*, 41(6):387–400, 2006.

[129] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, 1996.

[130] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266. ACM, 2016.

[131] Yulei Sui and Jingling Xue. Svf: Interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction*, page 265–266, 2016.

[132] Richárd Szalay and Zoltán Porkoláb. Flexible semi-automatic support for type migration of primitives for c/c++ programs. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 878–889. IEEE, 2022.

[133] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 48–62, Washington, DC, USA, 2013. IEEE Computer Society.

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

[135] Yoshiki Takashima, Ruben Martins, Limin Jia, and Corina S. Păsăreanu. Syrust: automatic testing of rust libraries with semantic-aware program synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 899–913, New York, NY, USA, 2021. Association for Computing Machinery.

[136] Adrian Taylor, Andrew Whalley, Dana Jansens, and Nasko Oskov. An update on memory safety in chrome. `https://security.googleblog.com/2021/09/an-update-on-memory-safety-in-chrome.html`, 2021.

[137] The Linux man-pages Project. time(1) — linux manual page, 2025.

[138] The LLVM Project. LLVM language reference manual. `https://llvm.org/docs/LangRef.html`.

[139] The Rust Project Developers. The rust standard library, 2025.

[140] The Rust Project Developers. Scoping rules. `https://doc.rust-lang.org/rust-by-example/scope.html`, 2025.

[141] The Rust Project Developers. tracing, 2025.

[142] The Rust Project Developers. vm-memory, 2025.

[143] The Servo Project Developers. Servo: The parallel browser engine project. `https://servo.org/`, 2019.

[144] The White House. Press release: Future software should be memory safe. `https://bidenwhitehouse.archives.gov/oncd/briefing-room/2024/02/26/press-release-technical-report/`, 2024.

[145] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing Forward-Edge Control-Flow integrity in GCC & LLVM. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 941–955, San Diego, CA, August 2014. USENIX Association.

[146] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *usenix-security*, 2014.

[147] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. Chopped symbolic execution. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 350–360, New York, NY, USA, 2018. Association for Computing Machinery.

[148] Erik Van Der Kouwe, Taddeus Kroes, Chris Ouwehand, Herbert Bos, and Cristiano Giuffrida. Type-after-type: Practical and complete type-safe memory reuse. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 17–27, 2018.

[149] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive cfi. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 927–940, New York, NY, USA, 2015. Association for Computing Machinery.

[150] Victor van der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 934–953, 2016.

[151] Matthew Van Gundy and Hao Chen. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *Network and Distributed System Security (NDSS) Symposium*, 2009.

[152] Emanuel Vintila, Philipp Zieris, and Julian Horsch. Evaluating the Effectiveness of Memory Safety Sanitizers . In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 88–88, Los Alamitos, CA, USA, May 2025. IEEE Computer Society.

[153] Dennis Volpano and Geoffrey Smith. A type-based approach to program security. In *TAPSOFT'97: Theory and Practice of Software Development: 7th International Joint Conference CAAP/FASE Lille, France, April 14–18, 1997 Proceedings 22*, pages 607–621. Springer, 1997.

[154] Lei Wang, Qiang Zhang, and PengChao Zhao. Automated detection of code vulnerabilities based on program analysis and model checking. In *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 165–173, 2008.

[155] Lian Kit Wee. Here comes the wave of insurance claims for the crowdstrike outage, 2024.

[156] Whole program LLVM. `https://github.com/travitch/whole-program-llvm`, 2023.

[157] Wikipedia contributors. Breadth-first search — Wikipedia, the free encyclopedia, 2023. [Online; accessed 7-October-2023].

[158] Wikipedia Contributors. 2024 crowdstrike-related it outages, 2025.

[159] Wikipedia contributors. Virtual method table — Wikipedia, the free encyclopedia, 2025. [Online; accessed 13-August-2025].

[160] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. *SIGPLAN Not.*, 30(6):1–12, jun 1995.

[161] Secure Code working group. Rustsec advisory database, 2025.

[162] A.K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[163] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. KEPLER: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1187–1204, Santa Clara, CA, August 2019. USENIX Association.

[164] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 662–676, 2017.

[165] Tianrou Xia, Hong Hu, and Dinghao Wu. DeepType: Refining indirect call targets with strong multi-layer type analysis. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 5877–5894, 2024.

[166] Tianrou Xia, Kaiming Huang, Dongyeon Yu, Yuseok Jeon, Jie Zhou, Dinghao Wu, and Taegyu Kim. LiteRSan: Lightweight memory safety via Rust-specific program analysis and selective instrumentation. *arXiv:2509.16389*, 2025.

[167] Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael R. Lyu. Memory-safety challenge considered solved? an in-depth study with all rust cves. *ACM Trans. Softw. Eng. Methodol.*, 31(1), sep 2021.

[168] Jun Xu, Dongliang Mu, Ping Chen, Xinyu Xing, Pei Wang, and Peng Liu. Credal: Towards locating a memory corruption vulnerability with your core dump. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 529–540, 2016.

[169] Jun Xu, Dongliang Mu, Xinyu Xing, Peng Liu, Ping Chen, and Bing Mao. Postmortem program analysis with hardware-enhanced post-crash artifacts. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 17–32, 2017.

[170] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. Precise and scalable detection of double-fetch bugs in os kernels. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 661–678, 2018.

[171] Zhiwu Xu, Bohao Wu, Cheng Wen, Bin Zhang, Shengchao Qin, and Mengda He. Rpg: Rust library fuzzing with pool-based fuzz target generation and generic support. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[172] Yutian Yang, Wenbo Shen, Xun Xie, Kangjie Lu, Mingsen Wang, Tianyu Zhou, Chenggang Qin, Wang Yu, and Kui Ren. Making memory account accountable: Analyzing and detecting memory missing-account bugs for container platforms. In *Proceedings of the 38th Annual Computer Security Applications Conference*, ACSAC '22, page 869–880, New York, NY, USA, 2022. Association for Computing Machinery.

[173] Yizhuo Zhai, Yu Hao, Hang Zhang, Daimeng Wang, Chengyu Song, Zhiyun Qian, Mohsen Lesani, Srikanth V. Krishnamurthy, and Paul Yu. Ubitect: A precise and scalable method to detect use-before-initialization bugs in linux kernel. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 221–232, New York, NY, USA, 2020. Association for Computing Machinery.

[174] Chao Zhang, Dawn Song, Scott A. Carr, Mathias Payer, Tongxin Li, Yu Ding, and Chengyu Song. Vtrust: Regaining trust on virtual calls. In *Network and Distributed System Security (NDSS) Symposium*. The Internet Society, 2016.

[175] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *2013 IEEE Symposium on Security and Privacy*, pages 559–573, 2013.

[176] Jinquan Zhang, Pei Wang, and Dinghao Wu. Libsteal: Model extraction attack towards deep learning compilers by reversing dnn binary library. In *18th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2023)*, pages 283–292, 2023.

[177] Mingwei Zhang and R. Sekar. Control flow integrity for COTS binaries. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 337–352, Washington, D.C., August 2013. USENIX Association.

[178] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M. Azab, and Ruowen Wang. PeX: A permission check analysis framework for linux kernel. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1205–1220, Santa Clara, CA, August 2019. USENIX Association.

[179] Yuchen Zhang, Yunhang Zhang, Georgios Portokalidis, and Jun Xu. Towards understanding the runtime performance of rust. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery.

Vita

Tianrou Xia

EDUCATION

Doctor of Philosophy, Informatics 2020–2025
The Pennsylvania State University

Master of Science, Computer Science 2018–2020
The Pennsylvania State University

Bachelor of Engineering, Information Security 2014–2018
Northeastern University

WORK EXPERIENCE

Teaching Assistant, The Pennsylvania State University

- SRA 221 - Information Security: Spring 2025
- IST 454 - Computer and Cyber Forensics: Spring 2023, Spring 2024
- CYBER 100 - Computer Systems Literacy: Fall 2023
- IST 597 - Fairness, Incentives, and Mechanism Design: Fall 2022
- DS 402 - Games, Algorithms, and Social Choice: Spring 2022, Fall 2022
- SRA 268 - Visual Analytics: Fall 2020

Research Assistant, Prof. Dinghao Wu's Group

- Jan 2021 – Dec 2021
- May 2022 - Aug 2022
- May 2024 - Dec 2024

Guest Lecture, The Pennsylvania State University

- IST 543 - Software Security: Fall 2025

PUBLICATION (As of Oct 2025)

T. Xia, K. Huang, D. Yu, Y. Jeon, J. Zhou, D. Wu, T. Kim. *LiteRSan: Lightweight Memory Safety Via Rust-specific Program Analysis and Selective Instrumentation*, arXiv, 2025.

T. Xia, H. Hu, D. Wu. *DeepType: Refining Indirect Call Targets with Strong Multi-layer Type Analysis*, USENIX Security Symposium, 2024.

T. Xia, Y. Sun, S. Zhu, Z. Rasheed, K. Shafique. *Toward A Network-Assisted Approach for Effective Ransomware Detection*, EAI Endorsed Transactions on Security and Safety, 2021.