FASTER AND MORE PRECISE POINTER ANALYSIS ALGORITHMS FOR AUTOMATIC BUG DETECTION

A Dissertation

by

PEIMING LIU

Submitted to the Graduate and Professional School of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Jeff Huang
Committee Members,	Guofei Gu
	Jiang Hu
	Riccardo Bettati
Head of Department,	Scott Schaefer

May 2022

Major Subject: Computer Science

ABSTRACT

Pointer Analysis is a fundamental technique with enormous applications, such as value-flow analysis, bug detection, etc. It is also a prerequisite of many compiler optimizations. However, despite decades of research, the scalability and precision of pointer analysis remain to be an open question. In this dissertation, I introduce my research effort to apply pointer analysis to detect vulnerabilities in software and more importantly, to design and implement a faster and more precise pointer analysis algorithm.

In this dissertation, I present my works on improving both the precision and the performance of inclusion-based pointer analysis. I proposed two fundamental algorithms, origin-sensitive pointer analysis and partial update solver (PUS), and show their practicality by building two tools, O2 and XRust, on top of them. Origin-sensitive pointer analysis unifies widely-used concurrent programming models: events and threads, and analyzes data sharing (which is essential for static data race detection) with thread/event spawning sites as the context. PUS, a new solving algorithm for inclusion-based pointer analysis, advances the state-of-the-art by operating on a small subgraph of the entire points-to constraint graph at each iteration while still guaranteeing correctness. Our experimental results show that PUS is 2x faster in solving context-insensitive points-to constraints and 7x faster in solving context-sensitive constraints. Meanwhile, the tool, O2, that is backed by origin-sensitive pointer analysis was able to detect many previously unknown data races in real-world applications including Linux, Redis, memcached, etc; XRust can also isolate memory errors in unsafe Rust from safe Rust utilizing data sharing information computed by pointer analysis with negligible overhead.

DEDICATION

To my grandmother, Li Li.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Professor Jeff Huang (advisor), Guofei Gu and Riccardo Bettati of the Department of Computer Science and Engineering, and Professor Jiang Hu of the Department of Electrical and Computer Engineering.

The work presented in Chapter 4 is co-first authored with Bozhen Liu, all other work conducted for the dissertation was completed by the student independently.

Funding Sources

Graduate study was supported by either Graduate Research Assistant or Graduate Teaching Assistant from the Department of Computer Science, Texas A&M University.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1 Overview

Pointer alias analysis is a fundamental technique in an enormous amount of static analyzers, such as value-flow analyses [4, 5, 6], deep bug detectors [7, 8, 9, 10], memory leak detectors [11, 12, 13, 14], etc. It is also the prerequisite of many compiler optimizations such as loop optimization and dead code elimination. On one hand, having a precise pointer analysis can usually dramatically eliminates the false positives reported by the static analyzer and/or strengthen the optimization performed by the compiler to generate more efficient code; on the other hand, a precise (context- and field-sensitive) pointer analysis are infamous for its poor scalability.

Although pointer analysis has been a focus of research for decades, it remains an open challenge to scale pointer analysis to large complex codebases. A crucial performance bottleneck is in solving the pointer analysis constraints. While precise pointer analysis is known to be undecidable [15, 16], any practical solution must over-approximate the exact answer. A state-of-the-art approach is the Andersen-style [17], *a.k.a. inclusion-based pointer analysis*, in which pointer assignments are constrained by inclusive relations. For example, a simple assignment $q = p$ from pointer p to q produces the contraint $pts(p) \subseteq pts(q)$, meaning that the points-to set of p, denoted as $pts(p)$, is a subset of points-to set of q. For a complex assignment involving pointer dereference, $q = *p$, it produces $\forall v \in pts(p) : pts(v) \subseteq pts(q)$. These inclusive constraints, while ensuring valid may-alias results, provide significantly higher precision than unification-based approaches (e.g., Steensgaard-style [18]).

In this dissertation, I present my work on improving both *precision* and *performance* of inclusion-based pointer analysis. The building stones are two new algorithms: 1) the origin-sensitive pointer analysis and 2) the partial update solver for inclusion-based pointer analysis. Origin-sensitive pointer analysis provides a new type of context abstraction that is much more precise when analyzing data sharing information between different components of the programs. Partial update solver

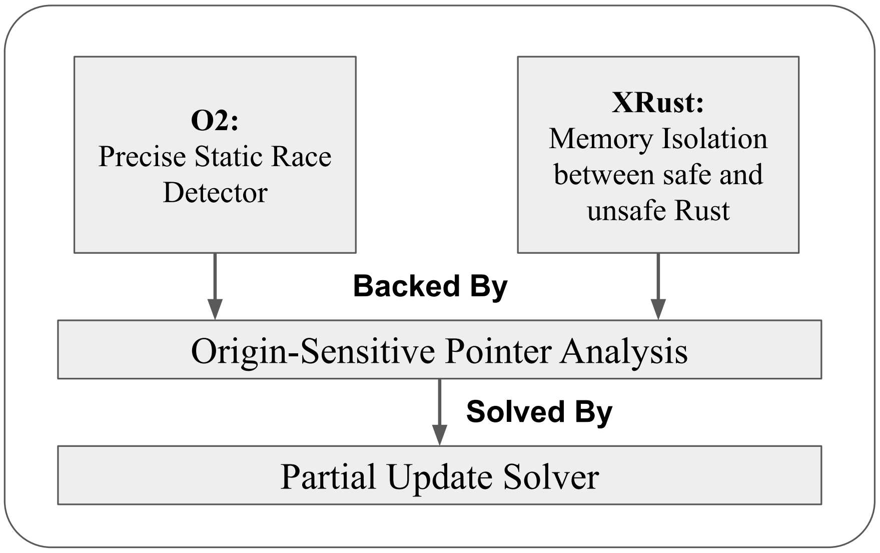

O2:
Precise Static Race Detector
XRust:
Memory Isolation between safe and unsafe Rust
Backed By
Origin-Sensitive Pointer Analysis
Solved By
Partial Update Solver

Figure 1.1: Connections between different tools/algorithms proposed in the dissertation

is a faster solving algorithm for inclusion-based pointer analysis that is more than 2x faster and 7x faster than the state-of-the-art when analyzing context-insensitive and context-sensitive constraints respectively. As shown in Figure 1.1, on top of these two fundamental techniques, we are able to build two applications: O2 and XRust to illustrate the strength of the new techniques. O2 is a static race detector that finds tens of previously unknown bugs in large complex systems including Linux kernel and XRust is a memory protection technique that isolates the memory errors in unsafe Rust with negligible overhead.

1.2 Partial Update Solver for Inclusion-based Pointer Analysis.

To mitigate the performance issue of inclusion-based pointer analysis, I present the partial update solver (PUS), Unlike previous algorithms, PUS only processes a *partial* constraint graph in each iteration, yet still guarantees the same *global* fixed point. The key insight behind our approach is that during the constraint solving process in each iteration, only a very small *causality subgraph* is subject to change due to the updates made in previous iterations. With the causality subgraph, PUS prunes the constraint graph to only operate on a small subset of the constraints in each iteration, which eliminates redundant computation across iterations, resulting in a much faster algorithm. Compared to prior approaches [19, 20, 10] that apply general graph processing techniques to pointer analysis, PUS is more efficient because it leverages two unique properties of pointer analysis. First, the sparsity of the constraint graph, which leads to our definition of causality subgraph. Second, the interconnections between different solving iterations provide the necessary information to minimize the set of causal constraints in the next iteration. Our extensive experiments show that PUS is more than $7\times$ faster than the state-of-the-art WP (Wave Propagation) and DP (Deep Propagation) algorithms [19] in solving context-sensitive pointer analysis, and more than $2\times$ faster in solving context-insensitive pointer analysis.

1.3 Origin-Sensitive Pointer Analysis

Most of the inter-procedural static analyses can be implemented in a context-sensitive way to improve precision, pointer analysis is no exception. To be more concrete, the context used by static analysis describes the calling environment when the analyzed function could be invoked. By analyzing the same function separately in different contexts, context-sensitive static analysis is able to infer the property of the target program more precisely. Especially for pointer analysis, there are two most commonly used contexts by existing works, they are, namely, callsite- and object-sensitive pointer analysis. Callsite-sensitive pointer analysis static computes the calling stack (usually up to a fixed k depth for scalability) to distinguish different calling contexts for the same function; object-sensitive pointer analysis uses the receiver object as the context and is typically used for object-oriented programming languages. While these two general abstractions might provide sufficient precision for certain types of tasks, they are not suitable for static data race detection as they fail to provide accurate thread-sharing information to identify concurrent data accesses (as discussed in Chapter 4).

Thus, we present origin-sensitive analysis, which instead uses *origin* as the context to analyze the same function under different concurrent executions. Origin, at a high level, represents a set of logical components of the program (for data race detection, it can threads/event handlers that are triggered asynchronously). By analyzing the information at the origin level instead of at the function level, origin-sensitive pointer analysis is more precise to analyze the connection between different components in the same program.

1.4 O2 and XRust

O2: Data races are among the worst bugs in software in that they exhibit non-deterministic symptoms and are notoriously difficult to detect. By utilizing the power of origin-sensitive pointer analysis, O2 abstracts different threads in the program into origins and precisely reasons the thread-sharing information between different threads. The experimental results provide strong evidence on the practicality of origin-sensitive pointer analysis by finding tens of previously unknown races in large complex systems.

XRust: Rust [21] is a rising language that tries to bridge the gap between memory safety and low-level systems programming. With new language features such as ownership, borrowing, and lifetime, Rust aims to guarantee that a program is memory safe if it could be compiled (in the absence of *unsafe Rust* code). The type system of Rust and its encapsulation of low-level operations have been formally proved to ensure memory safety [22, 23]. Despite the memory safety guarantees provided by Rust, the existence of *unsafe Rust* opens a security hole to the language. Unsafe Rust escapes from Rust's static checks [24]. By using it, programmers are able to manipulate raw pointers, perform unprotected type casting and other dangerous operations just like in C/C++. Unsafe Rust is needed, however, because (1) by nature, static analysis is conservative and will reject valid programs and (2) the underlying computer hardware is inherently unsafe and certain operations could not be done with safe Rust [25].

To mitigate the issue, I present XRust, which utilizes pointer analysis to analyze the set objects that could be potentially used by unsafe Rust (*unsafe objects*). Then by enforcing that unsafe objects are allocated separately with safe objects, XRust ensures that safe objects (protected by Rust's type system) will not be corrupted even when there are memory errors in unsafe Rust. Our experimental results show that XRust incurs only 0.15% median overhead on tested crates (2.8% on Rust standard libraries) and it effectively defends against attacks that exploit known real-world memory vulnerabilities in Rust.

1.5 Roadmap

The remainder of this dissertation is organized as follows. Chapter 2 introduces the background knowledge and the prior work on pointer analysis and data race detection. Chapter 3 addresses the scalability issue of inclusion-based pointer analysis by introducing partial update solver. Chapter 4 introduces origin-sensitive pointer analysis *together with* the static data race detector, O2, which is implemented on top of it. Chapter 5 introduces our work on applying pointer analysis to secure Rust programs due to the incorrect use of unsafe Rust. Last, Chapter 6 concludes the thesis and discusses the future work.

2. BACKGROUND AND RELATED WORKS

In the chapter, I introduce the necessary background needed for the dissertation as well as related works. I first introduce different types of pointer analysis and the previous attempts to improve its precision/performance. I also introduce data races as well as existing works on data race detection, where the origin-sensitive pointer analysis is applied.

2.1 Pointer Analysis

Pointer (points-to) analysis, which computes the set of potential objects that a pointer can point to during execution, is the prerequisite for many inter-procedural program analyses, with an enormous amount of applications in value-flow techniques [4, 5, 6], deep bug detectors [7, 8, 9, 10], memory leak detectors [11, 12, 13, 14], etc. It is also crucial for many compiler optimizations such as dead code elimination and loop optimization. Roughly, the problem can be modeled and solved in two ways by inclusion-based (Andersen-style) [17] or unification-based (Steensgaard-style) [18] pointer analysis. Each style abstracts programs into different types of constraints and computes the result by computing the minimal fixed point that satisfies all the constraints. Inclusion-based approach abstracts program according to the rules defined in Table 2.1, while unification-based approach abstracts program according to the rules defined in Table 2.2. In this dissertation, all the contributions are made on inclusion-based pointer analysis as it is more precise and more widely adopted.

2.1.1 Context-Sensitive Pointer Analysis

Like many other inter-procedural static analyses, pointer analysis can be implemented in both *context-sensitive* or *context-insensitive* ways. Context-sensitivity infers the property (the points-to set) of a particular variable with context information, the analysis only collapses information of the same variable over the possible executions that result in the same static context, while maintaining different copies for different contexts [26].

Two of the main types of contexts being studied are callsite- [27] and object-sensitivity [28].

Table 2.1: Constraints for inclusion-based pointer analysis

Category	Type	Statement	Constraints
Base	Address Taken	$v_1 \leftarrow \&o$	$loc(o)^1 \in pts(v_1)$
Simple	Assignment	$v_1 \leftarrow v_2$	$pts(v_1) \supseteq pts(v_2)$
Complex	Load	$v_1 \leftarrow *v_2$	$\forall v \in pts(v_2) : pts(v_1) \supseteq pts(v)$
Complex	Store	$*v_1 \leftarrow v_2$	$\forall v \in pts(v_1) : pts(v) \supseteq pts(v_2)$

^{1} $loc(o)$ denotes the memory location of object o.

Table 2.2: Constraints for unification-based pointer analysis

Category	Type	Statement	Constraints
Base	Address Taken	$v_1 \leftarrow \&o$	$loc(o)^1 \in pts(v_1)$
Simple	Assignment	$v_1 \leftarrow v_2$	$pts(v_1) = pts(v_2)$
Complex	Load	$v_1 \leftarrow *v_2$	$\forall v \in pts(v_2) : pts(v_1) = pts(v)$
Complex	Store	$*v_1 \leftarrow v_2$	$\forall v \in pts(v_1) : pts(v) = pts(v_2)$

^{1} $loc(o)$ denotes the memory location of object o.

Callsite-sensitivity is probably the oldest and best-known type of context sensitivity. It is also intuitive as it uses a sequence of call sites as the context. The sequence of the call sites, when analyzed statically, simulates the potential call stack when the target function is invoked during execution. Typically, the length of the call sites sequence is limited to a constant k for scalability. On the contrary, object-sensitivity is introduced for object-oriented programming languages such as Java, which uses object allocation sites as contexts. Specifically, the analysis qualifies a method's local variables with the allocation site of the receiver object of the method call. This kind of context information is non-local: it cannot be gathered by simple inspection of the call site, since it depends on what the analysis itself has computed to be the receiver object [26]. In most cases, object-sensitivity is considered to be superior when analyzing object-oriented programs [28]. The reason is that the constant k-limiting imposed on callsite sensitivity always leads to precision loss and different contexts will be merged when the length of the call sequence exceeds k, while object-sensitivity can tolerate it as long as the receiver object remains the same. Origin-sensitive is similar to object-sensitivity: it analyzes functions invoked in different origins (threads/events) separately. But it is more general and can be applied to imperative language such as C. Recently, selective context-sensitive techniques [29, 30, 31, 32, 33] have also been proposed. Although much progress has been made, context-sensitive pointer analysis remains difficult to scale.

Developing different algorithms and abstractions for pointer analysis has been researched for decades. For C programs, Steensgaard [18] proposed the first scalable pointer analysis based on a unification-based algorithm. Das et al. then extended the unification-based approach to include one level of context sensitivity [34]. For C program, Fahndrich et al [35] proposed an algorithm scale to 200K-line programs in field-insensitive ways, the algorithm computes the call graph on-the-fly. Whaley et al [36] proposed another context-sensitive pointer analysis by generating multiple instances of a method for every distinct calling context to prevent information from one context to flow to another. The approach uses the inclusion-based algorithm. It first computes a conservative call graph using context insensitive analysis and then using the context insensitive call graph to compute context-sensitive results. For Jave programs, Ruf et al [37] presented a summary-based approach to model context sensitivity (based on the unification-based algorithm) in the context of a specialized algorithm for synchronization, which requires a bottom-up traversal of the callgraph.

2.1.2 Pointer Analysis Solving Algorithms

Across the decades, Andersen's inclusion-based pointer analysis has emerged as the most popular pointer analysis [17]. Many works have been proposed to improve Andersen's analysis. Most of the previous research abstracts pointer analysis as a constraint graph and propagates the points-to information until a global fixed point. Heintze et al. [38] introduced a way to avoid the cost of computing the full transitive closure of the constraint graph. Instead, a dynamic transitive closure is computed on-demand and graph reachability queries are used to resolve points-to sets. As a result, cycle detection is achieved essentially for free as a result of the graph reachability queries. However, this technique also introduces the potential for redundant work across reachability queries. Later works [39, 40, 20] topologically sort the constraint graph to reduce redundant points-to set propagation. Pereira et al. [19] proposed a new constraint solving algorithm, wave propagation, by separating the algorithm into three phases; collapsing of cycles, points-to propagation and insertion of new edges. These three phases are performed as a wave and repeated until a fixed point is reached. PUS advances the state-of-the-art by performing SCC detection and points-to set propagation on the *causality subgraph*, thus avoiding redundant computation in each iteration.

As the difficulty in developing an efficient constraint solving algorithm remains, researchers recently turned their attention to tackle the problem at new angles. D4 [7] first introduced an incremental algorithm for inclusion-based pointer analysis to enable differential pointer analysis on code changes. The algorithm of D4 is orthogonal to ours and PUS can effectively integrate with D4 to speed up its bootstrapping constraint solving process. DEA [41] introduced a faster algorithm to deal with positive-weight cycle in field-sensitive pointer analysis, while it still relies on wave propagation to compute the fixed point.

Another line of research formulates pointer analysis as a CFL-reachability problem. Reps et al. [42] modeled the flow-insensitive pointer analysis into a CFL-reachability problem. Spath et al. [43] proposed a flow- and context-sensitive demand-driven pointer analysis that models the pointer analysis as an IFDS problem (which then can be solved by CFL-reachability). This line of research is orthogonal to PUS and details are omitted.

Graph simplification techniques can be applied to both constraint-based and CFL-reachability-based approaches to improve their scalability. Fahndrich et al. [44] first showed that collapsing SCC components in the constraint graph can significantly improve the performance of inclusion-based pointer analysis. Pearce et al. [39] introduced an algorithm for online cycle detection. By keeping the constraint graph topologically sorted, cycle detection need only be run when a new edge violates the existing topological ordering. Detecting cycles upon edge insertion proved to be too costly, and so Pearce et al. [40] introduced an efficient field-sensitive PTA that occasionally checks for and collapses cycles in the constraint graph. Hardekopf et al. [20] introduced Lazy Cycle Detection (LCD) and Hybrid Cycle Detection (HCD). LCD reduces runtime overhead even further by selectively triggering cycle detection only when identical points-to sets are discovered during transitive closure computation. HCD introduces an offline linear-time graph preprocessing stage that allows the online pointer analysis to detect cycles without the need for graph traversal at all. PUS extends the above techniques by not only applying general graph optimization techniques but also leveraging unique properties of constraint graph to only perform the SCC detection on causality subgraph. Thus, PUS can dynamically prune off most ineffective edges to avoid redundant points-to set propagation.

Recent work by Li et al. [45] proposed to simplify the input labeled graph in a CFL-reachability problem by eliminating useless graph edges. PUS is similar to this work from a very high level. However, this work primarily optimizes the labeled graph in CFL-reachability problems while PUS focuses on simplifying the constraint graphs in pointer analysis.

Besides improving the solving algorithm, researchers have also proposed to use Datalog [46] for fast and easy pointer analysis modeling. While the experimental result indicates a great potential along the direction, fully customized pointer analysis solvers are still desired and used by many of the most recent works [7, 6, 4] as they are easier to be extended and tailored for different needs.

2.2 Data Races and Data Race Detection

In multithreaded programs, a data race occurs when 1. two or more threads in a single process access the same memory location concurrently, and 2. at least one of the accesses is for writing, and 3. the threads are not using any exclusive locks to control their accesses to that memory. Data races are among the worst bugs in software in that they exhibit non-deterministic symptoms and are notoriously difficult to detect. The problem is exacerbated by interactions between threads and events in real-world software.

Due to its non-deterministic nature, race detection has been considered as an important research topic for decades. Both static and dynamic algorithm has been proposed to tackle the problem. One of the key challenges is how to compute and represent *Happens-Before* Relationships. Offset-span labeling [47], which is an online scheme that labels threads in a fork-join graph, labels each task with a vector of tuples. Vector Clock [48] records a clock for each thread in the system, and the virtual clock is increased upon every synchronization event. Two events are considered to be parallel if the two vector clock are not ordered. Flanagan et al [49] improve the vector clock algorithm by replacing heavyweight vector clocks with adaptive lightweight representation as they find the full generality of vector clocks is unnecessary in most cases.

Dynamic Race Detection Tools. Google's Thread Sanitizer [50], also known as TSAN, proposed a hybrid algorithm that uses both happens-before and lockset to detect data races. TSAN has been used to find hundreds of races in real-world applications. Helgrind [51] is a tool based on Valgrind [52]. Helgrind only detects happens-before relationships and it supports a subset o the dynamic annotations in TSAN. Intel's Inspector [53] is another dynamic data race detection tool that uses Intel PT [54] to trace the program. It uses a Concurrent Provenance Graph to record control, data and schedule dependencies.

Static Race Detection Tools. RacerD, developed at Facebook, is by far the most successful static race detector [55]. It is regularly applied to Android apps in Facebook and has flagged over 2500 issues that have been fixed by developers before reaching production [55]. RacerD's design favors reducing false positives over false negatives through clever syntactical reasoning, but it does not reason about pointers and thus can miss races due to pointer aliases. In contrast, O2 deals with both Java pointers and low-level pointers in C/C++ such as indirect function targets and virtual tables. Other classic static race detection tools (e.g., RacerX [56], RELAY [57]) have various difficulties when applied to modern software. RacerX contains many heuristics and engineering decisions, which are difficult to duplicate. RELAY depends on the CIL compiler front-end, which supports only a subset of C and has not been actively developed. Technically, RELAY uses a context- and field-insensitive pointer analysis, a major source of false positives. String-pattern-based heuristics are used in RELAY to filter out false aliasing. These heuristics are effective in reducing false positives, but are only specific to the code conventions in the target program and are unsound.

3. PARTIAL UPDATE SOLVER FOR INCLUSION-BASED POINTER ANALYSIS*

Although pointer analysis has been a focus of research for decades, it remains an open challenge to scale pointer analysis to large complex codebases. A crucial performance bottleneck is in solving the pointer analysis constraints. While precise pointer analysis is known to be undecidable [15, 16], any practical solution must over-approximate the exact answer. A state-of-the-art approach is the Andersen-style [17], *inclusion-based pointer analysis*, in which pointer assignments are constrained by inclusive relations. For example, a simple assignment $q = p$ from pointer p to q produces the contraint $pts(p) \subseteq pts(q)$, meaning that the points-to set of p, denoted as $pts(p)$, is included in the points-to set of q. For a complex assignment involving pointer dereference, $q = *p$, it produces $\forall v \in pts(p) : pts(v) \subseteq pts(q)$. These inclusive constraints, while ensuring valid may-alias results, provide significantly higher precision than unification-based approaches (i.e., , Steensgaard-style [18]). While the origin-sensitive pointer analysis introduced in Chapter 4 mitigates the performance problem of pointer analysis to some degree, it still suffers from the poor scalability inherent in inclusion-based pointer analysis.

As real-world programs often produce a huge number of constraints, quadratic to the number of pointers, the key challenge remained is how to efficiently solve these constraints. There was a significant effort over a decade ago by Pereira, Hardekopf, Pearce [19, 20, 40]. In their work, a naïve fixed-point algorithm is improved by separating complex constraints and propagating the points-to information into two stages; by applying different strongly connected component (SCC) detection strategies, e.g., lazy cycle detection and hybrid cycle detection, to reduce the size of the constraint graph [20]; or by sorting the constraint graph topologically to avoid redundant computation [40, 20]. More recently, Lei et al. [41] propose an efficient algorithm (DEA) for handling positive weight cycles in field-sensitive pointer analysis. Liu et al. [7] propose an incremental pointer analysis (D4) that only analyzes the updated code changes to dodge the performance over-

*Reprinted with permission from "PUS: A Fast and Highly Efficient Solver for Inclusion-based Pointer Analysis" by Liu, Peiming and Li, Yanze and Swain, Brad, and Huang, Jeff, 2022. Proceedings of the ACM/IEEE 44th International Conference on Software Engineering, Copyright 2022 by Peiming Liu.

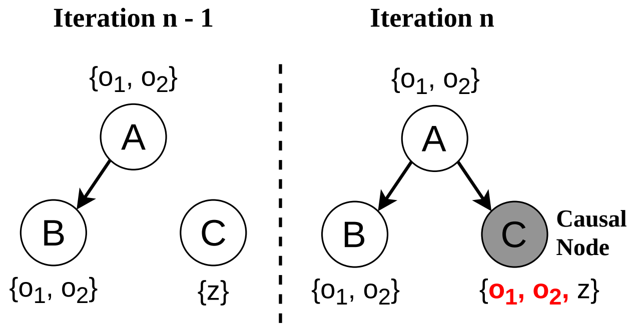


Figure 3.1: An example to illustrate the causality subgraph: with a new edge inserted after iteration $n - 1$, node C is identified as a causal node in iteration n. Reprinted From [1].

head introduced by a whole-program pointer analysis. While these approaches further improve the state-of-the-art in some specific aspects, their fundamental solving algorithm remains the same (e.g., DEA still relies on WP [19] to solve the constraints).

We tackle this challenge with a new fundamental solving algorithm. Unlike previous algorithms, our new algorithm, *Partial Update Solver* (PUS), only processes a *partial* constraint graph in each iteration, yet still guarantees the same *global* fixed point. The key insight behind our approach is that during the constraint solving process in each iteration, only a very small *causality subgraph* is subject to change due to the updates made in previous iterations. With the causality subgraph, PUS prunes the constraint graph to only operate on a small subset of the constraints in each iteration, which eliminates redundant computation across iterations, resulting in a much faster algorithm. Compared to prior approaches [19, 20, 10] that apply general graph processing techniques to pointer analysis, PUS is more efficient because it leverages two unique properties of pointer analysis. First, the sparsity of the constraint graph, which leads to our definition of causality subgraph. Second, the interconnections between different solving iterations provide the necessary information to minimize the set of causal constraints in the next iteration.

As illustrated in Fig. 3.1, suppose a new edge $A \rightarrow C$ is inserted in the previous iteration (due to complex constraints), C is identified as a *causal* node because the points-to information carried by A will *take effect* on C in the current iteration in order to satisfy the inclusive constraints.

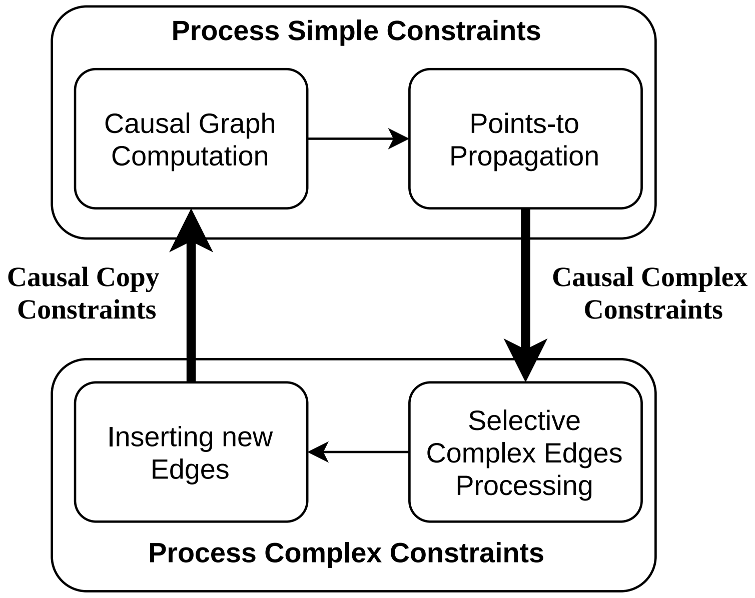

Process Simple Constraints
Causal Graph Computation
Points-to Propagation
Causal Copy Constraints
Causal Complex Constraints
Inserting new Edges
Selective Complex Edges Processing
Process Complex Constraints

Figure 3.2: An overview of PUS: partial update solver. Reprinted From [1].

However, B is not a *causal* node because its points-to information is not affected by the new edge. Our empirical results show that, on average, the causality subgraph includes less than *4%* of the nodes and edges in the full constraint graph, indicating a dramatic performance optimization opportunity.

Fig. 3.2 shows an overview of PUS. At a high level, PUS adopts a similar workflow to the existing *two-phase* constraint solving algorithms [19], in which the constraints are processed iteratively between two stages (for processing simple constraints and complex constraints, respectively). However, unlike the existing algorithms, which repeat the computation over the whole graph in each iteration, PUS *interactively* invokes the two processing phases such that the first phase computes a causality subgraph and selectively propagates the points-to information within the causality subgraph (based on the information provided by the second phase). Meanwhile, as new points-to information is propagated, the first phase also collects a subset of all the complex constraints to be processed in the second phase. In PUS, in each iteration, one phase provides necessary information for the other to infer a small set of causal constraints to be processed.

In principle, the time complexity of Andersen-stype pointer analysis is bounded by $O(N^2 max_x D(x)+$

NE) on a *k-sparse* program [58], where $max_x D(x)$ is the maximal number of statements dereferencing a pointer x, N/E is the number of nodes/edges in the constraint graph. The value of $max_x D(x)$ is bounded by a constant k (i.e., $max_x D(x) \leq k$) for real-world applications. The first portion, $O(N^2 max_x D(x))$, summarizes the complexity for handling complex constraints and the second portion, $O(NE)$, summarizes the complexity for propagating points-to information on the constraint graph. As PUS propagates points-to information only on the causality subgraph, it reduces the second portion to $O(N^2 max_x D(x) + N^* E^*)$, where N^*/E^* is the number of nodes/edges in the causality subgraph. In practice, this reduction leads to significant performance improvements because typically $N^* \ll N$ and $E^* \ll E$ in real-world programs.

3.1 Inclusion-based Pointer Analysis

The inter-procedural inclusion-based pointer analysis abstracts different program statements into the constraints listed in Table 2.1. It first scans the target program and generates three types of constraints: *base*, *simple* and *complex* [20]. It then abstracts the target program into a constraint graph (Definition. 2.1). Inclusion-based pointer analysis can then be solved by computing the transitive closure of the constraint graph such that for every pair of nodes $v_1, v_2 \in \mathcal{V}$, if there is an edge $e = \{v_1 \rightarrow v_2\} \in \mathcal{E}$, then $pts(v_1)$ and $pts(v_2)$ are the minimal points-to sets that ensure $pts(v_1) \subseteq pts(v_2)$.

The global fixed point is reached when all complex constraints and simple constraints are satisfied (complex constraints are satisfied by inserting new edges into the constraint graph): For each load constraint ($v_1 \leftarrow *v_2$) and every $v \in pts(v_2)$, we added a new edge $v \rightarrow v_1$ into the constraint graph; for each store constraint ($*v_1 \leftarrow v_2$) and every $v \in pts(v_1)$, we added a new edge $v_1 \rightarrow v$ into the constraint graph; and for each offset constraint ($v \leftarrow \&s.field$) and every $v \in pts(s)$, we insert $loc(v.field)$ into $pts(v)$.

Definition. 2.1: The ***constraint graph*** *(CG) of a program is an attributed graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}, pts\}$, in which $\mathcal{V}$ is a set of vertices, each of which corresponds to a variable v in the program; $\mathcal{E} \subseteq (\mathcal{V} \times \mathcal{V})$ is a set of directed edges (constraints) between vertices in $\mathcal{V}$, each of which represents a*

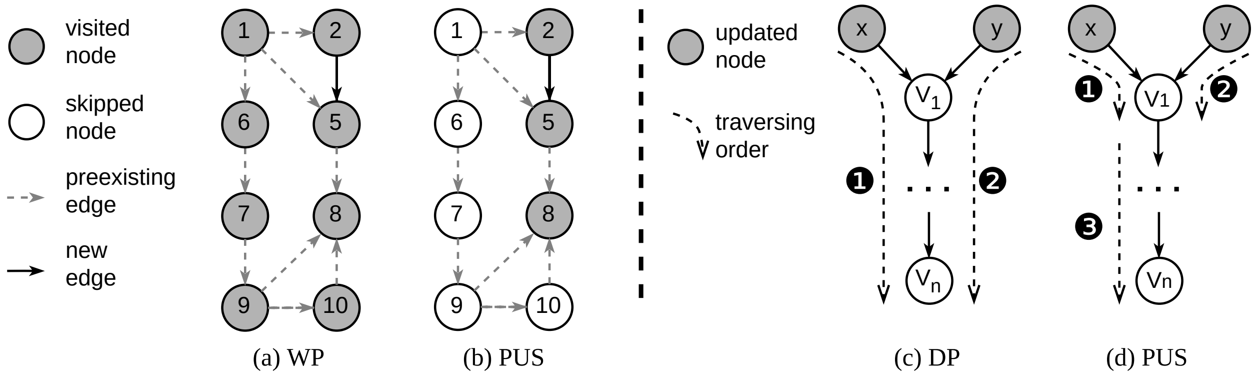


Figure 3.3: The comparison between PUS, WP and DP. (a) the solving process of WP (the entire graph need to be revisited) (b) the solving process of PUS (only marked node need to be visited) (c) the solving process of DP ($V_1 ... V_n$ are visited twice) (d) the solving process of PUS ($V_1 ... V_n$ are only visited once). Reprinted From [1].

simple constraint between two nodes (in the following text, the word 'edge' and 'constraint' are used interchangeably); and pts*:* $\mathcal{V} \rightarrow P(\mathcal{O})$ *(where* $P(\mathcal{O})$ *is the power set of the set of objects created by memory allocation operations in the program) is a function from* $v \in \mathcal{V}$ *to* $s \in P(\mathcal{O})$ *that maps a node (pointer) to its points-to set.*

3.2 Limitations of Existing Solving Algorithms

We divide the existing constraint solving algorithms for inclusion-based pointer analysis roughly into two categories and summarize their limitations as follows respectively.

Methods that process constraints in topological order: Performing a topological sorting on the constraint graph ensures that constraints are processed in the optimal order by guaranteeing that the points-to sets of all the predecessors of a node n have been updated before processing n. In this way, the points-to sets of the predecessors are the most recently updated before propagating to node n. Many algorithms [44, 39, 40, 38] adopt the topological sorting approach to boost the constraint solving time. Despite the benefits brought by it, performing SCC detection and topological sorting on large constraint graphs itself is time-consuming and could easily become a bottleneck that slows down the solving process.

Fig. 3.3 (a) and Fig. 3.3 (b) shows the solving process of WP and PUS on the example constraint graph respectively, as a new edge $(2 \rightarrow 5)$ is inserted, WP revisits the entire graph in topological order, on the other hand, PUS computes the same result by only visiting the three nodes in the causal subgraph (marked in grey).

Methods that process constraints in undetermined order: Methods that do not enforce SCC detection and topological order on the constraint graph (e.g., Deep Propagation (DP) [19], Lazy Cycle Detection (LCD) and Hybrid Cycle Detection (HCD) [20]) at each iteration unavoidably waste resources on redundant computation due to a sub-optimal order of constraint processing.

Fig. 3.3 (c) and Fig. 3.3 (d) show the solving process of DP and PUS on the example constraint graph respectively. When both $pts(x)$ and $pts(y)$ are updated, DP adopts a depth-first search to propagate from $x \rightarrow \cdots \rightarrow v_n$ and from $y \rightarrow \cdots \rightarrow v_n$ separately. As a result, the nodes and constraints between $v_1 \rightarrow \cdots \rightarrow v_n$ are visited twice. However, PUS shows that when analyzing the graph in topological order (i.e., $x \rightarrow v_1, y \rightarrow v_1$ and then $v_1 \rightarrow \cdots \rightarrow v_n$), every constraint only needs to be visited once.

The comparison between the existing two categories of algorithms reveals the dilemma of current algorithms: On one hand, full SCC detection and topological sorting are desired to eliminate redundant computation and to reduce the number of nodes by collapsing nodes in the same SCC in the constraint graph; on the other hand, applying a complete SCC detection on a large graph itself can introduce an unbearable overhead.

We found that the common problem for those works is that they all take a holistic view towards constraint graphs. Instead, PUS works on *causality subgraphs*. By only working on a small subgraph in each iteration, PUS can enjoy the benefit brought by topological sorting without introducing too much performance overhead. The rationale behind causality subgraphs are rooted in inherent properties of pointer analysis and the unique interconnection between different phases of a solving process summarized as follows:

- Constraint graphs for real programs are, by nature, *sparsely connected*. The sparsity of constraint graphs is a result of *modularization* of modern software (thus fewer connections between different modules) as well as the *locality* [2] of program statements (thus fewer connections between different statements). As constraint graphs are abstracted from programs, an update on one specific node in the constraint graph will likely only affect a limited number of neighboring nodes. Thus, in each iteration during the solving process and with limited nodes whose points-to sets are updated, only a very small subset (usually $\leq 4\%$ according to our experiments) of the nodes (casual nodes) are required to be processed, which means that topological sorting and points-to set propagation only need to be done on a small *causality subgraph* of the entire constraint graph in each iteration.

Being able to precisely infer a small subgraph in each iteration, PUS discovers another memory optimization opportunity: One of the most widely adopted optimization techniques used in existing methods and frameworks (e.g., WALA [59]) is to maintain a cached points-to set for every node in the constraint graph (Wave Propagation [19] even requires an additional cached points-to set for every edge in the graph). The cached points-to set is used to filter out *non-causal* nodes whose points-to sets do not get updated in the current iteration and to only process diffed points-to information. However, if the *causality subgraph* can be accurately inferred and most of the constraints in the subgraph are effective, i.e., by processing which, the points-to set will get updated, then the cached points-to set can be (optionally) eliminated to improve the memory efficiency without causing significant performance overhead.

3.3 Algorithm

In this section, we describe the detailed algorithm for PUS. We first present the overall structure of PUS in Algorithm 1, we then explain each component separately in detail in Algorithm 2, Algorithm 3 and Algorithm 4. For simplicity, we describe PUS under the context of *field-insensitive* pointer analysis. PUS can be extended for field-sensitive pointer analysis (as we implemented for experiments) by adding another type of constraint, the *offset* constraint, into complex constraints similar to the previous work [40].

[2] The locality here has a different meaning from the spatial/temporal locality in computer architecture. Here, it is used to explain that most of the statements in the program are irrelevant (e.g., `a++;` and `b++;`).

Algorithm 1: Partial Update Solver

```
Input : A unsolved constraint graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}, pts\}$
Result: The points-to information for every pointer in the program
1  $L_{comp} \leftarrow \varnothing$;
2  $L_{copy} \leftarrow \varnothing$;
3  for each $v \in \mathcal{V}$ do
4      if $pts(v) \neq \varnothing$ then
           // Nodes with address taken constraints have non-empty
              points-to set
5          $C_{copy} \leftarrow v$.getCopyConstraints();
           // get simple constraints started from the node and
              insert them into $L_{copy}$
6          $L_{copy}$.insert($C_{copy}$);
           // insert the node into $L_{comp}$
7          $L_{comp}$.insert($v$);
8      end
9  end
10 while $L_{copy} \neq \varnothing$ do
       // SCC detection on subgraphs of $\mathcal{G}$ based on $L_{copy}$
11     SCC Collapse and TopoSort on subgraphs of $\mathcal{G}$ (Algorithm 2);
12     $L_{comp} \leftarrow$ Partially Process Simple Constraints (Algorithm 3);
13     $L_{copy}.clear()$;
14     $L_{copy} \leftarrow$ Partially Process Complex Constraints (Algorithm 4);
15     $L_{comp}.clear()$;
16 end
```

3.3.1 Structure of the Algorithm

At a high level, PUS has a similar structure to WP [19] that separates the insertion of new constraints (handling complex constraints) from the propagation of points-to sets (handling simple constraints). However, PUS distinguishes itself by connecting the two constraint solving phases using two separate work lists:

- L_{copy}:$\{\mathcal{E}\}$ – A subset of simple constraints that is used to compute the causality subgraph used in the following stages.
- L_{comp}:$\{\mathcal{V}\}$ – A subset of nodes on which the complex constraints need to be recomputed.

At a high level, Algorithm 1 can be divided into initialization phase (from line 3 to 9), SCC detection and topological sort phase (line 11), Simple constraint processing phase (line 12) and Complex constraint processing phase (line 14), which are explained in detail in the following sections. We also relies on the following conventions to describe our algorithm: We refer to any edge $e \in L_{copy}$ used in Algorithm 1 as an **essential edge** and refer to any node $v \in L_{comp}$ used in Algorithm 1 as an **unsaturated node**. We use $dst(\mathcal{E})$ to denote the set of destination nodes for all edges $e \in \mathcal{E}$; we use $src(\mathcal{E})$ to denote the set of source nodes for all $e \in \mathcal{E}$; we use $in(n)$ to denote the set of incoming edges to node n; we use $out(n)$ to denote the set of outgoing edge from n; we use $pred(n)$, where n is a node, to denote the set of predecessor nodes of n; we use $succ(n)$ to denote the set of the successor nodes of n.

Algorithm 2: SCC Collapse and TopoSort on SubGraphs of *G*

```
Input  : A constraint graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}, pts\}$
         A list of starting edges $L_{copy} = \{\mathcal{E}\}$
Output: A toposorted vector $\mathcal{V}$ of SCCs that are reachable from at least one of $e \in L_{copy}$
1  $\mathcal{V}' \leftarrow \varnothing$;
2  $\mathcal{E}' \leftarrow \varnothing$;
3  $pts' \leftarrow pts$;
4  $\mathcal{G}' \leftarrow \{\mathcal{V}', \mathcal{E}', pts'\}$;
   // prune the graph $\mathcal{G}$ to a subgraph $\mathcal{G}'$
5  while $L_{copy} \neq \varnothing$ do
6      $e \leftarrow L_{copy}$.pop();
7      if visited($e$) then
8          continue ; // skip covered edges
9      end
10     setVisited($e$);
11     $\mathcal{E}'$.insert($e$);
       // add source and destination nodes of $e$ into $\mathcal{G}'$
12     $\mathcal{V}'$.insert($\{e.src, e.dst\}$);
13     $\mathcal{E}'$.insert($\{e' \mid e' \in v.outgoing\_edges() \wedge reachable(e.dst, v) = true\}$);
14     $\mathcal{V}'$.insert($\{v' \mid reachable(e.dst, v) = true\}$);
15 end
   // perform SCC detection on the subgraph $\mathcal{G}'$
   // also sort the graph internally
16 $\mathbb{V} \leftarrow$ Tarjan($\mathcal{G}'$);
   // return the toposorted vector $\mathbb{V}$
17 return $\mathbb{V}$;
```

3.3.2 Detailed Algorithm

In this section, we describe the detailed algorithms of all sub-components that are used in Algorithm 1.

Subgraph SCC Detection: As shown in Algorithm 2, the SCC detection is performed on the subgraph $\mathcal{G}'$ instead of the original graph $\mathcal{G}$. The set of edges and nodes in $\mathcal{G}'$ is computed according to the reachability from the constraints in L_{copy}.

The node set $\mathcal{N}'$ of $\mathcal{G}'$ consists of 1. the *source* and *destination* nodes of every constraints in L_{copy} and 2. all the nodes that are reachable for at least one of the *destination* nodes of the constraints in L_{copy} . The edge set $\mathcal{E}'$ of $\mathcal{G}'$ consists of 1. all the edges in L_{copy} and 2. all the *outgoing* edge of node n that are reachable from a least one of the *destination* nodes of the constraints in L_{copy}.

After SCC detection, a vector of nodes in topological order is returned by Algorithm 2 and used as one of the inputs for Algorithm 3. Note that although Algorithm 2 presents the computation of $\mathcal{G}'$ as a separate step, $\mathcal{G}'$ can be computed along with SCC detection utilizing the DFS traversal performed by Tarjan's algorithm internally.

The graph processed by Algorithm 3 defines the causality subgraphs in each iteration. In addition, we introduced the following definition to formally define the causality subgraph.

The graph processed by Algorithm 3 defines the causality subgraphs in each iteration. In addition, we introduced the following definition to formally define the causality subgraph.

Propagating points-to set on the causality subgraph: Algorithm 3 describes the procedure for processing simple constraints. Algorithm 3 takes a subgraph $\mathcal{G}'$ of $\mathcal{G}$ and a topologically sorted vector of nodes as the inputs. The topologically sorted vector of nodes ensures that simple constraints are processed in the optimal order to avoid redundant computation. L_{copy} is also passed in and used at line 5 to perform further pruning on the causality subgraph.

There are two important details that are worth noting in Algorithm 3:

1. During the points-to set propagation, the algorithm also computes and eventually outputs a

Algorithm 3: Partially Process Copy Constraints

```
Input  : A constraint graph: $\mathcal{G}' = \{\mathcal{V}', \mathcal{E}', pts'\}$
         A sorted vector of SCCs: $\mathbb{V} = \{\mathcal{N}\}$
         A list of effective copy constraints: $L_{copy}$
Output: A set of node $\mathcal{S}$ whose complex constraints need to be processed
1  $L_{comp} \leftarrow \varnothing$;
2  while $\mathbb{V}$.isNotEmpty() do
3      $n \leftarrow \mathbb{V}$.pop();
4      for each $e = \{src, dst\} \in n.getCopyConstraits()$ do
5          if $e = \{src, dst\} \in L_{copy} \vee src \in L_{comp}$ then
6              $changed \leftarrow$ PropagatePointsTo(src, dst);
7              if $changed$ then
8                  $L_{comp}$.insert(dst);
9              end
10         else
               // Prune the graph, skip unchanged subgraph
11             continue;
12         end
13     end
14 end
15 return $L_{comp}$;
   /* Process a simple constraint between src and dst, return
      true if the points-to information is updated            */
16 Function PropagatePointsTo($src, dst$):
17     $pts(dst) \leftarrow pts(dst) \cup pts(src)$;
18     if $dst.changed$ then
19         return true;
20     end
21     return false;
22 End Function
```

list of nodes, L_{comp}, to be used in Algorithm 4, which contains all the nodes on which the complex constraints need to be processed.

2. At line 11, the algorithm performs another pruning on the causality subgraph to further reduce the number of constraints processed by PUS.

The computation on L_{comp} is straightforward, Algorithm 3 simply inserts a node into L_{comp} if the points-to set of the node has been updated during the current iteration.

Algorithm 4: Partially Process Complex Constraints

```
   Input  : The constraint graph: $\mathcal{G} = \{\mathcal{V}, \mathcal{E}, pts\}$
            Nodes with effective complex constraints: $L_{comp}$
   Output: A set of processed: $L_{copy}$
1  while $L_{comp}$.isNotEmpty() do
2      $V \leftarrow L_{comp}$.pop();
3      for each $\{l \leftarrow *V\} \in$ V.getLoadConstraints() do
           // process load constraints
4          $newEdges \leftarrow$ processLoad($l, V$);
5          $L_{copy}$.insert($newEdges$);
6      end
7      for each $\{*V \leftarrow r\} \in$ V.getStoreConstraints() do
           // process store constraints
8          $newEdges \leftarrow$ processStore($V, l$);
9          $L_{copy}$.insert($newEdges$);
10     end
11 end
12 return $L_{copy}$;
```

The graph processed by Algorithm 3 defines the causality subgraphs in each iteration. In addition, we introduced the following definition to formally define the causality subgraph.

Definition. 4.1: *Given a constraint graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}, pts\}$ and a set of essential edges $\mathcal{E}^+ \subseteq \mathcal{E}$, the* ***essential-edge-covered graph*** *$\mathcal{G}' = \{\mathcal{V}', \mathcal{E}', pts'\}$ is a subgraph of $\mathcal{G}$, where $\mathcal{V}' = \mathcal{V}_1 \cup src(\mathcal{E}^+)$ and $\mathcal{V}_1 = \{v \mid \exists s \in dst(\mathcal{E}^+)$,v is reachable from s$\}$; $\mathcal{E}' = \mathcal{E}^+ \cup \{e \mid e = out(n) \wedge n \in \mathcal{V}_1\}$ and $pts' = pts$.*

Definition 4.2: *Given an* ***essential-edge-covered graph*** *$\mathcal{G}' = \{\mathcal{V}', \mathcal{E}', pts'\}$ and its corresponding essential edge set $\mathcal{E}^+ \subseteq \mathcal{E}'$, the set of* ***ineffective edges*** *$\mathcal{E}^-$ and the set of* ***ineffective nodes*** *$\mathcal{V}^-$ are determined dynamically during the points-to set propagation process. For node n, if $\forall p \in pred(n)$, $pts(p)$ does not get updated in the current iteration, then $n \in \mathcal{V}^-$. Similarly, $\mathcal{E}^- = \{e \mid e \in out(n) \wedge n \in \mathcal{V}^- \wedge e \notin \mathcal{E}^+\}$.*

Definition 4.3: *Given an* ***essential-edge-covered graph*** *$\mathcal{G}' = \{\mathcal{V}', \mathcal{E}', pts'\}$ and a set of ineffective edges $\mathcal{E}^-$, the* ***causality subgraph*** *$\mathcal{G}^* = \{\mathcal{V}^*, \mathcal{E}^*, pts^*\}$,which is processed by PUS, is a*

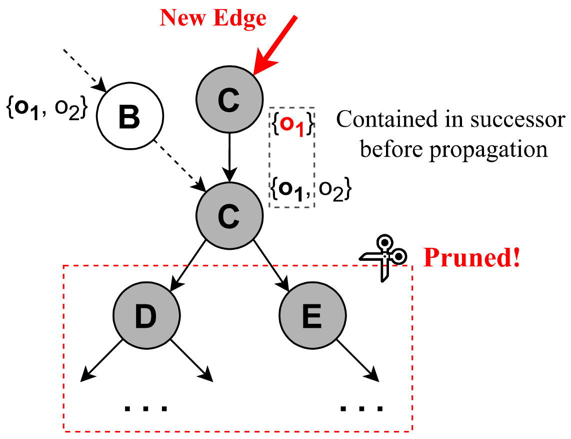


Figure 3.4: Further prune on the constraint graph during simple constraints processing phase. Reprinted From [1].

subgraph of $\mathcal{G}'$, *where* $\mathcal{V}^* = \mathcal{V}' - \mathcal{V}^-$, $\mathcal{E}^* = \mathcal{E}' - \mathcal{E}^-$, *and* $pts^* = pts'$.

Intuitively, definition 4.2 defines the set of nodes and edges that are pruned in Algorithm 3 at line 11, and the causality subgraph is defined by excluding the pruned nodes and edges from the *essential-edge-covered graph*. Fig. 3.4 offers an example that explains the rationale behind the graph pruning. In Fig. 3.4, the grey nodes and solid edges are within the essential-edge-covered graph $\mathcal{G}'$ for the current iteration. The corresponding points-to set is marked beside each node. In this example, the incoming update ($\{O_1\}$) to be propagated within the causality subgraph is already included in $pts(C)$ due to $B \rightarrow C$. To further propagate the points-to set from C does not make any update to C's successors (D and E in the example), thus the causality subgraph can be pruned by skipping $C \rightarrow D$ and $C \rightarrow E$. In Algorithm 3, since nodes are processed in topological order and all the nodes whose points-to sets have been updated in the current iteration are in L_{comp}, the test on $src \in L_{comp}$ at line 5 returns true only when $pts(src)$ gets updated in the current iteration. For node dst, if all the predecessors of dst are not included in L_{comp} and thus have not been updated, the outgoing edges of dst will be pruned.

By the end of the computation, Algorithm 3 outputs L_{comp} after draining the inputted node vector and passes L_{comp} to Algorithm 4.

Processing complex constraints: Algorithm 4 provides detailed information on how PUS handles complex constraints. The algorithm takes L_{comp}, the list of nodes provided by Algorithm 3, and locates all the nodes on which the complex constraints need to be processed.

The processing of the complex constraints follows a standard procedure by inserting new edges into the constraint graph. Algorithm 4 inserts all the newly added edges into the L_{copy} and eventually passes L_{copy} to both Algorithm 2 and Algorithm 3.

Note that whether or not a cached points-to set should be maintained so that PUS is able to process only the diffed points-to set [19] for complex constraints can be optionally applied. We omit the cached points-to set in our algorithm description as well as our implementation for better memory efficiency and our experimental results show that PUS is still much faster than techniques which apply the cached points-to set optimization.

3.3.3 Proof of Correctness

We prove that PUS will reach the global fixed point by the end of the computation in this section.

Definition. 4.4: *We say that a constraint graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}, pts\}$ is **points-to saturated** or reaches a **points-to saturated state** iff for any pair of nodes $v_1, v_2 \in \mathcal{V}$, if there is a path from v_1 to v_2, we have the minimal sets for $pts(v_1)$ and $pts(v_2)$ and $pts(v_1) \subseteq pts(v_2)$.*

Definition. 4.5: *We say that a constraint graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}, pts\}$ is **constraint saturated** or reaches a **constraint saturated state** iff for any node $v \in \mathcal{V}$, if there is a load constraint ($p = *v$) on v then there is an edge $e = \{v' \to p\} \in \mathcal{E}$ for every $v' \in pts(v)$; and if there are store constraints ($*v = p$) on v, then there is an edge $e = \{p \to v'\} \in \mathcal{E}$ for every $v' \in pts(v)$.*

By definition, the global fixed point is reached when the constraint graph is both points-to saturated and constraints saturated.

Lemma 4.1: Given an acyclic constraint graph, it will reach a *points-to saturated state* after processing the nodes once in topological order. □

Lemma 4.2: The *ineffective edge set $\mathcal{E}^-$* is empty during the first iteration in Algorithm 1. □

Theorem 4.1: At every iteration in Algorithm 1, the constraint graph ❶ is *points-to saturated* after processing simple constraints (line 13) and ❷ is *constraint saturated* after processing complex constraints on *unsaturated nodes* (line 14). □

Proof: We prove the theorem by induction.

For the first iteration ❶: By **Lemma 4.2**, the first iteration processes the entire *essential-edge-covered graph* $\mathcal{G}' = \{\mathcal{V}', \mathcal{E}', pts'\}$ with an essential edge set $\mathcal{E}^+ = \{e \mid e \in out(n) \wedge pts(n) \neq \varnothing\}$. By **Lemma 4.1**, the subgraph $\mathcal{G}'$ will reach a *points-to saturated state* after simple constraints processing. To prove the whole graph $\mathcal{G}$ will also be *points-to saturated*, it is equivalent to show that for nodes $n \notin \mathcal{V}'$, $pts(n) = \varnothing$: By contradiction, if there exists a node $n \notin \mathcal{V}' \wedge o \in pts(n)$, by the transitivity of constraint graph [7], there exists a path from address taken node of o to node n. However, since $\mathcal{E}^+$ includes all the address taken nodes' outgoing edges, node n should also be included in $\mathcal{V}'$ by definition, which contradicts with $n \notin \mathcal{V}'$. ❷: According to Algorithm 1, the *unsaturated node* set $\mathcal{V}^+ = \{v \mid pts(v) \neq \varnothing\}$ after processing simple constraints at line 13. It is obvious that the graph reaches a *constraint saturated state* after processing complex constraints on $\mathcal{V}^+$ as no edge needs to be inserted for node n whose points-to set is empty.

Combining ❶ and ❷, theorem 5.1 holds at the first iteration.

Suppose theorem 4.1 holds for the *n*-th iteration.

For the *n+1*-th iteration. We denote the constraint graph at *n*-th iteration before inserting new edges as $\mathcal{G}_n = \{\mathcal{V}_n, \mathcal{E}_n, pts_n\}$, the constraint graph at current iteration before inserting new edges as $\mathcal{G}_{n+1} = \{\mathcal{V}_{n+1}, \mathcal{E}_{n+1}, pts_{n+1}\}$ and the *causality graph* processed at current iteration as $\mathcal{G}^*_{n+1} = \{\mathcal{V}^*_{n+1}, \mathcal{E}^*_{n+1}, pts^*_{n+1}\}$

❶: According to Algorithm 1 and Algorithm 4, the essential edge set $\mathcal{E}^+_{n+1} = \mathcal{E}_{n+1} - \mathcal{E}_n$. To prove that a *points-to saturated state* will be reached, we prove the following two conditions hold:

1. for node $v \in \mathcal{V}_{n+1} - \mathcal{V}^*_{n+1}$, $pts_n(v) = pts_{n+1}(v)$ and thus need not to be processed, and

2. the pruning on $\mathcal{G}^*_{n+1}$ by removing ineffective constraints in $\mathcal{E}^-_{n+1}$ is sound.

For (1), assume there exists a node $v \in \mathcal{V}_{n+1} - \mathcal{V}^*_{n+1}$ and $\Delta_{n+1} = pts_{n+1}(v) - pts_n(v) \neq \varnothing$. By the transitivity of constraint graph, for $o \in \Delta_{n+1}$, there exists a path from the address taken node o' to v (denoted as a set $\mathcal{P} = \{o' \rightarrow v_1, v_1 \rightarrow v_2, ..., v_x \rightarrow v_y, v_y \rightarrow v\}$).

Case 1: If for every $e \in \mathcal{P}$, $e \notin \mathcal{E}^+_{n+1}$, then $e \in \mathcal{E}_n$. By induction hypothesis, the n-th iteration reached the *points-to saturated state*, thus $o \in pts_n(v)$ since there is a path $\mathcal{P}$ between o' and v, which is contradictory to the assumption $o \in \Delta_{n+1}$.

Case 2: If there exists a $e \in \mathcal{P}$, and $e \in \mathcal{E}^+_{n+1}$, then by definition $v \in \mathcal{V}^*_{n+1}$ and v is in the causality graph, which is contradictory to the assumption $v \in \mathcal{V}_{n+1} - \mathcal{V}^*_{n+1}$.

For (2), assume there exists an edge $e \in \mathcal{E}^-_{n+1}$ and by processing it, which is to compute $pts_{n+1}(e.dst) = pts_{n+1}(e.src) \cup pts_n(e.dst)$, $\Delta_{n+1} = pts_{n+1}(e.dst) - pts_n(e.dst) \neq \varnothing$. Since, by definition, $pts_{n+1}(e.src) - pts_n(e.src) = \varnothing$ as e is an ineffective edge. To satisfy $\Delta_{n+1} \neq \varnothing$, we have $pts_n(e.src) \nsubseteq pts_n(e.dst)$. However, since $e \in \mathcal{E}^-_{n+1}$, by definition $e \notin \mathcal{E}^+_{n+1}$, which equals $\mathcal{E}_{n+1} - \mathcal{E}_n$. We can conclude that $e \in \mathcal{E}_n$. By induction hypothesis, we have $pts_n(e.src) \subseteq pts_n(e.dst)$ and $e.dst$ is reachable from $e.src$ by $e \in \mathcal{E}_n$ and $\mathcal{G}$ is points-to saturated, which is contradictory to the assumption $pts_n(e.src) \nsubseteq pts_n(e.dst)$.

❷: According to Algorithm 3 and Algorithm 1, the set of *unsaturated nodes* $\mathcal{V}^+_{n+1} = \{v \mid pts_{n+1}(v) - pts_n(v) \neq \varnothing\}$. It is obvious that the algorithm will reach *constraint saturated state* after processing $v \in \mathcal{V}^+_{n+1}$. By induction hypothesis, in the previous iteration after inserting new edges, the constraint graph is *constraint saturated* and thus for $v' \in \{v \mid pts_{n+1}(v) = pts_n(v)\}$, they need not to be processed in the current iteration.

Combining ❶ and ❷, Theorem 4.1 holds at the $n+1$-th iteration provided that Theorem 4.1 holds in n-th iteration.

Theorem 4.2: Algorithm 1 guarantees the global fix point. □

Proof: We denote the constraint graph in the final iteration before inserting new edges as $\mathcal{G}_f = \{\mathcal{V}_f, \mathcal{E}_f, pts_f\}$ and the constraint graph in the final iteration after inserting new edges as $\mathcal{G}'_f = \{\mathcal{V}_f, \mathcal{E}'_f, pts_f\}$ Since Algorithm 1 returns when the *essential edge set* $\mathcal{E}^+_f = \mathcal{E}'_f - \mathcal{E}_f = \varnothing$, no edge is inserted by processing new complex constraints. Thus $\mathcal{G}_f = \mathcal{G}'_f$. By Theorem 4.1, $\mathcal{G}_f$ is *points-to saturated* and $\mathcal{G}'_f$ is *constraint saturated* and since $\mathcal{G}_f = \mathcal{G}'_f$, the final output $\mathcal{G}'_f$ are both points-to saturated and constraint saturated.

Table 3.1: Benchmarks and the constraint graph metrics (#Pointer, #Object and #Assign shows the number of pointers, objects and assignment statements in the tested program respectively). Reprinted From [1].

Benchmark	#LoC	#Pointer	#Object	#Assign
memcached	18.9K	15.2K	3.8K	6.0K
darknet	30.1K	91.3K	26.0K	44.1K
flatbuffers	156.1K	210.2K	83.5K	2659.5K
nfs-ganesha	251.5K	114.1K	33.5K	768.6K
curl	142.2K	70.0K	14.1K	578.5K
sqlite3	245.4K	129.3K	23.6K	1024.4K
keydb-server	259.1K	78.9K	20.0K	230.1K
vim	334.9K	267.9K	51.1K	1826.9K
cpython	564.9K	171.5K	52.2K	1770.0K
postgreSQL	1.0M	496.7K	106.3K	4677.6K

3.4 Experiments

The goal of our evaluation is to answer the following research questions.

- **RQ1:** How much reduction can PUS achieve by only processing the *causality subgraph* in each iteration? In other words, how large is the *causality subgraph* for real-world applications when compared to the entire constraint graph?

- **RQ2:** In terms of performance, how much faster is PUS when compared with state-of-the-art algorithms, namely WP and DP?

Table 3.2 and Fig. 3.5 provide strong evidence to support our key observation: The size of causality subraphs are small and updates on the points-to information of certain nodes only affect very limited set of neighboring nodes. From these experiments, we can easily understand why PUS is able to achieve such a dramatic reduction by analyzing small causality subgraphs instead of the entire constraint graph at each iteration.

3.4.1 RQ1: Reduction Achieved by PUS

To answer the first research question, we ran *context-insensitive* PUS on the benchmarks in Table 3.1 and collected statistics about the size of the causality subgraph processed by PUS in

Table 3.2: The size of the causality subgraphs processed by PUS. Reprinted From [1].

Benchmark	Constraint Graph		causality Subgraph				
	#Node	#Edge		#Node	%ratio	#Edge	%Ratio
memcached	19,033	14,792	min	1	0.01%	2	0.02%
			max	3,538	18.59%	11,175	75.53%
			avg.	**197**	**1.04%**	**703**	**4.74%**
darknet	117,272	110,622	min	1	0.00%	1	0.00%
			max	15,365	13.10%	34,406	31.10%
			avg.	**1,990**	**1.70%**	**7,171**	**6.48%**
flatbuffers	293,737	9,061,966	min	7	0.00%	11	0.00%
			max	48,533	16.52%	230,758	2.54%
			avg.	**6,679**	**2.27%**	**16,212**	**0.18%**
nfs-ganesha	147,550	824140	min	1	0.00%	2	0.00%
			max	28,865	20.07%	62,491	16.61%
			avg.	**726**	**0.50%**	**2,590**	**0.69%**
curl	84,311	646,020	min	2	0.00%	2	0.00%
			max	14,743	17.48%	51,365	7.95%
			avg.	**3,341**	**3.96%**	**17,688**	**2.74%**
sqlite3	152,919	1,114,272	min	1	0.00%	2	0.00%
			max	34,642	23.32%	120,412	10.79%
			avg.	**9,113**	**5.97%**	**30,938**	**2.77%**
keydb-server	99,015	270,497	min	2	0.00%	2	0.00%
			max	21,147	21.36%	44,993	16.60%
			avg.	**4,865**	**4.91%**	**12,229**	**4.51%**
vim	319,056	2,087,531	min	2	0.00%	6	0.00%
			max	68,600	21.50%	235,240	12.42%
			avg.	**10,512**	**3.29%**	**39,570**	**2.09%**
cpython	223,675	2,129,505	min	16	0.00%	16	0.00%
			max	52,424	23.44%	139,810	7.53%
			avg.	**9,885**	**4.42%**	**33,676**	**1.81%**
postgreSQL	603,061	4,988,935	min	1	0.00%	1	0.00%
			max	114,410	18.97%	333,764	6.90%
			avg.	**13,241**	**2.20%**	**46,621**	**0.96%**
avg.	**-**	**-**		**-**	**3.02%**	**-**	**2.69%**

each iteration. The detailed report is listed in Table 3.2. Table 3.2 compares the size of different causality subgraphs and analyzes the relative sizes of the causality subgraphs compared with the entire constraint graph.

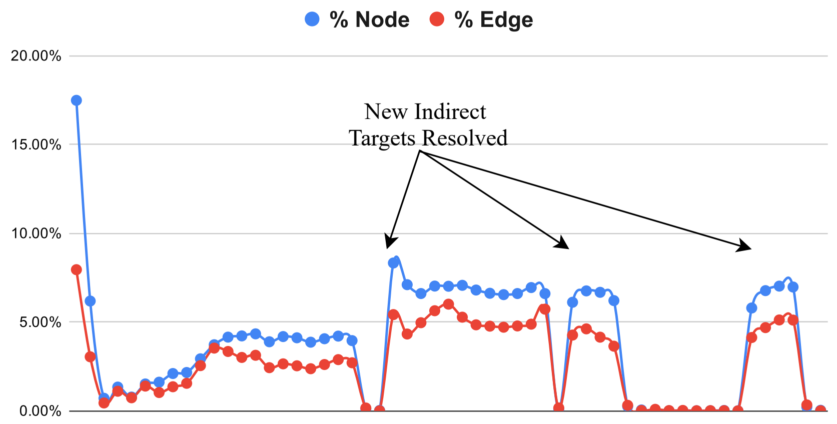
We report the *minimal*, *maximum* and *average* number of nodes and edges processed by PUS to summarize the characteristics of the causality subgraph because a different causality subgraph is computed by PUS in each iteration.

As shown in Table 3.2, on average a causality subgraph only contains around 3% of the nodes and 2.7% of the edges in the respective whole constraint graph. For most of the benchmarks, the size of the causality subgraph can be as small as just 1 or 2 nodes and edges, even for large benchmarks (e.g., cpython and postgreSQL) with more than 500K nodes and 300K edges. The minimal causality subgraph is usually observed in the last few iterations when the points-to sets of most of the nodes in the constraint graph are saturated. The result gives us more confidence on the performance improvement can be achieved by PUS, as algorithms like WP would still need to re-sort the entire constraint graph even when the number of effective nodes can be as low as 1.

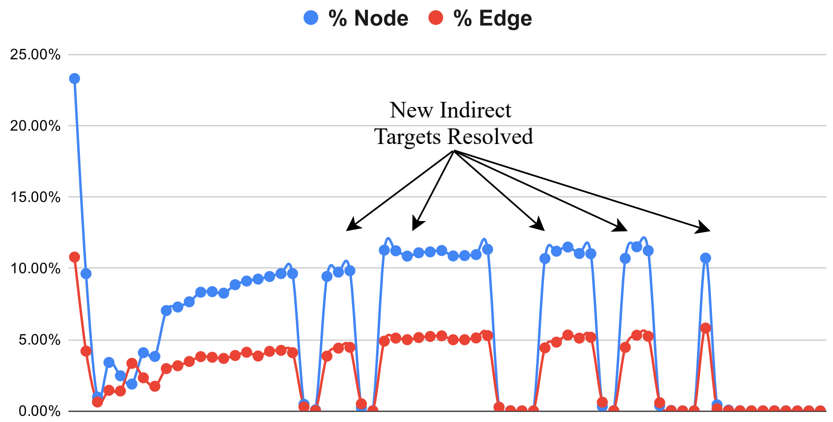
The result also shows that for most of the benchmarks, even the largest causality subgraph usually contains no more than 30% of the nodes and 30% of the edges in the complete constraint graph. More importantly, according to our observation, large causality subgraphs do not occur frequently, which is also why the average number of nodes and edges in the causality graph is still low despite the existence of some relatively large subgraphs. Our experiments shows that large causality graphs normally occur in the first few iterations at the beginning of the computation and/or after indirect calls are resolved and new nodes are inserted. These observations are validated in Fig. 3.5 and will be elaborated in the following paragraphs.

In order to gain insights into the entire 'lifetime' of the causality subgraphs and to understand how it 'evolves' as the analysis proceeds, we include two complete (also *typical*) footprints that show how the sizes of causality subgraphs fluctuate in each iteration of the whole solving process. The two data sets are collected by evaluating PUS on *curl* and *sqlite3* and are visualized in Fig. 3.5 (a) and Fig. 3.5 (b) respectively. It is clear that Fig. 3.5 (a) and Fig. 3.5 (b) exhibit several common patterns:

- The size of the causality graph normally increases greatly as new indirect calls are resolved. This is because each time when an indirect call is resolved, the newly resolved target functions introduce many unprocessed nodes and constraints into the constraint graph. Those



(a) The footprint of *curl*



(b) The footprint of *sqlite3*

Figure 3.5: The footprint of the size of the causality subgraphs processed by PUS at each iteration when analyzing *curl* and *sqlite3*. Reprinted From [1].

unprocessed nodes are likely to invalidate a large portion of the constraint graph, which in turn increases the size of the causality subgraph for the next iteration.

- After new nodes are inserted, the size of the causality subgraph normally reduces sharply after several iterations. The size then remains small until another set of new indirect calls get resolved. This indicates that the solving process converges quickly after a few iterations on most of the nodes, and then gradually approach the fixed point by only processing a very small number of nodes at each iteration.

Table 3.3: Performance of PUS comparing with wave propagation (WP) and deep propagation (DP) when running context-insensitive pointer analysis (%↑ shows the speedup). Reprinted From [1].

Benchmark	PUS	WP		DP	
		time	%↑	time	%↑
memcached	0.04s	0.35s	775.00%	0.1s	150.45%
darknet	0.34s	1.82s	435.29%	1.00s	194.12%
flatbuffers	95.9s	195.72s	104.08%	124.87s	30.28%
nfs-ganesha	11.17s	48.83s	327.15%	26.48s	137.06%
curl	18.45s	29.45s	59.62%	28.29s	53.27%
sqlite3	46.48s	98.77s	125.50%	128.73s	176.96%
keydb-server	4.84s	7.58s	56.61%	5.76s	19.00%
vim	81.17s	183.81s	126.41%	193.70s	138.55%
cpython	400.66s	619.55s	54.61%	655.91s	63.70%
PostgreSQL	1,381.2s	1,757.9s	27.27%	2,001.6s	44.91%
avg.	**-**	**-**	**3.09×**	**-**	**2×**

Table 3.2 and Fig. 3.5 provide strong evidence to support our key observation: The size of causality subraphs are small and updates on the points-to information of certain nodes only affect very limited set of neighboring nodes. From these experiments, we can easily understand why PUS is able to achieve such a dramatic reduction by analyzing small causality subgraphs instead of the entire constraint graph at each iteration.

3.4.2 RQ2: The Performance Improvement Achieved by PUS

PUS was evaluated in both context-insensitive and context-sensitive (k-callsite, with $k = 1$) settings. and compared with WP and DP. The experimental results are elaborated in Section 3.4.2.1 (when running context-insensitive analysis) and Section 3.4.2.2 (when running context-sensitive analysis).

3.4.2.1 Improvement when Running Context-Insensitive Pointer Analysis

In the context-insensitive setting, the execution time of each algorithm when running on different benchmarks is given in Table 5.2.

In summary, PUS achieves a significant performance improvement compared to WP and DP,

Table 3.4: Performance of PUS comparing with wave propagation (WP) and deep propagation (DP) when running k-callsite sensitive ($k = 1$) pointer analysis (%↑ shows the speedup). Reprinted From [1].

Benchmark	PUS	WP		DP	
		time	%↑	time	%↑
memcached	0.08s	0.68s	708.33%	0.26s	210.71%
darknet	1.09s	6.36s	484.30%	5.33s	389.81%
flatbuffer	673.39s	4580.5s	580.22%	2542.8s	277.62%
nfs-ganesha	33.28s	367.97s	1005.82%	459.54s	1281.0%
curl	37.58s	266.98s	610.40%	258.3s	587.31%
sqlite3	112.44s	639.64s	468.88%	961.3s	754.96%
keydb-server	20.38s	77.51s	280.42%	79.66s	290.98%
vim	367.62s	2587.1s	603.75%	3647.1s	892.09%
cpython	367.33s	3358.9s	814.43%	9559.7s	2502.5%
PostgreSQL	OOM	OOM	-%	OOM	-%
avg.	**-**	**-**	**7.17**×	**-**	**8.99**×

with more than $2\times$ speedup on average. For certain benchmarks, namely *memcached* and *darknet*, PUS can be $4\times$ as faster than WP. When compared with DP, PUS is $2\times$ faster on more than half of the tested benchmarks (namely *memcached*, *darknet*, *nfs-ganesha*, *vim* and *sqlite3*). Even in the worst cases, PUS can still be more than 20% faster than DP and WP.

In our experiments, we also made a similar observation as found in the original WP and DP paper [19]. The original paper observes that WP has an advantage over DP when analyzing relatively large program as WP is faster than DP on *sqlite3, vim* and *cpython* and DP is faster than WP on relatively smaller programs such as *memcached*, *redis-server* and *nfs-ganesha*. The one exception is *flatbuffer*, which has a relatively small number of lines of code (48.9K) while its corresponding constraint graph is nearly as big as that of large programs such as *vim*. However, unlike WP and DP, which have different advantages when analyzing programs of different scales, PUS outperforms both WP and DP on all the tested benchmarks with the sizes ranging from $18K$ to over $1M$ lines of code. PUS can be $8\times$ faster and achieves at least a 19% speedup. The fact that PUS is able to outperform both WP and DP on benchmarks of varying sizes (both large and small) indicates that is a much more general algorithm that can be applied to all kinds of programs.

3.4.2.2 Improvement when Running Context-Sensitive Pointer Analysis

In the context-sensitive setting (k-callsite, with $k = 1$), the execution time of each algorithm when running on different benchmarks is given in Table 5.2.

Surprisingly, PUS even achieved much higher speedups when solving context-sensitive constraints when compared to context-insensitive constraints. The results shows that on average, PUS is almost $7\times$ and $9\times$ faster than WP and DP respectively. For certain benchmarks, PUS can be more than $10\times$ faster than WP and DP (*ganesha*) and more than $25\times$ faster than DP (*python*). On all benchmarks, PUS is *at least* $2\times$ faster than both WP and DP. The result indicates that PUS has a great potential to be adopted widely as the computing power becomes stronger and stronger and more precise pointer analysis is desired in the future.

The reason why PUS is able to significantly outperform the state-of-the-art algorithms, especially when solving context-sensitive pointer analysis, is still rooted in our key insight that constraint graphs are *sparsely connected*. This property becomes even more essential in context-sensitive pointer analysis as one node in context-insensitive pointer analysis can correspond to multiple nodes in context-sensitive pointer analysis because the same variable is now analyzed separately under different contexts. This makes the constraint graph *sparser*. Thus, the effect brought by the update of one node becomes more *local* as it can only affect nodes under some particular contexts whereas one node can affect many neighbors in context-insensitive pointer analysis, even when the neighboring nodes represent variables in a mismatched context.

Despite all kinds of optimizations made when designing PUS, we still found it challenging to run context-sensitive pointer analysis on extremely large benchmarks using only commodity hardware. As the k-limiting for context-sensitive pointer analysis increases, the complexity of the algorithm and the size of the constraint graph grows exponentially, which makes the algorithm hard to scale on large benchmarks. During the experiments, we observed that more than 5 million nodes were created in the constraint graph for *PostgreSQL* in the first 5 minutes, which rapidly drains the memory of our machine. A more powerful machine is needed for evaluating PUS on *PostgreSQL*.

3.5 Summary

We have presented Partial Update Solver (PUS), a new constraint solving algorithm for inclusion-based pointer analysis. PUS significantly advances the state-of-the-art in reducing the time complexity by a quadratic factor. The key insight is that only a small portion of the constraint graph is effective for the points-to set propagation, which can be extracted efficiently into a subgraph, called *causality subgraph*. We have formally proved the correctness of and extensively evaluated the performance of on a wide range of real-world large complex programs. Our experimental results indicate that is high scalable and significantly more efficient than the state-of-the-art WP/DP algorithms. PUS achieves more than $7\times$ ($2\times$) speedups when comparing to WP/DP in solving context-sensitive and context-insensitive pointer analyses respectively.

4. ORIGIN-SENSITIVE POINTER ANALYSIS AND O2*

Data races are among the worst bugs in software in that they exhibit non-deterministic symptoms and are notoriously difficult to d etect. The problem is exacerbated by interactions between threads and events in real-world applications, which are two predominant programming abstractions for modern software such as operating systems, databases, mobile apps, and so on as they both lead to non-deterministic behaviors due to various types of race conditions.

Races in event-driven programs have attracted much attention in recent years [60, 61, 62, 63]. Event-based races can be more challenging to detect than thread-based races because most events are asynchronous and the event handlers may be triggered in many different ways. Moreover, the difficulty in detecting event-based races is exacerbated by interactions between threads and events, which are common in real-world software such as distributed systems. The state-of-the-art race detectors [50, 55] do not perform well in detecting event-based races, also due to the large space of casual orders among event handlers and threads.

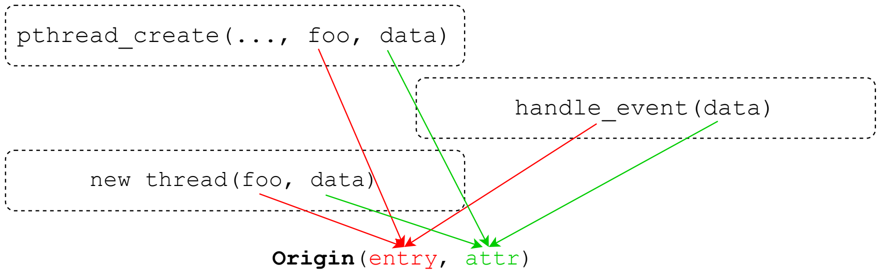


Figure 4.1: An “origin” view of threads and events.

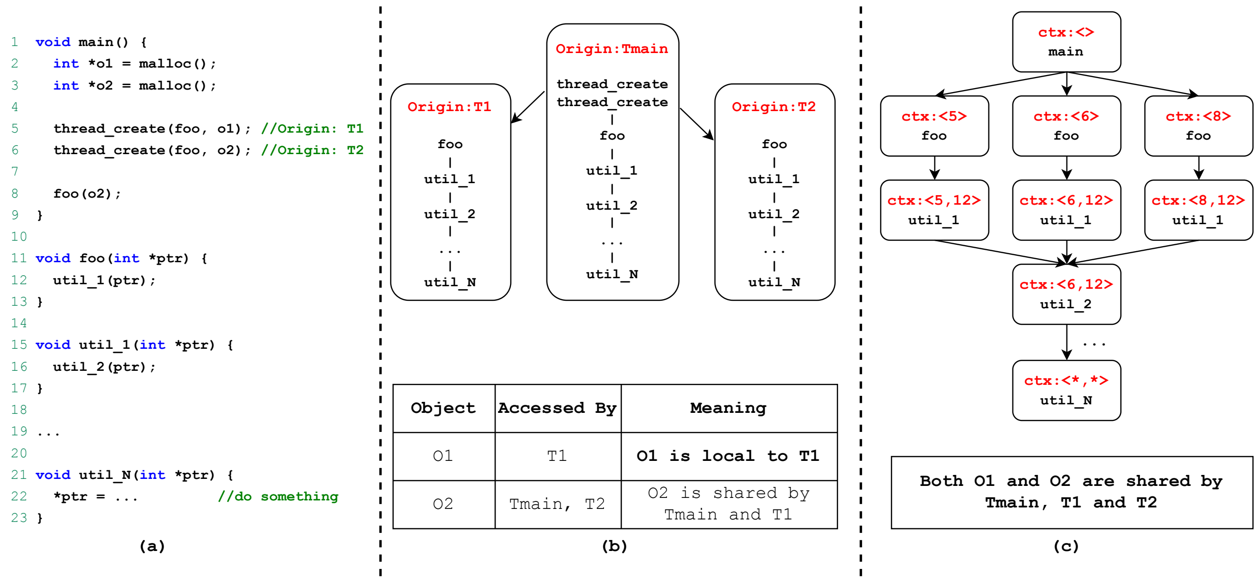
To mitigate the challenge, we preposed O2, a new system for detecting data races in complex multithreaded and event-driven applications. We show that conventional thread-sensitive static

Reprinted with permission from “When threads meet events: efficient and precise static race detection with origins” by Liu, Bozhen and Liu, Peiming* and Li, Yanze and Tsai, Chia-Che and Da Silva, Dilma and Huang, Jeff, 2021. Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Copyright 2021 by Peiming Liu.

analysis (with some tuning and care) is highly effective for finding races. A key concept behind O2 are *origins*, an extended notion of threads and events that unify them through two parts: 1) *an entry point* that represents the beginning of a thread or an event handler, and 2) *a set of attributes* that capture additional semantics, such as thread ID, event type, or pointers to memory objects that will be used in the thread or event handler. Figure 4.1 depicts an "origin" view for threads and events in C/C++. The origin attributes can be specified or inferred automatically at the origin's entry point. Rather than a straightforward unification, origins enables *origin-sensitive pointer analysis*, in which the conventional call-string-based or object-based context abstractions are replaced by origins. This has several advantages:

- Functions within the same origin share the same context, therefore the computation complexity inside an origin does not grow with the length of the call chain; and
- Computing k-most-recent calling contexts at every call site is redundant in many applications [29], e.g., when determining which objects are local to or are shared by which threads.
- The crucial origin entry point is preserved, not discarded as a trivial context in *k-limiting* [64] when the call stack's depth exceeds the context depth k.

To illustrate these advantages, consider an example in Figure 4.2. In this example, threads `T1` (line 5) does not share objects with `T2` and `Tmain`, while `T2` (line 6) and `Tmain` shares the same object `o2`. When marking `thread_create` as an origin entry points, the origin-sensitive callgraph (in Figure 4.2 (b)) is computed. It is clear that the same call chain are distinguished based on the origin ID, and the date flow within the same origin are analyzed separately with each other. Thus, origin-sensitive pointer analysis is able to conclude the correct sharing information between different threads. On the other hand, conventional k-call-site analysis (denoted *k-CFA*) can also be performed, in which k is the depth of the call chain [65] (in this example k is set to 2) used as the context to distinguish the same function when invoked in different context. When using this method, only 2 most recent calls are considered to analyze the function context-sensitively. As the result, it can only distinguish up to function `util_1`. Following functions after `util_2`



```
1  void main() {
2    int *o1 = malloc();
3    int *o2 = malloc();
4
5    thread_create(foo, o1); //Origin: T1
6    thread_create(foo, o2); //Origin: T2
7
8    foo(o2);
9  }
10
11 void foo(int *ptr) {
12   util_1(ptr);
13 }
14
15 void util_1(int *ptr) {
16   util_2(ptr);
17 }
18
19 ...
20
21 void util_N(int *ptr) {
22   *ptr = ...          //do something
23 }
```

(a)

Object	Accessed By	Meaning
O1	T1	O1 is local to T1
O2	Tmain, T2	O2 is shared by Tmain and T1

(b)

Both O1 and O2 are shared by Tmain, T1 and T2

(c)

Figure 4.2: (a) The example code. (b) The origin-sensitive call graph, where each origin consists of a sequence of calls of arbitrary length. The origin attributes precisely determine the call chain executed in each origin. (c) The context-sensitive (2-CFA) call graph without origin.

then become indistinguishable as their 2 most recent call context is identical. Sine the data flow information will also be merged at `util_2`, 2-CFA will conservatively assume that both `o1` and `o2` are shared among `T1`, `T2` and `Tmain`.

In addition to origin-sensitive analysis, there are a few important design choices we made in O2 that together make static race detection highly effective. First, O2's race detection engine is highly optimized to achieve scalability and precision. We construct a static happens-before graph (SHB) and use static "happens-before" instead of static "may-happen-in-parallel" as the foundational concept of the analysis. This allows pruning many infeasible race pairs by checking only graph reachability. Second, we develop several sound optimizations that scale race detection to large code bases, including:

- An efficient representation of origin-local happens-before relations, which further enables efficient checking and caching the happens-before relation between memory accesses;
- A compact representation of locksets, which enables a fast check of common locks and an

efficient cache policy of the intermediate results;

- A lock-region-based race detection that allows effectively merging many memory accesses into a representative one, which reduces the number of race checks significantly.

Note that the original work covered implementation details for both C/C++ and Java. While in this dissertation, the tool is introduced in the context of C/C++ as the implementation for Java is mainly accomplished by the other author.

4.1 Origin-Sensitive Pointer Analysis

In this section, we first present origin-sensitive pointer analysis, the use of which enables a more precise pointer analysis and identification of shared- and local-memory accesses by threads and events. Beyond race detection, it can benefit any analysis that requires analyzing pointers or ownership of memory accesses, e.g., deadlock, over-synchronization, and memory isolation.

4.1.1 Identifying Origins

In general, a program can conceptually be divided into many different origins, each represents a unit of the program's functionality. At the code level, an origin is a set of code paths all with the same starting point (i.e., the entry point) and data pointers (i.e., the origin attributes). In this way, origins divide a program into different sets of code paths according to their semantics where each origin represents a separate semantic domain. While origins can be specified by code annotations, we aim to extract them automatically from common code patterns in multithreaded and event-driven programs. Our system identifies two kinds of origins automatically by default: threads and event handlers. Finding static threads is not difficult in practice because threads are almost always explicitly defined, either at the language level or through common APIs such as POSIX Threads (Pthreads). Finding event handlers relies on code patterns such as Linux system call interfaces (all with prefix `__x86_sys_`). In cases where threads or events are implicit, such as customized user-level threads, developers may be willing to provide annotations to mark origins, since customized threads are likely to be an important aspect of the target application.

For `pthread` and `std::thread`-based and C/C++ programs, we automatically identify the

APIs as the origin *entry* points, which are frequently used to run code in parallel. We then reason about the origin *attributes* in order to distinguish different origins with the same entry point but different data. The origin attributes can be inferred at two places:

- *Origin Allocation* is the allocation site of a receiver object of an origin entry point. The attributes include the arguments passed to the allocation site. are the origin attributes of `T1`.
- *Origin Entry Point* may be invoked with parameters, of which pointers are also included in the attributes. For example, calls to `pthread_create` must specify the data pointer needed by the thread to share memory objects.

4.1.2 Origin-Sensitivity Rules

Pointer analysis typically uses the *pointer assignment graph* (PAG) [66] to represent points-to relations between pointers and objects. To achieve good precision, the PAG constructed by origin-sensitive PTA is built together with the call graph (a.k.a. on-the-fly pointer analysis [66]). The key difference is that the context of pointers in origin-sensitive PTA is represented by origins. The rules of origin-sensitive PTA for Java are summarized in Table 4.1. A set of similar rules can be inferred for other programming languages.

Intra-Origin Constraints: Statements ❶-❼ are in method $\langle m, \mathbb{O}_i \rangle$, and all the program elements created by them share the same origin $\mathbb{O}_i$ to indicate where they are originated from. For example, the allocated object by statement ❶ is represented as $\langle o, \mathbb{O}_i \rangle$ and assigned to pointer $\langle x, \mathbb{O}_i \rangle$, and their relation is represented by a points-to edge $\langle o, \mathbb{O}_i \rangle \rightarrow \langle x, \mathbb{O}_i \rangle$ in the PAG .

An object field pointer is distinguished by the origin of its receiver object. For statement ❹, each receiver object $\langle o, \mathbb{O}_k \rangle$ corresponds to an object field pointer $\langle o, \mathbb{O}_k \rangle.f$ that points to $\langle x, \mathbb{O}_i \rangle$. Note that a pointer and its points-to objects may have different origins, which shows how data flows across origins.

Although there exists a large body of work that can infer the content of arrays, analyzing array index *idx* in statements ❺❻ is statically undecidable and expensive. Hence, we do not distinguish different array indexes: array objects are modeled as having a single field $*$ that may point to any

Table 4.1: The origin-sensitive analysis rules for Java. Consider the following statements are in method `m()` with Origin $\mathbb{O}_i$, denoted $\langle m, \mathbb{O}_i \rangle$. The edges $\rightarrow$ are in the PAG and $\rightsquigarrow$ in the call graph. Reprinted From [3].

Statement	**Points-to Edge & Call Edge**
❶ $x = malloc()$	$\langle o, \mathbb{O}_i \rangle \rightarrow \langle x, \mathbb{O}_i \rangle$
❷ $x = y$	$\langle y, \mathbb{O}_i \rangle \rightarrow \langle x, \mathbb{O}_i \rangle$
❸ $*x = y$	$\dfrac{\forall \langle o, \mathbb{O}_k \rangle \in pts(\langle x, \mathbb{O}_i \rangle)}{\langle y, \mathbb{O}_i \rangle \rightarrow \langle o, \mathbb{O}_k \rangle}$
❹ $x = *y$	$\dfrac{\forall \langle o, \mathbb{O}_k \rangle \in pts(\langle y, \mathbb{O}_i \rangle)}{\langle o, \mathbb{O}_k \rangle \rightarrow \langle x, \mathbb{O}_i \rangle}$
❺ $x[idx] = y$	$\dfrac{\forall \langle o, \mathbb{O}_k \rangle \in pts(\langle x, \mathbb{O}_i \rangle)}{\langle y, \mathbb{O}_i \rangle \rightarrow \langle o, \mathbb{O}_k \rangle.*}$
❻ $x = y[idx]$	$\dfrac{\forall \langle o, \mathbb{O}_k \rangle \in pts(\langle y, \mathbb{O}_i \rangle)}{\langle o, \mathbb{O}_k \rangle.* \rightarrow \langle x, \mathbb{O}_i \rangle}$
❼ $x = y.f(a_1, ..., a_n)$ //non-origin entry	$\dfrac{\begin{array}{c}\forall \langle o, \mathbb{O}_k \rangle \in pts(\langle y, \mathbb{O}_i \rangle) \\ \langle f', \mathbb{O}_i \rangle = dispatch(\langle o, \mathbb{O}_k \rangle, f)\end{array}}{\begin{array}{c}\langle o, \mathbb{O}_k \rangle \rightarrow \langle f'_{this}, \mathbb{O}_i \rangle \\ \langle a_h, \mathbb{O}_i \rangle \rightarrow \langle p_h, \mathbb{O}_i \rangle, \text{where } 1 \leq h \leq n \\ \langle f'_{ret}, \mathbb{O}_i \rangle \rightarrow \langle x, \mathbb{O}_i \rangle \\ \textbf{\textit{add call edge}}\ \langle m, \mathbb{O}_i \rangle \rightsquigarrow \langle f', \mathbb{O}_i \rangle\end{array}}$
❽ $x = new\ thread(b_1, ..., b_n)$ //origin allocation	$\dfrac{\begin{array}{c}\textbf{\textit{Compute new origin}}\text{: } \mathbb{O}_j \\ \langle init, \mathbb{O}_j \rangle = dispatch(-, init)\end{array}}{\begin{array}{c}\langle o, \mathbb{O}_j \rangle \rightarrow \langle init_{this}, \mathbb{O}_j \rangle \\ \langle o, \mathbb{O}_j \rangle \rightarrow \langle x, \mathbb{O}_i \rangle \\ \langle b_h, \mathbb{O}_i \rangle \rightarrow \langle p_h, \mathbb{O}_j \rangle, \text{where } 1 \leq h \leq n \\ \textbf{\textit{add call edge}}\ \langle m, \mathbb{O}_i \rangle \rightsquigarrow \langle init, \mathbb{O}_j \rangle\end{array}}$
❾ $x.entry(c_1, ..., c_n)$ //origin entry point	$\dfrac{\begin{array}{c}\forall \langle o, \mathbb{O}_j \rangle \in pts(\langle x, \mathbb{O}_i \rangle) \\ \langle entry', \mathbb{O}_j \rangle = dispatch(\langle o, \mathbb{O}_j \rangle, entry)\end{array}}{\begin{array}{c}\langle o, \mathbb{O}_j \rangle \rightarrow \langle entry'_{this}, \mathbb{O}_j \rangle \\ \langle c_h, \mathbb{O}_i \rangle \rightarrow \langle p_h, \mathbb{O}_j \rangle, \text{where } 1 \leq h \leq n \\ \textbf{\textit{add call edge}}\ \langle m, \mathbb{O}_i \rangle \rightsquigarrow \langle entry', \mathbb{O}_j \rangle\end{array}}$

value stored in the array, e.g., $x[idx] = y$ is modeled as $x.* = y$. This model simply captures objects allocated by different origins that flow to an array without any complex index analysis. Besides, our algorithm can be easily integrated with existing array index analysis algorithms with no conflict.

❼ specifies how a complete callgraph can be computed in presence of virtual function call. When a non-origin entry method call ❼ invokes a target method `f'` within the same origin $\mathbb{O}_i$ as its caller, even though its receiver object $\langle o, \mathbb{O}_k \rangle$ might be allocated from a different origin $\mathbb{O}_k$. To determine a virtual call target and its context, we use the type of its receiver object o and the origin $\mathbb{O}_i$ of which thread/event-handler executes the target. The target's origin must be consistent with its caller's, regardless of whether it is an entry point or not.

Inter-Origin Constraints: We switch contexts from current origin $\mathbb{O}_i$ to a new origin $\mathbb{O}_j$ for an origin allocation ❽ and an origin entry point ❾.

Note that, to avoid false aliasing introduced by thread creations, we analyze every origin allocation in its new origin instead of its parent origin where it should be executed. Figure **??** shows two origins (`Ta` and `Tb`) allocated in Origin `Tmain`. The two origin allocations share the same super constructor `T()`. If we analyze them in their parent origin `Tmain`, only one object o_f will be allocated for field f on line 14. This will cause $pts(o_a.f) = pts(o_b.f) = \{\langle o_f, Tmain \rangle\}$, which introduces false aliasing. To eliminate such imprecision, origin-sensitive analysis creates two objects, $\langle o_f, Ta \rangle$ and $\langle o_f, Tb \rangle$, for each f under each origin by forcing the context switch at origin allocations on lines 2 and 3.

To identify origin allocations on-the-fly, we check the type of the allocated object, if it is a subclass of `std::thread` or event handler `handleEvent()`. Context switch on ❽ can efficiently separate data flows to the same origin constructor but from different allocation sites, Specifically, in this example a new and unique origin $\mathbb{O}_j$ is created for this new allocation $\langle o, \mathbb{O}_j \rangle$.

Both ❽ and ❾ designate the attributes for the new origin $\mathbb{O}_j$, including constructor arguments $(b_1, ..., b_n)$ and method parameters $(c_1, ..., c_n)$, which reveal significant information of the accessed data and the origin behavior. To reflect the ownership, the actual parameters use $\mathbb{O}_i$ as their contexts and the formal ones use $\mathbb{O}_j$. Meanwhile, call edges are added in the call graph, e.g., $\langle m, \mathbb{O}_i \rangle \rightsquigarrow \langle init, \mathbb{O}_j \rangle$ for ❽ and $\langle m, \mathbb{O}_i \rangle \rightsquigarrow \langle entry', \mathbb{O}_j \rangle$ for ❾.

Wrapper Functions and Loops: In practice, both ❽ and ❾ may be hidden in a wrapper function (e.g., cross-platform thread wrappers) invoked by multiple call sites. To efficiently sepa-

Table 4.2: The time complexity of different pointer analyses. Reprinted From [3].

Analysis	**Worst-Case Complexity**
0-context	O($p \times h^2$)
heap	O($p^3 \times h^2$)
2-CFA + heap	O($p^5 \times h^2$)
2-obj + heap	O($p^5 \times h^2$)
1-origin + heap	O($p^3 \times h^2$)

rate such origins, we can extend the entry point of an origin to also include its *k-call-site*. In our tools, we set k=1. Meanwhile, for an origin allocated in a loop, we always create two origins with identical attributes but different origin IDs.

***K*-Origin-Sensitivity:** In the same spirit as k-CFA and k-obj, a sequence of origins can be concatenated, denoted as *k-origin*. For example, a method `m()` can be denoted as follows:

$$\langle m, [\mathbb{O}_1, \mathbb{O}_2, ..., \mathbb{O}_{k-1}, \mathbb{O}_k] \rangle$$

where `m()` is invoked within Origin $\mathbb{O}_k$ that has a parent origin $\mathbb{O}_{k-1}$, etc. k-origin can further improve the precision when a pointer propagates across nested origins, and we observed such cases in many of our evaluated programs (e.g., Redis) where thread creations are nested.

Time Complexity: Table 4.2 summarizes the worst-case time complexity of different pointer analysis algorithms according to [67], where *p* and *h* are the number of statements and heap allocations, respectively. The complexity of k-CFA and k-obj varies according to the context depth k. However, their worst-case complexity can be doubly exponential [68]. The selective context-sensitive techniques [30, 31, 32, 33, 69] are also bounded by the context depth and have the same worst-case complexity as their corresponding full k-CFA and k-obj algorithms.

The 1-origin has the same complexity as 1-call-site-sensitive heap analysis (denoted *heap*). But the number of operations is increased linearly by a factor $(\#\mathbb{O} \times \mathbb{O}\%)$, where $\#\mathbb{O}$ is the number of origins and $\mathbb{O}\%$ is the ratio between the average number of statements within an origin and the total number of program statements, which is usually small (<10%) for most applications, according to

Table 4.3: SHB Graph with Origins: the following statements are in method `m()` with Origin $\mathbb{O}_i$. Reprinted From [3].

Intra-Origin Happen-before Rules

Statement	**Intra-Origin Node & HB Edge**
❸ $*x = y$	$\forall\langle o, \mathbb{O}_k\rangle \in pts(\langle x, \mathbb{O}_i\rangle)$, **write**($\langle o, \mathbb{O}_k\rangle$)
❹ $x = *y$	$\forall\langle o, \mathbb{O}_k\rangle \in pts(\langle y, \mathbb{O}_i\rangle)$, **read**($\langle o, \mathbb{O}_k\rangle$)
❺ $x[idx] = y$	$\forall\langle o, \mathbb{O}_k\rangle \in pts(\langle x, \mathbb{O}_i\rangle)$, **write**($\langle o, \mathbb{O}_k\rangle$)
❻ $x = y[idx]$	$\forall\langle o, \mathbb{O}_k\rangle \in pts(\langle y, \mathbb{O}_i\rangle)$, **read**($\langle o, \mathbb{O}_k\rangle$)
❼ $x = y.f(a_1, ..., a_n)$	$\forall\langle f, \mathbb{O}_i\rangle \in dispatch(\langle y, \mathbb{O}_i\rangle, f)$, *add HB edge*: **call**($\langle f, \mathbb{O}_i\rangle$) $\Rightarrow$ $\mathbf{f_{first}}$($\langle f, \mathbb{O}_i\rangle$), $\mathbf{f_{last}}$($\langle f, \mathbb{O}_i\rangle$) $\Rightarrow$ $\mathbf{call_{next}}$($\langle f, \mathbb{O}_i\rangle$)
🔓 $synchronized(x)\{ \ldots \}$	$\forall\langle o, \mathbb{O}_k\rangle \in pts(\langle x, \mathbb{O}_i\rangle)$, **lock**($\langle o, \mathbb{O}_k\rangle$), **unlock**($\langle o, \mathbb{O}_k\rangle$)

Inter-Origin Happen-before Rules

Statement	**Inter-Origin Node & HB Edge**
❾ $x.entry(c_1, ..., c_n)$	$\forall\langle entry, \mathbb{O}_j\rangle \in dispatch(\langle x, \mathbb{O}_i\rangle, entry)$, *add HB edge*: **entry**($\mathbb{O}_i, \mathbb{O}_j$) $\Rightarrow$ $\mathbf{origin_{first}}$($\mathbb{O}_j$)
❿ $x.join()$	$\forall\langle join, \mathbb{O}_j\rangle \in dispatch(\langle x, \mathbb{O}_i\rangle, join)$, *add HB edge*: $\mathbf{origin_{last}}$($\mathbb{O}_j$) $\Rightarrow$ **join**($\mathbb{O}_j, \mathbb{O}_i$)

our experiments.

4.2 O2: Race Detection Algorithm

In O2, we model both threads and events statically as functional units, each represented by a static trace of memory accesses and synchronization operations. Our race detection engine uses hybrid happens-before and lockset analyses similar to most prior work on dynamic race detection [70] (although ours is static). More specifically, our detection represents happens-before relations by a static happens-before (SHB) graph [71], which is designed to efficiently compute incremental changes from source code.

We modify the graph with origins as shown in Table 4.3. We record the field/array read and write accesses for statements ❸-❻ by creating read and write nodes. For statement ❼, we create a method call node (call) with two happens-before (HB) edges (denoted $\Rightarrow$): one points from the call node to the first node (f_{first}) of its target method `f` within the same origin $\mathbb{O}_i$, the other points from the last node (f_{last}) of $\langle f, \mathbb{O}_i \rangle$ to the next node after the call ($\text{call}_{\text{next}}$). Intra-origin HB edges are created by pointing from one intra-origin node to another in their statement order.

For lock operation 🔓, we create lock and unlock nodes to maintain the current lockset. For Java programs, we consider `synchronized` blocks and methods. For C/C++ programs, O2 currently only considers monitor-style locks (including both standard pthread mutexes and customized locks through configurations). And we aim to support atomics (e.g., `std::atomic`) and semaphores in our future work, by adding new happens-before rules from different origins to the atomic/semaphore operations.

For calls to an origin entry point ❾, we create an origin entry node (entry) to represent the start of a new origin $\mathbb{O}_j$ from its parent origin $\mathbb{O}_i$. And we add an inter-origin HB edge pointing to the first node ($\text{origin}_{\text{first}}$) of $\mathbb{O}_j$. For thread join statement ❿, we create a join node (join) to indicate the end of $\mathbb{O}_j$ that finally joins to $\mathbb{O}_i$. An inter-origin HB edge is created from the last node ($\text{origin}_{\text{last}}$) of current origin ($\mathbb{O}_j$) to the join node.

Existing static race detection (such as [72]) typically checks each pair of two conflict accesses from different threads: run a depth-first search (or breadth-first search) starting from one access and vice versa to check their happens-before relation on the SHB graph, and compute the locksets for both accesses to check whether they have common lock guards.

However, the efficiency is limited by the redundant work in graph traversals and lockset retrievals for all pairs of memory accesses. The straw man approach cannot scale to real-world programs which can generate large SHB graphs with millions of memory accesses.

4.2.1 Three Sound Optimizations

To address the performance challenges, we develop the following sound optimizations:

Check Happens-Before Relation: We only create inter-origin HB edges in the SHB graph. Instead of creating intra-origin HB edges, we assign a unique *integer ID* to each node, which is monotonically increased during the SHB construction. Therefore, we convert the traversal of visiting all intra-origin nodes along HB edges to a constant time integer comparison.

Check Lockset: Intuitively, a list of locks is associated with each memory access node in the

SHB graph in order to represent the mutex protection. We observe that the number of different combinations among mutexes is much smaller than the number of conflict memory accesses we need to check. Therefore, we assign each combination of mutexes (including the empty lockset) a *canonical ID* and associate each access node with such an ID. This not only reduces the memory for storing the SHB graph, but also speeds up the lockset checking process. All memory accesses with an identical lockset ID, or different IDs corresponding to overlapping locksets, are protected by the same lock(s), and the intersection of the IDs between two locksets can be cached for later checks.

Lock-Region-based Race Detection: We observe that a synchronization block or method often guards a large sequence of memory accesses on the *same* origin-shared object(s) (o_s), which incurs redundant race checking. Instead, we treat all the memory accesses on o_s within the same lock region as a single memory access on o_s, and check races on that single access *once*. This is sound because their happens-before relations and locksets are exactly the same. This optimization significantly boosts O2's performance by reducing the number of memory access pairs for detecting data races.

4.2.2 Other Implementation Details

Sequential and Relaxed Memory Models: Different from sequential consistency, a relaxed memory model may reorder certain reads and writes in the same thread and different threads may see different orders. O2 works for both sequential and relaxed memory models. The reason is that the SHB graph captures inter-origin happens-before relations at synchronization sites, and it does not assume a global ordering of reads and writes. Hence, our happens-before relations already relax the ordering constraints for reads and writes from the same origin.

Cross-Module and External Pointers: For C/C++, O2 always links the IR files into a single LLVM module and performs the analysis based on the whole module. Meanwhile, there is always a default origin (starting from the main entry point), so we do not have to deal with cross-module pointers. For JVM applications, O2 extends WALA's ZeroOneCFA to analyze all bytecode-level pointers loaded by the application classloader. When a pointer is passed from an external function

Table 4.4: Performance comparison on C/C++ benchmarks (in *sec.*). The slowdown (*SD*) is normalized with 0-ctx as the baseline. Reprinted From [3].

App	#KLOC	Metrics	0-ctx	O2	2-CFA
Memcached (#O = 12)	20.4	Time/SD	5.3	5.8/9%	7.5/41%
		#Pointer	8,400	12,883	15,772
		#Object	2,420	2,468	2,765
		#Edge	5,395	10,415	17,116
Redis (#O = 15)	116	Time/SD	9.3	15.0/61%	275.9/28x
		#Pointer	44,535	54,690	281,524
		#Object	14,458	14,913	32,401
		#Edge	598,981	963,654	13,530,084
Sqlite3 (#O = 3)	245	Time/SD	213	273/28%	OOM
		#Pointer	57,657	61,796	-
		#Object	10,093	10,310	-
		#Edge	7,909,626	8,879,155	-
Avg.	126	Time/SD	75.8	97.9/30%	-

call for which the IR file does not exist, we will create an anonymous object for that pointer.

4.3 Experiments

4.3.1 Performance

4.3.1.1 Origin-Sensitivity vs Other Pointer Analyses

Table 4.4 reports the performance for three C/C++ applications (*Memcached*, *Redis* and *Sqlite3*). The tested program covers small to large code bases that results in PAG ranging from 5 thousand 7 million edges. origin-sensitive analysis achieves upto 17x speedup over 2-CFA on *Redis* while only incurring 61% slowdown compared with context-insensitive analysis. Moreover, 2-CFA got killed when running on *Sqlite3* due to out of memory (OOM, 32GB) while origin-sensitive analysis only imposes 28% slowdown. On average, origin-sensitive analysis is only 30% slower than context-insensitive analysis while is able to produce much more accurate information for data race detection.

Table 4.5: New Races Detected by O2 (Confirmed by Developers). Reprinted From [3].

	Linux	TDengine	Redis/RedisGraph	OVS	cpqueue	mrlock	Memcached	Firefox	ZooKeeper	HBase	Tomcat
#Races	6	6	5	3	7	5	3	2	1	1	1

4.3.1.2 Race Detection Performance

We also tested RacerD on the three C/C++ programs in Table 4.4. However, RacerD could not run successfully on *Memcached* and *Redis*, and it reports no violations on *Sqlite3*.

4.3.2 New Races Found in Real-World Software

O2 has detected new races in every real-world code base we tested on, as summarized (partially) in Table 4.5. Most of them are due to a combination of threads and events. If considering events only or threads only, or considering them separately, these races will be missed. In the following, we elaborate the races found in several high-profile C/C++, Android apps, and distributed systems.

4.3.2.1 Linux Kernel

We evaluated O2 on the Linux kernel (commit `5b8b9d0c` as of April 10th, 2020), compiled with tinyconfig64, clang/LLVM 9.0. We define four types of origins: system calls with function prefix: `__x64_sys_xx`, driver functions over file operations (`owner`, `llseek`, `read`, `write`, `open`, `release`, etc), kernel threads with origin entries `kthread_create_on_cpu()` and `kthread_create_on_node()`, and interrupt handlers with origin entries `request_threaded_irq()` and `request_irq()`). There are 398 system calls included in our build. For each system call, we create two origins representing concurrent calls of the same system call, and a shared data pointer if the system call has a parameter that is a pointer (e.g., `__x64_sys_mincore`). In total, 1090 origins are created, including 796 from system calls and 294 from others.

In total, O2 detects 26 races in less than 8 minutes. We manually inspected all these races and confirmed that 6 are real races, 7 are potential races, and the other 13 are false positives. The 6 real races are all races to the linux kernel bugzilla, and all of them have been confirmed at the time of writing. The 7 potential races are difficult to manaully inspect due to very complex code paths involving the races. For the false positives, a majority of them are due to mis-recognition of spinlocks (such as `arch_local_irq_save.38`) or infeasible branch conditions which O2 does not handle. The code snippet below shows a real bug found by O2, which detects concurrent writes on the same element of array *vdata* (with array index `CS_HRES_COARSE`).

```
void update_vsyscall_tz(void){//in class time.vsyscall
  struct vdso_data *vdata = __arch_get_k_vdso_data();
  vdata[CS_HRES_COARSE].tz_minuteswest = sys_tz.tz_minuteswest; //RACE
  vdata[CS_HRES_COARSE].tz_dsttime = sys_tz.tz_dsttime; //RACE
  ...
}
```

In addition, we found that among the 71459 allocated objects by the kernel (within the configured origins), 329 of them are origin-shared. And 1051 accesses are on origin-shared memory locations from a total of 36321 memory accesses. The result indicates that the majority of memory used by the kernel is origin-local, which can be beneficial to region-based memory management.

We also discovered that the system call paths do not create any new kernel threads or register interrupts. However, driver functions can do both operations. For example, the driver of GPIO requests a thread to read the events by the kernel API `request_threaded_irq` [2]. And the interrupt requests can create kernel threads by API `kthread_create` [3].

4.3.2.2 Memcached

Memcached is a high performance multithreaded event-based key/value cache store widely used in distributed systems. We applied O2 to commit `14521bd8` (as of May 12th, 2020). O2 is able to finish analyzing memcached within 5s, and reports 16 new races in total. All these races are previously unknown. We manually confirmed that 11 of them are real and the rest of them are potential races. A majority of the real races are on variables such as `stats`, `settings`, `time_out`, or `stop_main_loop`. There are also three races that are not on these variables but look more harmful. We reported the three races to the developers and all of them have been

[2]/linux-stable/drivers/gpio/gpiolib.c@1104:8

[3]/linux-stable/kernel/irq/manage.c@1279:7 and @1282:7

confirmed. The other five potential races all involve pointer aliases on queued items. One of the reported races is shown below with the simplified code snippet:

```
void *do_slabs_reassign(){ ... //event
  if (slabsclass[id].slabs > 1){
    return cur;//RACE: missing lock
  }
}
void *do_slabs_newslabs(){ ... //thread
  pthread_lock();
  p->slab_list[p->slabs++] = ptr;//with lock
  pthread_unlock() ...
}
```

The listed bug is related to Memcached's *slab-base memory allocation*, which is used to avoid memory fragmentation by storing different objects using different *slab classes* based on their size. Since the accesses in the event handler is not protected by the lock, there is a data race between the event handler and all the running threads that try to allocate new slabs. Although another lock-protected check on the same variable is made later in the function, the data race can still lead to undefined behaviors. This case is interesting as it shows that unlike previous tools, which only reason about *inter-thread* races, O2 is able to unify events and threads to find races in complex programs that leverage both concepts for concurrency.

4.4 Summary

We have presented O2, a new system for static race detection. O2 is powered by a novel abstraction, *origins*, that unifies threads and events to effectively reason about shared memory and pointer aliases. Our extensive evaluation with Java and C/C++ programs demonstrates the potential of O2, finding a large number of new races in mature open-source code bases and achieving dramatic performance speedups and precision improvement over existing static analysis tools. O2 has been integrated into Coderrect, a commerical static analyzer [73].

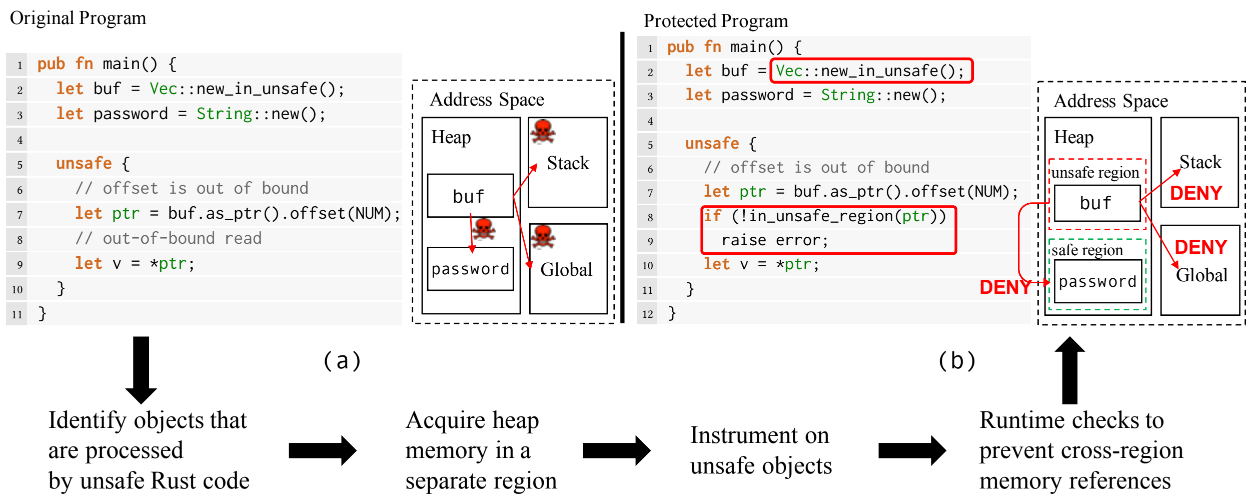
5. SECURING UNSAFE RUST PROGRAMS*

In this chapter, we show that XRust uses pointer analysis to secure unsafe Rust programs. Long-existing systems programming languages such as C/C++ offer programmers the ability to manipulate low-level resources but in error-prone ways. Countless severe bugs have been found due to the unsafe nature of these languages [74, 75, 76]. Rust [21] is a rising language that tries to bridge the gap between memory safety and low-level systems programming. With new language features such as ownership, borrowing, and lifetime, Rust aims to guarantee that a program is memory safe if it could be compiled (in the absence of *unsafe Rust* code). The type system of Rust and its encapsulation of low-level operations have been formally proved to ensure memory safety [22, 23].

However, the static restrictions of Rust can be too strict to admit many valid programs due to reasons including (1) by nature, static analysis is conservative and (2) the underlying computer hardware is inherently unsafe and certain operations could not be done with safe Rust [25]. This problem is addressed by *unsafe Rust*, which escapes from Rust's static checks [24]. With unsafe Rust, programmers are able to manipulate raw pointers, perform unprotected type casting and other dangerous operations just like in C/C++. Therefore, a Rust program is free of memory errors only when its unsafe code is correctly implemented and does not violate memory safety properties [22]. However, requiring all the unsafe Rust code to be correctly implemented is difficult. Bugs in unsafe Rust code may result in severe vulnerabilities, as witnessed by several memory errors discovered recently [77, 78, 79]. What is worse is that a memory error in unsafe Rust may corrupt arbitrary data in the whole address space, e.g., bugs in unsafe Rust can be exploited to hijack function pointers or steal sensitive data in safe Rust.

To understand how the unsafe portion of Rust is used in real-world applications, we randomly selected 500 crates from `crates.io` and counted the number of lines of unsafe code (shown

*Reprinted with permission from "Securing unsafe rust programs with XRust" by Liu, Peiming and Zhao, Gang and Huang, Jeff, 2020. Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, Copyright 2020 by Peiming Liu.


Original Program
Address Space
Heap
Stack
buf
password
Global
(a)
Protected Program
Address Space
Heap
unsafe region
buf
Stack
DENY
safe region
password
DENY
Global
DENY
(b)
Identify objects that are processed by unsafe Rust code
Acquire heap memory in a separate region
Instrument on unsafe objects
Runtime checks to prevent cross-region memory references

```
pub fn main() {
  let buf = Vec::new_in_unsafe();
  let password = String::new();

  unsafe {
    // offset is out of bound
    let ptr = buf.as_ptr().offset(NUM);
    // out-of-bound read
    let v = *ptr;
  }
}
```

```
pub fn main() {
  let buf = Vec::new_in_unsafe();
  let password = String::new();

  unsafe {
    // offset is out of bound
    let ptr = buf.as_ptr().offset(NUM);
    if (!in_unsafe_region(ptr))
      raise error;
    let v = *ptr;
  }
}
```

Figure 5.1: A technical overview of XRust (using instrumentation-based memory isolation). Reprinted From [2].

Table 5.1: Unsafe Rust code in practice (Rust-lang contains the code for Rust compiler and all the Rust standard libraries).

	LoC	**LoC (unsafe)**	**unsafe %**
collected crates	2,480,761	18,490	0.75%
Rust-lang	327,792	3,163	0.96%

in Table 5.1). The result indicates that most real-world Rust programs only rely on a very small fraction of unsafe code ($< 1\%$) on average. Although in practice most memory objects in Rust are statically protected by Rust's type system, a bug residing in unsafe Rust code could simply ruin the entire effort and put the whole program at the risk of being attacked!

In this paper, we present XRust, a novel approach to mitigate the security threat brought by unsafe Rust while imposing minimal overhead to Rust programs. While there exist several prior attempts [80, 81, 82, 83, 84] on C/C++ to retrofit full memory safety of the language (which is often expensive), our goal is not to bring memory safety to unsafe Rust, but to ensure the integrity of data in safe Rust (in the presence of memory errors in unsafe Rust code). In XRust, the heap is logically divided into two mutually exclusive regions: an unsafe region and a safe region. The set of memory objects created in and/or accessed by unsafe Rust (referred to as *unsafe objects*) are recognized by pointer analysis and are placed in the unsafe region and can be corrupted. All other *safe objects* are stored in the safe region and can never be corrupted. The separation between safe and unsafe objects can be enforced by in-process memory isolation techniques [85, 86]. In this work, we explore two methods using instrumentation and memory guard pages respectively to achieve in-process isolation.

As depicted in Figure 5.1, XRust works as follows:

- In the original code, the two objects `buf` and `password` are treated equally and are placed in the same heap region. A heap-based attack exploiting a memory corruption of `buf` in unsafe Rust code can cause arbitrary write to the whole address space, including corrupting `password`;

- In the XRust-protected code, `buf` is placed in the unsafe region separated from `password`, because `buf` is used in unsafe Rust. When using instrumentation, runtime checks are inserted to prevent cross-region data flows from the unsafe region to the safe region. When using guard pages, isolation is enforced by placing inaccessible memory pages between the two regions.

We note that XRust does not attempt to guarantee full memory safety of Rust, but only the safety of memory objects in safe Rust. The main goal of XRust is to provide effective protection while imposing negligible overhead. Also, XRust only targets memory corruption on heap objects but not stack corruption. Stack protection techniques such as stack canaries [87] and SafeStack [88] have been deployed widely in real systems. Proposals [89, 90] to support SafeStack in Rust have also been implemented.

5.1 Overview

In this section, we first discuss the rationale behind the design of XRust. We then illustrate how XRust works on a motivating example based on a real vulnerability in Rust.

The clear separation between safe and unsafe Rust naturally divides heap objects into two *mutually exclusive* sets: the sets of *safe* and *unsafe* objects, based on whether they are used in unsafe Rust. At a high-level view, since only unsafe objects are under the risk of being corrupted in Rust programs, the memory isolation enabled by XRust ensures that potential memory corruptions can only impact the unsafe objects and can never cross the boundary to corrupt safe objects.

In this subsection, we first discuss how unsafe Rust is used in practice, and then discuss the protection strength of XRust with respect to both *spatial* and *temporal* memory safety.

5.1.1 Unsafe Rust in practice

We studied several popular open-source Rust projects as well as Rust's standard libraries to understand the usage of unsafe Rust in the real world.

As summarized in Table 5.1, Rust programs only contain less than 1% unsafe code on average, and unsafe Rust is typically used only for low-level operations and optimizations. The statistics provide strong evidence that most objects are only processed by safe Rust and by isolating the side-effect of unsafe Rust, XRust is able to protect all of them. In addition, we also conducted in-depth inspections of the source code with respect to the usage of unsafe Rust. We summarize our findings into three categories:

Unbounded Memory Accesses. Instead of using object references, programmers sometimes use raw pointers and unchecked pointer arithmetic to access a piece of consecutive memory. *E.g.,* in `base64`, instead of using a `vector`, the developers access the encoding buffer directly through a raw pointer and iterate over the memory by adding offsets to the pointer. This pattern is normally used to access an internal buffer and to skip default bound checkings (in `image`, `base64`, `vec`, etc)

Unchecked Conversions. This includes both *type* conversion as well as *data format* conversion (e.g., *utf-8* to *utf-16*). This is mainly used for developing low-level functionalities such as decoding/encoding binary data and serialization as in `string`, `byteorder`, `bytes`, etc.

Internal States Override. When using safe API, the internal states of an object is normally maintained internally by Rust (e.g., pushing an element into a vector increases the size of the vector). However, when developers access an object in unexpected ways, the internal states need to be manually adjusted. E.g., after initializing the buffer of a vector using raw pointers, the size of the vector needs to be overridden accordingly. The operation is unsafe as programmers are responsible to provide the correct value and unmatched internal states may lead to undefined behaviors. This is typically used for the purpose of low-level optimizations as in `vecdeque`, `vec`, etc.

5.1.2 Observations behind XRust

Based on the empirical studies above, we make two observations:

Observation #1: Being aware that unsafe Rust code is not checked by the compiler, Rust programmers tend to avoid heavy usage of unsafe Rust in practice and only rely on unsafe features to perform necessary low-level operations [91]. This indicates that in reality, it is likely that most objects in a Rust application are safe objects, and critical data such as `password` (with high-level semantics) is unlikely to be processed in unsafe Rust.

Observation #2: Unlike C++ which stores the *virtual function table* (vtable) pointers of an object adjacent to its data members [92], Rust stores them separately. Internally, Rust achieves polymorphism and dynamic dispatching by transforming objects into *trait objects* [93]. As illustrated in Figure 5.2, the reference to a trait object is a *fat pointer* consisting of two pointers: one points to the data members of the object and the other points to the vtable. This implicitly puts the heap data and vtable pointers into two regions. For unsafe objects, only its data members are allocated in the unsafe heap region. Thus, overflow to corrupt vtable pointers is a cross-region reference and will be prevented by XRust.

5.1.3 Protection Strength of XRust

The two observations above lead to the following properties of XRust:

Spatial Memory Safety The observations imply that by preventing cross-region references, XRust can efficiently defend Rust programs against:

1. Non-control data attacks in unsafe Rust code that corrupt objects outside the unsafe region;

2. Control-oriented attacks in unsafe Rust code that corrupt the vtable pointer of a trait object or raw function pointers outside the unsafe region, e.g., to hijack control flow to malicious

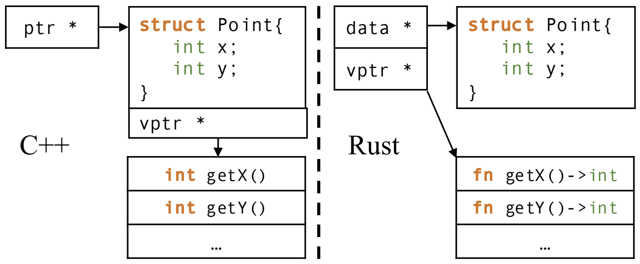


Figure 5.2: Memory layout of objects in C++ vs Rust. Reprinted From [2].

code.

These protections are valuable in practice because (1) there is a high chance that most sensitive data in Rust applications are safe objects (Observation #1), and (2) vtable pointers of trait objects are the major source of indirect jumps in Rust and they are protected by XRust (Observation #2).

Temporal Memory Safety XRust is able to prevent temporal memory errors from corrupting safe objects as well. In Rust, temporal errors can only happen on unsafe objects because safe Rust code statically eliminates all temporal errors by analyzing the *lifetime* of references and the *ownerships* of objects. So, when a temporal error (e.g., use after free) occurs, the pointer used to access memory must point to an unsafe object. Since our multi-region allocator will not reuse memory previously used for unsafe objects to allocate any safe object (Section 5.3.2), the freed memory of an unsafe object will only be used to hold another unsafe object. When a temporal error occurs, the memory access on the freed pointer will still be within the unsafe heap region so that the temporal error cannot escape the unsafe region to corrupt safe objects.

5.1.4 A Motivating Example

Listing 5.1 shows a code fragment simplified from the `rust-base64` library. For versions before 0.5.1, the library contains an integer overflow bug that eventually leads to a heap buffer overflow. On line 4, the vulnerable function first tries to reserve a buffer on the heap and the size of the buffer is calculated by the vulnerable function `encoded_size` containing an integer

```
1  pub fn encode_config_buf<T>(buf: &T, ..) {
2    // reserve a large enough buffer to
3    // store the encoded string
4    buf.reserve(encoded_size(len));
5    // using unsafe operation to store encoded
6    // string to buffer
7    unsafe {
8      // buf object is used in unsafe code!
9      let mut output_ptr = buf.as_mut_ptr();
10     while condition {
11       // do pointer arithmetic and accessing
12       // memory directly
13       ptr::write(output_ptr.offset(..), ...);
14       ...
15     }
16   }
17 }
```

Listing 5.1: A real buffer overflow in `rust-base64` due to unsafe Rust code (CVE-2017-1000430).

overflow[2]. A heap overflow can happen when the integer overflow leads to a smaller buffer and this vulnerability can be exploited to overwrite data in safe Rust. For Rust applications using this library, the unsafe code may only account for a small fraction of the entire code. However, this bug can still result in memory corruptions in the entire address space.

XRust significantly mitigates this vulnerability. It first identifies `buf` as an unsafe object because it is used in unsafe Rust (line 9), by analyzing the data flow from the safe Rust to unsafe Rust. Then instead of reserving heap memory for the objects normally (line 4), it reserves the

[2]Note that Rust checks integer overflows for the debugging build by default, but does not check in the optimized release build.

memory for `buf` in the unsafe region, by rewriting the function to call an extended API. Finally, accesses to `buf`, which is an unsafe object, are restricted in the unsafe memory region. When using instrumentation, the memory reference on line 13 will be instrumented as follows:

```
let ptr = output_ptr.offset(..);
if (!in_unsafe_region(ptr))
  panic!("cross region data flow detected");
write(ptr, ...);
```

At runtime, attempts to access addresses outside the unsafe heap region will be detected by XRust, thus the vulnerability cannot be exploited to perform attacks on safe objects.

We observe that, even with instrumentation which often imposes high overhead for other languages such as C/C++ by other techniques, XRust is still fast (3.6% overhead on median). This is because XRust only checks memory references on unsafe objects, which avoids heavy instrumentation to propagate the meta information as required by techniques such as SoftBound [80], and it avoids the expensive whole-program reaching definition analysis as required by DFI [81] to determine valid data flows. Moreover, XRust checks only cross-region data flow (rather than object bounds), which can be achieved in constant time with the help of our heap allocator (Section 5.3.2). In our design, we also leverage guard pages to protect cross-region references (Section 5.4.2), which is even more efficient than using instrumentation.

Figure 5.3 shows the technical design of XRust, which consists of three key components: 1) extensions made to Rust and the Rust compiler to provide high-level APIs for allocating objects in the unsafe regions; 2) a new heap allocator that supports an unsafe heap region; and 3) runtime protections to prevent cross-region memory references. In the next three sections, we present the details of each component.

5.2 Language Extensions

In this section, we first introduce necessary background on how Rust encapsulates its heap allocation interfaces and then present our extensions.

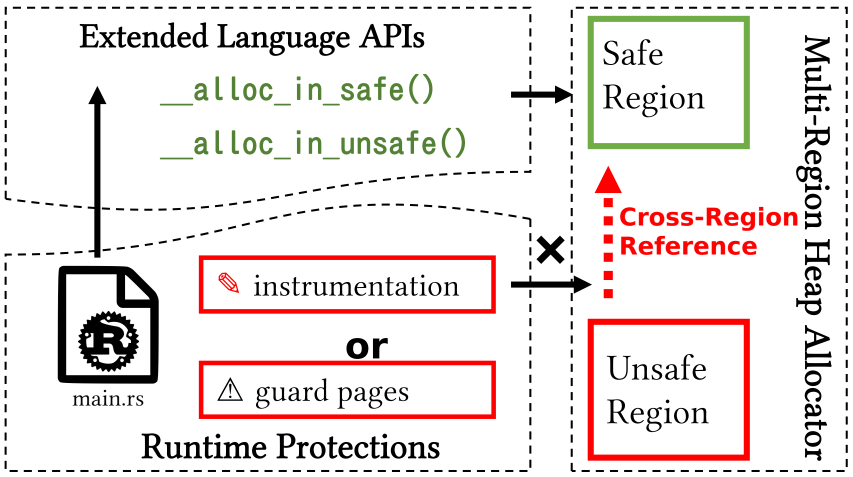


Figure 5.3: Three key components of XRust. Reprinted From [2].

5.2.1 Heap Allocation in Rust

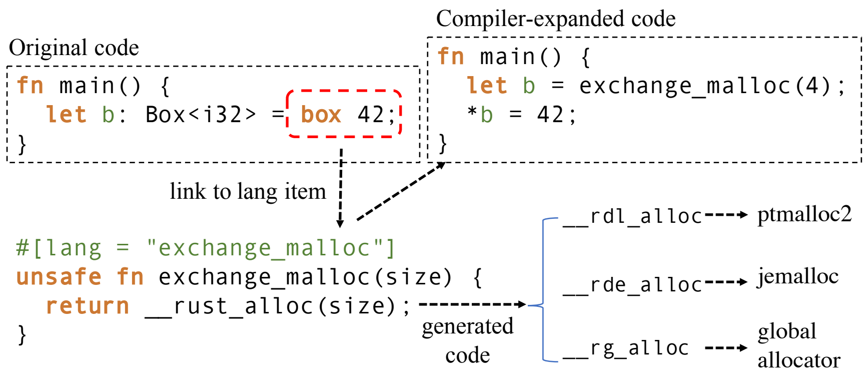
Instead of allowing programmers to acquire and release heap memory directly through `malloc` and `free`, Rust provides high-level abstractions on heap memory through encapsulation on heap operations. The release of a heap object is automatically inserted by Rust compiler and programmers are not allowed to free the memory manually to avoid errors like double frees. It also gives Rust the flexibility of changing the allocator globally (even for pre-compiled libraries) without recompiling the code by defining a *global allocator* [3] [94]. These encapsulations and the loose connection between the language and the allocator implementation require extra abstraction layers between these two components.

There are two ways to acquire a piece of heap memory in Rust[4]. In most cases, this can be achieved by creating a `Box<T>` object. For low-level library developers, it could be done by directly interacting with the `Alloc` *trait* (trait is similar to Java's *interface*). The `Box<T>` objects are wrapped pointers that can only point to heap objects and are internally created using `box` expressions[5]. For example, the expression `box 42` allocates four-byte heap memory that stores a

[3]The feature of switching allocators globally is not in a stable state yet. The description in this paper is based on the latest Rust (version 1.32) by the time of writing.

[4]Calling malloc-like function through FFI is out of the scope.

[5]box expression is an unstable feature as well.



```
fn main() {
  let b: Box<i32> = box 42;
}
```

```
fn main() {
  let b = exchange_malloc(4);
  *b = 42;
}
```

```
#[lang = "exchange_malloc"]
unsafe fn exchange_malloc(size) {
  return __rust_alloc(size);
}
```

Figure 5.4: Rust workflow for linking heap allocations. Reprinted From [2].

32 bit integer of value 42, and it returns a `Box<i32>` object pointing to the allocated heap object as the result. Those `Box<T>` objects will be dropped later by the compiler-inserted code when their owners go out the scope, i.e., the owner function returns or the owner block terminates. In Rust's standard libraries, neither `box` expressions nor the default implementation of the `Alloc` trait is bounded to a specific allocator. They both rely on the Rust compiler to generate glue code to bind the program to a specific allocator during code generation phase.

For heap allocation through the `Alloc` trait, the default implementation delegates all its tasks to a set of functions with the `__rust` prefix. Specifically, `__rust_alloc()` for heap allocation, `__rust_dealloc()` for heap deallocation, and `__rust_realloc()` for heap reallocation, etc. These functions do not have actual implementations, but are treated as special internal symbols by the Rust compiler and implemented by compiler generated code to invoke different allocators, e.g., the allocator for static libraries and for executable binaries.

For heap allocation through `box` expressions, it requires two *lang items*: "`exchange_malloc`" for allocation and "`box_free`" for deallocation. Lang items [95] are pluggable features in Rust whose functionalities are not hard-coded into the language but are implemented in libraries, using a special marker (`#[lang = "..."]`) to indicate their existence. Figure 5.4 illustrates the workflow. At compile time, for each `box` expression, the Rust compiler searches all the dependent libraries to find functions marked by these two lang items. The compiler then generates code by calling the function marked as `exchange_malloc` to allocate heap memory, and inserts calls to the function marked as `box_free` to drop `Box<T>` objects. In Rust's standard libraries, the default implementation of `exchange_malloc` delegates heap allocation to `__rust_alloc()`.

5.2.2 Language Support for Unsafe Region

To support a different heap region, we add corresponding "`unsafe`" interfaces for each of the allocation functions. For example, we add `__rust_unsafe_alloc` as the entry point for allocating heap memory in the unsafe region. The compiler is also extended to generate code to invoke these extended functions for handling the unsafe heap region.

We then build high-level APIs for the extended interfaces by extending Rust's standard library. Additional methods are added to the `Alloc` and `GlobalAlloc` traits to deal with the unsafe heap region. For example, the function `unsafe_alloc()` is added to the `Alloc` trait to provide interfaces for allocating memory in the unsafe region. Based on this, high-level classes in the standard libraries can be extended as well. For example, `Vec::unsafe_with_capacity()` is added to the `Vec` structure to create a vector that puts the internal memory buffer in the unsafe heap region, which allows programmer to interact with unsafe allocation interfaces on their own demands.

The newly added interfaces are backward compatible with existing Rust programs. By default, calls to the extended interfaces (e.g., `unsafe_alloc()`) are delegated to the pre-existing functions (e.g., `alloc()`). The compiler-generated code also delegates the requests from `__rust_unsafe_alloc` to the standard API if the underlying allocator does not support a separate unsafe region. In this way, all existing Rust programs can be compiled without modification. When programmers use the extended interfaces but with an allocator that does not support the unsafe region, the allocation can still be completed, but the allocated heap chunks will not be placed in a separate unsafe heap region. The default implementation is then overridden by our extended allocator and linked properly by the compiler. Invocations on them are passed to the proper API to allocate and free heap memory in the unsafe region.

For `box` expressions, we add a new operator `unsafe_box` to create a `Box` object in the unsafe heap region. The grammar of `unsafe_box` expressions is identical to `box` expressions, and the result of `unsafe_box` expressions has the same type (`Box<T>`) as the result of `box` expressions. Generating the same type ensures that the `unsafe_box` operator can fit into the existing Rust type system. The only difference between `Box` objects created by `box` and `unsafe_box` expressions is that internally they are put into different heap regions, but all other operations (dereference, type casting, pattern matching, etc.) are identical. Similarly, `unsafe_box` will be linked to a new lang item `unsafe_exchange_malloc` at compile time, which handles the allocation of unsafe objects.

5.3 Multi-Region Heap Allocator

Our allocator implementation is based on `ptmalloc2` [96] and supports multi-threading. We first introduce the architecture of `ptmalloc2` and then present our extensions.

5.3.1 Architecture of ptmalloc2

`ptmalloc2` was forked from `dlmalloc` [97] and later merged into `glibc` with threading support. `ptmalloc2` maintains separate heap segments and freelist data structures using multiple *per-thread arenas*, such that threads can rely on different arenas to perform heap allocation/deallocation simultaneously without synchronization.

In `ptmalloc2`, each arena can manage a list of heap segments (except for the main arena, which only has one heap segment). A heap segment is a large piece of mmapped memory from where free chunks are retrieved and returned to users. Arenas also keep the freelist data structures of their heap segments (i.e., *bins*) used to hold free chunks. Bins are divided into four different types based on chunk sizes: *fast*, *unsorted*, *small*, and *large*, and each is handled differently. To handle a heap allocation, `ptmalloc2` chooses the appropriate bins based on the requested size. More details can be found in [98].

5.3.2 XRust Extensions on ptmalloc2

In our design to extend `ptmalloc2` for handling heap (de)allocation in the unsafe region, we followed most of its current design. The interactions with the unsafe heap region are achieved through extended APIs such as `unsafe_malloc()`. These APIs are merged into Rust and linked with extended language APIs. The data structures used by the unsafe region are lazily initialized upon the first request for allocating memory in the unsafe region. For applications that do not use the unsafe region, the extended allocator acts the same as unmodified `ptmalloc2` and no overhead is imposed.

Unsafe Region in Heap. In our heap allocator, the set of *arenas* for handling allocations in unsafe region and those for allocations in safe region are disjoint, i.e., the unsafe arenas will not be reused for allocating objects in safe region and *vice versa*. This ensures that for every internal heap segment managed by the allocator, it only contains the objects in the same region so that overflow originated from unsafe objects will not corrupt safe objects.

We extend the architecture of `ptmalloc2` to enable fast checks on cross-region errors. Intuitively, cross-region references can be checked by determining whether the pointer is within the range of any unsafe heap segment. This could lead to huge runtime overhead since the number of unsafe heap segments is unbounded (especially for multi-thread programs). To address the issue, we use a pre-allocated bitmap to record the type of heap segments (*safe* or *unsafe*), which can be quickly indexed by the start addresses of heap segments. This introduces negligible memory overhead since the heap segments in `ptmalloc2` is 1 megabytes aligned by default, thus the memory overhead is 1 bit *per* megabyte. The bitmap is protected by `PROT_READ` and can only be accessed inside the allocator upon the creation of an unsafe heap segment. Under this design, checking an memory reference takes only constant time regardless how many unsafe heap segments have been allocated.

Multi-thread Support. To maximize the performance of multi-thread programs, we adopt the *per-thread arena* mechanism to allow accessing the free lists for unsafe heap region concurrently. For multi-thread programs, threads are assigned with different arenas to allocate heap memory. Since every arena manages a disjoint set of heap segments, they can be accessed concurrently without synchronization.

Our design of the multi-region heap allocator also renders time-of-check-to-time-of-use (TOCT-

TOU) attacks almost impossible. To trigger such an attack, an unsafe pointer needs to be verified to be within unsafe region first (time of check) and later be used to corrupt a safe object (time of use) because the unsafe object is first freed and the same address is reused for a safe object by other threads before the time of use. However, since unsafe heap segments are maintained separately by different arenas in our allocator, a freed unsafe chunk will only be reused to hold another unsafe object, which makes the attack difficult. Besides, Rust prohibits programmer from calling `drop` manually to deallocate objects to avoid errors, which makes it even harder to launch the attack.

Cross-Region References inside Allocator. To fully prevent cross-region memory references, the allocator need to be free of cross-region errors as well. For example, memory errors can be exploited to corrupt the metadata of heap chunks because `ptmalloc2` stores the metadata adjacent to user data [99].

To address this problem, we insert runtime checks to ensure that the unsafe region inside the allocator would never be able to reference data outside the region. For example, the free chunks in the unsafe region would only be linked to other free chunks within the unsafe region. Whenever the allocator attempts to access the metadata of chunks in the unsafe region, checks are added to ensure that the allocator can never perform cross-region references based on corrupted data.

5.4 Cross-Region Reference Prevention

Cross-region memory references can be prevented by in-process memory isolation techniques, which have already been widely studied. It has been shown that isolation can be enforced with negligible overhead through hardware-based protection, e.g., by using Intel MPK [85] or ARM memory domain [86, 100]. In our prototype implementation, we explored two schemes to detect cross-region references: 1) instrumenting memory references on unsafe objects; and 2) utilizing memory protection pages (i.e., guard pages) to detect overflows.

5.4.1 Code Instrumentation

We first perform an inter-procedural data flow analysis to identify the allocation sites of unsafe objects in Rust programs, based on a recent data flow framework [101]. Any allocated object that is later accessed by unsafe code is considered as an unsafe object. Every allocation site is a taint source, and every unsafe instruction is a taint sink. We record every object that flows from a source to a sink. Based on the results, we rewrite the program to allocate objects in the unsafe memory region.

Shared Unsafe Objects in Safe Rust. We also revealed a crucial technical caveat during the process of developing XRust: To completely isolate the side effect of unsafe Rust code, the instrumentation should be applied not only on unsafe Rust code but *all unsafe objects*, which means that not only the data directly touched by unsafe code, but also everything transitively reachable from such data needs to be instrumented.

The following code creates a vector of length 3 on line 1 and calls an unsafe function `set_len()` on line 3 to set the length of the vector to 10 manually (without resizing the buffer).

```
1 let v = vec![1,2,3];
2 unsafe {
3   v.set_len(10);
4 }
5 let elem = v[9];
```

The memory reference on line 5 is an out-of-bound read because the vector has only allocated the memory space for storing three integers. The code passes the Rust compiler because the vector length is changed by unsafe code. Moreover, no exception will be thrown at runtime by the assertion inserted for the memory reference on line 5, which only checks if the vector index is less than the vector length.

The example above shows how unsafe Rust can be used to override the internal states of an object, which can lead to a memory corruption outside unsafe Rust (but on unsafe objects). Therefore, to provide complete protection, all memory references on unsafe objects (inside or outside unsafe Rust) need to be checked. In fact, one of the real vulnerabilities in Rust (`VecDeque`, see Section 5.5.4) belongs to this category. This also indicates that the existing work (FC [102]) fails to provide a complete isolation from unsafe Rust code because it only protects unsafe Rust code.

For instrumentation, we apply a context-insensitive pointer analysis [4] to identify memory ref-

```
let mut ptr;
if (condition) {
  ptr = __rust_alloc();
  shadow[ptr] = SAFE;
} else {
  ptr = __rust_unsafe_alloc();
  shadow[ptr] = UNSAFE;
}
if (shadow[ptr] == UNSAFE) {
  if(!in_unsafe_region(ptr))
    raise error;
}
let v = *ptr;
```

Listing 5.2: An example for distinguishing between safe and unsafe objects.

erences on unsafe objects (a further improvement might be using origin-sensitive pointer analysis instead to improve the precision of the pointer analysis). Since static pointer analysis is conservative, the points-to set of a pointer can contains both safe and unsafe objects. To address this issue, we use shadow memory to mark the types of pointers and only perform checks on pointers of unsafe objects at runtime.

Listing 5.2 shows an example. The points-to set of `ptr` contains both safe and unsafe objects. To check only references on unsafe objects, a shadow memory is allocated and indexed by the pointer's address. This method is inspired by SoftBound [80], but instead of storing bound information of a pointer in shadow memory, we only use 1 bit to store whether a pointer points to an unsafe object at runtime. Compared to SoftBound, this method incurs much lower space overhead: As heap objects managed by `ptmalloc2` are 16 bytes aligned, XRust imposes at most 1 bit overhead for 16 bytes memory, thus the memory overhead is $< 1\%$. The protection on shadow memory can be done by using approaches discussed in CPI [88] with negligible overhead.

Table 5.2: Performance of XRust and DFI on real-world Rust applications and standard Rust libraries (grayed rows). Reprinted From [2].

App	Ver.	LoC	#Download	Native (ms/iter)	XRust g-page	XRust overhead	XRust inst.	XRust overhead	DFI exec.	DFI overhead
base64	0.5.1	2K	2.32M	3527.72	3529.59	0.06%	7167.63	103.15%	9721.60	175.58%
byteorder	1.2.7	2.3K	4.70M	25.91	26.01	0.03%	26.76	3.28%	64.53	149.05%
json	0.11.13	4.3K	0.39M	2213.17	2260.96	2.16%	2298.91	3.87%	12985.72	486.75%
bytes	0.4.10	7.9K	1.80M	6.24	6.84	9.62%	7.25	16.19%	33.471	436.39%
image	0.20.1	13.3K	0.54M	2151.26	2152.87	0.07%	2189.92	1.77%	13426.83	524.14%
regex	1.0.6	48.1K	6.03M	2157.80	2162.48	0.22%	2187.68	1.17%	15251.78	606.82%
Median	-	-	-	2154.53	2157.68	0.15%	2188.80	3.6%	11353.66	461.57%
Average	-	-	-	1680.35	1689.79	2.03%	2313.03	21.57%	8580.66	396.46%
vec	1.30.0	-	-	0.40	0.42	4.08%	0.90	123.08%	4.32	555.00%
string	1.30.0	-	-	2.00	2.03	1.52%	4.16	108.30%	5.57	178.50%
linked-list	1.30.0	-	-	0.16	0.17	6.76%	0.20	13.70%	0.52	225.00%
vec-deque	1.30.0	-	-	0.71	0.71	1.13%	0.72	2.26%	4.01	464.79%
btree	1.30.0	-	-	21.97	22.58	2.80%	23.88	8.69%	114.81	422.58%
Median	-	-	-	0.71	0.71	2.80%	0.90	13.70%	4.32	422.58%
Avg.	-	-	-	5.05	5.18	3.26%	5.92	51.21%	25.85	369.17%

5.4.2 Guard Page

A more efficient approach can be implemented by imposing two guard pages below and above each heap segment. Since the guard pape cannot be accessed, cross-region references can be detected when it touch the guard page. To bypass this protection, cross-region references must stride across an entire guard page to avoid being detected.

This approach is often more efficient than code instrumentation, though in theory guard page is incomplete (e.g., a direct long jump from the unsafe region to the safe region without touching the guard page). There are reports on how the Linux's stack guard page can be bypassed to launch attacks [103], and it could be mitigated by enlarging the size of guard pages [104]. Nevertheless, complete and efficient hardware-based techniques such as Intel MPK and ARM memory domains can also be integrated into XRust, as explored in recent work [85, 86].

Using guard pages also avoids pointer analysis needed by instrumentation. Unlike instrumentation, which requires pointer analysis to locate unsafe objects to insert assertions before memory accesses on them, guard page enforces isolation automatically after the objects are allocated into different regions. If a cross-region data flow occurs on an unsafe object, the guard page will be accessed and a segment fault will be issued automatically by the operating system.

5.5 Evaluation

We have conducted extensive experiments to evaluate the effectiveness and efficiency of XRust. To evaluate the efficiency, we deployed XRust on six widely-used real-world applications. We also studied five core components of Rust's standard library where unsafe Rust is used ubiquitously to examine XRust in extreme cases. We measured the overhead of XRust under two protection schemes: guard page and instrumentation. Our experimental results show that XRust incurs 0.15% overhead on median (2% on average) when applying guard pages and 3.6% overhead on median (21% on average) when using instrumentation. We also compared XRust with DFI [81] to understand the overhead that could be introduced by imposing a full protection of data-flow integrity.

To evaluate the effectiveness, we studied all the three publicly reported memory corruption errors that we could find in real Rust programs [77, 78, 79]. We designed attacks to exploit these errors and applied XRust to defend against them.

The experiments ran on an AMD Ryzen 2600X with 6 cores@3.6GHz processor in 64 bit mode with 32GB RAM. All experiments were done on Ubuntu Bionic Beaver (18.04 LTS).

5.5.1 Efficiency

All the real-world Rust applications are popular projects (with more than one million downloads) collected from `crates.io`, the official package central repository of Rust, and they all contain unsafe Rust code. We use their built-in benchmarks to measure the performance of XRust for fairness. All the Rust standard libraries are measured using the benchmarks from the Rust compiler. The results are reported in Table 5.2 (averaged over 50 runs).

When using guard page, the overhead comes from the inserted checks performed in the heap allocator to avoid errors caused by corrupted metadata in the unsafe region (discussed in Section 5.3.2). Most cross-region references outside the heap allocator are automatically detected and reported by the operating system upon illegal accesses on guard pages. This approach also introduces 8 KB (two pages) memory overhead for each unsafe heap segment to place guard pages right below and above every unsafe segment.

As reported in Table 5.2, the overhead is negligible for most real-world applications (less than 0.5% for `base64`, `byteorder`, `image` and `regex`). One important factor that affects the performance is the frequency of heap allocations performed in the unsafe heap region. The highest overhead (9.6%) is reported on `bytes`, which heavily relies on unsafe heap allocation. We discuss the allocation statistics in Section 5.5.2.

When using instrumentation, the overhead is higher than using guard pages, because it requires a region check before each memory reference on unsafe objects. Nevertheless, the overhead is still low in most cases (less than 5% for `base64`, `byteorder`, `image` and `regex`, and 16% for `bytes`). The highest overhead (103%) is reported on `base64` (different from that of using guard pages). By analyzing the instrumented program, we found the reason is that in `base64` the checks are inserted into a performance-critical function, which occupies over 98% execution time of the program.

For the Rust standard libraries, the overhead is slightly higher than the real Rust applications, with approximately 3% for guard pages and 50% for instrumentation. The reason is that to bridge between unsafe low-level operations and high-level Rust language features, Rust's standard library typically uses more unsafe Rust code, which increases the number of inserted runtime checks. Also, the performance overhead is highly-related to the tests performed on the benchmarks. Since we used the original test suites (for evaluation fairness) and the tests examine different aspects of the benchmarks, it could lead to different performance numbers. For example, one of the four test cases for `vec-deque` aims at testing the speed of allocating new objects, which imposes little overhead as no memory references need to be instrumented. It explains the differences between the overhead reported in Table 5.2.

Comparison to data-flow integrity (DFI). DFI [81] provides a strong protection against control and data attacks, by ensuring the integrity of data flows at runtime with respect to a statically computed data-flow graph. Unfortunately DFI incurs prohibitive overhead in practice (e.g., around

Table 5.3: Allocation statistics in safe and unsafe heap regions. Reprinted From [2].

App	#allocation (safe)	#allocation (unsafe)	% unsafe allocation
base64	57.50M	25.41M	30.64%
byteorder	13.47K	4	0.02%
json	18.07M	0.39M	2.11%
bytes	34.06M	126.98M	78.85%
image	9.21M	10	0.00%
regex	19.83M	675	0.00%
Avg.	23.11M	25.46M	18.60%
vec	611.89M	37.49M	5.78%
string	516.80M	40.69M	7.21%
linked-list	2.93K	68.70M	99.95%
vec-deque	1.40K	12.54M	99.98%
btree	2.99K	20.51K	87.28%
Avg.	115.21M	42.26M	60.04%

4X runtime overhead on average in our experiments[6]). In their original work, Castro et al. use a static reaching definition analysis to determine the set of write instructions for each memory read, and maintain a runtime definition table (RDT) to record the last write instruction to each memory location at runtime. This incurs both large runtime overhead (for checking all reads and writes) and space overhead (for storing the RDT) even after several optimizations. Differently, XRust ensures the data-flow integrity from unsafe Rust to safe Rust by isolating the unsafe memory region, thus it is much faster (over an order of magnitude) than the full DFI, as reported in Table 5.2.

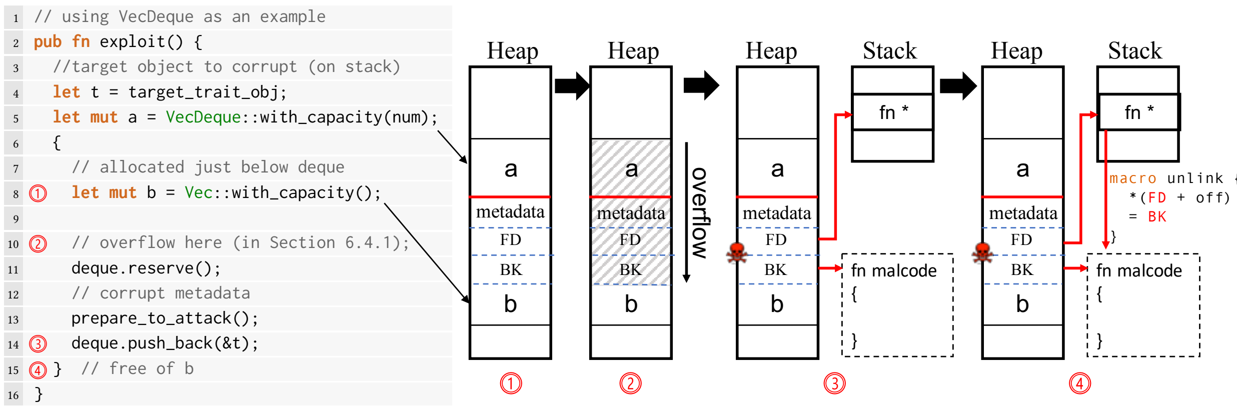
5.5.2 Allocation Statistics

Table 5.3 reports the statistics of heap allocations in safe and unsafe regions in our experiments. For most applications, they only use unsafe Rust in limited locations and thus the allocations in the unsafe region only account for a small fraction of the total amount of heap allocations. One exception is `bytes`, which has around 80% allocations in unsafe code. It is because `bytes` is a library that deals with low-level data structure. It relies on unsafe Rust heavily to access the low-level binary data. For other applications such as `regex` and `json`, almost all objects are safe.

[6]The DFI prototype was implemented by ourselves on top of LLVM following the paper [81], since DFI is not available.

Table 5.4: Performance of the heap allocator with different numbers of unsafe heap segments. Reprinted From [2].

Size (byte)	#Thread: 1			#Thread: 2			#Thread: 4			#Thread: 8			**Avg.**
	ptm-alloc [1]	*unsafe ext.*	*over-head*	*ptm-alloc*	*unsafe ext.*	*over-head*	*ptm-alloc*	*unsafe ext.*	*over-head*	*ptm-alloc*	*unsafe ext.*	*over-head*	*over-head*
16∼1k	25.9M	24.5M	5.3%	5.2M	5.1M	1.3%	3.8M	3.5M	6.6%	2.3M	2.2M	3.9%	4.3%
32∼2k	25.6M	23.7M	7.8%	4.3M	4.2M	3.7%	2.7M	2.7M	0.5%	2.2M	2.2M	0.3%	3.1%
64∼4k	16.0M	15.5M	3.1%	1.7M	1.6M	5.5%	1.3M	1.2M	8.3%	1.9M	1.8M	5.3%	5.6%
128∼8k	14.9M	13.8M	7.9%	1.4M	1.3M	11.4%	1.0M	1.0M	3.8%	1.8M	1.7M	7.9%	7.7%
256∼16k	13.7M	12.9M	6.5%	1.1M	1.1M	3.3%	1.2M	1.1M	7.9%	1.7M	1.6M	6.4%	6.0%
Avg.	19.2M	18.1M	6.1%	2.8M	2.7M	5.0%	2.0M	1.9M	5.4%	2.0M	1.9M	4.7%	5.3%



```
// using VecDeque as an example
pub fn exploit() {
  //target object to corrupt (on stack)
  let t = target_trait_obj;
  let mut a = VecDeque::with_capacity(num);
  {
    // allocated just below deque
①   let mut b = Vec::with_capacity();

②   // overflow here (in Section 6.4.1);
    deque.reserve();
    // corrupt metadata
    prepare_to_attack();
③   deque.push_back(&t);
④ }  // free of b
}
```

Figure 5.5: A proof-of-concept attack performed on `VecDeque`. Reprinted From [2].

The data also confirms our observation that in high-level user applications, programmers typically avoid heavy use of unsafe Rust.

For the standard Rust libraries, the statistics are the opposite. For three out of the five libraries, almost all the objects are unsafe. However, this is not surprising since unsafe Rust is widely used in standard libraries to deal with low-level operations.

5.5.3 Performance of the Allocactor

The customized heap allocator is a core part of XRust and the checks inserted inside the allocator can affect performance. To quantify its performance, we have heavily tested the allocator using a benchmark from `rpmalloc` [105] and compared with unmodified `ptmalloc2`. In our settings, the benchmark iterates 20,000 times in total and in each iteration it allocates and frees

30,000 heap objects of various sizes. In addition, all the objects are allocated via extended interfaces and placed in the unsafe region. This experiment could be viewed as a worst case stress testing since the only functionality of the benchmark is to allocate and deallocate heap memory, and hence provides insights on the worst case performance of the allocator.

The results are reported in Table 5.4. The overall performance of the extended allocator is about 5% slower than `ptmalloc2` on average when tested with 1, 2, 4 and 8 threads.

5.5.4 Effectiveness on Real Vulnerabilities

By the time of writing, we found three reported memory corruption errors in real-world Rust programs. We carefully studied each of them and found that XRust is capable of preventing all these errors.

Corruption in `VecDeque`: `VecDeque` is a double-ended queue implemented with a growable ring buffer and it is a part of Rust's standard library. A buffer overflow vulnerability (CVE-2018-1000657) was discovered inside the `VecDeque::reserve` function only recently. The simplified code is listed below:

```
1 pub fn reverse(&mut self, additional: usize) {
2   let new_cap = used_cap + additional;
3   if new_cap > self.capacity() {
4     self.buf.reserve(..);
5     unsafe {
6       self.handle_cap_increase(..);
7 } } }
```

The root cause of the bug is on line 4, where the function mixes up its internal capacity with its user-visible capacity. Because the user-visible capacity is one element smaller than the actual size of the buffer, The unsafe function `handle_cap_increase` can cause the pointer to point to out-of-bound memory address and upon next push, a value can be written outside the buffer.

The vulnerability can be exploited to overflow one element outside the buffer, Because there is no public attack on this vulnerability, we manually built a proof-of-concept case with a vulnerable program using the function, and performed an `unsafe unlink` exploit [106] to make an arbitrary write to vtable pointers as in Figure 5.5. This attack would fail on recent glibc since extra security checks were added into the library. Workaround to bypass the checks could be found in [106]. The result shows that XRust is able to detect the attack consistently because both the stack and the data segment are outside the unsafe heap region. A cross-region write to corrupt the stack data and vtable pointers is detected by the heap allocator since the metadata is corrupted by the overflow.

Corruption in `str::repeat`: A buffer overflow bug was reported in the function `str::repeat` (CVE-2018-1000810), which is also a part of Rust's standard library. The root cause of the bug is an instance of integer overflow to buffer overflow bugs. The simplified code is listed below:

```
1  pub fn repeat(&self, u: usize) -> Vec<T> {
2    let mut buf = Vec::with_capacity(n * len);
3
4    while condition {
5      unsafe {
6        ptr::copy(buf.as_ptr(),
7                  buf.as_ptr().add(len),
8                  len);
9        ...
10 }   }   }
```

The function is used to create a string that repeats a fixed number of times. On line 2, when calculating the capacity of the `Vec` to hold the string by $n * len$, an integer overflow could happen, which in turn results in a smaller buffer and causes an overflow when using unsafe code to store the value on line 6. We similarly conducted the same proof-of-the-concept attack on it as on `VecdDeque`, and XRust can detect the overflow as well.

Corruption in `Base64`: The details of this error (CVE-2017-1000430) have been presented in Section 5.1.4. Attacking this vulnerability is more difficult than the previous two cases, because it requires triggering an overflow on a 64 bit integer. To perform a proof-of-concept attack, we changed the `Base64` code to use 16 bit integer. The experiment setting is similar to the other two cases, and XRust is able to defend the attack in this case as well.

5.6 Discussions and Limitations

We note that XRust targets memory safety issues brought by unsafe Rust only. XRust assumes Rust's memory safety guarantees to be valid, which requires a correct design and implementation of Rust and its framework (including its standard libraries) so that no memory error will occur in the absence of unsafe Rust code. This is essential because safe abstractions provided by programming languages are inherently encapsulations on unsafe operations. Attackers cannot modify the code segments since they are unwritable and they cannot control the program's loading process. These requirements ensure that the integrity of the instrumented dynamic checks and the heap allocator can safely set up the isolation between safe and unsafe memory regions.

XRust does not handle dynamic code generation. This is a difficult problem because the new code cannot be analyzed or instrumented statically by a compiler. This limitation is shared by techniques relying on static analysis, e.g., SoftBound [80], DFI [81], and CPI [88]. A potential solution is to track dynamically generated code and continue the analysis at runtime. We leave it as future work.

5.7 Summary

We have presented XRust, a novel approach to protect safe memory objects in Rust from being corrupted by unsafe Rust code. The key idea is to separate the address space of a Rust program into two non-overlapping regions with a customized heap allocator, and then automatically insert runtime checks to efficiently detect cross-region references on unsafe objects. Our extensive evaluation on both popular real-world Rust applications and standard Rust libraries shows that XRust is highly effective and efficient: it prevents attacks on the all the known Rust vulnerabilities while exhibiting small or negligible overhead. We stress that it is promising to apply XRust to secure Rust applications in practice.

[1]Measured by the number of memory operations per CPU second.

6. CONCLUSION

This thesis makes contribution to improving precision and performance of inclusion-based pointer analysis. For precision, this thesis proposed origin-sensitivity as a new context for data race detection, which is able to compute the thread-sharing information accurately. For performance, this thesis proposed partial update solver, a new solving algorithm for inclusion-based pointer analysis. Both of techniques leads to promising experimental results: O2, origin-sensitivity enabled race detector, was able to find tens of previously unknown bugs in popular real-world applications including Linux. PUS, the new solver, was able to achieve over 2x speedup for context-insensitive pointer analysis and 7x speedup for context-sensitive (1-CFA) analysis.

Compared to existing context-sensitive abstractions, origin-sensitivity provides better precision and performance when used for static race detection (and potentially other applications that need data sharing information between components). We observed that instead of spawning different contexts on each call site, it is more precise to use only *crucial* contexts (*origin entry points*) to distinguish functions statically.

Compared to existing inclusion-based pointer analysis solving algorithms, PUS significantly improve the performances by only operates on a small *causality subgraph* at each iteration. We observed that only a small portion of the constraint graph are subject to update at each iteration and topological sort needs only to be done on the small subgraph to boost the performance.

Lastly, we proposed XRust, which relies on the information computed by pointer analysis to forbid illegal data flows between safe and unsafe Rust. Our evaluation showed that only negligible overhead was imposed on the target program and XRust can effectively protect against previously found memory corruption CVEs.

REFERENCES

[1] P. Liu, Y. Li, B. Swain, and J. Huang, "Pus: A fast and highly efficient solver for inclusion-based pointer analysis," in *Proceedings of the ACM/IEEE 44nd International Conference on Software Engineering*, 2022.

[2] P. Liu, G. Zhao, and J. Huang, "Securing unsafe rust programs with xrust," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 234–245, 2020.

[3] B. Liu, P. Liu, Y. Li, C.-C. Tsai, D. Da Silva, and J. Huang, "When threads meet events: efficient and precise static race detection with origins," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pp. 725–739, 2021.

[4] Y. Sui and J. Xue, "Svf: interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th international conference on compiler construction*, pp. 265–266, 2016.

[5] R. Bodík and S. Anik, "Path-sensitive value-flow analysis," in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 237–251, 1998.

[6] Q. Shi, X. Xiao, R. Wu, J. Zhou, G. Fan, and C. Zhang, "Pinpoint: Fast and precise sparse value flow analysis for million lines of code," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 693–706, 2018.

[7] B. Liu and J. Huang, "D4: fast concurrency debugging with parallel differential analysis," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 359–373, 2018.

[8] Y. Li, B. Liu, and J. Huang, "Sword: A scalable whole program race detector for java," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 75–78, IEEE, 2019.

[9] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta, "Fast and accurate static data-race detection for concurrent programs," in *International Conference on Computer Aided Verification*, pp. 226–239, Springer, 2007.

[10] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for java," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 308–319, 2006.

[11] Y. Sui, D. Ye, and J. Xue, "Static memory leak detection using full-sparse value-flow analysis," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pp. 254–264, 2012.

[12] G. Fan, R. Wu, Q. Shi, X. Xiao, J. Zhou, and C. Zhang, "Smoke: scalable path-sensitive memory leak detection for millions of lines of code," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 72–82, IEEE, 2019.

[13] Y. Xie and A. Aiken, "Context-and path-sensitive memory leak detection," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 115–125, 2005.

[14] S. Cherem, L. Princehouse, and R. Rugina, "Practical memory leak detection using guarded value-flow analysis," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 480–491, 2007.

[15] S. Horwitz, "Precise flow-insensitive may-alias analysis is np-hard," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 19, no. 1, pp. 1–6, 1997.

[16] G. Ramalingam, "The undecidability of aliasing," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 5, pp. 1467–1471, 1994.

[17] L. O. Andersen, *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Cophenhagen, 1994.

[18] B. Steensgaard, "Points-to analysis in almost linear time," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 32–41, 1996.

[19] F. M. Q. Pereira and D. Berlin, “Wave propagation and deep propagation for pointer analysis,” in *2009 International Symposium on Code Generation and Optimization*, pp. 126–135, IEEE, 2009.

[20] B. Hardekopf and C. Lin, “The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code,” in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 290–299, 2007.

[21] S. Klabnik and C. Nichols, *The Rust Programming Language*. No Starch Press, 2018.

[22] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, “Rustbelt: Securing the foundations of the rust programming language,” *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, p. 66, 2017.

[23] E. Reed, “Patina: A formalization of the rust programming language,” *University of Washington, Department of Computer Science and Engineering, Tech. Rep. UW-CSE-15-03-02*, 2015.

[24] Rust-Team, “Unsafe rust,” 2017. `https://doc.rust-lang.org/book/second-edition/ch19-01-unsafe-rust.html`.

[25] cmr, “Unsafe rust,” 2018. `https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html`.

[26] Y. Smaragdakis and G. Balatsouras, “Pointer analysis,” *Found. Trends Program. Lang.*, vol. 2, pp. 1–69, Apr. 2015.

[27] M. Sharir, A. Pnueli, *et al.*, *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences . . . , 1978.

[28] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, “Pick your contexts well: Understanding object-sensitivity,” in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’11, (New York, NY, USA), pp. 17–30, ACM, 2011.

[29] T. Tan, Y. Li, and J. Xue, "Making k-object-sensitive pointer analysis more precise with still k-limiting," in *Static Analysis* (X. Rival, ed.), (Berlin, Heidelberg), pp. 489–510, Springer Berlin Heidelberg, 2016.

[30] Y. Li, T. Tan, A. Møler, and Y. Smaragdakis, "Scalability-first pointer analysis with self-tuning context-sensitivity," in *Proc. 12th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, November 2018.

[31] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras, "Introspective analysis: context-sensitivity, across the board," in *ACM SIGPLAN Notices*, vol. 49, pp. 485–495, ACM, 2014.

[32] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis, "Precision-guided context sensitivity for pointer analysis," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, p. 141, 2018.

[33] S. Z. Guyer and C. Lin, "Client-driven pointer analysis," in *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, (Berlin, Heidelberg), pp. 214–236, Springer-Verlag, 2003.

[34] M. Das, "Unification-based pointer analysis with directional assignments," *Acm Sigplan Notices*, vol. 35, no. 5, pp. 35–46, 2000.

[35] M. Fähndrich, J. Rehof, and M. Das, "Scalable context-sensitive flow analysis using instantiation constraints," in *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pp. 253–263, 2000.

[36] J. Whaley and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," in *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pp. 131–144, 2004.

[37] E. Ruf, "Effective synchronization removal for java," in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming language design and implementation*, pp. 208–218, 2000.

[38] N. Heintze and O. Tardieu, "Ultra-fast aliasing analysis using cla: A million lines of c code in a second," *ACM SIGPLAN Notices*, vol. 36, no. 5, pp. 254–263, 2001.

[39] D. J. Pearce, P. H. Kelly, and C. Hankin, "Online cycle detection and difference propagation for pointer analysis," in *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 3–12, IEEE, 2003.

[40] D. J. Pearce, P. H. Kelly, and C. Hankin, "Efficient field-sensitive pointer analysis of c," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 30, no. 1, pp. 4–es, 2007.

[41] Y. Lei and Y. Sui, "Fast and precise handling of positive weight cycles for field-sensitive pointer analysis," in *International Static Analysis Symposium*, pp. 27–47, Springer, 2019.

[42] T. Reps, "Program analysis via graph reachability," *Information and software technology*, vol. 40, no. 11-12, pp. 701–726, 1998.

[43] J. Späth, L. Nguyen Quang Do, K. Ali, and E. Bodden, "Boomerang: Demand-driven flow-and context-sensitive pointer analysis for java," in *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[44] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken, "Partial online cycle elimination in inclusion constraint graphs," in *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pp. 85–96, 1998.

[45] Y. Li, Q. Zhang, and T. Reps, "Fast graph simplification for interleaved dyck-reachability," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 780–793, 2020.

[46] Y. Smaragdakis and M. Bravenboer, "Using datalog for fast and easy program analysis," in *International Datalog 2.0 Workshop*, pp. 245–251, Springer, 2010.

[47] J. Mellor-Crummey, "On-the-fly detection of data races for programs with nested fork-join parallelism," in *Supercomputing'91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pp. 24–33, IEEE, 1991.

[48] F. Mattern *et al.*, *Virtual time and global states of distributed systems*. Citeseer, 1988.

[49] C. Flanagan and S. N. Freund, "Fasttrack: efficient and precise dynamic race detection," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 121–133, 2009.

[50] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: Data race detection in practice," in *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, (New York, NY, USA), pp. 62–71, ACM, 2009.

[51] A. Jannesari, K. Bao, V. Pankratius, and W. F. Tichy, "Helgrind+: An efficient dynamic race detector," in *2009 IEEE International Symposium on Parallel & Distributed Processing*, pp. 1–13, IEEE, 2009.

[52] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[53] J. Thalheim, P. Bhatotia, and C. Fetzer, "Inspector: data provenance using intel processor trace (pt)," in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pp. 25–34, IEEE, 2016.

[54] A. Kleen and B. Strong, "Intel processor trace on linux," *Tracing Summit*, vol. 2015, 2015.

[55] "Infer : Racerd." `http://fbinfer.com/docs/racerd.html`, 2021.

[56] D. Engler and K. Ashcraft, "Racerx: Effective, static detection of race conditions and deadlocks," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, (New York, NY, USA), pp. 237–252, ACM, 2003.

[57] J. W. Voung, R. Jhala, and S. Lerner, "Relay: Static race detection on millions of lines of code," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering*

Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC-FSE '07, (New York, NY, USA), pp. 205–214, ACM, 2007.

[58] M. Sridharan and S. J. Fink, "The complexity of andersen's analysis in practice," in *International Static Analysis Symposium*, pp. 205–221, Springer, 2009.

[59] WALA, "Wala." `http://wala.sourceforge.net/wiki/index.php/Main_Page`, 2018.

[60] C.-H. Hsiao, S. Narayanasamy, E. M. I. Khan, C. L. Pereira, and G. A. Pokam, "Asyncclock: Scalable inference of asynchronous event causality," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, (New York, NY, USA), p. 193–205, Association for Computing Machinery, 2017.

[61] C.-H. Hsiao, J. Yu, S. Narayanasamy, Z. Kong, C. L. Pereira, G. A. Pokam, P. M. Chen, and J. Flinn, "Race detection for event-driven mobile applications," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, (New York, NY, USA), p. 326–336, Association for Computing Machinery, 2014.

[62] P. Maiya, A. Kanade, and R. Majumdar, "Race detection for android applications," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, (New York, NY, USA), p. 316–325, Association for Computing Machinery, 2014.

[63] X. Fu, D. Lee, and C. Jung, "nadroid: Statically detecting ordering violations in android applications," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, (New York, NY, USA), p. 62–74, Association for Computing Machinery, 2018.

[64] M. Sharir, A. Pnueli, *et al.*, *Two approaches to interprocedural data flow analysis*, ch. 8, p. 189–233. New York University. Courant Institute of Mathematical Sciences, 1978.

[65] O. Shivers, "Control-flow analysis of higher-order languages," tech. rep., 1991.

[66] O. Lhoták, “Spark: A flexible points-to analysis framework for Java,” Master’s thesis, McGill University, December 2002.

[67] K. T. Tekle and Y. A. Liu, “Precise complexity guarantees for pointer analysis via datalog with extensions,” *CoRR*, vol. abs/1608.01594, 2016.

[68] M. Sagiv, T. Reps, and R. Wilhelm, “Solving shape-analysis problems in languages with destructive updating,” *ACM Trans. Program. Lang. Syst.*, vol. 20, pp. 1–50, Jan. 1998.

[69] S. Z. Guyer and C. Lin, “Error checking with client-driven pointer analysis,” *Sci. Comput. Program.*, vol. 58, pp. 83–114, Oct. 2005.

[70] R. O’Callahan and J.-D. Choi, “Hybrid dynamic data race detection,” in *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP ’03, (New York, NY, USA), p. 167–178, Association for Computing Machinery, 2003.

[71] B. Liu and J. Huang, “D4: Fast concurrency debugging with parallel differential analysis,” in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, (New York, NY, USA), pp. 359–373, ACM, 2018.

[72] S. Zhan and J. Huang, “Echo: Instantaneous in situ race detection in the ide,” in *Proceedings of the ? International Symposium on the Foundations of Software Engineering*, FSE ’16, 2016.

[73] “Coderrect race detector.” `https://coderrect.com/download/`, 2021.

[74] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, *et al.*, “The matter of heartbleed,” in *Proceedings of the 2014 conference on internet measurement conference*, pp. 475–488, ACM, 2014.

[75] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, “Non-control-data attacks are realistic threats.,” in *USENIX Security Symposium*, vol. 5, 2005.

[76] J. Pincus and B. Baker, “Beyond stack smashing: Recent advances in exploiting buffer overruns,” *IEEE Security & Privacy*, vol. 2, no. 4, pp. 20–27, 2004.

[77] R. base64 project, “Cve-2017-1000430,” 2017. `https://www.cvedetails.com/cve/CVE-2017-1000430/`.

[78] P. Sampaio, “Cve-2018-1000657,” 2018. `https://bugzilla.redhat.com/show_bug.cgi?id=1622249`.

[79] P. Sampaio, “Cve-2018-1000810,” 2018. `https://bugzilla.redhat.com/show_bug.cgi?id=1632932`.

[80] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, “Softbound: Highly compatible and complete spatial memory safety for c,” *ACM Sigplan Notices*, vol. 44, no. 6, pp. 245–258, 2009.

[81] M. Castro, M. Costa, and T. Harris, “Securing software by enforcing data-flow integrity,” in *Proceedings of the 7th symposium on Operating systems design and implementation*, pp. 147–160, USENIX Association, 2006.

[82] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, “Cets: compiler enforced temporal safety for c,” in *ACM Sigplan Notices*, vol. 45, pp. 31–40, ACM, 2010.

[83] G. C. Necula, S. McPeak, and W. Weimer, “Ccured: Type-safe retrofitting of legacy code,” in *ACM SIGPLAN Notices*, vol. 37, pp. 128–139, ACM, 2002.

[84] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, “Cyclone: A safe dialect of c.,” in *USENIX Annual Technical Conference, General Track*, pp. 275–288, 2002.

[85] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, P. Druschel, and D. Garg, “Erim: Secure, efficient in-process isolation with memory protection keys,” *arXiv preprint arXiv:1801.06822*, 2018.

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

[87] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.," in *USENIX Security Symposium*, vol. 98, pp. 63–78, San Antonio, TX, 1998.

[88] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity.," in *OSDI*, vol. 14, p. 00000, 2014.

[89] Rust-Team, "issue 26612," 2015. `https://github.com/rust-lang/rust/issues/26612`.

[90] japaric, "issue 39699," 2017. `https://github.com/rust-lang/rust/issues/39699`.

[91] B. Smith, "Stop using unsafe code," 2017. `https://github.com/alicemaz/rust-base64/issues/29`.

[92] S. B. Lippman, *Inside the C++ object model*, vol. 242. Addison-Wesley Reading, 1996.

[93] Rust-team, "Trait objects," 2011. `https://doc.rust-lang.org/book/trait-objects.html`.

[94] A. Crichton, "Tracking issue for changing the global, default allocator (rfc 1974)," 2015. `https://github.com/rust-lang/rust/issues/27389`.

[95] R. team, "Lang items," 2015. `https://doc.rust-lang.org/1.5.0/book/lang-items.html`.

[96] W. Gloger, "Wolfram gloger's malloc homepage," 2006. `http://www.malloc.de/en/`.

[97] D. Lea, “A memory allocator,” 1996. http://g.oswego.edu/dl/html/malloc.html.

[98] splotifun, “Understanding glibc malloc,” 2015. bhttps://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/.

[99] andigena, “ptmalloc fanzine,” 2016. https://lwn.net/Articles/725832/.

[100] A. documentation, “Arm memory domain.” http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0056d/BABBJAED.html.

[101] Phasar-Team, “Phasar framework,” 2018. https://github.com/secure-software-engineering/phasar.

[102] H. M. Almohri and D. Evans, “Fidelius charm: Isolating unsafe rust code,” in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, pp. 248–255, ACM, 2018.

[103] J. Corbet, “The stack clash,” 2017. https://blog.qualys.com/securitylabs/2017/06/19/the-stack-clash.

[104] J. Corbet, “Preventing stack guard-page hopping,” 2017. https://lwn.net/Articles/725832/.

[105] M. Jansson, “rpmalloc-benchmark,” 2017. https://github.com/rampantpixels/rpmalloc-benchmark.

[106] Shellphish, “unsafe unlink,” 2011. https://github.com/shellphish/how2heap/blob/master/glibc_2.26/unsafe_unlink.c.