Safer at Any Speed: Automatic Context-Aware Safety Enhancement for Rust

NATALIE POPESCU*, Princeton University, USA
ZIYANG XU*, Princeton University, USA
SOTIRIS APOSTOLAKIS, Google, USA
DAVID I. AUGUST, Princeton University, USA
AMIT LEVY, Princeton University, USA

Type-safe languages improve application safety by eliminating whole classes of vulnerabilities–such as buffer overflows–by construction. However, this safety sometimes comes with a performance cost. As a result, many modern type-safe languages provide escape hatches that allow developers to manually bypass them. The relative value of performance to safety and the degree of performance obtained depends upon the application context, including user goals and the hardware upon which the application is to be executed. Since libraries may be used in many different contexts, library developers cannot make safety-performance trade-off decisions appropriate for all cases. Application developers can tune libraries themselves to increase safety or performance, but this requires extra effort and makes libraries less reusable. To address this problem, we present NADER, a Rust development tool that makes applications safer by automatically transforming unsafe code into equivalent safe code according to developer preferences and application context. In end-to-end system evaluations in a given context, NADER automatically reintroduces numerous library bounds checks, in many cases making application code that uses popular Rust libraries safer with no corresponding loss in performance.

CCS Concepts: • **Software and its engineering** → *Software maintenance tools*; *Runtime environments*; **Software development process management**; *Compilers*; • **Security and privacy** → **Software and application security**; • **General and reference** → **Empirical studies**; *Performance*.

Additional Key Words and Phrases: Rust, bounds checks, safety-performance trade-off

ACM Reference Format:
Natalie Popescu, Ziyang Xu, Sotiris Apostolakis, David I. August, and Amit Levy. 2021. Safer at Any Speed: Automatic Context-Aware Safety Enhancement for Rust. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 103 (October 2021), 23 pages. https://doi.org/10.1145/3485480

1 INTRODUCTION

A decades-long struggle against memory safety vulnerabilities in critical systems has motivated developers to use languages with compiler-enforced type-safety. Such type safety guarantees can eliminate entire classes of memory-safety bugs by ensuring that data types are not accessed outside of their allocated memory regions. For instance, language-enforced bounds checks could have prevented OpenSSL's [OpenSSL 2021] recent Heartbleed vulnerability [Durumeric et al. 2014] (in

*Both authors contributed equally to this research.

Authors' addresses: Natalie Popescu, Princeton University, USA, npopescu@princeton.edu; Ziyang Xu, Princeton University, USA, ziyangx@princeton.edu; Sotiris Apostolakis, Google, USA, apostolakis@google.com; David I. August, Princeton University, USA, august@princeton.edu; Amit Levy, Princeton University, USA, aalevy@cs.princeton.edu.

2475-1421/2021/10-ART103
https://doi.org/10.1145/3485480

C code), which had access, via buffer overflow, to memory outside of its allocated region. While type-safe languages help prevent security critical bugs, overheads from dynamic memory and type checks have historically impeded their use in performance-sensitive contexts such as kernels, databases, Web browsers, etc. This leaves developers in a bind when both safety and performance are critical, and developers building performance sensitive applications have thus shied away from type-safety. But recent languages specifically target these applications and provide type-safety with performance comparable to C [Apple 2021; Donovan and Kernighan 2015; Matsakis and Klock 2014].

Rust [Matsakis and Klock 2014] is one such language in which some safety checks are free due to it's strict type system that enforces most memory safety properties statically; others require dynamic enforcement and *may* have a cost. Rust is particularly interesting because it prioritizes memory safety, yet developers still circumvent some dynamic checks with Rust's `unsafe` constructs. A predominant example of this is accessing Rust slices through functions like `get_unchecked` that avoid bounds checks to improve performance–we refer to this pattern as unchecked indexing (Definition 2.2) and the checked counterpart as checking indexing (Definition 2.1). In general, unchecked indexing is a long-standing problem, but even in the ecosystem of a language that enforces bounds checks by default we find them manually elided in as many as 10% of the 500 most downloaded libraries, many of which are used in applications that prioritize safety.

We perform a study to evaluate the top Rust libraries that use unchecked indexing, finding that 76.4% of their benchmarks show little-to-no or even a negative performance impact from unchecked indexing in a given context, suggesting that manually trading safety for performance in this way is rarely effective. Not only do developers incorrectly identify the most expensive checks to elide, but they fundamentally *cannot* correctly do so for every possible context. We find that simply using an older compiler version can change the checked indexing overhead of a sample application from 7.7% to 6.5%. Using a different architecture changes this overhead to 2.5%, and using a different application workload changes it again to 4.4%. Furthermore, an overhead of 4.4% may be acceptable in a safety-critical application but not in a performance-sensitive one, so a blanket approach that converts all unchecked indexing back into checked indexing is not necessarily realistic. We also investigate how unchecked indexing from libraries permeates into a set of 27 Rust applications, finding that on average there are 86 times more unchecked indexing operations in application dependencies than in the applications themselves.

Based on these findings, we suggest an approach that, rather than treating unchecked indexing as a requirement, treats unchecked indexing as a suggestion for an indexing operation that *may* improve end-to-end performance were it unchecked. This way, applications need not be tied to the trade-off decisions made by library developers, but library developer insights about which unchecked indexing operations may be profitable are still taken into consideration.

We implement this approach as NADER, a prototype tool that provides a systematic way for Rust developers to improve safety subject to their chosen performance cost threshold. NADER aims to automatically restore as many bounds checks as possible within this threshold in the context of the target compiler and hardware. This tool works well for a variety of applications: it shows that no unchecked indexing is used in two benchmarks, it converts all unchecked indexing to checked indexing in 6 benchmarks, and it finds better trade-off options for two benchmarks.

Contributions. We summarize our contributions as:

- **A study** that shows that developers do not and *cannot* correctly identify the most expensive bounds checks to elide–especially problematic for libraries that are used in a variety of contexts;

- **The NADER approach** that treats unchecked indexing as a suggestion and automatically restores safety up to a user-specified performance overhead threshold;
- **A prototype implementing the NADER approach** that finds better safety-performance trade-offs for a wide range of benchmarks.

Availability. Our research artifact is available at https://doi.org/10.5281/zenodo.5484436.

2 MOTIVATING STUDY

While run-time safety checks increase the safety of a program, they often come at a cost and are not integrated by default in systems programming languages like C and C++. Many tools designed for C and C++ strive to provide low-cost checks in hopes of motivating programmers to use them more frequently. Among these tools is ASAP [Wagner et al. 2015], a tool that automatically introduces safety checks into C++ code up to a specified overhead threshold. ASAP leverages the insight that most safety-critical checks occur in cold code and discovers that many checks can be introduced with low performance overheads.

Unlike C/C++, Rust provides a different solution: it adds such checks by default while also giving users the option of an `unsafe` escape hatch for when they are too limiting. While Rust compiler developers try to reduce the performance overhead of these checks through optimizations, expensive ones can still slip through. The perception that there are *many* such checks has motivated developers to take performance matters into their own hands and manually elide the checks they deem expensive. Studies [Rust Language Team 2021] have found that programmers commonly use `unsafe` Rust, and a survey [Evans et al. 2020] claims that 55% of participants use `unsafe` Rust for better performance.

Note that unlike C++, which is inherently unsafe, `unsafe` code in Rust needs to be explicitly introduced by programmers. Consequently, while work like ASAP uncovers an interesting insight–that inserting safety checks from an unsafe baseline can be cheap in most cases–the question remains open for user-elided checks from a safe baseline. Primarily, it is not clear whether run-time checks elided with `unsafe` Rust are justified by their performance. An often overlooked factor is the role of context: we suspect that run-time checks elided with `unsafe` Rust may be justified in some contexts but not in others, as context can change their overhead. Finally, we seek to understand how this manual elision of run-time checks permeates the Rust ecosystem.

2.1 Study Setup

We perform a study to investigate these factors with a focus on bounds checks. Bounds checking is a longstanding problem in the community [Bodík et al. 2000; Gupta 1993; Kolte and Wolfe 1995; Patterson 1995; Qian et al. 2002; Rugina and Rinard 1999], and unchecked indexing is often credited as a frequent pattern of sacrificing safety for performance in Rust. One study [Qin et al. 2020] cites performance as a reason for 22% of the unsafe code they studied, of which unchecked indexing and `ptr::copy_nonoverlapping()` are significant contributors. Another study notes that unchecked indexing "plays a significant role for some performance-oriented crates" [Astrauskas et al. 2020], where a crate is analogous to a Rust library. This motive is additionally demonstrated by code comments like "it would be great if the bounds checks here could be optimized out…this improves performance by about 7%" in the `hc128.rs` file of the `rand_hc-0.2.0` library, and "this function skips the bounds check in optimized builds. Using it in the hottest two call sites gives a 15%…speed boost" in Firefox's `modules/libpref/parser/src/lib.rs`. Moreover, manually elided checks are easy to syntactically isolate in Rust.

Definition 2.1 (Checked Indexing)**.** A checked index dereferences an offset into an array-like structure (e.g., `slice`, `array`, and `str` in Rust) subject to a successful check that the offset is within

the bounds of the structure (e.g. the length of the `slice`). In Rust, checked indexing is exposed with the `get` or `get_mut` methods as well as with the index operator (`a[i]`), as shown in `copy_checked` below:

```
fn copy_checked(src: &[u8], dst: &mut [u8]) {
  for i in 0..src.len() {
    dst[i] = src[i]; // or, dst.get_mut(i).unwrap() = src.get(i).unwrap();
  }
}
```

Definition 2.2 (Unchecked Indexing)**.** An unchecked index dereferences an offset into an array-like structure **without** performing a bounds check. Unchecked indexing is exposed in Rust through the `get_unchecked` or `get_unchecked_mut` methods, like so:

```
fn copy_unchecked(src: &[u8], dst: &mut [u8]) {
  unsafe {
    for i in 0..src.len() {
      *dst.get_unchecked_mut(i) = *src.get_unchecked(i);
    }
  }
}
```

Given this information, we formulate the goals of our study into three concrete questions:

Q1: In a given context, is using unchecked indexing justified by performance?
Q2: Are decisions to use unchecked indexing appropriate for all contexts?
Q3: How prevalent is unchecked indexing in the Rust ecosystem?

2.2 Results and Analysis

Q1: In a given context, is using unchecked indexing justified by performance?
What makes the cost of bounds check particularly hard to measure is that the effect of a single bounds check depends not only on the direct dynamic overhead of a couple of added instructions but also on the compiler passes and optimizations it may prevent. This complicated interplay can cause a single bounds check to affect program performance significantly in both intuitive and unintuitive ways. Within a narrow scope, a checked index could cause a performance hit that becomes unimportant within a larger scope with different performance bottlenecks. The opposite can also occur, where a checked index can block vectorization or identification of idioms such as `memcpy`.

Consider `copy_checked` and `copy_unchecked`, two functions that both implement equivalent `memcpy` semantics: the former using checked indexing and the latter using unchecked indexing. When called by the exact same code, `copy_unchecked` runs 7x faster. `copy_checked` compiles into a canonical loop with an automatically-inserted bounds check guarding each iteration, while `copy_unchecked` turns into a single `memcpy` call in the generated LLVM IR. Notably, the performance overhead of the checked version is not the result of executing a bounds check in every iteration of the loop. Instead, it stems from the additional complexity in the loop's control flow, which complicates the code pattern and ultimately inhibits vectorization.

Given the complexity of assigning an overhead to every bounds check, we experimentally measure the overhead of checked indexing on a handful of libraries where bounds checks are already manually elided. We show the results as a histogram in Figure 1. Each of the 203 benchmarks belongs to one of 7 libraries in the top 250 most downloaded Rust libraries on crates.io that a) contains unchecked indexing and b) has a benchmark suite that compiles with `rustc-1.52.0`. To

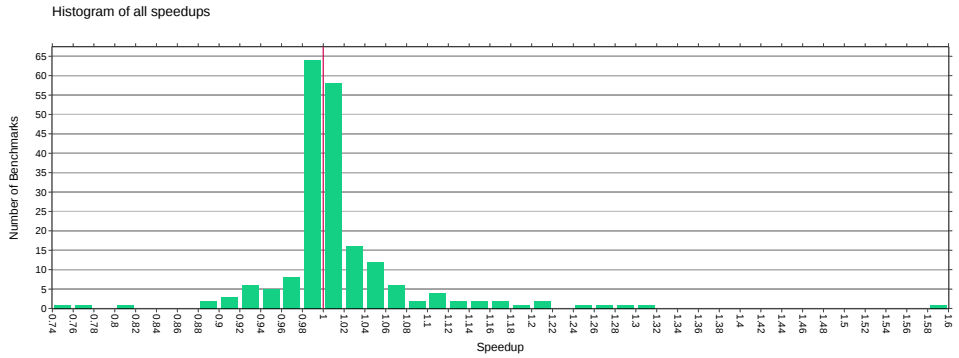

Fig. 1. A histogram of benchmarks (y-axis) exhibiting a particular performance impact (x-axis) when all unchecked indexing is converted to checked indexing. Consists of 203 benchmarks over 7 Rust libraries. 131 (64.5%) benchmarks show less than a 1% change in performance, 48 (23.6%) are more than 1% slower with checked indexing, and 24 (11.8%) are more than 1% faster with checked indexing. All but two benchmarks (one in the 0.96-0.98 Speedup bucket and another in the 0.94-0.96 Speedup bucket) have non-overlapping 95% confidence intervals.

measure the overhead of checked indexing, we compare the performance of each unmodified library with a version of the library where all unchecked indexing is converted to checking indexing. We observe a large variance in the overheads of checked indexing: 23.6% of benchmarks *do* report significant performance hits from checked indexing, but 64.5% report little-to-no impact and, surprisingly, 11.8% report improved performance! All but two benchmarks (one in the 0.96-0.98 Speedup bucket and another in the 0.94-0.96 Speedup bucket) are statistically significant at the 95% confidence level. **Ultimately, while unchecked indexing *can* improve performance, most of the time it does not.**

The counter-intuitive performance improvements from checked indexing could be caused by affected heuristics within the compiler. For example, introducing a checked indexing operation–and thus increasing the instruction count of a function–can potentially prevent the inlining of the function if it barely satisfies the inlining heuristic without the added check. If this inlining decision ends up being harmful (by increasing i-cache or i-TLB misses), the added check could improve performance. Adding a check could also affect code layout, where the alignment of branch targets is critical for performance. An additional check could prevent misalignment or reduce padding.

An explanation for the large variance of our results could be that the benchmarks may not be exercising the indexing operations at all, or may only be exercising them a bit, such that the overhead is not apparent. In the `glam-0.14.0` library for example, the `mat3_transform_point2` benchmark shows a 21% slowdown from checked indexing, whereas in the `mat3_inverse` benchmark shows a 6.4% speedup from checked indexing. This indicates that, while `glam-0.14.0` as a whole contains unchecked indexing, different benchmarks in the library will naturally exercise those operations to different extents and may also be susceptible to different downstream interactions that will affect the ultimate overhead.

Q2: Are decisions to use unchecked indexing appropriate for all contexts?
The overall context of a library consists of the entire environment in which it runs. This includes the application that uses the library, the compiler, the operating system, the underlying ISA, and the microarchitecture. To illustrate the impact of context on performance, we use `brotli-decompressor` [Dropbox 2020], a drop-in replacement for the C implementation of the

Table 1. The overhead of checked indexing in four different contexts. For each context, the overhead represents the relative slowdown of a converted `brotli-decompressor` compared to the unconverted original. The baseline context–the same as the deployment context used in Section 5–is compiled with `rustc-1.52.0`, run on a machine with two Intel Xeon E5-2697 v3 processors, and profiled using `brotli-decompressor`'s data at compression level 5. Each variation only changes one part of this context.

Deployment Context	Baseline	Different Compiler (rustc 1.46.0)	Different Architecture (Apple M1)	Different Workload (Compression Level=11)
Overhead	7.7%	6.5%	2.5%	4.4%

Brotli generic-purpose lossless compression algorithm [Alakuijala et al. 2019]. Table 1 shows the overhead of checked indexing (i.e. the performance difference between converted and unconverted versions of `brotli-decompressor`) in four different contexts. The overhead of checked indexing in the baseline context is 7.7%, but using an older version of `rustc` changes this overhead to 6.5%. Similarly, running on a different architecture or with a different compression level reduces the overhead of checked indexing more by varying amounts. Note that the overhead percentages shown are not all comparing against a single baseline. For example, the absolute performance of both converted and unconverted versions with the newer `rustc-1.52.0` is improved over the older `rustc-1.46.0`, but the relative overhead is larger with `rustc-1.52.0`. One possible explanation is that a newly-introduced optimization is applicable to just the unchecked version, thus widening the gap.

Table 2. The simplified context of a generic application. The overhead of a checked indexing operation can be affected by factors in every layer of the system software stack (some of which are mentioned under Examples), so predicting it for every context is simply not possible.

Layer	Examples
Applications	safety-performance goals; full context of dependencies
Libraries	semantics of safe/unsafe code; performance bottlenecks
Compiler	vectorization; phase-ordering problem
Operating System	exception handling; memory management
Instruction Set Architecture	vector extensions; branch instructions
Microarchitecture	branch prediction; memory management

Furthermore, operating system and microarchitectural implementation details for performance-critical components, like memory management and caching, add to the already-present complexity of reasoning about performance. A summary of all the system layers that would need to be fully understood in order to accurately predict performance for every context is shown in Table 2. In cases where safety checks lead to frequent exception handling, the implementation of such procedures will also affect performance. Additionally, common intuitions about how additional branches impact performance–that they largely become no-ops–are still occasionally erroneous, e.g., due to the mitigation of speculative execution side channels revealed by Meltdown [Lipp et al. 2018] and Spectre [Kocher et al. 2019]. **Therefore, the measured overhead of checked indexing in one context cannot be used to predict its overhead in another context.**

Q3: How prevalent is unchecked indexing in the Rust ecosystem?
We find that 10% of the 500 most downloaded Rust libraries on `crates.io` contain at least one instance of unchecked indexing. We also find that, across a set of 27 applications which we selected according to criteria described in Section 5, there tends to be on average 86 times more unchecked

indexing in application dependencies than in the application itself, as seen in Table 4. While reasoning about and auditing every unchecked indexing operation in one's own application may be feasible, doing this for every dependency can overwhelm even large engineering teams. This calls for a more systematic way of identifying and defending against all of the potential unchecked indexing that an application may use.

Insight: Developers cannot always correctly identify the most expensive checks to elide. Based on our results from Figure 1, the naive solution would be to always use checked indexing and leave it at that. Unfortunately, this is not a realistic option for two reasons. The first is that, even if we started this instant, many libraries still contain counterproductive unchecked indexing that would end up in dependencies everywhere. The second reason is that expensive checks may not be acceptable for some performance-sensitive applications. A solution is needed that can convert all unchecked indexing in an application and its dependencies, but that is also flexible enough to leave the expensive checks out of applications that cannot tolerate them. Furthermore, the solution should work in a per-benchmark, per-context fashion. We build a prototype of such a tool that, given a benchmark and an overhead threshold, returns the best trade-off between safety and performance.

3 THE NADER APPROACH

Library developers introduce unchecked indexing into their code when they believe it is correct to do so and they intuit that it might improve performance. As shown in Figure 2, the status quo forces application developers to accept the choices that library developers make. They do not have a simple way of discovering the unchecked indexing hidden in all dependent libraries nor to make them checked if they wish to improve safety. The NADER approach makes the application developers aware of hidden unchecked indexing in all dependent libraries, tests the performance impact of existing unchecked indexing in a specific deployment context, and automatically improves safety by converting unchecked indexing to checked indexing subject to a performance overhead threshold.

Figure 2 compares the workflow of library and application development and deployment under NADER with the status quo. The NADER approach does not impose any changes to the workflow of library developers because library developers do not have adequate information to make informed decisions about unchecked indexing. This also allows the tool to be more compatible with existing libraries. In the application developer's workflow, NADER first discovers which of the application's dependencies use unchecked indexing. This information, not readily available to developers today, enables them to make more informed decisions about whether to incorporate libraries, audit them, choose between alternative libraries, or implement the library functionality themselves. We refer to unchecked indexing in the application itself as *direct unchecked indexing* and that in dependencies as *indirect unchecked indexing*. Before deploying an application, developers use the NADER toolchain to determine whether direct or indirect unchecked indexing has an impact on performance in their particular context. Finally, NADER transforms unchecked indexing to checked indexing until performance degrades to a developer-provided threshold using developer-provided benchmarks to complete the context.

Concretely, NADER enables the following improvements to an application developer's workflow:

Awareness: Make application developers aware of unchecked indexing. When application developers are aware of the unchecked indexing used in a dependent library, they can make an informed decision to trust it, audit its code, or choose an alternative option.

Safety: Improve application safety up to a performance overhead threshold. NADER converts direct and indirect unchecked indexing to checked alternatives. Application developers supply

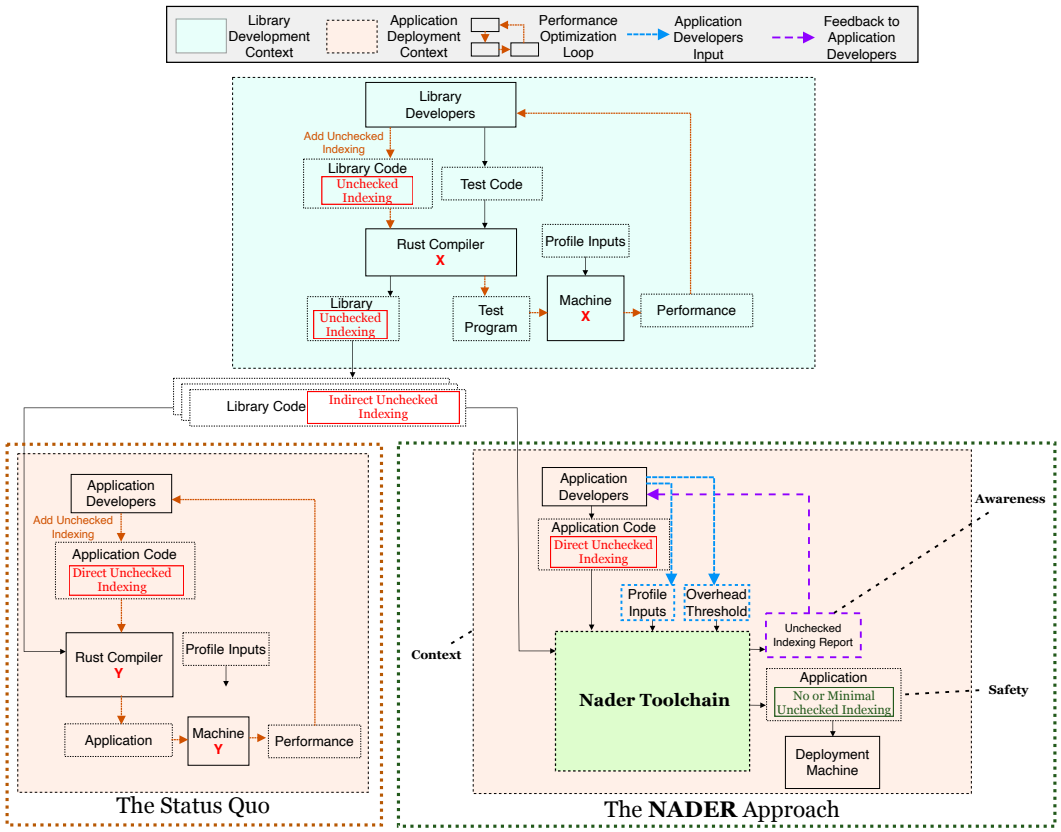

Fig. 2. Comparison of the status quo and the NADER approach. NADER makes application developers aware of unchecked indexing and automatically optimizes the safety-performance trade-off in the deployment context to generate binaries with minimal unchecked indexing.

test inputs and a performance overhead threshold to NADER, which then automatically converts as much unchecked indexing as possible without exceeding the overhead threshold.

Context: Optimize safety-performance trade-offs for an application *and* deployment context. NADER explores the safety-performance trade-off automatically, considering the entire computing stack in Table 2 with no additional developer effort.

With the NADER approach, application developers can fulfill different safety-performance objectives by adjusting the overhead threshold. In all cases, the NADER approach improves safety with minimal or no overhead.

Performance-Driven. If users are looking for the best possible performance, they can set the overhead threshold to zero and force it to maintain optimal performance while introducing as many bounds checks as possible. As shown in Section 5, a significant portion of bounds checks can be reintroduced to improve safety without sacrificing any performance.

Safety-Driven. If the application is sensitive to safety but not as much to performance, the application developer can choose to enforce maximum safety by specifying an infinite overhead threshold, essentially making all unchecked indexing checked.

Balance Between Performance and Safety. Users in-between the two extremes can specify an ideal performance threshold, automatically optimize performance by identifying a set of highly profitable unchecked indexing operations (relative to the threshold) and improve safety by converting all the others to their checked counterparts.

4 NADER TOOLCHAIN

NADER helps application developers regain control over unchecked indexing in their applications and dependent libraries. It consists of three components: NADER-Analyzer, NADER-Converter, and NADER-Explorer. We begin our discussion of NADER by describing the interactions between components, also summarized in Figure 3, and follow with details of the components themselves.

4.1 NADER Overview

NADER takes as input an application with a benchmark (as source code) and an overhead threshold; it returns to the developer an application binary that is optimized for their safety-performance needs. NADER assumes that the given benchmark is representative of the workload the developer wants to optimize the application for.

First, NADER-Analyzer uncovers all direct and indirect unchecked indexing compiled by the benchmark, revealing the level of compromised safety the application is exposed to. As an optimization, if no unchecked indexing is seen in this step, then no unchecked indexing will end up in the binary, and NADER returns the application binary as is.

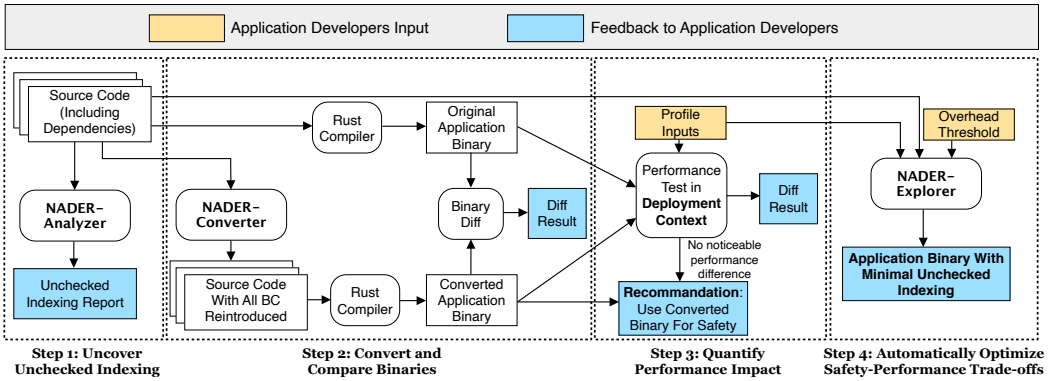

Fig. 3. NADER identifies all potentially-used unchecked indexing operation and determines if *any* of them improve performance. If some do, then NADER identifies them and, subject to the overhead threshold, converts the cheapest ones to their checked counterparts. If no unchecked indexing improves performance, NADER makes them all checked.

If NADER-Analyzer finds any unchecked indexing during compilation, NADER-Converter: i) converts all found unchecked indexing to checked indexing, ii) compiles both the original, unmodified source code and the converted source code, and iii) compares the resulting binaries. If the two binaries are identical, NADER knows that the original application is, in practice, no less safe than the converted application; as another optimization, NADER exits early in these cases. Binaries that differ could indicate the presence of unchecked indexing or simply be the result of unrelated non-determinism. At this point NADER switches to empirical methods.

In its third step, NADER executes the two versions of the compiled benchmark. The difference in performance between these two binaries effectively measures the overhead of checked indexing. If

the overhead of checked indexing is not significant in the given context, there is no need to make a safety-performance trade-off, and NADER returns the fully-converted application binary.

Finally, if there is a significant performance difference, NADER-Explorer automatically explores the safety-performance trade-off space to identify the most expensive checks. It converts as many unchecked indexing uses as possible to their checked counterparts while keeping the performance overhead below the specified threshold, and returns the maximally-converted binary to the application developer.

4.2 NADER Analyzer: Uncovering Direct and Indirect Unchecked Indexing

NADER-Analyzer first compiles the developer-provided benchmark using a custom `rustc`, augmented with an MIR pass that identifies all of the unchecked indexing calls and returns a list of their corresponding source code locations. In particular, as function calls are fully resolved at the MIR-level, the MIR pass identifies calls to just the Rust core and library's implementation of unchecked indexing, which ensures that NADER-Converter preserves the correct semantics.

Then, NADER-Analyzer presents a report to the developer that, among with the source code locations of each unchecked indexing operation found, includes: the number of direct unchecked indexing (in the application itself), the number of indirect unchecked indexing (in the dependencies), and the number of libraries that the application depends on that contain unchecked indexing. NADER-Analyzer ignores dependencies marked `dev` in the application manifest (a file named `Cargo.toml` by Rust convention) because these libraries are not available when compiling the application binary. [1]

4.3 NADER Converter: Converting Unchecked Indexing to Checked Indexing

NADER relies on the Rust core library's implementation of checked gets for correctness and trusts other unsafe code within the core and standard libraries. In addition to core and standard libraries that ship with Rust, the Rust ecosystem relies heavily on open-source, third party libraries that also contain unsafe code, but are less-vetted and, therefore, less trustworthy than the standard library. NADER acts on this set of unsafe code.

To convert unchecked indexing to checked indexing, we study the semantics of all `pub unsafe fn get_unchecked<I>(&self,index: I)` functions in Rust's core and standard libraries and its safe alternative `get` (also `get_unchecked_mut` and its safe alternative `get_mut`). As documented in the Rust `slice` library, "calling this method…with an out-of-bounds index is undefined behavior even if the resulting reference is not used", meaning the programmer should ensure that the index is within bounds. The `get` alternative does the bounds checking before indexing into the slice and returns `None` if the bounds check fails or `Some(...)` if the bounds check succeeds. Otherwise, all pre- and post-conditions of calling these two methods are the same.

To implement NADER-Converter, we replace all the calls of `get_unchecked` with its safe alternative `get` and chain an `unwrap()` after it. `unwrap()` will panic if the returned `Option` is `None`, which simulates the effect of the out-of-bounds panic from intrinsic indexing, `s[i]`.

4.4 NADER Explorer: Automatically Exploring the Safety-Performance Trade-Off

At the center of our proposed approach is NADER-Explorer, shown in Figure 5, a tool that can automatically optimize for the safety-performance trade-off while taking into account the deployment context. In order to optimize for the performance-safety trade-off, both performance and safety need to be concretely defined.

[1] dev dependencies are common in many language ecosystems and allow developers to leverage the language's package manager to install tools useful for testing and development, such as a unit test runner or code formatting tools.

Definition 4.1 (Performance Metric)**.** We define our performance metric as the average throughput of the application with its typical workloads in the deployment context.

Safety, on the other hand, is less straightforward to define. When only using Safe Rust, we can rely on the Rust compiler to provide safety guarantees [Matsakis and Klock 2014]. However, if even one line of unsafe code is introduced, the compiler can no longer guarantee the safety of related code. The premise of our safety metric is that every line in an unsafe block needs to be manually audited to ensure it does not violate the type- or memory-safety assumptions made by the compiler elsewhere. In order to minimize the auditing burden, we want minimize the number of lines that need to be audited.

Definition 4.2 (Safety Metric)**.** We define our safety metric as the number of lines that do not need to be audited, reflected by the amount of source code that is free of unchecked indexing (where the unchecked indexing is eventually introduced into the application binary). ASAP [Wagner et al. 2015] uses a similar metric.

Given our concrete definitions of both safety and performance, our goal is to minimize the amount of unchecked indexing (i.e., maximize safety) without exceeding the acceptable overhead threshold. We formalize our goal as the following optimization problem:

Definition 4.3 (Optimization Problem)**.** Given an initial set of unchecked indexing operations M within an application A and a user-specified performance threshold T, find a set N, such that:

(1) $N \subseteq M$;
(2) $\phi(N) \leq$ T (acceptable performance);
(3) $\forall K. K \subseteq M \wedge \phi(K) \leq T \implies |N| \leq |K|$ (maximum safety)

where $\phi(S)$ is the execution time of A modified such that every previously-unchecked index $a \in S$ is now protected with a bounds check.

Problem 4.3 is essentially a search problem, where each search point corresponds to a trial run consisting of recompilation and execution, taking seconds to minutes to complete in practice. If the count of unchecked slice accesses is 20, and each search point takes one second, the exhaustive search will take more than 11 days to finish. Thus, an exhaustive approach is not acceptable and we must rely on some heuristic to expedite the search.

By measuring the impact of individual bounds checks and ranking them in ascending order based on their performance impact, we can adopt a greedy search algorithm by adding the checks back to the original source code one by one and measuring the overall performance after every addition. The search stops when the overhead reaches the threshold. This brings the complexity of the search problem from exponential down to linear. To effectively capture the impact of a single bounds check, we rely on two observations.

Observation 4.1. Bounds checks mainly affect performance by introducing new instructions and by potentially blocking compiler optimizations.

The introduced instructions will likely make a single indexing operations take more cycles to finish, increasing the dynamic instruction count. If performance-oriented optimizations like vectorization and inlining are blocked by the introduced instructions, the resulting code performs much worse compared to the unchecked version.

Observation 4.2. If a bounds check is in cold code, its impact on performance is limited.

If a bounds check is rarely executed given a typical workload, the introduced instructions barely change the total number of dynamic instructions. Similarly, blocking compiler optimizations on a cold code path would not considerably affect performance.

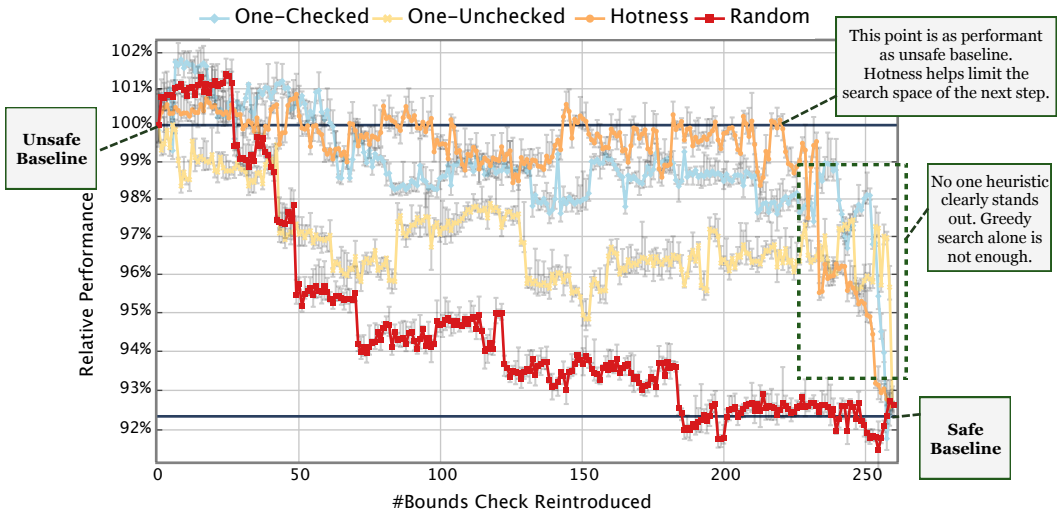

Fig. 4. Comparison of four different heuristics for ordering the bounds checks to reintroduce. No single heuristic is strictly the best, but getting hotness information is O(1)–combining heuristics is likely to be most effective. Each experiment runs ten times and we present the median time with the error bars depicting 95% confidence intervals.

Based on these observations, we devise three heuristics to estimate the impact of an individual bounds check: i) **hotness** of the unchecked indexing; ii) a **one-checked** slowdown, the performance difference between the original source code (unsafe baseline) and making one unchecked indexing operation in the unsafe baseline checked; and iii) a **one-unchecked** speedup, the performance difference between converting all unchecked indexing to checked (safe baseline) and then making one checked indexing operation unchecked again. For the hotness heuristic, we use the `callgrind` profiler [Weidendorfer et al. 2004] to gather the statistics of the binary, and parse the generated profile to pinpoint the instances of unchecked indexing. Remaining experiments are run natively. The hotness heuristic relies on Observation 4.2 and the other two rely on Observation 4.1.

We perform a greedy search with these three heuristics to get the impact of a single bounds check, and also include a random ordering for comparison. The detailed evaluation setup is the same as in Section 5. The performance of each heuristic on one benchmark (`brotli-decompressor`) is presented in Figure 4. All three heuristic variants perform better than the random ordering. The variance of the results can be explained by several secondary performance effects: the introduced instructions can change the binary layout, affecting the instruction cache performance, and the introduced branch instructions may also affect the accuracy of the branch predictor. While these effects are secondary compared to those in Observation 4.1, their variance makes the final exploration rather imprecise.

By using just the hotness heuristic, NADER-Explorer can limit unchecked indexing to a very small set (23 instances out of the original 263 as shown in Figure 4) without sacrificing any performance. To further optimize the search algorithm and minimize unchecked indexing, NADER-Explorer iteratively converts the remaining unchecked indexing operations. In each round, it first measures the performance impact of the one-checked slowdown heuristic on each unchecked indexing operation, only converting those that fall below a small sensitivity threshold. If none fall below this threshold, NADER-Explorer chooses the cheapest one to convert. The search stops when the developer-specified overhead threshold is reached.

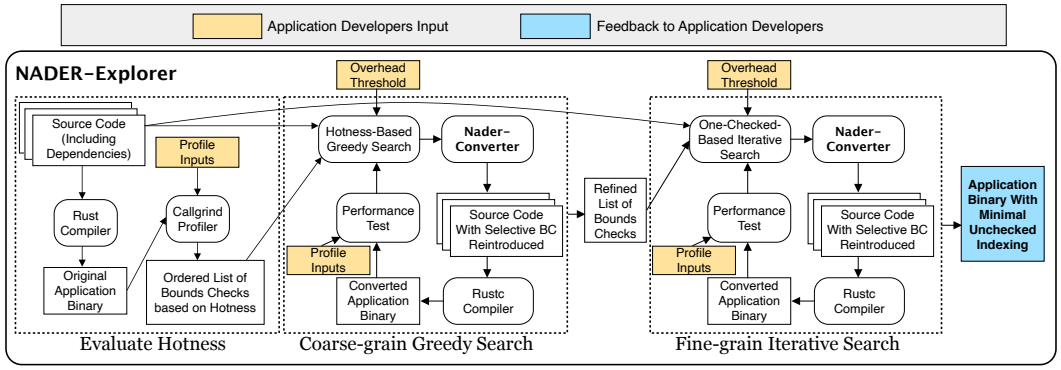

Fig. 5. BC=Bounds Check. NADER-Explorer uses a combination of hotness and one-checked heuristics. It first gets hotness with `callgrind` and uses a hotness-based greedy search to find the minimal set of unchecked indexing that performs no worse than the unsafe baseline. Then, it iteratively gets one-checked slowdowns and adds bounds checks to locations where converting from unchecked to checked indexing introduces little or no overhead. NADER-Explorer continuously makes sure the overhead is below the threshold.

The hotness-based greedy search part of NADER-Explorer has a time complexity of $O(n * T)$, where `n` is the total number of unchecked indexing operations and T is the time needed to run one benchmark. The iterative search part of NADER-Explorer has a time complexity of $O(m^2 * T)$ where `m` is the number of leftover unchecked indexing operations after using the hotness-based greedy search. In practice, `n` is on the order of tens to hundreds, T varies from 0.1s to 30s, and `m` is on the order of tens. The total running time of NADER-Explorer varies from one minute to an hour. Note that the two steps of the search are both embarrassingly parallel and NADER supports using as many cores on a system as possible. The output of NADER-Explorer is a binary that minimizes unchecked indexing without exceeding the performance overhead threshold.

4.5 Implementation Limitations

NADER does not support programs where libraries are distributed as pre-compiled binary objects or LLVM IR. However, since Rust programs by default build libraries from source, we do not foresee this as a major issue.

NADER does not currently handle macros gracefully. For applications in which macros fundamentally contribute to the code, such as `brotli-decompressor`, macros can be expanded manually using `cargo-expand` (we do this for our evaluation of `brotli-decompressor`). However, `cargo-expand` does not work correctly in all cases so, in general, NADER will not convert unchecked indexing operations that are called from macros.

NADER relies on end-to-end performance measurements for the last two steps. The current implementation can only handle a single performance metric with one benchmark input for an application. However, some applications are essentially bundles of multiple relatively independent parts. For these applications, one single profile input is not enough to cover all functionalities. Due to the relatively small performance difference between unchecked and checked indexing, NADER is sensitive to performance variability and currently requires a controlled running environment. Applications with inherently large variance are not good candidates. We plan to incorporate more sophisticated profiling tools in the future to address these issues.

5 EVALUATION

We evaluate NADER on a set of Rust applications that are safety- and/or performance-sensitive. We initially assess the amount of unchecked indexing in the source code of all selected applications and their dependencies (Section 5.2). Then, for the applications for which we can reliably measure end-to-end performance, we demonstrate that NADER automatically makes all of them safer with little performance overhead (Section 5.3).

Table 3. Summary of the wide variety of applications we evaluate NADER on.

Category	Application	Description
Data Processing	brotli-decompressor [Dropbox 2020]	Decompressor of Brotli format
	COST [McSherry et al. 2015]	Graph computation
	tantivy [tantivy Team 2021]	Search engine library
	flatbuffers [Google 2021]	Memory efficient serialization library
	vector [Timber 2021]	Tool for building observability pipelines
Languages and Tools	RustPython [RustPython Team 2021]	A Python interpreter
	fnm [fnm Team 2021]	Node.js version manager
	wasmer [Wasmer 2021]	WebAssembly Runtime
Cryptography	BLAKE3 [BLAKE3 Team 2021]	BLAKE3 hash function
	rage [rage Team 2021]	Encryption tool
Databases	tikv [TiKV Project 2021]	Distributed transactional key-value database
	diesel [Diesel Team 2021]	ORM and Query Builder
	flux [InfluxData 2021]	Scripting language for querying databases
	sonic [Sonic Team 2021]	Schema-less search backend
	splinter [Kulkarni et al. 2018]	Key-value store
The Web	swc [swc Team 2021]	Speedy web compiler
	warp [Warp Team 2021]	Web server framework
	iron [Iron Team 2021]	Web framework
	zola [Zola Team 2021]	Static site generator
	servo [Servo Project Developers 2021]	The Servo browser engine
	gecko [Mozilla 2021]	Firefox browser
	tonic [Hyperium 2021]	Native gRPC client & server implementation
Hypervisor	firecracker [AWS 2021]	MicroVMs for serverless computing
Graphics	gfx [Rust Graphics Mages 2021]	Vulkan-like GPU API
Networking	boringtun [Cloudflare 2021]	Implementation of WireGuard protocol
	NetBricks [Panda et al. 2016]	New network function framework
	quiche [Cloudflare 2021]	Implementation of QUIC and HTTP/3

5.1 Datasets and Experiment Setup

We select 27 popular Rust-based applications that satisfy the following criteria: they 1) are reasonably well-maintained applications, 2) have benchmarks, 3) have dependencies, and 4) are plausibly either security-sensitive, performance-sensitive, or both. While not an automated process, we systematically select satisfying applications from sources that include official packages in Linux distributions, popular GitHub repositories, and otherwise well-known applications written in Rust. These applications typically depend on one or more of the libraries from Figure 1, but we do not select applications based on this criteria; while the cost of checked indexing may be more pronounced or visible in libraries, it is ultimately more important in the context of applications. Table 3 lists all of the applications we selected, in what general computing category they lie, and a concise description of each.

Table 4. UI=Unchecked Indexing. Although application developers may not use unchecked indexing directly, there are many cases where unchecked indexing is hidden in application dependencies. For the safety-performance trade-off exploration, NADER requires a representative benchmark (provided by the application) that can run locally and has stable performance: ✗ means that this requirement is not satisfied and ● means that the application does not perform unchecked indexing at all so there is no need to continue.

Category	Application	#Direct UI	#Indirect UI	#Total Deps	#Deps w/ UI (%)	Continue with NADER?
Data Processing	brotli-decompressor	263	0	5	0 (0%)	✓
	COST	6	3	22	1 (4.5%)	✓
	tantivy	0	105	131	11 (8.4%)	✓
	flatbuffers	0	0	11	0 (0%)	●
	vector	0	449	988	37 (3.7%)	✗
Languages and Tools	RustPython	1	157	293	17 (5.8%)	✓
	fnm	0	168	252	9 (3.6%)	✗
	wasmer	0	55	199	5 (2.5%)	✗
Cryptography	BLAKE3	0	2	11	1 (9.1%)	✗
	rage	0	117	207	8 (3.9%)	✓
Databases	tikv	18	0	707	0 (0%)	✗
	diesel	0	44	167	6 (3.6%)	✗
	flux	0	43	53	4 (7.5%)	✗
	sonic	0	9	103	3 (2.9%)	✗
	splinter	3	604	311	20 (6.4%)	✗
The Web	swc	7	74	202	11 (5.4%)	✓
	warp	0	47	164	8 (4.9%)	✓
	iron	0	54	172	9 (5.2%)	✓
	zola	0	324	461	28 (6.1%)	✓
	servo	1	331	1031	35 (3.4%)	✗
	gecko	7	462	834	33 (4.0%)	✓
	tonic	0	102	302	15 (5.0%)	✗
Hypervisor	firecracker	0	31	66	1 (1.5%)	✗
Graphics	gfx	0	136	241	16 (6.6%)	✗
Networking	boringtun	0	0	46	0 (0%)	●
	NetBricks	0	0	1	0 (0%)	●
	quiche	0	6	13	1 (7.7%)	✗

We conduct all experiments with `rustc-1.52.0` (nightly-2021-02-11) and compile using release profiles. All performance experiments run directly–without using virtual machines or simulators–on a machine with two Intel Xeon E5-2697 v3 processors running at 2.60GHz (turbo-boost disabled) with 768GB of memory. The operating system is 64-bit Ubuntu 20.04 LTS. Note that we use `callgrind` to gather hotness information (execution count per line) for NADER-Explorer, but it is not a part of our performance measurements.

5.2 Ubiquitous Hidden Unchecked Indexing

In our selected set of 27 Rust applications, we uncover the extent to which unchecked indexing is used. This helps us narrow down the set of applications to those that NADER can have an effect on; an application with no unchecked indexing in its own code or in any of its dependencies would not benefit from NADER. But it also helps us understand and establish a baseline for how much unchecked indexing permeates popular Rust applications. Table 4 shows the results of this initial exploration. These numbers are reported as a static source code count of the unchecked indexing

Table 5. Perf Diff=Performance Difference, Bmark=Benchmark, Ex=Example, App=Application, UI=Unchecked Indexing, D=Direct UI, I=Indirect UI (# deps), Elaps=Time Elapsed, Thrpt=Throughput, Cstm=Custom Measurement (gecko provides an accumulated result value for multiple subtests). Most applications show no performance difference when all indexing operations are checked. Two applications do not perform any unchecked indexing by default, and two have at least one impactful checked indexing operation. Time elapsed is measured by inserting a timer into the source code to avoid timing setup, and throughput (requests per second) is measured using wrk [wrk Project 2021]. Getting gecko's "#Hot UI" values times out.

Application	Binary	#Compiled UI	Same Bin?	Perf Metric	Perf Diff	#Hot UI	Libs Introducing Hot UI	#Converted UI
tantivy	Bmark	D: 0, I: 75 (11)	Y	-	-	-	-	-
rage	Bmark	D: 0, I: 13 (5)	Y	-	-	-	-	-
swc	Bmark	D: 7, I: 53 (10)	N	Elaps	<1%	0	-	All (60)
warp	Ex	D: 0, I: 30 (8)	N	Thrpt	<1%	2	httparse	All (30)
iron	Ex	D: 0, I: 22 (4)	N	Thrpt	<1%	6	crossbeam-deque, httparse	All (22)
RustPython	App	D: 1, I: 88 (14)	N	Elaps	<1%	1	siphasher	All (89)
zola	App	D: 0, I: 246 (23)	N	Elaps	<1%	4	regex, crossbeam-deque	All (246)
gecko	Bmark	D: 6, I: 259 (29)	N	Cstm	<1%	-	-	All (265)
COST	Bmark	D: 6, I: 0 (0)	N	Elaps	11%	2	(self)	threshold-dependent
brotli-decompressor	App	D: 261, I: 0 (0)	N	Elaps	7.7%	170	(self)	threshold-dependent

operations that may end up in the application binary. We observe that far more unchecked indexing exists in application dependencies than in the applications themselves. On average, the ratio of direct to indirect unchecked indexing is 1 to 86. This indicates that writing a safe application is much more involved than just avoiding unchecked indexing in application code; not only do applications typically depend on tens if not hundreds of libraries, but the libraries are also constantly being updated and changed.

As mentioned in Section 4.5, NADER currently requires a representative benchmark provided by the application, or a representative workload together with the application binary. If the performance measurement cannot be conducted in our experimental setup or there is no reasonably synthetic profiling workload, we do not continue with exploring the safety-performance trade-off. Given this limitation, we are left with ten applications out of the initial 27 for the next part of the evaluation, as shown by the ✓in Table 4.

5.3 Automatically Optimizing the Safety-Performance Trade-off

For each application in Table 5, we run Steps 1-4 of the workflow presented in Section 4.1. In Step 1, NADER-Analyzer uncovers the MIR-level unchecked indexing that is compiled into each of the 10 applications. In Step 2, NADER-Converter modifies the source code to convert the uncovered

unchecked indexing to checked indexing and compares the generated binary of the unmodified application with that of the modified application. The "Same Bin?" column in Table 5 lists any binaries that we find to be identical in our deployment context. The two versions of an application may compile to the same binary for a couple reasons: 1) there was no unchecked indexing to begin with; thus both versions of the application are identical, and Rust often generates the same binary, or 2) the bounds checks from the converted indexing operations are elided during compilation, producing identical results to the original unchecked indexing operations and hence the same binary. If the binaries of the two application versions are identical, no performance or safety difference is possible, so NADER returns the application binary as is. Applications `tantivy` and `rage` fall under this category. We use the `default-tokenize-alice` benchmark for `tantivy` and the `decrypt` benchmark for `rage`.

For the remaining applications, NADER explores the safety-performance trade-off by measuring the difference in performance between the original and the modified versions of the application. We use the benchmark `bench_full` for `swc`; run the `hello` example for `warp`; run the `hello` binary for `iron`; test the application `RustPython` with `pystone` input where the size is 30000; use the `perf_reftest talos` benchmark for `gecko`; and use the `pagerank` binary with the `hilbert` algorithm and `LiveJournal` input for `COST`. For `brotli-decompressor`, we perform several modifications to convert it to a measurable benchmark. We manually expand the macros for inserting unchecked indexing in 263 locations across 50 functions in the source code (using the `cargo-expand` tool), which could otherwise be enabled by the `unsafe` feature. We also use a single compressed file (60MB from an uncompressed 212MB tarball) generated from the complete Silesia corpus [Deorowicz 2003] with compression level 5. For `warp` and `iron`, two server applications, we use `wrk` [wrk Project 2021] to measure the server throughput as requests per second, locally. For `gecko` we use the accumulated result value that `perf_reftest` provides. For the rest, we use elapsed time as the performance metric.

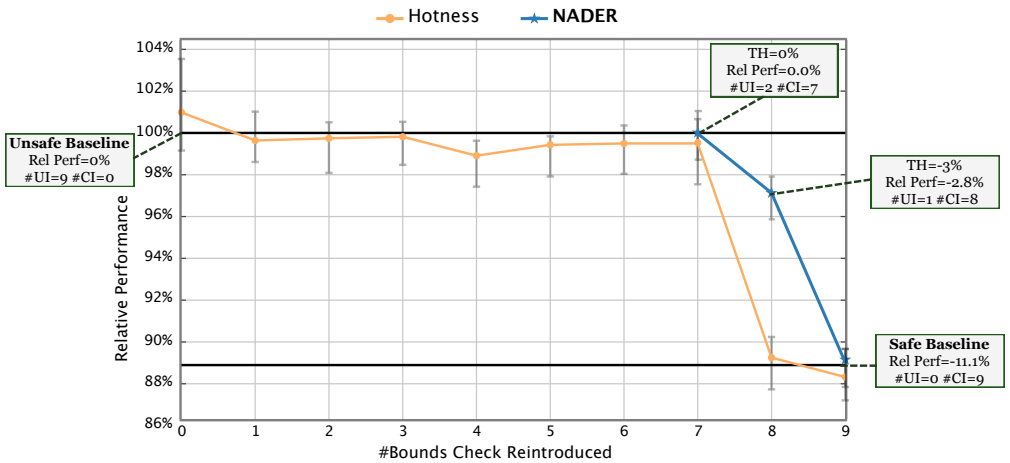

Fig. 6. NADER-Explorer on `COST`. TH=Threshold, Rel Perf=Relative Performance, UI=Unchecked Indexing, and CI=Checked Indexing. NADER automatically identifies **2** impactful bounds checks for the `pagerank` binary with threshold **0-2%** and the most impactful bounds checks with threshold **3-11%**. Each experiment runs ten times and we present the median time with the error bars showing 95% confidence intervals.

As shown in the "Perf Diff" column of Table 5, 6 out of the 8 remaining applications show a performance difference below 1%. Given that this small performance difference is below the

inherent performance variability of our experimental setup, NADER concludes that there is no performance benefit and makes all unchecked indexing checked. To keep the developers informed, NADER also conducts a profiling run with `callgrind` and reports the source code locations of the unchecked indexing that gets executed during the profiling run. In the "#Hot UI" column of `swc` we can see that all unchecked indexing is actually in cold code. For `warp`, `iron`, `zola`, and `RustPython`, a few instances of unchecked indexing are executed.

After this step, two applications remain: `COST` and `brotli-decompressor`. For these two, NADER proceeds with identifying the most impactful instances of unchecked indexing. NADER uses NADER-Explorer to search for the smallest set of performance-critical unchecked indexing according to the overhead threshold and converts all other cases.

Figure 6 shows the results of NADER on `COST`. The exploration process takes around 5 minutes to execute in our experimental setup. Using the hotness-based greedy search, NADER-Explorer constrains the search space to only two (out of nine) source locations corresponding to unchecked indexing. Then, NADER-Explorer uses the one-checked slowdown iterative search, yielding a better result than solely using the hotness heuristic. Note that the two source code locations are in the same statement and are of the same hotness (same execution count). Thus, the hotness-based greedy approach can not distinguish the precedence and sorts them arbitrarily. NADER instead returns a deterministic result and reports that one bounds check affects the performance more than the other.

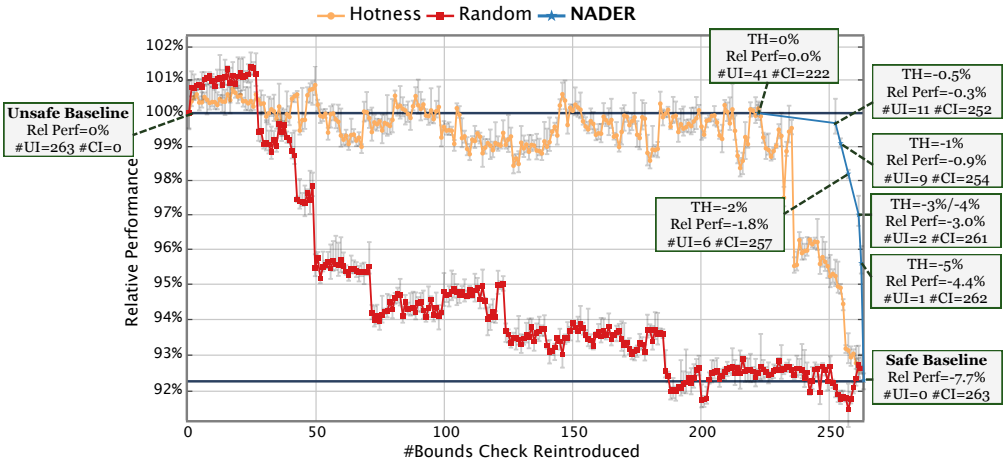

Fig. 7. NADER-Explorer on `brotli-decompressor`. TH=Threshold, Rel Perf=Relative Performance, UI=Unchecked Indexing, and CI=Checked Indexing. NADER finds the minimal set of unchecked indexing given each threshold, and significantly outperforms the hotness-based greedy search. The random heuristic results used in Figure 4 are included for comparison. Each experiment runs ten times and we present the median time with the error bars showing 95% confidence intervals.

Figure 7 shows the results of NADER on `brotli-decompressor`. The exploration process takes less than 30 minutes in our experimental setup. Using the hotness-based greedy search, NADER-Explorer isolates a minimal set of 41 unchecked indexing operations that yield the same performance as the unsafe baseline and then proceeds to convert them with the one-checked-based iterative search. By setting -0.5%, -1%, -2%,..., -8% as the overhead threshold, NADER-Explorer yields a minimal set of unchecked indexing for each threshold, balancing the safety-performance trade-off. As shown in Figure 7, using only one unchecked index reduces the overhead from 7.7% to 4.4%,

using two reduces overhead to 3.0%, and using nine reduces overhead to a mere 0.9%. NADER generates the binary for the application developers to use immediately, as well as a detailed report of the source code locations of the unchecked indexing left in the binary. With a drastically reduced number of unchecked indexing (from 263 to only 9 with 0.9% performance overhead), the application developers can manually audit these cases and then either verify that all these unsafe code locations are indeed sound or rewrite the code to avoid them.

Note that all of these performance measurement results only apply to our evaluation context. Yet, NADER is context-aware by construction and makes different trade-offs in different deployment contexts. In addition, the application developer can specify different thresholds to express different fine-grained safety-performance trade-off requirements in different deployment contexts. For example, when deploying `brotli-decompresor` server-side–assuming the environment is more contained behind layers of protection–one can be more performance-driven and specify a threshold of 0% while still enjoying some safety improvement; whereas one might be more concerned about safety client-side and prefer minimal unchecked indexing there.

6 RELATED WORK

6.1 Safety-Performance Trade-offs

Trading Safety for Performance. Similar to Rust, other type-safe languages have some unsafe features that are sometimes used for performance reasons. For example, OCaml has `Obj.magic` (unsafely forces a type cast between any two OCaml types), while Haskell has loopholes (e.g., `unsafeCoerce`) that can bypass typing and module encapsulation [Terei et al. 2012].

Trading Performance for Safety. For inherently unsafe and performance-oriented languages such as C, prior work has proposed sacrificing performance for safety. One prime example is the introduction of bounds checks to enforce spatial safety of C with a sizeable (67%) runtime overhead [Nagarakatte et al. 2009]. Others offer weaker security guarantees but with more efficient checks [Akritidis et al. 2008; Akritidis et al. 2009; Dhurjati et al. 2006; Ruwase and Lam 2004].

6.2 Cost of Runtime Safety Checks

Safety Checks Blocking Optimizations. Apart from the local cost of executing safety checks, prior work has identified the potential cascading effects of safety checks. Notably, Bodik et al. [Bodík et al. 2000] underline that bounds checks in Java introduce exception points that cannot be bypassed due to the precise exception semantics, and thus restrict the applicability of optimizations. We make similar observations about the collateral damage of bounds checks in the context of Rust and LLVM.

Mitigating the Cost of Safety Checks. To avoid the high cost of runtime safety checks, prior work has employed various compiler-based techniques. For type-safe languages that protect the code with bounds checks, prior work has proposed automatic, compile-time elimination of fully redundant bounds checks using value-range analysis [Patterson 1995; Rugina and Rinard 1999] and of partially redundant checks using iterative data-flow analysis [Gupta 1993; Kolte and Wolfe 1995].

In Java, most bounds check elimination tools work at runtime [Bodík et al. 2000; Qian et al. 2002; Würthinger et al. 2007], whereas Rust relies on the LLVM compiler infrastructure to eliminate the cost of bounds checking statically. LLVM supports elimination of redundant expressions and hoisting loop invariant checks out of loops. But given the LLVM community's focus on C/C++, there is room for improvement in terms of mitigating the cost of bounds check patterns observed in Rust code. NADER can guide compiler engineers towards mitigating the costs of more types of bounds checks, enabling them to automatically make safe Rust code more performant.

6.3 Performance Profiling

Software profilers identify where programs spend most of their time but without any other meaningful feedback to programmers about where optimizations would matter the most. COZ [Curtsinger and Berger 2018] introduced causal profiling to address this problem. Causal profiling calculates the impact of potential optimizations by virtually speeding up parts of the code at runtime. COZ would not help quantify the performance benefit of bounds check removal, as speeding up specific source code lines cannot capture the cascading effect of a bounds check removal within the compiler. Similar to COZ, NADER performs empirical experiments, but contrary to COZ, NADER fully evaluates the overheads of bounds checks by observing end-to-end application performance, including the effects of bounds checks on compiler optimizations.

6.4 Automated Threshold-Based Trade-offs

Log20 [Zhao et al. 2017] automatically places log statements to satisfy a user-specified performance threshold. Similarly, NADER can automatically perform threshold-based decision-making, but the trade-off is between performance and safety.

ASAP [Wagner et al. 2015] automatically and selectively adds sanity checks to increase code safety without exceeding the user-specified performance overhead. Similarly to NADER, ASAP enables developers to selectively trade-off minimal performance overheads for maximal security. But contrary to ASAP's focus on the inherently insecure C and C++, NADER focuses on the inherently safe Rust. Therefore, instead of adding safety checks in the context of an unsafe language, NADER evaluates the performance impact of manually-removed checks in a safety-first language. Our insight is not that some safety checks do not have performance impact (e.g., in cold code) but that expert developers often do not realize the performance impact of the default safety checks and unnecessarily harm the safety of their code.

6.5 Formal Methods for Proving Rust's Safety

The RustBelt project [Jung et al. 2017] is very promising, and the world that it envisions – where memory- and type-safety need not be sacrificed to achieve better performance – is superior to the world NADER envisions – where the choice of points on this trade-off is ceded to the application developer or user, rather than library developers operating with limited information. The proof burden in RustBelt and other verification tools is likely too high to expect widespread adoption in a vast ecosystem of libraries like Rust's. RustBelt and NADER are complimentary: NADER can be used to eliminate unnecessary uses of unchecked indexes, while RustBelt could be used to prove safety in the remaining cases.

7 CONCLUSION

We conduct a study that indicates that using unchecked indexing is rarely justified by the performance in a given context. Furthermore, different contexts can change the overhead of checked indexing, so different checked-unchecked indexing choices may need to be made per context. We also show that most unchecked indexing occurs in application dependencies, thus writing a safe application requires constant supervision of all dependencies. In response to our findings, we propose an approach that automatically frees application developers from these hard-coded and often counterproductive trade-offs and optimizes applications according to end-user contexts. We implement NADER, a tool that reintroduces checked indexing up to a user-specified overhead threshold, and show that using NADER on several end-to-end Rust-based applications enables a significant reduction of unchecked indexing without sacrificing performance. In the future, we plan to extend NADER to automatically make safe code more performant, further reducing the need

for unsafe code like unchecked indexing. We can use NADER to pinpoint a narrow set of the most impactful bounds checks in a given application and direct static analysis techniques to elide these high-impact bounds checks automatically. This would improve application performance without sacrificing safety and would enable a relatively low compilation overhead due to the narrow target scope. Future work will also involve exploring other types of performance-motivated unsafe code in Rust.

ACKNOWLEDGMENTS

We thank the reviewers for their invaluable comments and suggestions. This work was supported by Google and the National Science Foundation (NSF) through Grants CCF-2028733, CCF-1814654, and CCF-2119070. All opinions, findings, conclusions, and recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES

P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. 2008. Preventing Memory Error Exploits with WIT. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*. 263–277. https://doi.org/10.1109/SP.2008.30

Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense Against Out-of-Bounds Errors. In *18th USENIX Security Symposium (USENIX Security '09)* (18th usenix security symposium (usenix security '09) ed.). USENIX. https://www.microsoft.com/en-us/research/publication/baggy-bounds-checking-an-efficient-and-backwards-compatible-defense-against-out-of-bounds-errors/

Jyrki Alakuijala, Andrea Farruggia, Paolo Ferragina, Eugene Kliuchnikov, Robert Obryk, Zoltan Szabadka, and Lode Vandevenne. 2019. Brotli: A General-Purpose Data Compressor. *ACM Transactions on Information Systems* 37, 1 (Jan. 2019), 1–30. https://doi.org/10.1145/3231935

Apple. 2021. Swift Programming Language. https://swift.org/. [Online; accessed April-2021].

Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. 2020. How Do Programmers Use Unsafe Rust? *Proc. ACM Program. Lang.* 4, OOPSLA, Article 136 (Nov. 2020), 27 pages. https://doi.org/10.1145/3428204

AWS. 2021. Firecracker. https://firecracker-microvm.github.io/. [Online; accessed April-2021].

BLAKE3 Team. 2021. BLAKE3. https://github.com/BLAKE3-team/BLAKE3. [Online; accessed April-2021].

Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. 2000. ABCD: Eliminating Array Bounds Checks on Demand. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 321–333. https://doi.org/10.1145/349299.349342

Cloudflare. 2021. Boring Tun. https://github.com/cloudflare/boringtun. [Online; accessed April-2021].

Cloudflare. 2021. quiche. https://github.com/cloudflare/quiche. [Online; accessed April-2021].

Charlie Curtsinger and Emery D. Berger. 2018. Coz: Finding Code That Counts with Causal Profiling. *Commun. ACM* 61, 6 (May 2018), 91–99. https://doi.org/10.1145/3205911

S Deorowicz. 2003. Silesia Compression Corpus. Silesian University of Technology, Poland.

Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. 2006. SAFECode: Enforcing Alias Analysis for Weakly Typed Languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 144–157. https://doi.org/10.1145/1133981.1133999

Diesel Team. 2021. DIESEL: A safe, extensible ORM and Query Builder for Rust. https://github.com/diesel-rs/diesel. [Online; accessed April-2021].

Alan A.A. Donovan and Brian W. Kernighan. 2015. *The Go Programming Language* (1st ed.). Addison-Wesley Professional.

Dropbox. 2020. Dropbox/Rust-Brotli-Decompressor. https://github.com/google/brotli.

Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. 2014. The Matter of Heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (Vancouver, BC, Canada) *(IMC '14)*. Association for Computing Machinery, New York, NY, USA, 475–488. https://doi.org/10.1145/2663716.2663755

Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. 2020. Is Rust Used Safely by Software Developers?. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 246–257. https://doi.org/10.1145/3377811.3380413

fnm Team. 2021. Fast Node Manager. https://github.com/Schniz/fnm. [Online; accessed April-2021].

Google. 2021. FlatBuffers. https://google.github.io/flatbuffers/. [Online; accessed April-2021].

Rajiv Gupta. 1993. Optimizing Array Bound Checks Using Flow Analysis. *ACM Lett. Program. Lang. Syst.* 2, 1–4 (March 1993), 135–150. https://doi.org/10.1145/176454.176507

Hyperium. 2021. Tonic. https://github.com/hyperium/tonic. [Online; accessed April-2021].

InfluxData. 2021. Flux - Influx data language. https://github.com/influxdata/flux. [Online; accessed April-2021].

Iron Team. 2021. Iron. https://github.com/iron/iron. [Online; accessed April-2021].

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–34.

Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

Priyadarshan Kolte and Michael Wolfe. 1995. Elimination of Redundant Array Subscript Range Checks. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation* (La Jolla, California, USA) *(PLDI '95)*. Association for Computing Machinery, New York, NY, USA, 270–278. https://doi.org/10.1145/207110.207160

Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. 2018. Splinter: Bare-Metal Extensions for Multi-Tenant Low-Latency Storage. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 627–643. https://www.usenix.org/conference/osdi18/presentation/kulkarni

Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*.

Nicholas D. Matsakis and Felix S. Klock. 2014. The Rust Language. *Ada Lett.* 34, 3 (Oct. 2014), 103–104. https://doi.org/10.1145/2692956.2663188

Frank McSherry, Michael Isard, and Derek G. Murray. 2015. Scalability! But at what COST?. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. USENIX Association, Kartause Ittingen, Switzerland. https://www.usenix.org/conference/hotos15/workshop-program/presentation/mcsherry

Mozilla. 2021. Gecko. https://developer.mozilla.org/en-US/docs/Mozilla/Gecko. [Online; accessed April-2021].

Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 245–258. https://doi.org/10.1145/1542476.1542504

OpenSSL. 2021. OpenSSL. https://www.openssl.org/. [Online; accessed April-2021].

Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. 2016. NetBricks: Taking the V out of NFV. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 203–216. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/panda

Jason R. C. Patterson. 1995. Accurate Static Branch Prediction by Value Range Propagation. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation* (La Jolla, California, USA) *(PLDI '95)*. Association for Computing Machinery, New York, NY, USA, 67–78. https://doi.org/10.1145/207110.207117

Feng Qian, Laurie Hendren, and Clark Verbrugge. 2002. A comprehensive approach to array bounds check elimination for Java. In *International Conference on Compiler Construction*. Springer, 325–341.

Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding memory and thread safety practices and issues in real-world Rust programs. 763–779. https://doi.org/10.1145/3385412.3386036

rage Team. 2021. rage. https://github.com/str4d/rage. [Online; accessed April-2021].

Radu Rugina and Martin Rinard. 1999. Automatic Parallelization of Divide and Conquer Algorithms. In *In Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 72–83.

Rust Graphics Mages. 2021. gfx. https://github.com/gfx-rs/gfx. [Online; accessed April-2021].

Rust Language Team. 2021. Unsafe Rust. https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html. [Online; accessed April-2021].

RustPython Team. 2021. RustPython. https://github.com/RustPython/RustPython. [Online; accessed April-2021].

Olatunji Ruwase and Monica S Lam. 2004. A Practical Dynamic Buffer Overflow Detector. In *NDSS*, Vol. 2004. 159–169.

Servo Project Developers. 2021. Servo project. https://servo.org/. [Online; accessed April-2021].

Sonic Team. 2021. Sonic. https://github.com/valeriansaliou/sonic. [Online; accessed April-2021].

swc Team. 2021. swc. https://github.com/swc-project/swc. [Online; accessed April-2021].

tantivy Team. 2021. tantivy. https://github.com/tantivy-search/tantivy. [Online; accessed April-2021].

David Terei, Simon Marlow, Simon Peyton Jones, and David Mazières. 2012. Safe Haskell. *SIGPLAN Not.* 47, 12 (Sept. 2012), 137–148. https://doi.org/10.1145/2430532.2364524

TiKV Project. 2021. TiKV. https://github.com/tikv/tikv. [Online; accessed April-2021].

Timber. 2021. Vector. https://github.com/timberio/vector. [Online; accessed April-2021].

J. Wagner, V. Kuznetsov, G. Candea, and J. Kinder. 2015. High System-Code Security with Low Overhead. In *2015 IEEE Symposium on Security and Privacy*. 866–879. https://doi.org/10.1109/SP.2015.58

Warp Team. 2021. warp. https://github.com/seanmonstar/warp. [Online; accessed April-2021].

Wasmer. 2021. Wasmer. https://wasmer.io/. [Online; accessed April-2021].

Josef Weidendorfer, Markus Kowarschik, and Carsten Trinitis. 2004. A tool suite for simulation based analysis of memory access behavior. In *International Conference on Computational Science*. Springer, 440–447.

wrk Project. 2021. wrk. https://github.com/wg/wrk. [Online; accessed April-2021].

Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. 2007. Array bounds check elimination for the Java HotSpot™ client compiler. In *Proceedings of the 5th international symposium on Principles and practice of programming in Java*. 125–133.

Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. 2017. Log20: Fully Automated Optimal Placement of Log Printing Statements under Specified Overhead Threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 565–581. https://doi.org/10.1145/3132747.3132778

Zola Team. 2021. zola. https://github.com/getzola/zola. [Online; accessed April-2021].