Thesis Dissertation

# MIRAV: MIR-ASSISTED VALIDATOR FOR RUST PROGRAM BINARIES

**Giorgos Paphitis**

**UNIVERSITY OF CYPRUS**

**COMPUTER SCIENCE DEPARTMENT**

May 2026

# UNIVERSITY OF CYPRUS
## COMPUTER SCIENCE DEPARTMENT


**MIRAV: MIR-Assisted Validator for Rust Program Binaries**


**Giorgos Paphitis**

Supervisor
Dr. Elias Athanasopoulos


Thesis submitted in partial fulfilment of the requirements for the award of degree of Bachelor in Computer Science at University of Cyprus

I declare that this dissertation and any accompanying software are my own original work, conducted in full compliance with the University of Cyprus Code of Conduct for Research (https://www.ucy.ac.cy/legislation/kodikas-deontologias-kai-politikes/), and with full accountability on my part for the accuracy, integrity, and validity of all content.

May 2026

# Acknowledgments

I would like to express my sincere gratitude to my supervisor, Dr. Elias Athanasopoulos, for their invaluable guidance, continuous support, and constructive feedback throughout the course of this thesis. Their expertise and encouragement greatly contributed to the completion of this work. I am also thankful to the faculty and staff of the Computer Science Department for providing the resources and academic environment necessary for my studies. Finally, I would like to thank my family and friends for their patience, motivation, and unwavering support during my undergraduate journey.

# Summary

Rust offers strong memory safety with minimal runtime support, therefore producing fast programs. These properties are primarily enforced during compilation, eliminating common memory bugs often found in C/C++ programs. However, the absence of runtime verification means that the compile-time safety guarantees may be invalidated in the binary prior to execution. As a result, vulnerabilities can be injected into binaries through rewriting techniques, diminishing the security Rust aims to provide.

In this thesis, we implement a validator that verifies whether a Rust-produced binary adheres to Rust's safety guarantees enforced during its compilation. We explore the idea of using the program's Mid-level Intermediate Representation as safety proof to compare against the binary and detect inconsistencies. By combining this idea with static analysis, we develop a scalable validator that checks both spatial and temporal memory rules. We evaluate our tool on multiple Rust projects hosted on GitHub, containing real-world applications, and achieve a 76% classification accuracy on the unmodified versions, while their modified (unsafe) versions were all correctly classified.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Programming languages can be categorized as safe or unsafe. This separation depends on how they handle memory during execution, particularly whether it is handled by the language or the developer. A traditional example of a memory-unsafe language is C/C++. Memory allocation and deallocation are managed exclusively by the developers using functions like malloc and free, offering full control. It also minimizes execution overhead, as it avoids runtime components such as a garbage collector, which introduce additional computational costs. These features are ideal in systems programming and constrained environments, such as IoT devices [18]. The limited computational power and memory of IoT devices require efficient memory management and minimal runtime overhead. However, solely relying on developers to correctly handle memory leads to significant security risks. Developers can make mistakes and critical memory vulnerabilities may be missed in programs, such as buffer overflows and Use-After-Free (UAF).

To address the above security concerns, safe programming languages have been introduced. These languages remove the responsibility of managing memory from the developers. Instead, a runtime is used to manage memory allocation and access, minimizing memory bugs. An example of such a language is Java. Java programs are not compiled directly to native binaries, but instead to bytecode [4]. Bytecode is a platform independent binary format. To execute it, the Java Virtual Machine (JVM) is used. The JVM executes the bytecode and handles all memory allocations, deallocations and accesses [21]. As a result, programs are safe from common memory bugs with no development effort required. However, due to the need for a runtime, significant overhead is introduced, originating from components like the garbage collector [6].

A new type of memory-safe programming languages has been increasingly adopted for development in critical systems and constrained environments. The goal of these languages is to offer strong memory safety, similar to safe languages, while incurring minimal overhead. A prime example is Rust. Rust is a memory-safe programming language with execution speeds comparable to C/C++ [16, 29]. Rust programs run natively on the

machine and do not use a runtime to manage memory. To achieve this, it enforces strict ownership and borrowing rules during compilation and rejects programs that violate them. In cases where safety cannot be determined during compilation, checks are embedded in the compiled binary which verify safety during the program's execution.

The absence of a runtime creates the potential for a different kind of attack. Rust enforces its memory safety rules during compilation. However, there are no guarantees that after compilation these rules remain valid when the binary is executed. This leaves Rust programs susceptible to vulnerabilities introduced through binary rewriting techniques. It has already been shown that attackers can stealthily inject vulnerabilities in programs, submit them to an app store, and remotely exploit them once a large number of users has downloaded the app [27]. Producing vulnerable programs in memory-unsafe languages such as C is straightforward. Doing this in Rust, though, has not yet been widely explored. Louka, A. et al. [19] have shown that such attacks on Rust binaries are possible. As a result, app stores supporting Rust programs may falsely advertise them as secure. This highlights the need to validate Rust-produced binaries to ensure that the safety guarantees enforced by Rust at compile-time remain valid.

There are many techniques for analysing programs and detecting bugs, each presenting different advantages and limitations. One approach is using static analysis and binary lifting to an intermediate representation (IR). Static analysis examines the code of the binary without executing it. Binary lifting attempts to translate the assembly instructions into an IR which can help perform more complex analysis. A popular example is the Binary Analysis Platform (BAP) [15]. The goal of BAP is to lift machine code into an IR that allows formal validation of code. It achieves complete code coverage and lower overhead compared to other approaches. Nevertheless, lifting assembly code is not trivial due to features like vector instructions and indirect jumps.

Another widely used approach for bug detection is dynamic analysis using binary instrumentation techniques. These techniques analyse programs while they are running by dynamically inserting instructions. A commonly used tool is Valgrind [24]. Valgrind instruments a binary and maintains a shadow memory to track bytes and validate memory accesses. As a result, it can accurately detect memory violations that occur during execution. Consequently, it often introduces high execution and memory overhead. This can be mitigated by employing coarse-grained memory tracking, but it may worsen accuracy. Furthermore, code paths are validated only if execution reaches them, so the quality of test cases is crucial for complete analysis. Assuming the aforementioned attacker who deliberately injects vulnerabilities in their program, the test cases they provide can be crafted to avoid the execution paths containing the injected vulnerabilities.

Hybrid approaches are also being explored as a way to mitigate the limitations of the previous techniques. Driller [26] combines fuzzing and concolic execution to eliminate

the dependence on test cases and alleviate the path explosion problem often encountered with concolic execution. These ideas, however, still suffer from the inherent limitations of the methods they use such as significant time and computing resources due to input space size and the SMT solver.

In this thesis, we explore a static analysis approach for evaluating if Rust's safety properties are valid in binaries. We develop a tool that can be used by app stores to verify the safety of Rust-produced binaries before signing them. We adopt the concept of proof-carrying code explored by Necula, G.C [23]. In this scenario, the author assumes an untrusted code producer who provides a binary that needs to be verified. In order to verify it, a safety proof must also be given. This proof certifies the program's adherence to a predefined safety policy. Using this proof, the program's compliance to the safety policy can be quickly verified mathematically. In our approach, the safety policy is Rust's safety rules during compilation. As the proof, we use the Mid-level Intermediate Representation (MIR) of the program. By using the MIR, we lose the mathematical guarantees of formal verification, allowing for a more scalable solution combining traditional static analysis techniques with the additional information provided by the MIR. In this thesis, we employ the validation techniques for Rust binaries introduced by Louka, A. et al. [19] and use the MIR information as the ground-truth to cross-validate them and detect discrepancies. To retrieve the serialization of the MIR of a program during its compilation we use the modified Rust compiler, rustc++ [20].

Our main contributions are as follows.

- We explore the idea of using the MIR as a safety proof to compare against the binary and detect inconsistencies.
- We implement a tool that can evaluate whether Rust's spatial and temporal safety guarantees are valid in a binary.

# Chapter 2

# Background

## 2.1 Rust

Rust is a memory safe programming language with a minimal runtime. Its goal is to provide strong memory safety while minimizing its runtime overhead. It is increasingly being adopted in systems programming, performance-critical software and embedded devices [18]. To achieve this, it enforces strict rules during development. These rules are then validated by the compiler which rejects programs that violate them. In cases where safety cannot be verified at compile-time (e.g. dependence on runtime values like inputs), it embeds checks in the compiled binary to protect it at runtime.

### 2.1.1 Mid-level Intermediate Representation

When a Rust program is compiled, it is lowered to multiple intermediate representations, each containing less semantic information [9]. In particular, source code is converted into the High-level Intermediate Representation (HIR). The HIR is a simplified version of the Abstract Syntax Tree (AST) preserving the program's high-level structure. After this, the HIR is lowered to the MIR [8]. Then, it reaches the code generation stage where the MIR is transformed to LLVM-IR and finally machine code. The MIR is comprised of basic blocks with statements and terminators. It has a similar level of abstraction to LLVM-IR but maintains type and explicit move information. These properties make it the ideal stage for the compiler to perform most of its analysis and validation, like the borrow checker that we discuss further below.

## 2.2 Rust's Spatial Memory Safety

Common vulnerabilities in languages such as C are buffer overflows, which occur when a program accesses memory outside the allocated bounds. Such vulnerabilities can lead

to critical security breaches, like remote code execution. Safe programming languages address these vulnerabilities using a runtime that keeps track of the allocated memory of a program during execution and verifying that all memory accesses do not exceed the allowed bounds. Rust ensures spatial safety at compile-time and, when necessary, embeds runtime checks in the compiled program to prevent out-of-bounds access. Most data types, like vectors and strings, are not directly indexed in the binary. Instead a library function is called which checks whether the desired index is within the object's bounds before returning it. However, fixed-size buffers and slices are directly accessed and bounds checks may be embedded to protect them.

### 2.2.1 Fixed-size Buffers

Fixed-size buffers are allocated on the stack. They are always initialized with either an array literal or a repeat expression [1]. Buffers can be accessed directly in the program using the traditional square bracket notation. During compilation, the compiler attempts to identify the index used every time a buffer is accessed and ensure that it is not out-of-bounds. Complex indexing expressions or when the index depends on a value known at runtime can prevent the compiler from ensuring safety at compile-time. In such cases, it inserts an inline safety check in the compiled program that will first check the index before accessing the buffer and panic if it detects an attempted out-of-bounds access [2].

### 2.2.2 Slices

Slices [11] are references to a contiguous block of elements in a collection, like buffers. Particularly, a slice is a fat pointer containing the address of the first element and the length. Slices can only be indexed if the object they originate from can also be indexed. For example, string slices cannot be indexed using the square bracket notation as this is not allowed for strings either. However, slices of vectors and buffers can be indexed. When a slice is indexed, it is directly accessed using pointer arithmetic, and if required, an inline bounds check is embedded, similar to fixed-size buffers.

## 2.3 Rust's Temporal Memory Safety

Common examples of temporal safety bugs are dangling pointers and UAF bugs which pose great security risks. In C/C++, a dangling function pointer or VTable pointer can lead execution to attacker controlled code. These bugs originate from mishandling of memory from developers. To mitigate such bugs, safe languages move the responsibility of handling memory from the developer to the garbage collector. A garbage collector is a

runtime component that manages dynamic memory allocation and deallocation [5]. Discovering unreachable memory blocks in order to reclaim their space is not trivial. Many algorithms are used to improve accuracy and performance, but they can incur substantial overhead [17]. Rust achieves temporal safety entirely at compile-time, introducing no runtime overhead. It does this by employing strict ownership and borrowing rules during development.

### 2.3.1 Borrow Checker

The borrow checker is one of the most important components of the Rust compiler. It operates on the MIR of the program and imposes strict temporal rules to guarantee temporal safety [3], eliminating bugs like UAF and double free. To achieve this, it introduces concepts like ownership and lifetimes which we explore below.

### 2.3.2 Ownership

Ownership [13] is a core concept in Rust that enables it to ensure temporal safety without needing a garbage collector. It enforces a set of rules that allow it to manage memory allocation and reclamation entirely at compile-time. These rules are as follows. (1) Every value has exactly one owner at any time. (2) When the owner goes out-of-scope the value is dropped. A consequence of the first rule is that when a value is moved into another variable, the original variable loses ownership of it and can no longer access it. Together, these rules allow memory to be allocated and reclaimed with no runtime overhead, eliminating the possibility of temporal bugs.

### 2.3.3 References and Borrowing

According the the aforementioned ownership rules, a memory value can have only one owner. But the value can be borrowed as well. When a value is borrowed, its ownership is not transferred but a reference is created. References are similar to pointers, but they have to follow some rules to ensure safety [10]. A reference can be either mutable or immutable. If a reference is mutable, then it can be used to alter the value. The rules for references are as follows. (1) References must not outlive the owner of the value they borrow. (2) At any point, there can exist either multiple immutable references or only one mutable reference for a value. With the above rules, Rust provides the ability to borrow values while maintaining the strong safety guarantees offered by ownership.

### 2.3.4 Lifetimes

To verify whether a reference's scope exceeds the owner's scope, lifetimes are used. Lifetimes are a generic concept that define the scope of a reference [7]. Every reference must have a lifetime. Most of the time, a reference's lifetime can be automatically inferred by the compiler. For example, the scope of a locally defined reference is the block it is defined in. In some scenarios, the compiler cannot figure out the scope of a reference and requires the developer to explicitly set it. One such scenario is when a function takes references as arguments and returns one of them [12]. Reasoning about the function's possible return paths may lead the compiler to infer an incorrect lifetime for the returned reference. To avoid such mistakes, the compiler requires the developer to specify the lifetimes to ensure safety.

## 2.4 Static Binary Analysis

Static binary analysis is the process of analysing a binary at rest. This method is considerably faster and requires less resources than dynamic analysis techniques, such as binary instrumentation. Furthermore, it is safer as the binary is not executed and it enables analysis of binaries built for different platforms or require a special environment. Finally, it offers full code coverage because it does not depend on execution reaching new code paths. However, static analysis suffers from indirect control flows, like jump tables for switch statements. Another important limitation is obfuscation, which is designed to confuse disassemblers. For example, a jump instruction that targets the middle of another instruction. This takes execution on a different path than the one assumed by the disassembler. Tools based on static analysis often have to trade-off between completeness and soundness. Trying to achieve complete detection of vulnerabilities often requires over-approximations which may result in many false positives. Whereas, more precise approaches might miss vulnerabilities.

## 2.5 Threat Model

We consider the adversary to be an application developer who deliberately alters the produced binary of their Rust program to violate Rust's safety guarantees and introduce vulnerabilities before submitting it to an app store. Once a large number of users download the malicious app, the adversary can remotely attack the users by exploiting the inserted vulnerabilities. An example of this has been presented by Wang et al. [27]. In their scenario, a program that contained hidden malicious control flow paths passed the app review process of iOS. The hidden paths were then remotely activated through planted vulnera-

bilities in the program. Because there was no change in the code of the program, it did not violate its signature and the apps were allowed to be executed. Analysis performed by app store providers is limited to binaries only, as access to source code will be refused by organizations since it may contain company secrets. In contrast, languages like Java submit their programs to app stores as a collection of class files containing bytecode. In Rust, the MIR follows a similar level of abstraction to Java bytecode. We argue that requesting developers to submit the MIR alongside the binary is feasible as it is similar to Java bytecode. Most of Rust's memory checks during compilation, such as the borrow checker, as well as the MIR Interpreter (Miri) [14], run on the MIR. Due to this, we assume the possibility of restoring the compiler's state from the MIR serialization. While this is challenging due to its dependence on the compiler's internal state, it enables validation of the provided MIR itself to ensure its adherence to the safety guarantees before cross-validating it with the binary. This assumption is necessary because the attacker may also modify the submitted MIR in order to hide injected vulnerabilities.

# Chapter 3

# Architecture

We have developed a static analysis tool capable of receiving a binary and its MIR serialization and evaluate whether the program is safe to execute. In particular, it detects whether the program complies with Rust's memory safety guarantees that were enforced during its compilation. We have also implemented the functionality of automatically injecting buffer overflow vulnerabilities into the binary by altering the emitted bounds checks. The pipeline of the tool can be seen in Fig. 3.1.

## 3.1 Design Goals

Based on the threat model defined before, validation of source code is not relevant as we aim to detect binaries that have been altered after compilation. As a result, binary analysis is the only feasible solution. However, binaries lack semantic information that can be beneficial during analysis, such as type information. To address this, we combine static analysis of the binary with information extracted from its MIR. We treat the information from the MIR as the ground-truth state of the program and use it to examine whether it matches the binary. We selected static analysis due to its scalability and complete code coverage. We adopt a conservative approach where programs are classified as unsafe not only when an explicit safety violation is detected, but also when discrepancies between the binary and the MIR are observed, such as less bounds checks than expected.

## 3.2 Safety Violations Detected

### 3.2.1 Spatial Violations

Our tool is able to detect spatial violation attempts related to bounds checks of fixed-size buffers and their slices. Specifically, it is able to detect three kinds of violations. (1) Modifying the size checked in the bounds check to allow for out-of-bounds indices on

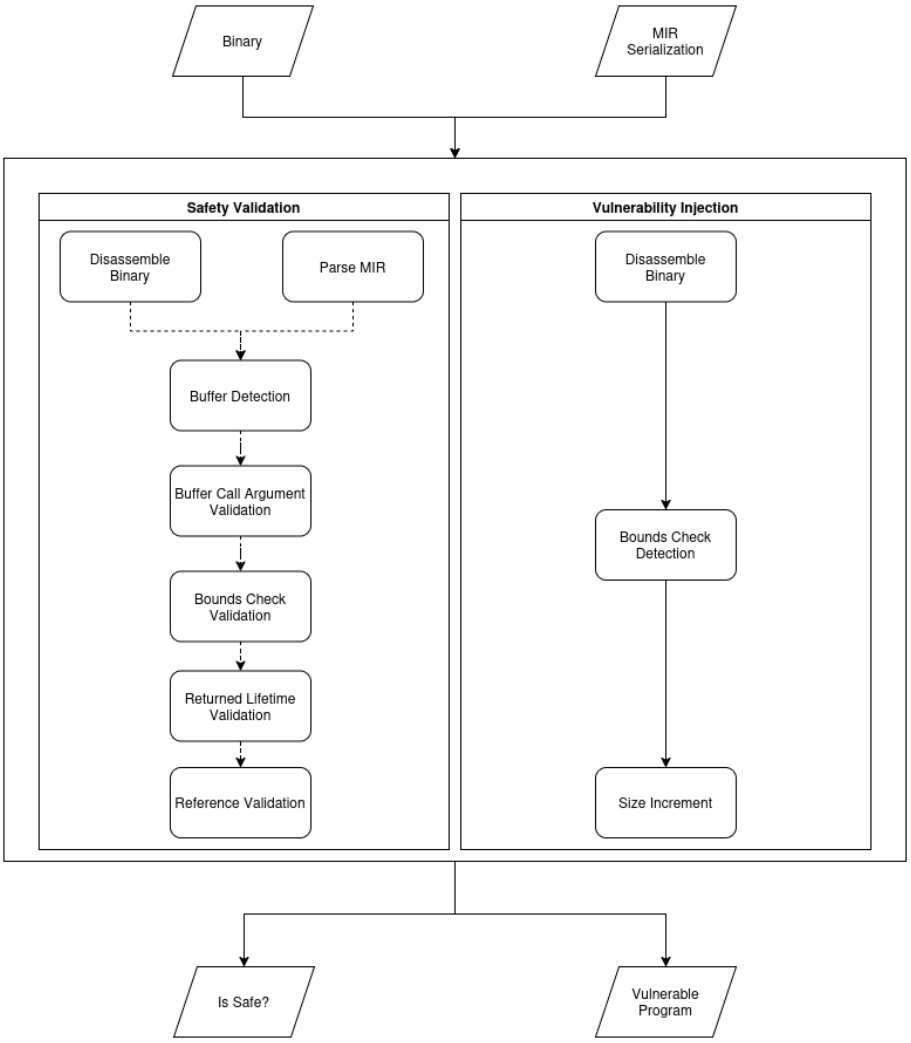


Figure 3.1: Tool pipeline

these objects. (2) Completely removing a bounds check. (3) Providing a larger value for a buffer's size when it is passed as an argument to a function.

### 3.2.2 Temporal Violations

Temporal violations relate to the borrowing rules Rust enforces at compile-time. These rules are as follows. (1) A reference's scope must not exceed the owner's scope. (2) At any point, there may exist either multiple immutable references or one mutable reference. We currently support validation for references of fixed-size buffers and their slices. To gather scope information about the binary, we use the DWARF symbols found in the binary when compiled with debugging information. Furthermore, we verify that any function returning a reference returns one of its arguments, where that argument is itself a reference.

## 3.3 Safety Validation

This is the tool's main functionality. It takes a binary and its serialized MIR as input, cross validates them and classifies the binary as safe or unsafe to execute. First, the binary is fully disassembled and its MIR is parsed. Then, the following stages of the pipeline are executed on each function independently. For crates, we extract the compiled Rust library and validate each object file independently.

### 3.3.1 Buffer Detection

The first stage is to identify the fixed-size buffers to track, along with their slices. We use the MIR to locate them using type information of local declarations and identified initialization patterns. Using the extracted information, we match the buffers and slices in the binary. Failing to discover a buffer or slice identified from the MIR flags the binary as unsafe.

### 3.3.2 Buffer Call Argument Validation

When a buffer is passed as an argument, its size may also be provided, depending on the argument type. In such cases, we verify that the correct size is passed alongside the buffer. To do this, we first discover all call sites in the MIR where a tracked buffer is used as an argument. Then, we find the identified call sites in the binary and checks if the expected buffers with their correct sizes are passed. Failure to discover a call site marks the binary as unsafe. Buffer arguments that don't need to pass their size are verified along the function's locally defined buffers.

### 3.3.3 Bounds Check Validation

The final spatial check is to validate all bounds checks for both the identified buffers and slices. We locates all bounds checks contained in the function and discover the object they are protecting. If the check is protecting a tracked buffer or slice, then we verify that the check uses their correct size. We also count how many bounds checks we find for each of the tracked objects and compares it against the number of checks found in the MIR for them. If we find less checks in the binary than in the MIR for an object then it is classified as unsafe.

### 3.3.4 Returned Lifetime Validation

If the function analysed returns a reference, then it must reference either some global data or one of its arguments. Otherwise, returning a reference to a local variable would lead to

a UAF bug. When it returns one of the arguments, Rust must know which of the arguments it may be in order to verify their scopes in the call sites. To verify such cases, we parse the signature of the function from the MIR and check if it returns a reference. If it does, then we find its arguments with the same lifetime, and using their type, match the register they will be passed through based on the x86-64 calling convention. We then recursively begin to follow the binary's execution flow, tracking the arguments and verifying if each branch returns one of the identified arguments. Any branch that does not return a tracked argument causes the program to be considered as unsafe.

### 3.3.5 Reference Validation

The last stage of the pipeline is to verify the borrowing rules. In this stage, no additional information is extracted from the MIR. We build a hierarchy of the function's scopes using the binary's DWARF debugging symbols. We then discover all references created for the tracked objects along with the scope they are created in. Whenever we find a reference being used, we check whether it is out-of-scope based on the scope the reference was created in. If a reference is used to mutate the original object, we also check if the reference's scope overlaps with any other reference for that object.

## 3.4 Vulnerability Injection

The second functionality offered by our tool is buffer overflow injection. It only takes the binary as an input and exports a modified copy of it. To inject the vulnerabilities, we locate all bounds checks and increment the size they check against by one. As a result, we introduce the ability to overflow a buffer by one assuming no further validation was performed by the developer (e.g. manual if check in original source code).

## 3.5 Limitations

Our tool supports detection of buffers with scalar values, like integers and floats, and their slices. This limits safety validation as only these buffers and slices are tracked. Our proposal inherits the limitations of static binary analysis such as indirect control flow and obfuscation. It may yield a considerable amount of false positives due its conservative design regarding binary and MIR discrepancies. For example, indirect control flow may lead to failure in discovering a call site that should have been checked. Currently, we have developed and evaluated the tool on Rust programs compiled in debug mode (non-optimised). Finally, it depends on debug information like the symbol table for MIR and binary function matching and the DWARF symbols for scope information.

# Chapter 4

# Implementation

## 4.1 Spatial Memory Safety Validation

Most of Rust's data types are safe from out-of-bounds bugs because indexing is performed through implicit or explicit calls to Rust's library methods, which ensure that the desired index is within bounds. For example, the square bracket notation is not allowed for strings, whereas for vectors it is converted into a function call during compilation. However, fixed-size buffers and slices are exceptions since they compiled into direct memory operations. During compilation, Rust attempts to verify that all accesses to these types are valid. Whenever it is unable to do so, it emits an inline bounds check to protect them at runtime. If an out-of-bounds access is attempted, the program will panic and terminate execution.

Our tool analyses programs and their serialized MIR to verify that the compiler-emitted checks have not been manipulated. It supports validation of buffers with scalar values and their slices. Furthermore, we verify that any call site passing one of the tracked buffers as an argument also provides the correct size.

### 4.1.1 Buffer Bounds Check Validation

Fixed-size buffers are stored in the stack and are directly accessed by the program. The check is a simple comparison of the desired index with an immediate value representing the buffer's size. The immediate value can easily be altered to allow for out-of-bounds access.

To detect such cases, our tool validates each function independently, performing the following steps: (1) Match each function's serialized MIR to the disassembled function based on their Fully Qualified Name (FQN). (2) Retrieve the function's fixed-size buffers from its MIR serialization. (3) Find the initialization of each buffer in the disassembly and store its starting stack offset. (4) Locate all buffer bounds checks in the binary. (5) Find the buffer associated with each check. (6) Compare the associated buffer's size to

the immediate value used in the check. (7) Count the number of checks found in the MIR for each buffer and compare it with the number found in the disassembly.

**Step 1.** Each function's MIR serialization contains its FQN. We use the name to match it to its corresponding function in the binary by demangling the symbols in the symbol table. However, the FQN in the MIR may not match exactly with a symbol in the binary. This is because the compiler needs to uniquely identify multiple implementations for the same type in the source code as shown in Listing 4.1. To do so, it emits an implementation ID inside the FQN in the form of {impl#0}. This is not maintained in the final symbol name but is replaced by the type of the implementation. To mitigate this, we search for these symbols by matching the rest of the FQN.

```
1 struct Foo;
2 impl Foo {
3   // FQN: Foo::{impl#0}::a
4   fn a(&self) {}
5 }
6 impl Foo {
7   // FQN: Foo::{impl#1}::b
8   fn b(&self) {}
9 }
```

Listing 4.1: Multiple implementation blocks for the same type

**Step 2.** We search the MIR to find all of the function's fixed-size buffers. In particular, we analyse all statements to locate buffer initializations. Two kinds of initializations have been identified, an explicit array literal and a repeat expression. For each discovered initialization, we store the destination buffer's local declaration index, size, values, and entry type. In addition to the locally defined buffers, we also extract the function's fixed-size buffer arguments, as they follow an identical bounds checking procedure.

**Step 3.** Using the information gathered from Step 2, we search for the buffers in the binary more accurately. Consider the explicit array literal initialization shown in Listing. 4.2. This could correspond to one buffer of size 5. However, it could also correspond to two buffers of sizes 3 and 2. Using the buffer information from the MIR, we can minimize this ambiguity. Based on the type of initialization used for buffers, we have identified different patterns in the binary to initialize them. For explicit array literals, a series of mov instructions is used to move each value in the stack. For repeat expressions, four main techniques were identified: (1) Using AVX instructions. (2) Using a memset call. (3) Using a loop. (4) Using a series of word moves (a buffer of 32 8-bit values initialized with four quad-word moves). These methods are displayed in Listing. 4.3. In the second case, calls to memset may not be directly resolved through the call's target due

to Position Independent Code (PIC). Instead, we identify the callee through relocation information.

For the buffer arguments with fixed size, we use the function's signature from the MIR to estimate the register or stack offset the buffer will be passed through. Specifically, from the signature we resolve the types for all of the function's arguments and the return type. Using the types and their order, we estimate the register each argument will be passed in. For example, string arguments require two registers since they must also pass their size. We then search from the start of the function to find where these arguments are stored on the stack.

```
1  // Explicit Array Literal
2  mov dword [rsp + 0xc], 0xa
3  mov dword [rsp + 0x10], 0xb
4  mov dword [rsp + 0x14], 0xc
5  mov dword [rsp + 0x18], 0xd
6  mov dword [rsp + 0x1c], 0xe
```

Listing 4.2: Ambiguous buffer initialization

```
1  // Explicit Array Literal
2  mov dword [rsp + 8], <first value>
3  mov dword [rsp + 0xc], <second value>
4  ...
5  // Repeat expressions (e.g. [0 as u8; <size>])
6  // 1. AVX instructions
7  xorps xmm0, xmm0 // value to set (zero)
8  movaps xmmword [rsp + 0x100], xmm0  // first 128 bits set
9  movaps xmmword [rsp + 0x1b0], xmm0  // second 128 bits set
10 ...
11 // 2. Memeset  call
12 mov rdi, rsp    // start of buffer
13 mov esi, <value>
14 mov edx, <size> // size
15 call <memset address>
16 // 3. Loop
17 mov rax, qword [rsp]
18 mov qword [rsp], rax
19 cmp rax, <size>
20 jae 0x7ce4    // finished
21 mov rax, qword [rsp]
22 mov dword [rsp + rax*4 + 8], <value> // start of buffer is 0x8
23 add rax, 1 // increment index
24 mov qword [rsp], rax
25 jmp 0x7cbe    // next iteration
26 // 4. Word mov instructions
27 // qword moves 64 bits -> 8 values if entry size is 8 bits
28 mov qword [rsp + 8], 0
29 mov qword [rsp + 0xc], 0
30 ...
```

Listing 4.3: Buffer initialization patterns

**Step 4.** Once we have identified all buffers and matched them to the binary, we validate their bounds checks. When a bounds check detects an out-of-bounds attempt, it terminates execution by calling the panic_bounds_check() function. It does not call the function directly, but instead jumps to a stub that prepares its arguments and then calls it. Each call site consists of five instructions as shown in Listing. 4.4. We discover the bounds checks by locating all jumps targeting any instruction within the identified call sites. Each check has one of the two forms shown in Listing. 4.5. In both cases, they start with a comparison that checks if the index is less than the buffer's size. If the index is valid, the successful branch leads to the code that accesses the buffer, otherwise it jumps to one of the call sites for panic_bounds_check(). In crates, we resolve the call to the panic function through relocation information, similarly to memset, due to PIC.

```
1 mov rdi, qword [rsp + 0x10] // argument setup
2 lea rdx, [rip + 0x515e5]
3 lea rax, [rip - 0x95d] //loads panic_bounds_check() address
4 mov esi, 3
5 call rax
```

Listing 4.4: Panic bounds check all site

```
1 // First form
2 // Fallthrough edge is the successful branch
3 cmp rax, 6 // correct size
4 jae <panic_call_site>
5
6 // Second form
7 cmp rax, 6 // correct size
8 jb <successful_branch>
9 jmp <panic_call_site>
```

Listing 4.5: Buffer bound check

**Step 5.** To find the buffer associated with each bounds check, we follow the successful branch of the check. We identify if any of the first five instructions access the offset of a tracked buffer. The access must also use the checked register from the comparison. We have encountered two buffer accessing patterns as shown in Listing 4.6. The first uses a single operation to access the buffer. The second loads the base of the buffer and then increments the pointer by the index register. If a buffer is correctly identified, then we increment the number of checks discovered for that buffer.

```
1 // Buffer bounds check
2 cmp rax, 6
3 jae <panic_call_site>
4 // Successful branch - First Version
5 mov rax, qword [rsp + 0x10]
6 mov ecx, dword [rsp + rax*4 + 0x1c] // buffer accessed
7
8 // Second Version
```

```
 9  lea rcx, [rsp + 0x1c]
10  mov rdx, rax
11  shl rdx, 1 // Apply stride
12  add rcx, rdx
```

Listing 4.6: Buffer bounds check associated buffer

**Step 6.** For each discovered bounds check, we have identified the buffer they are protecting. Therefore, we compare the immediate value used in the check against the buffer's size. If the value is larger than the size, then there is a buffer overflow vulnerability and the program is classified as unsafe.

**Step 7.** The above process is able to detect if a check has been altered to allow for out-of-bounds indices. However, a bounds check may be erased to prevent detection. For example, exchanging the jump to the panic call site with nop instructions bypasses protection and is missed by the validation procedure, since we use these jumps to locate the bounds checks. To address this, we utilise bounds check information from the MIR. In particular, we count how many bounds checks the MIR contains for each tracked buffer. Then, we compare this with the number discovered in the binary for that buffer. If fewer checks have been identified in the binary, then the program is considered unsafe.

### 4.1.2 Slice Bounds Check Validation

A slice is a fat pointer that is accessed directly in the binary using pointer arithmetic. This means that bounds checks will be emitted for them similarly to buffers. In order to validate these, we discover the slices created for each of the tracked buffers from the MIR and try to identify their range. This is not always possible as their range may be declared using variables instead of immediate values. We then match these initializations in the binary as shown in Listing 4.7. An index function is called with the buffer and range as arguments and returns the created slice along with its size. Both the slice and its size are stored at consecutive stack offsets. Statically resolving the callee from the call instruction is not guaranteed. The index function may be indirectly called which prevents static resolution of the target address due to PIC. To minimize this, along with the possibly unknown range bounds, we perform a best match of the slice initialization with the information available from the MIR.

Slice bounds checks follow a very similar pattern to buffer bounds checks. The detection and slice association procedure is identical to buffers. The only difference is the value they compare against in the check. The slice's size is not a compile-time constant but it is returned by the index function that creates it. Therefore, the check does not use

an immediate value but it uses the stored size from the initialization as shown in Listing 4.8. To validate these we perform two checks. First, we check that after the slice's initialization, the stack offset at which its size is stored at is not altered. Then, for every bounds check found, we verify that from the moment the size is loaded into the register, the register is not altered until it reaches the comparison.

```
1 lea rdi, [rsp + 0xe9] // buffer address
2 mov esi, 0xf
3 call qword [rip + 0x2a7a7d] // index function
4 mov qword [rsp + 0x100], rax // slice address
5 mov qword [rsp + 0x108], rdx // slice size
```

Listing 4.7: Slice initialization

```
1 mov rcx, qword [rsp + 0x10] // slice size
2 mov qword [rsp + 0x18], rax
3 mov qword [rsp + 0xc0], rax
4 cmp rax, rcx // compares against loaded size
5 jae <panic_call_site>
6 mov rax, qword [rsp + 8] // slice
```

Listing 4.8: Slice bounds check

### 4.1.3 Buffer Arguments Validation

When a function takes a buffer as an argument, it's size may also be passed. This depends on the callee's signature. If the argument's type explicitly sets the size, then the size does not need to be passed. When no size is specified, all call sites must provide the buffer's size along with it. We locate the call sites that contain variable-sized buffer arguments and verify if they pass the correct size along with it. First, we discover all call sites that contain a tracked buffer as an argument in the MIR. Through the argument's type we determine if its size will also be provided. After this, we estimate the register they will be passed through based on the calling convention. Using this information, we find the call sites in the binary and check whether the correct buffer with its corresponding size are passed. If a call site is missing, a wrong buffer is passed or an incorrect size is given, then the program is deemed unsafe.

## 4.2 Temporal Memory Safety Validation

Rust's main temporal safety protection comes from the borrow checker which enforces the ownership and borrowing rules. Using these rules, it minimizes temporal memory bugs, such as UAF, at compile-time.

We have developed experimental procedures to validate these rules in the binary. In particular, we perform the following three checks. (1) If references are used out-of-scope.

(2) If the scope of mutable references overlaps with any other reference for the same object. (3) If functions that return a reference return one of their reference arguments. Currently, we support reference tracking for fixed-size buffers and their slices. We have attempted to utilise scope and reference information from the MIR to match in the binary. However, the number of scopes found in the MIR for a function often mismatched the number found in the binary due to optimisations. For this reason, we opted to use only the scopes in the binary for validation and utilise the MIR for lifetime information needed for the third check.

### 4.2.1 Out-of-Scope Reference Access

When a reference is used to access a value, we verify if the instruction is within the scope the reference was created in. We achieve this using the DWARF debugging information. When the program is compiled with debugging symbols, it emits scope information as shown in Listing 4.9. Specifically, for each function a tree structure of lexical blocks is created, each containing a list of addresses the scope applies to. Using this, we determine the scope references are created in and therefore the range they are valid for. We first identify all references created in the binary. A reference is created when the address of a stack offset corresponding to a tracked object is loaded into a register, which is then stored into a second offset. We store the new offset and the scope it was created in. Then, we scan for instructions that access the reference's value as shown in Listing 4.10. Whenever we identify an instruction accessing a reference, we check whether it is allowed to access that reference. It is allowed to access the reference if its scope is the same as, or a descendant of, the scope in which the reference was created.

```
1 <d9e>: Abbrev Number: 29 (DW_TAG_subprogram) // main function
2 <d9f>   DW_AT_low_pc       : 0x7a80 // start address
3 <da7>   DW_AT_high_pc      : 0x1cb // end address
4 <dad>   DW_AT_linkage_name: (indirect string, offset: 0x97f):
5                         _ZN14buffer_ref_mut4main17he0fe62c8d547ffddE
6 <db7>: Abbrev Number: 30 (DW_TAG_lexical_block) // first scope
7 <db8>   DW_AT_ranges       : 0x50
8 <dca>: Abbrev Number: 30 (DW_TAG_lexical_block) // second scope
9 <dcb>   DW_AT_ranges       : 0x80
```

Listing 4.9: DWARF debugging information

```
1 // Creation
2 lea rax, [rsp + 0x588] // tracked object
3 mov qword [rsp + 0x80], rax
4 // Access
5 mov rsi, qword [rsp + 0x80]
```

Listing 4.10: Reference creation and access

### 4.2.2 Mutable and Immutable Reference Validation

During the reference validation procedure we described above, we also keep track of the mutability of each reference. Initially, all references are considered immutable. A reference is marked as mutable if we find it being used to mutate the value it points to. When a reference is moved into a register, the address of the value's owner is stored in the register. If that address is then used as the target of an instruction, then the reference is considered mutable as shown in Listing 4.11. Once a reference is identified as mutable, we check if its scope overlaps with any other reference for the same object. If it does then the program is considered unsafe.

```
1 mov rax, qword [rsp + 0x60] // reference
2 mov qword [rax], rdi  // dereferencing to mutate value
```

Listing 4.11: Mutable reference detection

### 4.2.3 Functions Returning References

When a function returns a reference, it cannot point to a local variable of the function as this would lead to a UAF bug. The returned reference must be either for some global data or one of its arguments. If it if for global data, then it will be marked with the static lifetime. However, lifetime names are not maintained in the serialized MIR. This creates the following limitations. (1) We cannot determine whether a function returns a reference to a global variable since the static name is erased. (2) Arguments are all assumed to have the same lifetime name. As a result, we analyse all functions assuming they return one of their arguments and not global data, and consider all arguments with a lifetime as candidates to be returned by the function.

For each function, we check if its return type is a reference and retrieve its lifetime from the MIR. If it is, then we extract all arguments that are references and estimate the register or stack offset they will be passed through following the x86-64 calling convention. These registers and stack offsets are added to the active list. The active list holds the values that currently contain one of the arguments that are allowed to be returned by the function. After this, we start recursively following the program's execution flow and update our active list as follows. (1) A register/offset is activated when an active register/offset is copied to it. (2) An active register/offset is deactivated when a non active register/offset is copied to it. When the execution encounters a jump, we follow it with a copy of the current active list. If it is a conditional jump, we also follow the fallthrough edge. When a branch reaches a return, we check whether register %RAX is active. Because references are pointers, they are returned through %RAX and not through a given argument. If it is not, then the program is marked as unsafe.

## 4.3 Vulnerability Injection

To evaluate our bounds check validation process, we developed a buffer overflow vulnerability injection mechanism. We locate all bounds checks in the binary that compare against an immediate value using the same procedure examined in Step 4 of Section 4.1.1. For each identified check, we increment the immediate value of the comparison by one through binary rewriting as shown in Listing 4.12. As a result, we allow a one-byte out-of-bounds access to buffers.

```
1 // Last 4 bytes correspond to the immediate value
2 0x7ee8: 48 3D 80 00 00 00  cmp rax, 0x80
```

Listing 4.12: Buffer overflow injection

# Chapter 5

# Evaluation

## 5.1 Dataset

We composed a dataset consisting of 25 binaries and 10 crates. The binaries are further categorized in two classes based on how we discovered them. (1) 14 controlled binaries. (2) 11 real-world binaries The controlled binaries were found through GitHub by searching for local buffer initialization patterns. The real-world binaries were discovered from websites listing popular Rust projects. Furthermore, we modified the controlled binaries using both our tool's vulnerability injection procedure and manual rewriting, producing 14 vulnerable variants containing buffer overflows and out-of-scope accesses. The code of each project was retrieved from GitHub and was compiled using the rustc++ [20] compiler which is based on the 1.81 version of the Rust compiler, with no optimisations (debug mode) and debugging symbols.

## 5.2 Binary Results

### 5.2.1 Controlled Binary Results

Table 5.1 displays the evaluation results of the controlled binaries. In particular, it shows the number of fixed-size buffers and slices discovered, the expected number of bounds checks and how many were missing, the number of varaible-sized buffer arguments, the number of call sites not found in the binary, and the number of references found. Through our evaluation, we noticed that references are not commonly found in binaries. Even if some exist, they are very rarely used. This may be because, at the binary level, references lose their meaning. The notion of ownership is lost in the binary and, instead, values are accessed by their memory address. Another observation is the amount of missing checks in the binary compared to the MIR. The MIR is serialized before any optimisation passes take place. This means that for any access to a buffer, a bounds check is emitted for it

despite its complexity. Further down the compilation pipeline, checks may be removed if the compiler is able to ensure safety statically. To confirm this, we have implemented a mostly automatic, source code mutation procedure which requires minor manual intervention discussed in Subsection 5.2.1. Tables 5.2 and 5.3 show the validator's accuracy. We observe 21% classification accuracy on the non-mutated binaries and 71% on their mutated versions. The remaining of the misclassified binaries were mostly due to failing to discover call sites in the binary that contained tracked buffers as arguments which needed to be validated. The main reason for these was indirect calls for which we could not statically resolve their location due to PIC. The vulnerable versions of the binaries were all correctly classified as unsafe because of the larger immediate value found compared to the buffer's size and the out-of-scope reference usage. These results highlight the conservative implementation of our tool which leads to a higher rate of false positives.

#### Mutation Procedure

We operate on the source code of the project and alter all buffer accesses, without affecting their logic, preventing the compiler from verifying them statically. An example is shown in Listing. 5.1. We use Rust's black_box() standard library function. This function tells the compiler that its argument is modified in some unknown way, therefore stopping it from optimising the code based on knowledge of that value. Here, we use it to hide the buffer's index, which prevents the compiler from removing the bounds check. We tested this procedure on all projects with missing bounds checks in the binary and in most cases they were no longer missing.

```
1  // Original
2  for i in 0..count {
3      buf_2[i] = buf_1[i];
4  }
5
6  // Mutated
7  use std::hint::black_box;
8  ...
9  for i in 0..count {
10     let idx_0 = black_box(i);
11     let idx_1 = black_box(i);
12     buf_2[idx_0] = buf_1[idx_1];
13 }
```

Listing 5.1: Source code mutation example

### 5.2.2 Real-world Binary Results

Table 5.4 displays the same results for the real-world binaries. We noticed that even though buffers exist, there are no bounds checks for them. To understand why, we

| Binary | Buffers | Slices | Bounds Checks | Missing Checks | Buffer Arguments | Missing Call Sites | References |
|---|---|---|---|---|---|---|---|
| sha256sum | 2 | 0 | 9 | 8 | 0 | 0 | 0 |
| norx | 10 | 8 | 6 | 0 | 8 | 0 | 0 |
| icmp-rust | 3 | 6 | 0 | 8 | 0 | 0 | 0 |
| rustun | 4 | 2 | 0 | 1 | 0 | 0 | 1 |
| sflow-collector | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| Rust-AES | 3 | 0 | 6 | 6 | 0 | 0 | 0 |
| ricv32i | 2 | 0 | 1 | 0 | 0 | 0 | 0 |
| qft | 6 | 0 | 0 | 6 | 0 | 0 | 3 |
| tetris-cli | 21 | 0 | 0 | 1 | 0 | 1 | 0 |
| rust-aes | 26 | 0 | 17 | 32 | 1 | 0 | 0 |
| iz-cpm | 12 | 0 | 4 | 5 | 0 | 3 | 0 |
| mpl | 16 | 1 | 4 | 64 | 14 | 1 | 0 |
| retrocompressor | 3 | 12 | 0 | 2 | 5 | 0 | 0 |
| hextazy | 11 | 15 | 2 | 8 | 0 | 6 | 0 |

Table 5.1: Evaluation results of controlled binaries

| | Safe | Unsafe |
|---|---|---|
| Predicted Class | 3 | 11 |

Table 5.2: Classification of controlled binaries

searched all of the identified buffers in the source code and examined their usage. In most cases, buffers were only used as arguments to functions or were filled by copying the contents of another slice. No direct access using the square bracket notation was found. We also observe that boringtun contains the most buffers with a considerable difference. By looking at how they use them, we understand that fixed-size buffers are more commonly used for network related code, such as sending and receiving data. Code authors may prefer to use other data structures, such as vectors, as they offer more capabilities. Table 5.5 show the accuracy of the validator for these binaries. It successfully marks 82% of the binaries as safe. The misclassifications occur from a missing call site and a missing slice initialization in the binary.

## 5.3 Crate Results

Table 5.6 displays the evaluation results of the crates analysed. We observe again that most bounds checks are found in networking related crates. Many of the missing checks are recovered through our proposed mutation procedure. The remainder of the undiscovered checks originate from incorrect buffer identification in the binary or disassembly difficulties. In particular, during analysis of the brotli crate we encountered missing disassembled code due to the use of a jump table. Small buffers (e.g. size 1) containing zeroes may result in inaccurate detection in the binary, since a single move of a zero to the stack could correspond to the buffer's initialization. Furthermore, validation of temporal safety

| | Safe | Unsafe |
|---|---|---|
| Predicted Class | 10 | 4 |

Table 5.3: Classification of mutated controlled binaries

| Binary | Buffers | Slices | Bounds Checks | Missing Checks | Buffer Arguments | Missing Call Sites | References |
|---|---|---|---|---|---|---|---|
| env_logger | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| boringtun | 20 | 8 | 0 | 0 | 0 | 1 | 0 |
| xsv | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| eza | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| bend | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| bandwhich | 12 | 0 | 0 | 0 | 0 | 0 | 0 |
| tokei | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| just | 3 | 0 | 0 | 0 | 0 | 0 | 0 |
| awesome-rust | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| tickrs | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| broot | 2 | 1 | 0 | 0 | 1 | 0 | 0 |

Table 5.4: Evaluation results of real-world binaries

was omitted due to the lack of scope information from debugging symbols.

## 5.4 Performance Analysis and Limitations

### 5.4.1 Performance Analysis

During evaluation, we have measured the execution time of the validator for each project. Fig. 5.1 and Fig. 5.2 show the validator's performance on binaries and crates, respectively. We observe that the validation time scales superlinearly with considerable variance with respect to the project's size. Most of the validator's execution time originates from the disassembly process, which scales with the file size. Once disassembly is finished, the validation procedure's execution time varies depending on the number of user-defined functions checked and their size. This causes the variance shown in the graph, as smaller projects could contain more user-defined functions leading to higher overhead during the validation procedure. However, each function's validation is independent, allowing for parallelization, which could reduce variance and bring performance closer to linear scaling. During crate validation, the crate's object files are extracted and analysed independently. As a result, we observe worse scaling compared to analysis of binaries, but parallel analysis of each file may narrow the performance gap.

### 5.4.2 Limitations

Due to using an older version of the Rust compiler (1.81), evaluation of the real-world binaries and crates was limited, as most required more recent versions. The validator

| | Safe | Unsafe |
|---|---|---|
| Predicted Class | 9 | 2 |

Table 5.5: Classification of real-world binaries

| Crate | Buffers | Bounds Checks | Missing Checks | Appeared Checks | Missing Call Sites |
|---|---|---|---|---|---|
| jobserver | 2 | 0 | 3 | 2 | 0 |
| ipnet | 5 | 4 | 20 | 20 | 0 |
| brotli | 51 | 70 | 30 | 16 | 2 |
| brotli-decompressor | 7 | 66 | 12 | 3 | 0 |
| sha1-smol | 4 | 1 | 16 | 16 | 0 |
| cesu8 | 1 | 0 | 4 | 0 | 0 |
| pulldown-cmark | 4 | 2 | 6 | 6 | 0 |
| libsecp256k1 | 9 | 2 | 1 | 1 | 0 |
| byte-unit | 1 | 0 | 1 | 1 | 0 |

Table 5.6: Evaluation results of crates

currently supports tracking of buffers that contain scalar values, such as integers. Furthermore, we adopt a conservative approach, which has shown to produce false positives. Reference validation focuses only on references for buffers and their slices, which constrained our evaluation. The limited set of references tracked prevented us from performing thorough validation and understanding their usage in binaries. Finally, results for functions returning references have been omitted. In most cases, functions were returning references to either global data or a field from one of its structure arguments. As we discussed before, lifetime names are missing in the MIR serialization. As a result, the validator could not differentiate a function returning a reference to some global variable through the expected static lifetime name, but instead assumed no such possibility. In addition, references to structures presented another implication. Assume a function that takes a reference to a structure as an argument. It is allowed to return a reference to a field of the structure. However, the validator fails to detect cases where the return value is an offset within a reference argument.

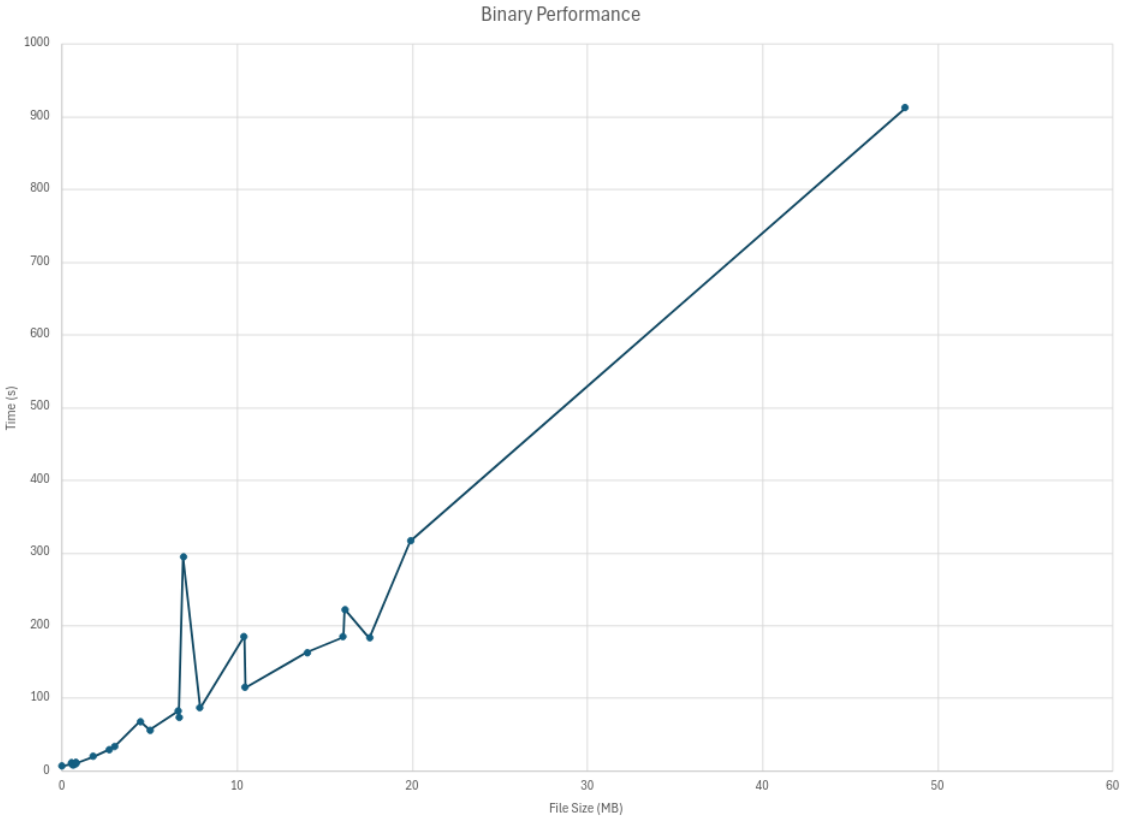


Figure 5.1: Performance results of binaries

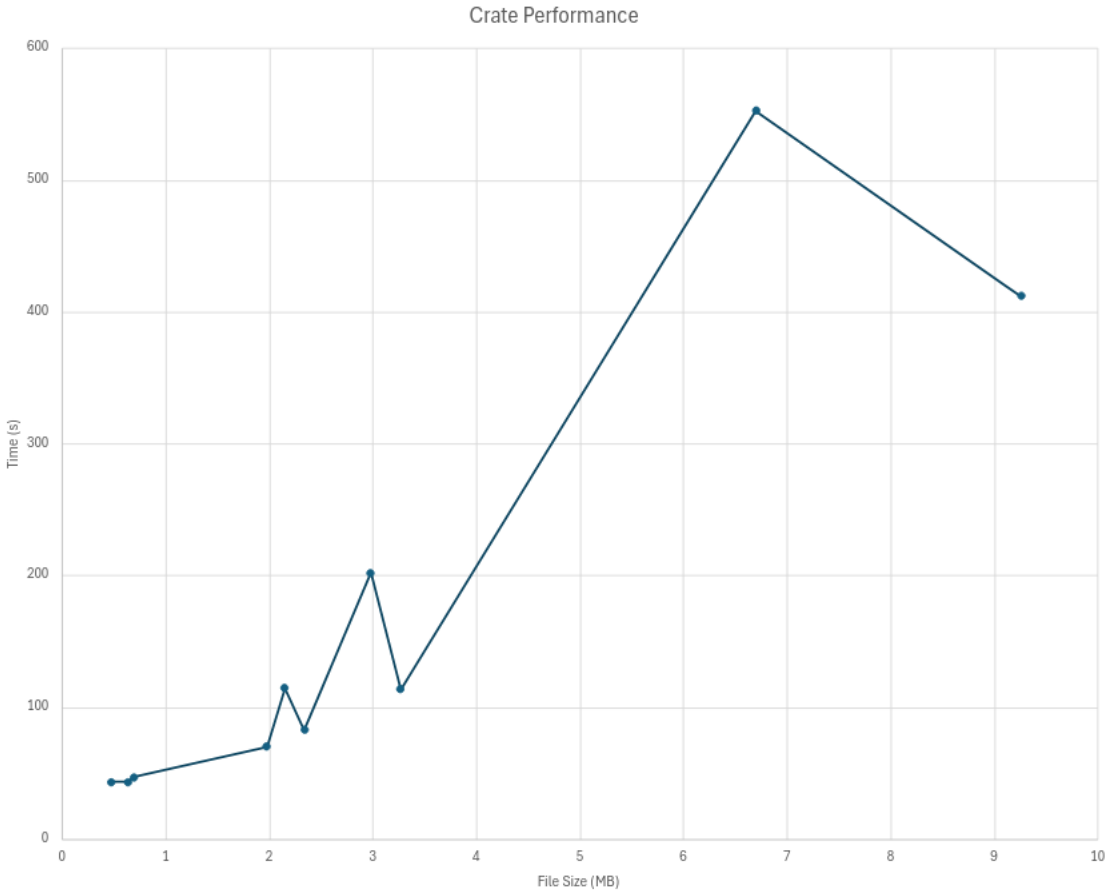


Figure 5.2: Performance results of crates

# Chapter 6

# Related Work

Static analysis approaches attempt to detect vulnerabilities in programs without executing them. Brumley et al. [15] developed a program verification platform that lifts binary code into an intermediate language, representing the side effects of instructions. This approach offers faster detection with full code coverage. However, these methods often yield many false positives and suffer with lifting indirect jumps or obfuscated code.

Another approach is dynamic analysis through binary instrumentation. Nethercote et al. [24] have developed Valgrind, a dynamic binary instrumentation framework for detecting memory bugs in binaries. It maintains a shadow memory table during execution which holds metadata about the program's memory. Specifically, it instruments all allocations, deallocations, and accesses in both the stack and heap to update its table and verify the validity of each memory access. Two important limitations of binary instrumentation are its significant runtime overhead and limited code coverage. The granularity at which memory is tracked (per byte, word, etc.) determines the additional memory required to maintain metadata about each memory region. Finer granularity requires more metadata but provides better accuracy, whereas coarser granularity reduces metadata size but may worsen accuracy. In addition, the code coverage of such techniques is limited as they can only analyse the executed paths. Wide code coverage depends on the provided test cases by the developers, an assumption that does not hold in our threat model.

Hybrid approaches are another evolving approach which combine different techniques to address the limitations of each. Stephens et al. [26] introduced Driller, a vulnerability detection tool which combines selective concolic execution with fuzzing. It eliminates the need for test cases and instead uses a fuzzer to generate inputs and guide execution to new paths. When the fuzzer gets stuck, a concolic execution engine utilises the generated inputs to help it reach deeper program states. This approach requires significant time and computing resources due to the size of the input space and the SMT solver. Hybrid techniques have gained increasing attention and numerous ideas are proposed to help address their current limitations, like QSYM [28] which provides a faster concolic execution

engine through dynamic binary translation.

Tools also exist that require compilation of source code in order to validate memory safety. Serebryany et al. [25] introduced the AddressSanitizer (ASan) which injects code before any memory access during compilation. The injected code maintains a shadow memory of the program. This approach often induces lower overhead compared to binary instrumentation approaches. Min et al. [22] provided an efficient, Rust specific version of ASan which is achieved by reducing unnecessary memory checks. In addition, the Rust development team maintain Miri [14], an MIR interpreter that emulates MIR execution and detects unsafe code.

# Chapter 7

# Conclusion

In this thesis, we addressed the problem of introducing memory violations in Rust binaries through binary modification. We presented a static analysis validation tool that detects these violations, using the binary's MIR to enhance analysis. In particular, we developed a validation methodology that covers spatial and temporal memory rules, utilising the MIR as the ground-truth representation of the program and comparing it against the binary's state. Through our evaluation, we successfully detected all buffer bounds violations introduced by our automated injection procedure, and out-of-scope references. The results also displayed a considerable number of false positives due to compiler optimisations and indirect control flows. This is a consequence of our conservative design, prioritizing soundness over completeness. However, we believe that these can be significantly reduced by obtaining the MIR serialization after optimisation passes have been performed. Furthermore, our results indicate that using the MIR is a viable approach for improving detection of memory violations, and enables validation of cases requiring additional information, such as type information (e.g., functions returning references).

# Bibliography


[1] Array and index expressions - The Rust Reference. https://doc.rust-lang.org/reference/expressions/array-expr.html.

[2] Array and index expressions - The Rust Reference. https://doc.rust-lang.org/reference/expressions/array-expr.html#array-and-slice-indexing-expressions.

[3] The borrow checker - Rust Compiler Development Guide. https://rustc-dev-guide.rust-lang.org/borrow_check.html.

[4] Bytecode. https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html#jvms-2.1.

[5] Garbage Collection Tuning. https://docs.oracle.com/en/java/javase/21/gctuning/introduction-garbage-collection-tuning.html.

[6] JVM Heap. https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html#jvms-2.5.3.

[7] Lifetimes - The Rust Programming Language. https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html.

[8] The MIR (Mid-level IR) - Rust Compiler Development Guide. https://rustc-dev-guide.rust-lang.org/mir/index.html.

[9] Overview of the compiler - Rust Compiler Development Guide. https://rustc-dev-guide.rust-lang.org/overview.html.

[10] References and Borrowing - The Rust Programming Language. https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html.

[11] The Slice Type - The Rust Programming Language. https://doc.rust-lang.org/book/ch04-03-slices.html.

[12] Validating References with Lifetimes - The Rust Programming Language. https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html#generic-lifetimes-in-functions.

[13] What is Ownership? - The Rust Programming Language. https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html#ownership-rules.

[14] Rust-lang/miri. https://github.com/rust-lang/miri, Apr. 2026.

[15] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A Binary Analysis Platform. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification*, pages 463–469, Berlin, Heidelberg, 2011. Springer.

[16] W. Bugden and A. Alahmar. Rust: The Programming Language for Safety and Performance, June 2022.

[17] Z. Cai, S. M. Blackburn, M. D. Bond, and M. Maas. Distilling the Real Cost of Production Garbage Collectors. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 46–57, May 2022.

[18] J. Harris. IoT Survey 2024. https://newsroom.eclipse.org/news/announcements/eclipse-foundation-unveils-2024-iot-embedded-developer-survey-results, Dec. 2024.

[19] A. Louka, A. Dionysiou, and E. Athanasopoulos. Validating Memory Safety in Rust Binaries. In *Proceedings of the 17th European Workshop on Systems Security*, EuroSec '24, pages 8–14, New York, NY, USA, Apr. 2024. Association for Computing Machinery.

[20] A. Louka, G. Portokalidis, and E. Athanasopoulos. Rustc++: Facilitating Advanced Analysis of Rust Code. In *Proceedings of the 18th European Workshop on Systems Security*, EuroSec'25, pages 63–69, New York, NY, USA, Apr. 2025. Association for Computing Machinery.

[21] S. Maring, R. Sapir, M. Wiesenberg, V. Iyer, B. Wright, T. Smith, M. Kalfane, and T. Das. Overview of Oracle JVM. https://docs.oracle.com/en/database/oracle/oracle-database/26/jjdev/Oracle-JVM-components.html.

[22] J. Min, D. Yu, S. Jeong, D. Song, and Y. Jeon. ERASan: Efficient Rust Address Sanitizer. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4053–4068, May 2024.

[23] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 106–119, New York, NY, USA, Jan. 1997. Association for Computing Machinery.

[24] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. 42(6):89–100.

[25] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, 2012.

[26] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings 2016 Network and Distributed System Security Symposium*, San Diego, CA, 2016. Internet Society.

[27] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee. Jekyll on iOS: When Benign Apps Become Evil. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 559–572, 2013.

[28] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 745–761, 2018.

[29] Y. Zhang, Y. Zhang, G. Portokalidis, and J. Xu. Towards Understanding the Runtime Performance of Rust. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, pages 1–6, New York, NY, USA, Jan. 2023. Association for Computing Machinery.