Validating Memory Safety in Rust Binaries

Antonis Louka
louka.antonis@ucy.ac.cy
University of Cyprus
Nicosia, Cyprus

Antreas Dionysiou
dionysiou.antreas@ucy.ac.cy
University of Cyprus
Nicosia, Cyprus

Elias Athanasopoulos
athanasopoulos.elias@ucy.ac.cy
University of Cyprus
Nicosia, Cyprus

ABSTRACT

Without depending on heavy runtime support, Rust can realize fast machine code that mitigates most of the common attacks associated with memory-corruption and can appear in all unsafe machine code developed using C/C++. Most of the work for producing machine code with security guarantees is carried out at compile-time by the Rust compiler. However, at runtime, there is no mechanism to ensure that the produced security guarantees, as computed at compile-time, are still in place.

In this paper, we explore the possibilities of an attacker fabricating Rust binaries so that they are on purpose vulnerable. We show that it is possible to modify automatically, and at large scale, binaries so that certain defences, placed by the Rust compiler, are removed. We introduce a generic problem, and focus on concepts of spatial and temporal safety. Finally, we produce a validator that assesses if all checks ensuring spatial and temporal safety remain intact within a Rust binary, before executing it. Our work is a step towards validating Rust binaries at load time so that security guarantees computed at compile-time are effective at runtime.

KEYWORDS

Rust, Memory Safety, Static Analysis, Buffer Overflow

ACM Reference Format:
Antonis Louka, Antreas Dionysiou, and Elias Athanasopoulos. 2024. Validating Memory Safety in Rust Binaries. In *The 17th European Workshop on Systems Security (EuroSec '24), April 22, 2024, Athens, Greece.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3642974.3652281

1 INTRODUCTION

Programming systems can be considered safe or unsafe based on how they manage memory. Traditionally, systems code is built using a loose memory model, such as the one provided by C/C++. This gives developers flexibility in handling memory as they like, however, it also opens the space for devastating security issues, such as memory corruption. Nevertheless, programmers hesitate in switching to memory-safe programming systems, since the overheads of memory management may be critical, especially when developing low-level code. As an answer to this, lately we have seen new programming systems that provide memory safety without the need of heavy runtime support. A prime example of such a system is Rust.

Rust provides a new programming paradigm where the compiler rejects programs that are potentially vulnerable. Additionally, the Rust compiler enhances the produced machine code to adhere to several checks that can ensure both spatial and temporal memory safety without a runtime system. This approach, which does not rely on a special runtime system, can produce fast machine code, comparable to the one produced by a C or C++ compiler. However, the absence of a runtime system allows for modifying the produced machine code so that certain checks are intentionally omitted.

Producing deliberately vulnerable programs is not new; in the past, it has been shown that an attacker can introduce vulnerabilities in applications that can be downloaded from an appstore with the purpose of exploiting them, once they are installed in a large user-base [23]. Although producing deliberately vulnerable programs in memory-unsafe programming systems, such as Objective-C, C, and C++, is almost straight-forward, performing a similar task with memory-safe programming systems, such as Rust, is unexplored. The implications of doing this with Rust can be significant. As an example, consider an appstore that advertises the distribution of *secure code*, by means of supporting only Rust, or other similar systems, as their development platform.

In this paper, we explore the possibilities of an attacker for deliberately altering the machine code produced by a Rust compiler so that memory safety is not preserved, anymore. Through our attacks, we argue that security that is vetted at compile time, such as the one imposed by the Rust compiler, should be also reviewed at runtime. In principle, we argue that Rust's machine code should be validated before executing, in order to ensure that the binary is indeed memory safe. Validating compiled code is not new; for example NaCl code [7], which is sandboxed by means of SFI, is also validated during loading, another example is JavaCard with verification of JVM bytecode [22]. In this paper, we do the first steps of producing a Rust validator that assesses if certain checks produced by the Rust compiler have not been altered in a Rust binary.

Our main contributions can be summarized as follows.

- We explore how machine code produced by a Rust compiler can be modified to be intentionally vulnerable;
- We implement a tool that can modify automatically buffer overflow checks in Rust binaries so they are not memory safe, anymore;
- We develop a validator that can assess, whether the checks introduced by the Rust compiler have been modified.

2 BACKGROUND

Memory Safety. A recent survey showed that 60-70% of vulnerabilities in iOS and macOS [11] and 70% and 90% of the ones found by Microsoft and Google [1, 19] are memory-safety related. These bugs

EuroSec '24, April 22, 2024, Athens, Greece

ACM ISBN 979-8-4007-0542-7/24/04…$15.00
https://doi.org/10.1145/3642974.3652281

lead to unexpected behavior, crashes, and arbitrary code execution, since there is no runtime to enforce a type of runtime exception. An overread bug can expose sensitive information (e.g., cookies, passwords) or security-related data for bypassing defences, such as stack canaries [5] and Address Space Layout Randomization (ASLR) [18]. An overwrite bug can be exploited for writing past the end of a buffer and corrupt control data (e.g., a return address or VTable pointer) for hijacking the control flow of the program.

Rust. Rust is a systems programming language adhering to design goals of both performance and memory safety. It ensures memory safety at compile time by enforcing certain rules about how code is written and structured, without using runtime support or any Rust specific review process.

Rust's Spatial Safety. The Rust compiler allocates statically sized buffers on the stack and prevents buffer overflows by emitting snippets of bounds checking code when using them. Similar code can be emitted for integer overflows if opted by the programmer.

Rust's Temporal Safety. Rust, enforces temporal safety using the "Borrow Checker" [15]; that operates on the Mid-level Intermediate Representation (MIR) [16] from which the LLVM IR is produced. The Borrow Checker enforces the Ownership, Borrowing and Lifetime concepts. With the Ownership rules [20], rustc dictates memory management with automatic memory allocation and deallocation. Each variable is called the *owner* of a value and only one owner can exist at a time. When the owner goes out of scope, the value is deallocated, when the value is assigned to another variable the ownership is transferred to the new variable. With the Borrowing concept [21], Rust instead of transferring ownership, permits another variable to borrow the value. This allows arbitrary number of immutable references (read-only references), but only one mutable reference (read-write) at a time. Lifetimes [8] are compiler or programmer assigned stubs, to enforce the ownership model and prevent dangling pointers.

Threat Model. We consider an adversary that deliberately tampers with the code of a compiled Rust binary to introduce intentionally a memory-corruption vulnerability. Normally, this is not possible, since the Rust compiler generates memory-safe code. We follow a similar scenario demonstrated by Wang et al. [23], where a developer creates deliberately a buggy application and submits it to an appstore. Since the malicious user is the actual app-developer, the digital signature cannot be used for detecting whether the application has been tampered with, and the binary can be published to the appstore as a code-signed binary. Recompilation cannot be a remedy to this problem since companies refuse to give access to their source code due to industrial secrets, and the app-store is limited to reviewing the binary only. This further highlights the validator that we contribute in this work, verifying whether the security checks introduced by the Rust compiler remain intact.

These applications are commonly known as *Jekyll apps*; applications that are remotely exploitable with malicious control flows that are not present during the review process, but manifest from the signed code when the application reaches the end-user device. The attacker does not introduce these changes a priori as the screening process can capture them. We assume that the changes are stealthily embedded in the binary to avoid detection; binaries with entirely removed checks do not follow Rust rules and are rejected by the app-store screening process. Although Jekyll apps have been extensively studied in the context of memory-unsafe programming systems, such as Objective-C, where memory corruption vulnerabilities can be trivially introduced to the code, in this paper, we explore the feasibility of this approach in Rust, which, in principle, is memory safe. Note that attacks exploiting bugs other than those prevented by the use of Rust security checks are not the focus of this paper. These bugs can anyway be exploited even without tampering with the Rust checks. Instead, in this paper, we demonstrate the possibility of tampering with a compiled with Rust, and thus seemingly safe, binary to introduce bugs.

3 METHODOLOGY

Violating Rust's Spatial Safety Techniques. Rust's buffer overflow checks compare the current index to the length of the array. If the index is greater than the length, the program terminates, preventing memory corruption. The format of this check varies based on the compilation's optimization level (debug and release modes). We create examples using statically allocated arrays, locate the safety checks the Rust compiler generates for protecting code and modify the safety check using binary re-writing.

Buffer overflow attack. We use two buffers, copying the contents of the larger one to the smaller; by manipulating the buffers we force *rustc* to emit compiler checks. An example of compiler check is shown in Listing 1. This check evaluates the indexing variable of the buffer against its length; if the index exceeds the buffer length, the program crashes and raises an exception of type `panic_bounds_check`. The next step is to modify this check by allowing the index to be greater than the buffer length, thereby, introducing an overwrite bug, copying the larger buffer into the smaller one and overwriting the stack.

We also create a shellcode that overwrites the return address of the currently executing function with the address of a "malicious" function planted in the binary, redirecting the execution flow of the program to the malicious function and executing arbitrary code.

```
1 cmp    $0xf,%rax -modified-> cmp $0x100,%rax
2 setb   %al
3 test   $0x1,%al
4 jne    0x55555555d766
5 jmp    0x55555555d7b5
```

Listing 1: Modified buffer overflow check.

Depending on the program, the shellcode may need to contain other artifacts, e.g., the address of the buffer, its length, since it uses the stack to manipulate spatial data. In these cases, an overread bug is needed to leak contents of the stack; even though Rust does not use stack smashing protection outside nightly builds [9]. A key requirement for the attack is that the modified compare instruction (`cmp`) uses the *same* number of bytes as the original one. An adversary cannot use any arbitrary value to patch the check, as the x86 instruction set uses variable length commands, and the number of bytes used can vary.

Detecting Buffer Overflow Checks. For detecting buffer overflow checks in Rust binaries, we rely on heuristics that reflect observations from a large pool of compiled binaries covering various types of checks including optimized ones.

Heuristic 1. Buffer bounds check has a branch to a panic exception function called `panic_bounds_check`. This function is executed when there is an out-of-bounds access on arrays or slices, including

the `vector` and `string` types of Rust. We locate that call to ensure all identified checks specifically relate to bounds checking.

Heuristic 2. Before invoking the exception function, an argument preparation occurs, typically consisting of 2–3 assembly instructions. We gather the addresses of the instructions preceding the call and create a list ("destination address list"). We explore the disassembly of each function locating branches and check whether the branch target address is in the destination address list. Heuristics 1 & 2 work for all types of checks since the check format is not involved. They focus on locating the exception function calls (`panic_bounds_check` calls), and create a list with addresses that are target addresses of branches that jump to the `panic_bounds_check` function call.

Heuristic 3. Each unoptimized check typically comprises a maximum of five instructions. To locate the instructions responsible for bounds checking, we detect branch instructions targeting addresses in the destination address list, and backtrack until a compare instruction is found or maximum number of instructions (5), is reached.

We realize our check detection methodology, incorporating the aforementioned heuristics, through an automated locating tool, namely *locator*, that searches for the compiler safety checks in the disassembly of a given binary. The output of the locator is a file containing all checks found in the binary and a JSON log containing the checks found per function (used by the validator –see Sec. 4).

The check location procedure requires the detection of the `panic_bounds_check` function address, which cannot be done through symbols since we assume stripped binaries. To accurately find the address of the `panic_bounds_check`, we track all the calls (direct and indirect) in the binary. We use multiple instances of disassembled code of the `panic_bounds_check` function from different binaries to extract general regular expressions for its instructions. When this call-tracking process is finished, the locator creates a list of the addresses called. For every address found, we disassemble 21 instructions (length of `panic_bounds_check`) and match them to the regular expressions.

In summary, the locator's procedure is as follows: (1) Locate all functions in the binary, (2) Locate the address of the `panic_bounds_check` function, by inspecting all functions — Heuristic #1, (3) Inspect all functions and create the "destination address list" (addresses in argument preparation) — Heuristic #2, (4) Locate all branches in the binary and filter out branches that do not target addresses in the "destination address list", (5) For each branch with target address in "destination address list", backtrack until a `cmp` instruction is reached (max. 5 instructions) — Heuristic #3.

The locator is capable of classifying detected checks into different categories based on their characteristics. Binaries compiled in debug mode have checks that consist of 5 instructions, compared to 2 instructions in release mode. The locator categorizes checks into *optimized* and *regular* based on the number of instructions involved. Each category includes subcategories like "`cmp reg, reg`", "`cmp reg, value`" and, "`cmp reg, symbol`" along with a *general* subcategory for other types of checks not matching these criteria. The tool classifies the checks into subcategories based on the type of instructions used, e.g., "`cmp rax, rcx`" or "`cmp rax, 0x100`".

Evaluating the Locator's Performance. We evaluate the locator by comparing its output against a ground truth provided from a symbol-based tool, which locates the `panic_bounds_check` function using function symbols and using the same heuristics as the locator. We evaluate the performance of the locator on Github and artificial binaries. Artificial binaries are small programs we create where we manipulate buffers to force the compiler to add buffer overflow checks. We also gather applications from GitHub written in Rust and serve a utility i.e. a Rust implementation of *du* tool, a JSON log file viewer, a command line csv toolkit etc; we clone their source code and compile them using default optimization.

In total, we evaluate the locator using 36 binaries — 12 artificial and 6 binaries from Github, and their equivalent stripped versions. In Table 1, we compare the checks found by our locator on the unstripped versions (column 3) with the ground-truth (column 2) and the results of the locator on the stripped versions (column 4). For each binary, we report the number of detected checks along with the respective accuracy. The locator manages to find all (100% of) the checks in unstripped binaries and a large percentage of the checks in stripped ones. The average deviation of the locator's performance on unstripped and stripped binaries is 28.1%.

The process of disassembling, splitting code to Basic Blocks (BBL) and identifying functions is harder on larger stripped binaries. Since the locator operates on a function-wide level the function identification process can affect the accuracy of the locator tool.

Table 1: Performance of the locator tool in detecting checks in unstripped binaries and their stripped versions.

Binary Name	Ground Truth	Located Checks	Located Checks (stripped version)
#0	104	104 (100%)	93 (89.42%)
#1	103	103 (100%)	92 (89.32%)
#2	95	95 (100%)	79 (83.15%)
#3	103	103 (100%)	92 (89.32%)
#4	93	93 (100%)	83 (89.25%)
#5	106	106 (100%)	92 (86.79%)
#6	90	90 (100%)	85 (94.44%)
#7	103	103 (100%)	92 (89.32%)
#8	96	96 (100%)	86 (89.58%)
#9	89	89 (100%)	57 (64.04%)
#10	86	86(100%)	54 (62.79%)
#11	103	103 (100%)	71 (68.93%)
dust [2]	397	397 (100%)	278 (70.02%)
fblog [3]	416	416 (100%)	166 (39.9%)
runiq [24]	280	280 (100%)	90 (32.14%)
xsv [4]	486	486 (100%)	316 (65.02%)
sk [13]	424	424 (100%)	238 (56.13%)
ssurl [17]	526	526 (100%)	306 (58.17%)

Rust's Temporal Safety Techniques. *Multiple mutable references on the same object.* Temporal safety is provided by the borrow checker which ensures that there is only one mutable reference to the same value in memory. When our code is invalidated it means that the Rust compiler will not produce an executable. In this Proof Of Concept (POC), we use a buffer (b1) with a mutable reference (b2), and another reference (b3) to a second buffer (s), and use b2, b3 to manipulate the two buffers. In this example we perform binary rewriting, to transfer the address of b1 to the b3 reference after compilation. This means that we create three references that can be used to modify the memory pointed by b1, an operation prohibited by Rust. We showcase that with binary re-writing we can create arbitrary references to the same buffer.

The assembly code of this POC before and after the modification is provided in Listing 2. An important aspect of buffers in Rust is that they are always initialized before use. In our case, we replaced

"`lea rax, [s]`" instruction, with "`lea rax, [b1]`". The new instruction loads the effective address of buffer b1 to `rax`. Thus, when the program executes the instruction "`mov qword [b3], rax`", it moves the address of b1 buffer to b3 instead of `s`. Since the new instruction is not the same length as the previous one, we realign the binary using `nop` operations while in a real-world scenario this would be obfuscated.

```
# pass reference of b1 to b2
0x00009027  lea rax, [b1]
0x0000902c  mov qword [b2], rax
# initialization of the buffer of b3 reference
0x00009034  lea rdi, [s]
0x0000903c  xor esi, esi
0x0000903e  mov edx, 0x28
0x00009043  call sym.imp.memset
# pass reference of s to b3
0x00009048  lea rax, [s]
0x00009050  mov qword [b3], rax

# Lines 10 and 11 after modification:
# pass reference of b1 to b3
0x00009048  lea rax, [b1]
0x0000904d  nop
0x0000904e  nop
0x0000904f  nop
0x00009050  mov qword [b3], rax
```

Listing 2: Assembly code of POC before and after changes.

Use-After-Free. Borrowing and Ownership rules along with lifetimes are concepts that Rust uses to avoid dangling pointers and use after free bugs (automatic memory allocation and deallocation). We introduce such bugs by creating a reference that points to a freed memory address, and utilize the code in Listing 3 as a POC.

```
pub struct User {
    logged: bool,
    name: String,
    password: String,
}
fn create_users() -> Box<User> {
    let mut u1 = Box::<User>::new(User::new("Antonis", "1234")
        );
    let mut u2 = Box::<User>::new(User::new("Cassandra", "
        myfavpassword"));
    u1.log_in("1234".to_string());
    return u2;
}
fn main() {
    let mut u2 = create_users();
    u2.log_in("myfavpassword".to_string());
}
```

Listing 3: POC example for Use-After-Free bug.

`create_users()` function creates two different users $u1$ and $u2$. Each user has the ability to log in; in this example, $u1$ logs in and $u2$ is returned to the caller function. Based on Borrowing and Ownership rules we know that once the scope of a variable ends that variable is dropped if no other reference exists for that variable. Thus, $u1$ is dropped when the function ends and $u2$ changes ownership from `create_users()` function to `main()` function.

With similar binary rewriting, we make $u2$ point to the same address space as $u1$. Then variable $u2$ is returned to `main()` function as a dangling pointer. When this pointer is used instead of accessing the address space allocated for $u2$ it accesses the address for $u1$, resulting to a use-after-free bug. To properly use binary re-writing in this scenario we use padding code to accommodate the code of the attack. We add one `jmp` instruction with target address the code in the padding section, and one `jmp` instruction at the end of the injected code, to return back to the original program execution. The added code performs the reference change, restores the initial values of the registers used to the values before the jump. Such modifications are restricted due to binary misalignment. Since changes can break the binary all the corrupted instructions must be restored before returning the control to the initial code. With similar binary rewriting we can bypass other concepts like lifetimes, thus introducing dangling pointers.

4 VALIDATOR

Validator Procedure. We formalize a validation procedure (*validator*) for the checks identified by the locator (Sec. 3 focusing on checks similar to Listing. 1. The validator takes a compiled binary and a JSON log file generated by the locator as input, which maps function names to a list of checks found within each function. The output of the validator is a JSON file containing function names and their corresponding validation class: *benign, undefined* or *malicious* for functions with tampered checks or borrowing rules violations.

The validator analyzes the binary, identifies existing buffers, and calculates their lengths. It then matches each check with the buffer it safeguards. Once the corresponding buffer is determined, the validator reconstructs its version of the check using the buffer's length and compares it with the found check. The reconstructed check, using the actual length from buffer initialization, is considered the ground truth. If the found check contains a value smaller or equal to the reconstructed buffer length, it is considered valid. The validation methodology comprises five (5) distinct phases: (1) reading input log and analyzing binary, (2) locating buffers, (3) tracking buffers, (4) validating buffer checks, and (5) validating references.

Reading input log and analyzing binary. The validator reads the input log file, analyzes the binary, and locates functions in the binary along with `memcpy` and `memset` functions.

Locating buffers. To precisely locate buffers, we rely on the initialization patterns utilized by the Rust compiler. Safe Rust code necessitates buffer initialization upon declaration. We've identified three primary initialization patterns: i) usage of `memset` function (address extracted in stage (1)), ii) using loops, and iii) using of AVX type registers with multiple `mov` instructions. Utilizing these patterns, we construct regular expressions to identify the initialization code and extract the buffer's address and length in bytes.

The use of `memset` and loops are easier to understand; with the AVX initialization the compiler initializes a vector register and uses multiple move instructions to transfer the initialization value to memory. There can be an arbitrary number of moves, however the addresses start from a higher offset and go to lower, with the lowest being the buffer base address. Each move initializes a certain amount of bits based on the AVX register used. These bits may not be part of the buffer length in source code but serve as alignment bits, and affect the ground truth length of the validator (larger than the actual buffer). However, they do not affect the actual validation procedure, as this memory belongs to the buffer and a valid check should be smaller or equal to the ground truth. An example for each initialization pattern is provided in Listing 4. We use buffer access patterns to convert buffer length from bytes to the source code buffer length, used by the checks.

```
# (1) Memset Initilization
lea rdi, [s]            # buffer address
mov esi, 0xa            # initialization value
```

```
4  mov edx, 0xdac          # buffer length
5  call sym.imp.memset     # memset call
6
7  # (2) AVX Initilization
8  xorps xmm0, xmm0 # initialization value (zero)
9  movaps xmmword [rsp + 120], xmm0
10 movaps xmmword [rsp + 110], xmm0
11 ...
12 movaps xmmword [rsp + 40], xmm0
13 movaps xmmword [arr_a], xmm0 # buffer address
14
15 # (3) Loop Initilization
16 lea rax, [rsp + 20]     # buffer start address
17 lea rcx, [rsp + 20]     # buffer end address
18 add rcx, 0x50          # buffer length in bytes
19 mov qword [rsp + 10], rcx   # destination
20 mov qword [rsp + 18], rax   # start
21 ...
22 cmp rax, rcx              # comparison
23 je 0x90fa                 # loo end
24 mov rax, qword [rsp + 8]
25 mov qword [rax], 0xa      # initialize buffer
26 ...
27 jmp 0x90cf                #loop
```

Listing 4: The three buffer initilization patterns in Rust.

Tracking Buffers. Locating buffers through initialization patterns does not locate buffers passed as arguments to other functions. In Rust, argument passing happens in two ways, by reference and by value. Rust uses pass-by-value by default and pass-by-reference on reference passing (using &). In Rust we distinguish references as mutable (default) and immutable (using `mut` keyword); this does not affect the assembly code but is only checked at compile time by the borrow checker. In pass-by-value, the compiler employs `memcpy` to copy the buffer's contents to a different memory location relative to the stack pointer (`rsp`). The new base address is then passed as a parameter to the callee function, opposed to pass-by-reference, where the original address is given.

The validator captures buffers given as variables to other functions by tracking call instructions. It maintains a representation of the buffers initialized in each function. For each buffer the validator locates all the addresses that point to the checked buffer. Then for each call found in the function, both direct and indirect, it identifies if the buffer is passed as argument. The validator covers both pass-by-value and pass-by-reference argument passing. It backtracks from the function call and verifies if an address that corresponds to the checked buffer is loaded to a register used for argument passing in x86. If the checked buffer is an argument, the validator first visits the callee function before proceeding to the next call in the caller function, repeating the same procedure. This recursive process continues for every callee function that the checked buffer is a parameter to. The validator represents the buffers as: (a) the function the buffer was initialized in, (b) a list with function addresses that access the buffer, (c) a list with the registers, that pass the buffer as argument, (d) a list per function with the buffer references, (e) the length of the buffer, and (f) the initial address of the buffer in each function. After the buffer tracking process, the validator generates a representation for each function, including buffer addresses, their respective lengths, and a list of addresses referencing each buffer.

Validating Checks. In a function-wide manner the validator evaluates each check separately. Each check has two branches, as seen in Listing 1. Specifically, the `jmp` instruction leads to the call of the `panic_bounds_check` function, and the `jne` instruction leads to the buffer accessing instructions[1]. We develop regular expressions to match access-type instructions, and extract the buffer's address guarded by the check and the stepping offset (size of each element in the buffer in bytes). With this information we transform the buffer length from bytes to the length used in the check. For instance, for a buffer containing ten 64-bit integers, we find a ground truth length of 80 bytes, and the check uses a length value of 10. We reconstruct the ground truth check by dividing the ground truth length (80) with the stepping offset (8 bytes), to get the actual length (10).

Validating References. We provide a validation method that works on the disassembly of a binary for attacks on the borrowing and ownership concepts; it works on a function level and it invalidates a function with either more than one mutable references (read-write), or one mutable and one immutable (read-only) references to the same buffer. Our method validates references independent of scope (blocks of code where outside the block the reference does not exist) and catches the attack in Listing 2. We implement this method for buffer reference validation, but can be expanded to other objects; provided we locate their initial assigned address.

As a first step, the validator identifies the starting address of each distinct buffer within the function, then proceeds to verify the related buffer references. The validator maintains metadata, of active registers, and references. For active registers the validator maintains a key-value pair mapping, where key is the register and value is the address loaded to that register at each point of time. For the references the validator maintains three different sets, the previously used reference set, the read-only and read-write references. The read-only set contains reference addresses used only for reading from the specific buffer, while the read-write set keeps addresses used for reading and writing the buffer. The previously used set contains all the addresses correlated to the buffer until that point in the procedure (union of read-only and read-write sets).

The validator updates these structures during execution, simulating the dataflow from the initial address to other registers and addresses. We create regular expressions to locate when an address is used as read-write address and match the reference transfer from an address to a register and from an active register to another address. The reference validation process demands the concept of context, which can be thought as the active reference and number of used references and registers at a point of time. When a branch instruction is present the context can vary between the two branches, and thus the validator follows each branch separately, with the help of two structures: (1) visited addresses, and (2) to-be-visited addresses. The tool adds the taken and fallthrough edge addresses to the to-be-visited list and evaluates the addresses with a Last-In-First-Out (LIFO) manner to keep context relevant.

Steps taken to validate references. In summary, the steps taken to validate references are: (1) When an active address or the initial address is moved to a register, add the register-address pair to the active register mapping, (2) If an active register is used as destination operand, remove it from active register mapping, (3) If an active register is used as a source operand and another register is used as destination operand, add the new register to active register mapping, (4) When an active register moves value to address retire

[1]In this paper, we focus on this type of checks. Note, however, that the validator can be easily extended for considering other checks as well; we leave this as future work.

active address and mark new address as active; add the old address to previously used set and the new address to read-only set, (5) When a regular expression for a write memory access matches an instruction using an active register, that address is used as a read-write reference (add reference to read-write set and remove it from read-only set), and (6) If the active address in validation is in the read-write set and the validator finds a memory access instruction where neither operand matches the active address and are in the read-only or read-write set, the function is deemed malicious.

Evaluation. *Dataset Composition.* To evaluate the validator, we compose a dataset consisting of 66 binaries, 42 valid (following Rust's rules) and 24 malicious (invalid/modified). From the 42 valid binaries, 6 are real-world applications taken from GitHub used also in Sec 3, 18 are toy examples created by us, and the rest are stripped binaries (6 from GitHub examples and 12 stripped versions of the toy examples). For the 24 malicious binaries we have: (1) binaries with modified checks — altered offset value used in the check, to a larger (invalid) value, allowing for memory corruption; we create 17 binaries by altering 6 GitHub applications and 11 toy examples, and (2) binaries with modified references — we create 7 binaries altering 4 GitHub applications and 3 toy examples, using binary rewriting to create a second invalid reference for a certain buffer.

All binaries are compiled using version 1.70 of the Rust compiler (rustc), with the default optimization level. In future rustc versions, slight changes in code generation are expected, necessitating *minor* adaptations to the tools presented in this paper, as our methodology, in principle, remains the same. Minor adaptations to our tools may also be necessary when facing large code bases and optimized binaries, which we consider future work.

Results. For our experiments we define as true positives the number of times the validator classified a malicious binary as malicious, false positives the number of times the validator classified a benign binary as malicious, true negatives the number of times it classified a benign binary as benign, and false negatives the number of times it classified a malicious binary as benign.

The summary of the results of the validator prototype are shown in Table 2. For a binary to be benign it needs to successfully pass both buffer and reference validation procedures. The validator classified 41 binaries as benign and 25 as malicious, successfully classifying 65 out of the total 66 binaries. Specifically, the validator correctly classified 41 out of the 42 benign binaries (misclassifying one), and 24 out of the 24 malicious binaries, resulting in a 98% accuracy rate. These results indicate that the validator successfully detects invalid code in the given binaries, subsequently marking them as malicious. Notice that while the validator primarily targets checks akin to those in Listing 1, it can be extended to validate other types of checks mentioned in this paper as well.

Limitations. Notice that the validator prototype is not able to cope with scenarios where functions use a buffer, which is given as argument, and that function is never called. Rust compiler performs dead-code elimination in such functions, but with *code decorators* this can be prevented and keep the code in the binary; such scenario is apparent as a misclassification in Table 2. Furthermore, the validator cannot verify checks that correspond to buffers not found in stage (2) of the validation process, since the respective ground truth cannot be created. We currently do not consider these checks in the validator evaluation, and mark them as *undefined* in the produced log file. Note that the accuracy of the locator tool (Sec.3) directly impacts validator's performance, as checks cannot be evaluated for correctness if they are never located.

Table 2: Confusion matrix showing the performance of the validator in classifying benign and malicious binaries.

Binaries (66)		**Actual Class**	
		Benign (42)	Malicious (24)
Predicted Class	Benign	41	0
	Malicious	1	24

5 RELATED WORK

Wang et al. [23] introduce a novel method that avoids code signing and app review mechanisms, and hides the malicious behavior of the application in code that is rearranged remotely after installation to the user device. Han et al. [10] introduce an attack vector with applications that have passed Apple's vetting process and works on non-jailbroken iOS devices. Dharsee et al. [6] introduce a novel type of hardware trojans, namely, *Jinn* that corrupt general-purpose hardware. They demonstrate the effectiveness of *Jinn* trojans in bypassing compiler-based security using the Rust language. *Jinn* trojans can manipulate compiler checks on the fly during execution. Papaevripides et al. [14] discuss attacks on mixed binaries where "safe code" can assist in bypassing code hardening in C/C++ code, such as Control-Flow Integrity and SafeStack. Zhuohua et al. [12] propose an automated bug detection framework for Rust programs, namely, *MIRChecker*. This framework operates in Rust's MIR and using numerical and symbolic information detects potential runtime crashes and errors.

6 FUTURE WORK

The locator prototype identifies buffer-overflow checks, but further (taint) analysis is needed for locating input-dependent guarding buffers that can be reached by an adversary. In addition, to further enhance the accuracy of the validator, we aim at accounting for buffers in objects, e.g., vectors, strings. The validator manages to find buffers contained in objects, using the initialization patterns. However, the tool is not (yet) able to correlate the buffers found in objects with their accesses and, therefore, the checks that guard them. Moreover, another optimization is to track buffers returned from functions. The tool tracks buffers that are passed to functions as arguments; it is possible that checks also correspond to buffers returned from functions. Implementing these optimizations can mitigate the *undefined* results mentioned in Sec. 4.

7 CONCLUSIONS

In this paper, we showed that Rust's safety measures against buffer overflows can be neutralized by modifying specific checks in the binary after compilation, along with safety measures against dangling pointers. In particular, we introduced a heuristic-based methodology that: (a) locates safety checks in compiled binaries, and (b) automates the binary modification process. Finally, we proposed a preliminary validation methodology that determines whether a binary has been tampered with.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback. The authors are grateful for the financial support provided by the project "Collaborative, Multi-modal and Agile Professional Cybersecurity Training Program for a Skilled Workforce In the European Digital Single Market and Industries" (CyberSecPro) project. This project has received funding from the European Union's Digital Europe Programme (DEP) programme under grant agreement No. 101083594. The views expressed in this paper represent only the views of the authors and not of the European Commission or the partners in the above-mentioned project. Moreover, the work of this paper was also supported by the European Union's Horizon Europe research and innovation programmes under grant agreement No. 101070599 (SecOPERA).

REFERENCES

[1] 2019. Queue the hardening enhancements. https://security.googleblog.com/2019/05/queue-hardening-enhancements.html
[2] Bootandy. [n. d.]. Bootandy/Dust: A more intuitive version of DU in rust. https://github.com/bootandy/dust.
[3] Brocode. [n. d.]. Brocode/fblog: Small command-line json log viewer. https://github.com/brocode/fblog.
[4] BurntSushli. [n. d.]. Burntsushi/XSV: A fast CSV command line toolkit written in rust. https://github.com/BurntSushi/xsv.
[5] Thurston HY Dang, Petros Maniatis, and David Wagner. 2015. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. 555–566.
[6] Komail Dharsee and John Criswell. 2023. Jinn: Hijacking Safe Programs with Trojans. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 6965–6982. https://www.usenix.org/conference/usenixsecurity23/presentation/dharsee
[7] Bennet Yee et al. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *IEEE Symposium on Security and Privacy*.
[8] The Rust Foundation. [n. d.]. The rust programming language. https://doc.rust-lang.org/rust-by-example/scope/lifetime.html
[9] The Rust Foundation. 2022. *The Rust Programming Language: Exploit Mitigations*. Accessed on November 20, 2023.
[10] Jin Han, Su Mon Kywe, Qiang Yan, Feng Bao, Robert Deng, Debin Gao, Yingjiu Li, and Jianying Zhou. 2013. Launching Generic Attacks on iOS with Approved Third-Party Applications. In *Applied Cryptography and Network Security*, Michael Jacobson, Michael Locasto, Payman Mohassel, and Reihaneh Safavi-Naini (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 272–289.
[11] Paul Kehrer. [n. d.]. https://langui.sh/2019/07/23/apple-memory-safety/
[12] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C.S. Lui. 2021. MirChecker: Detecting Bugs in Rust Programs via Static Analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 2183–2196. https://doi.org/10.1145/3460120.3484541
[13] Lotabout and Contributors. 2023. Skim: Fuzzy Finder in Rust. https://github.com/lotabout/skim. Accessed on November 20, 2023.
[14] Michalis Papaevripides and Elias Athanasopoulos. 2021. Exploiting Mixed Binaries. *ACM Trans. Priv. Secur.* 24, 2, Article 7 (jan 2021), 29 pages. https://doi.org/10.1145/3418898
[15] rust-borrow checker. [n. d.]. Rust Compiler Development Guide. https://rustc-dev-guide.rust-lang.org/borrow_check.html.
[16] Rust-Lang. [n. d.]. Rust MIR. https://rustc-dev-guide.rust-lang.org/mir/index.html
[17] Shadowsocks Contributors. 2023. Shadowsocks-Rust: A Rust port of Shadowsocks. https://github.com/shadowsocks/shadowsocks-rust. Accessed on November 20, 2023.
[18] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 574–588.
[19] MSRC Team. 2019. https://msrc-blog.microsoft.com/2019/07/18/we-need-a-safer-systems-programming-language/
[20] The Rust Programming Language contributors. 2022. The Rust Programming Language: Understanding Ownership. https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html. Accessed on November 20, 2023.
[21] The Rust Project Developers. 2022. *The Rust Programming Language: References and Borrowing*. Accessed on November 20, 2023.
[22] USENIX Association 2002. *2nd Workshop on Industrial Experiences with Systems Software (WEISS '02)*. USENIX Association, USENIX Association. https://www.usenix.org/legacy/events/wiess02/tech/full_papers/deville/deville_html/
[23] Tielei Wang, Kangjie Lu, Long Lu, Simon Chung, and Wenke Lee. 2013. Jekyll on iOS: When Benign Apps Become Evil. In *22nd USENIX Security Symposium (USENIX Security 13)*. USENIX Association, Washington, D.C., 559–572. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/wang_tielei
[24] Whitfin. [n. d.]. Whitfin/runiq: An efficient way to filter duplicate lines from input. https://github.com/whitfin/runiq.