Finding Rust Allocations Referenced in Unsafe Code

Andrew Chin
School of Cybersecurity and Privacy
Georgia Institute of Technology
Atlanta, United States
achin34@gatech.edu

Jiarui Xu
School of Computer Science
Georgia Institute of Technology
Atlanta, United States
jxu605@gatech.edu

***Abstract*—Rust is a programming language that provides strong memory safety benefits while remaining efficient and low-level. *Unsafe* Rust does not provide memory safety guarantees, so bugs in unsafe code could violate memory safety for the entire program.**

Our tool collects allocations in Rust code that are referenced in unsafe code in order to identify the values that are at risk of such memory bugs. We collect information about unsafe code regions unique to Rust and bundle it with the code's LLVM IR to collect a points-to set for references used in unsafe code. Our tool then finds the location of the pointer analysis results in Rust source code.

Code on https://github.com/azchin/raw-pointer-analysis.

I. Introduction

Rust is a systems programming language that aims to be a safer alternative to C and C++ by enforcing restrictions on pointer usage to prevent memory bugs and data races [1]. However, compromises to memory safety must be made to allow for complete functionality through the use of unsafe code. Developers can perform operations that bypass compiler checks for memory safety in unsafe code and must exercise caution to manually reason about the unsafe code's safety. When developers incorrectly reason about unsafe code, not only unsafe regions, but also safe regions are at risk of memory bugs.

Rust introduces the concept of ”ownership”, in which a value may only be bound (owned) to one variable at any given time. The value can be ”borrowed” by either using read-only references, or at most one mutable reference [2]. References cannot exist if the original value goes out of scope. Ownership and borrowing is checked in the compiler, through the ”borrow-checker”. Ownership allows Rust to deallocate a value when the its current owner goes out of scope. In doing so, Rust does not require a garbage collector. Memory bugs are easily detectable. For example, a use-after-free bug is detect at compile time because a dangling reference usage would be caught as an error because it is no longer in scope. After the value's owner goes out of scope, references are also automatically deallocated.

While ownership and borrowing is appropriate for most code, there are some cases where developers need to bypass the borrow-checker due to the conservative nature of static analysis. Unsafe Rust addresses these cases by allowing developers to dereference raw pointers, call unsafe functions, and other operations [3]. Unsafe code must be contained in an unsafe block or an unsafe function. Traditional memory bugs can occur from unsafe code and can even affect safe Rust code. This is undesirable because the memory safety guarantees of safe Rust code become void and the presence of unsafe code reduces the confidence that these guarantees are upheld.

LLVM is a compiler framework that defines a unified low-level code representation [4]. The framework supports LLVM backends for different architectures (e.g. ARM, x86). The Rust compiler acts as an LLVM frontend which transforms Rust source code into LLVM Intermediate Representation (IR).

The Static Value-Flow (SVF) tool is a static analysis framework for LLVM-based languages. [5]. It includes many analysis tools, such as Andersen's pointer analysis [6]. Because SVF operates on LLVM IR, it can be used to perform more powerful static analysis on Rust.

Our tool builds on top of SVF's Andersen pointer analysis to gather Rust values that are used in unsafe code.

II. Motivation

```
 1 /// Creates a vector by repeating a slice `n` times.
 2 pub fn repeat(&self, n: usize) -> Vec<T> {
 3     let mut buf = Vec::with_capacity(self.len() * n);
 4     buf.extend(self);
 5     while condition { // until we repeated buf n times
 6         unsafe {
 7             ptr::copy_nonoverlapping(
 8                 buf.as_ptr(),
 9                 (buf.as_mut_ptr() as *mut T).add(buf.len()),
10                 buf.len(),
11             );
12         }
13     }
14 }
```

Fig. 1. CVE-2018-1000810

Consider the simplified code in Fig. 1 from Rust standard library's `VecDeque` [7]. An integer overflow in line 3 can lead to an underallocation of the `buf` vector's capacity. The `while` loop on line 5 and subsequent unsafe `memcpy` on line 7 assumes that the `buf` has been properly allocated with size `len * n`. However, this assumption is never verified before performing the unsafe operation. This leads to a buffer overflow when the loop keeps copying memory past the true capacity of `buf`.

This is an example of unsafe code affecting values in safe code. The buffer overflow overwrites other values in the stack. These unauthorized writes break Rust's borrow semantics because the checker does not account for such operations.

The development of this tool is inspired by XRust [8], which partitions allocations into two memory regions: those allocated in unsafe code regions and everything else. XRust uses a similar SVF analysis to determine all pointers aliasing a pointer used in unsafe code and instrument runtime checks to see if the memory being accessed is from the correct partition.

TRust is a tool that extends XRust by using Intel Memory Protection Keys (MPK) for increased efficiency [9]. It also uses SVF, but adds a context-sensitive value-flow analysis on top of pointer analysis to collect allocation sites. Our tool borrows some design choices from TRust such as propagating unsafe location metadata, but extends the output by tracing analysis results back to Rust source code.

III. Design and Implementation

Our tool finds references and pointers used in unsafe code and traces the references to their allocation site. For a given Rust project, the tool outputs a mapping of original owners to their references and pointers in unsafe code.

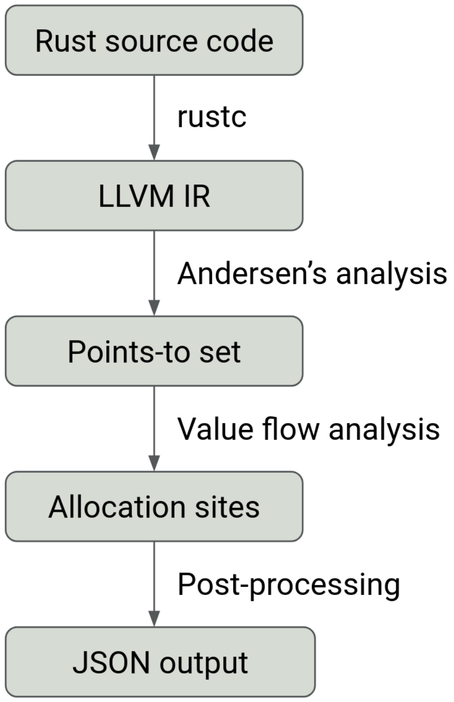


Fig. 2. High-level architecture

We track unsafe code regions by querying AST's of all the project files. Then, we collect a points-to set after pointer analysis. Finally, we trace the results of the points-to set back to the Rust source code and report locations in the project files. The pipeline is illustrated in Fig 2.

A. Unsafe code locations

Unsafe code blocks are a unique concept to Rust that is not shared by LLVM. The issue is that we want to perform Andersen's pointer analysis (further discussed in III-B) using SVF, which only operates on LLVM IR. It is thus necessary to propagate the location of unsafe code down to LLVM IR. We can do so by parsing the Rust source code, finding the line numbers for the unsafe code blocks, and using debuginfo embedded in LLVM instructions to find instructions in unsafe regions. We use the tree-sitter parser framework with its Rust grammars to query for the unsafe blocks [10].

B. Pointer analysis

After parsing the LLVM IR of the Rust project and identifying pointers in unsafe code, we perform Andersen's pointer analysis using SVF. Andersen's algorithm is a context-insensitive and flow-insensitive interprocedural analysis [6]. The points-to set provides us knowledge about the potential owners of the pointer. At this stage, we have a mapping of unsafe pointers to possible allocation sites. To aid further formatting, we invert this mapping to have allocation sites as keys and unsafe pointers as values.

C. Value-flow analysis

To improve the precision of our results, we use sparse value-flow analysis via SVF. For each potential allocation site, a Sparse Value Flow Graph (SVFG) is constructed and we traverse the graph to filter out any unsafe pointers which aren't reachable.

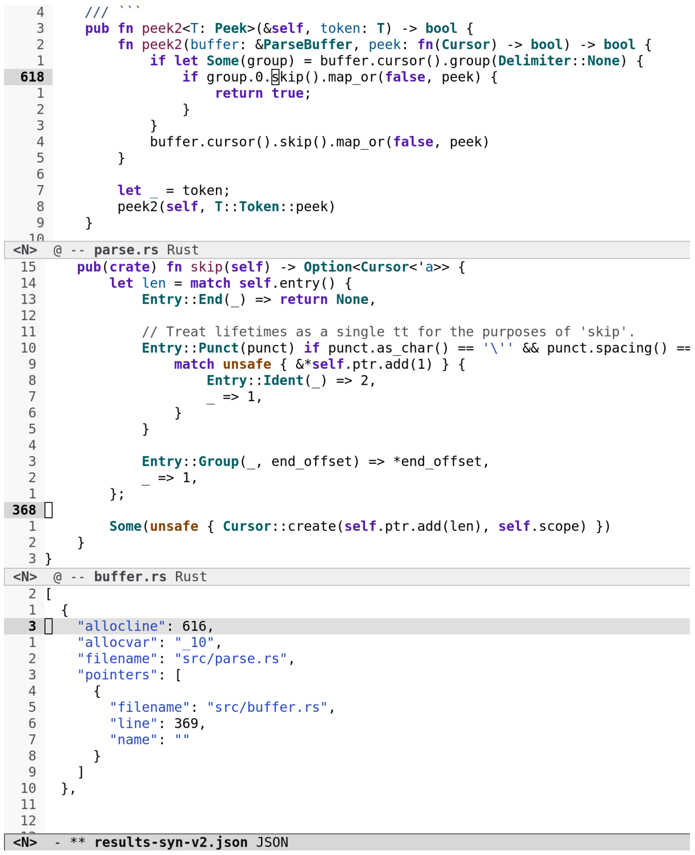
D. Finding original allocation sites

Obtaining a set of allocation sites in Rust code from a points-to set requires a pairing of LLVM IR allocations and Rust code allocations. We find the LLVM allocations by searching for the `alloca` instruction for the points-to values. We once again use debuginfo embedded in the LLVM instructions to figure out the allocation's location in Rust source code. Finally, we output the results in JSON so that our output is both reasonably human-readable and machine-readable.

IV. Discussion

We evaluate our tool by manually inspecting its output when run on some popular Rust crates. We are constrained to crates that can be compiled with an older Rust toolchain version. We use Rust 1.64.0 in order to compile with an LLVM version that supports non-opaque pointers, which is necessary for SVF's pointer analysis. SVF, LLVM 14, and all of our Rust crates are built on a Ubuntu 20.04 LTS machine. The crate "syn" in particular was used because it contains a non-trivial amount of unsafe code.

The manual analysis shows that our tool provides promising results. For example in Fig 3, the top-most pane shows the usage of `skip()`, the second pane shows the function definition of `skip()` as well as the unsafe code, and the



```
    /// ```
    pub fn peek2<T: Peek>(&self, token: T) -> bool {
        fn peek2(buffer: &ParseBuffer, peek: fn(Cursor) -> bool) -> bool {
            if let Some(group) = buffer.cursor().group(Delimiter::None) {
                if group.0.skip().map_or(false, peek) {
                    return true;
                }
            }
            buffer.cursor().skip().map_or(false, peek)
        }

        let _ = token;
        peek2(self, T::Token::peek)
    }
<N>  @ -- parse.rs Rust
    pub(crate) fn skip(self) -> Option<Cursor<'a>> {
        let len = match self.entry() {
            Entry::End(_) => return None,

            // Treat lifetimes as a single tt for the purposes of 'skip'.
            Entry::Punct(punct) if punct.as_char() == '\'' && punct.spacing() ==
                match unsafe { &*self.ptr.add(1) } {
                    Entry::Ident(_) => 2,
                    _ => 1,
                }
            }

            Entry::Group(_, end_offset) => *end_offset,
            _ => 1,
        };

        Some(unsafe { Cursor::create(self.ptr.add(len), self.scope) })
    }
}
<N>  @ -- buffer.rs Rust
[
  {
    "allocline": 616,
    "allocvar": "_10",
    "filename": "src/parse.rs",
    "pointers": [
      {
        "filename": "src/buffer.rs",
        "line": 369,
        "name": ""
      }
    ]
  },


<N>  - ** results-syn-v2.json JSON
```

Fig. 3. Example in the ”syn” crate

bottom pane shows our analysis results which indicates the location of an allocation and its unsafe pointer usage.

There is a loss of source code information in the LLVM IR, and thus also in our results. As seen in Fig 3, the variable names are not the same as the ones in the source code (i.e. ”_10” as opposed to ”group.0”). This is caused by Rust's Middle Intermediate Representation (MIR) pass, which transforms the names of all local variables. There are also intermediate LLVM registers that don't have names in the original source code (not even in MIR), and thus are reported as nameless. The line numbers could be inaccurate due to the inlining of some Rust block expressions.

We compared the usage of Andersen's pointer analysis + wave propagation with versioned flow sensitive pointer analysis. For the crates we evaluated, we did not find any difference in results. This can likely be attributed to the lack of control flow complexity when tracing allocations to raw pointers in library crates. A hypothesis for a future experiment is that flow sensitivity produces more precise results when tracing allocations to raw pointers in dependency crates. However, for the purposes of analyzing library code such as ”syn” (47.5k lines of code), using flow-insensitive analysis seems to suffice with the benefit of improved performance (runs in 7.7 seconds compared to flow sensitive analysis running in 20.0 seconds).

An approach to resolve the legibility of results from optimizations is to modify the Rust compiler to embed custom LLVM metadata which indicates the line of Rust code where each optimized value originates from.

An extension to our tool would be to track raw pointers in dependency crates and determine the allocation sites in the target project that reach unsafe code. This extension is useful for applications such as run-time protection on unsafe pointers. The easiest integration with the cargo build system would be to dump the unsafe region metadata from the rustc compiler.

A. Future work

By collecting a mapping of owner allocations to pointers used in unsafe code in a systematized fashion, we enable extensions such as runtime integrity checks like in XRust [8]. Knowing about allocation sites allows for the application hardware feature hardening such as ARM Pointer Authentication Codes or ARM Memory Tagging Extensions onto pointers that we know will be used in unsafe code. If we want to perform taint analysis to track unsafe functions from an untrusted crate or a Foreign Function Interface call, we can leverage this tool to also taint the parameters' or return's owner and other references. Our tool can also be used for IDE plugin messages which can indicate the line number of a variable allocation that is eventually passed to unsafe code.

References

[1] *Rust Programming Language*, en-US. [Online]. Available: https : / / www . rust - lang . org/ (visited on 10/19/2023).

[2] *What is Ownership? - The Rust Programming Language*. [Online]. Available: https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html (visited on 10/19/2023).

[3] *Unsafe Rust - The Rust Programming Language*. [Online]. Available: https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html (visited on 10/19/2023).

[4] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” en, in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, San Jose, CA, USA: IEEE, 2004, pp. 75–86, ISBN: 978-0-7695-2102-2. DOI: 10.1109/CGO.2004.1281665. [Online]. Available: http://ieeexplore.ieee.org/document/1281665/ (visited on 10/19/2023).

[5] Y. Sui and J. Xue, “SVF: Interprocedural Static Value-Flow Analysis in LLVM,” en,

[6] L. O. Andersen, “Program Analysis and Specialization for the C Programming Language,” en,

[7] *NVD - CVE-2018-1000810*. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2018-1000810 (visited on 10/19/2023).

[8] P. Liu, G. Zhao, and J. Huang, “Securing unsafe rust programs with XRust,” en, in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, Seoul South Korea: ACM, Jun. 2020, pp. 234–245, ISBN: 978-1-4503-7121-6. DOI: 10.1145/3377811.3380325. [Online]. Available: https://dl.acm.org/doi/10.1145/3377811.3380325 (visited on 08/31/2023).

[9] I. Bang, M. Kayondo, H. Moon, and Y. Paek, “TRUST: A Compilation Framework for In-process Isolation to Protect Safe Rust against Untrusted Code,” en,

[10] *Tree-sitterIntroduction*. [Online]. Available: https://tree-sitter.github.io/tree-sitter/ (visited on 11/10/2023).