Detecting Cross-Language Memory Management Issues in Rust

Zhuohua Li[1], Jincheng Wang[1], Mingshen Sun[2], and John C.S. Lui[1]

[1] The Chinese University of Hong Kong
{zhli, jcwang, cslui}@cse.cuhk.edu.hk
[2] Individual Researcher
bob@mssun.me

Abstract. Rust is a promising system-level programming language that can prevent memory corruption bugs using its strong type system and *ownership*-based memory management scheme. In practice, programmers usually write Rust code in conjunction with other languages such as C/C++ through Foreign Function Interface (FFI). For example, many notable projects are developed using Rust and other programming languages, such as Firefox, Google Fuchsia OS, and the Linux kernel. Although it is widely believed that gradually re-implementing security-critical components in Rust is a way of enhancing software security, however, using FFI is inherently unsafe. In this paper, we show that memory management across the FFI boundaries is error-prone. Any incorrect use of FFI may corrupt Rust's ownership system, leading to memory safety issues. To tackle this problem, we design and build FFIChecker, an automated static analysis and bug detection tool dedicated to memory management issues across the Rust/C FFI. We evaluate our tool by checking 987 Rust packages crawled from the official package registry and reveal 34 bugs in 12 packages. Our experiments show that FFIChecker is a useful tool to detect real-world cross-language memory management issues with a reasonable amount of computational resources.

Keywords: Static Analysis · Rust · Bug Detection.

1 Introduction

Rust is an emerging programming language that is famous for its strong security guarantees and high performance. Many companies and open source communities have been re-writing their software in Rust in an incremental manner, i.e., while most of the source code remains intact, some security-critical components are re-written in Rust. For example, Firefox contains a considerable amount of Rust code [4], and the Linux kernel is in the process of integrating Rust as its second language for kernel development [30,21]. New Rust projects also usually integrate with third-party C/C++ libraries to avoid reinventing the wheels. Rust can be used in conjunction with other languages because it supports Foreign Function Interface (FFI), which enables Rust to call external interfaces and exchange arbitrary data.

The incremental development of Rust code is widely believed to improve the security of software. However, calling external code is inherently unsafe in Rust because the Rust compiler cannot perform security checks across the FFI boundaries. Programmers may accidentally misuse the unsafe abilities that lead to vulnerabilities. In addition, different assumptions made by different languages make it possible for attackers to maneuver between the FFI boundaries and exploit these vulnerabilities [24]. Recent empirical studies [12,41] have shown that the incorrect use of FFI is one of the most significant causes of real-world memory-safety bugs. Even for Rust packages written in pure safe Rust (i.e., without using FFI), they may still be affected because they may depend on other packages that include FFI. According to our statistics (§2.2), among around 77,000 packages on the official Rust package registry[3], more than 72% of the packages depend on at least one package that contains unsafe FFI calls. Therefore excluding FFI is unrealistic in the current Rust ecosystem; instead, people have made lots of efforts to secure the use of FFI. For example, the Rust community has drafted several guidelines for writing unsafe code, including FFI [39,34,37,36]. Some Rust packages such as `rust-bindgen` and `safer_ffi` can automatically generate FFI, preventing developers from misusing it. However, they can only help developers to write correct interfaces with appropriate data types. Memory corruption caused by heap memory allocation/deallocation across the FFI boundaries remains an open problem. Moreover, Rust has a unique ownership system for memory management (§2.1), which creates its own paradigm of memory safety issues [41,31,22]. Hence existing works on misusing FFI for other memory-safe programming languages [20,19] such as Java and Python are no longer applicable.

In this paper, we study the security impacts of heap memory management issues across the FFI boundaries, especially those caused by the combination of Rust's ownership-based memory management and C/C++'s manual memory management. To tackle this problem, we propose to use static analysis techniques to detect potential memory management bugs across the FFI boundaries. Our method is based on the theory of Abstract Interpretation [7,8,9]. We design an augmented taint analysis algorithm to keep track of the states of heap memory, which captures the paradigms created by the ownership-based memory management. We implement our tool called FFIChecker, which automatically collects all the generated LLVM intermediate representation (IR) for both Rust and C/C++ code, then performs static analysis and outputs diagnostic reports. Security analysts can then inspect the reports and determine whether there are any real bugs. Our evaluation shows that FFIChecker can successfully detect real-world memory safety issues within acceptable time and with reasonable precision. To our knowledge, our work is the first effort that addresses the memory management issues across FFI boundaries in Rust programs.

We summarize our contributions as follows.

- We show the potential security and memory management issues when programmers intermix Rust and C/C++ via FFI.

[3] https://crates.io

- We propose an augmented abstract domain that captures the memory states in the ownership-based memory management scheme.
- We design and build FFIChecker, an automated static analyzer that can detect potential memory management bugs across the FFI boundaries in Rust packages and report informative diagnostic messages. The source code is available online[4], which can be the basis of other research in the future.
- We perform extensive evaluations in the Rust ecosystem. We evaluate 987 packages crawled from the official package registry and detect 34 bugs among 12 packages. All the detected bugs have been manually confirmed and reported to the authors and 15 of them have been fixed at the time of writing.

2 Background

In this section, we provide the background knowledge needed to understand the rest of the paper. We first introduce the Rust programming language and its security guarantees. Then we illustrate the prevalence of FFI and how Rust's memory management scheme interacts with it.

2.1 The Rust Programming Language

Rust is famous for its ability to build high-performance and secure programs. As a strongly-typed and compiled language, its rigorous type system and the unique *ownership* system enforce strict disciplines to eliminate memory safety issues. The ownership system is an automated memory management strategy derived from *linear logic* [13] and *linear types* [40]. Under the ownership system, each value has a unique *owner* (called owner variable), which keeps track of the lifetime of the value. Once the owner variable goes out of its scope, the ownership system automatically releases the memory allocated for the value. Note that the scope of each variable is determined at compile time so that the Rust compiler can insert appropriate memory reclamation routines to the generated binary. Thus neither reference counting nor garbage collection is needed. This enables Rust to build fast programs since no runtime overhead is introduced.

To pass a value to other parts of code, one can either *copy/clone*, *move*, or *borrow* the owner variable. Copying/cloning is usually used for data types that have semantics where copying their bytes is a valid way of creating a real copy, e.g., basic data types like integers. For more complicated data types, especially those that maintain internal heap memory (e.g., vectors), Rust's assignments *move* the ownership by default. After the ownership is *moved*, due to the uniqueness of the owner, the previous owner is immediately invalidated. A value can also be *borrowed* by taking a reference of it, through which the value can be temporarily accessed without changing the ownership. The references can be either *mutable* or *immutable*. The Rust type system regulates that there are no "mutable aliases", meaning that a read-only value can be immutably referenced multiple times; when the value is writable, only one mutable reference is allowed at a time.

[4] https://github.com/lizhuohua/rust-ffi-checker

The Rust compiler enforces the above rules to make security guarantees as follows. On the one hand, since the ownership system keeps track of the lifetime of each value, it ensures that the lifetime of a reference cannot exceed the value it points to. Therefore memory safety issues caused by dangling pointers such as use-after-free can be effectively prevented. On the other hand, since the ownership system eliminates mutable aliases, many security issues caused by concurrent reading/writing, such as race conditions and iterator invalidation, are avoided.

2.2 Foreign Function Interface (FFI) and Memory Management

As a system-level programming language, Rust can easily collaborate with other languages through the Foreign Function Interface (FFI). In this paper, we consider the case where the external code is written in C/C++ since this is the most common usage of FFI. Integrating Rust code with C/C++ code is prevalent and necessary because (1) Many C/C++ projects integrate Rust into existing codebases (e.g., the Linux kernel and Firefox) to enhance their security. (2) It can avoid duplicated work and benefit from the rich ecosystem of libraries written in C/C++. (3) C/C++ can be used for performance-critical scenarios.

However, since the Rust compiler cannot reason about the security of external code, calling FFI is inherently unsafe. Programmers need to explicitly use the `unsafe` keyword to bypass the security check enforced by the compiler. Therefore, using FFI is extremely error-prone. Existing studies [12,41,24] have shown that the incorrect use of FFI has become a severe source of memory safety bugs.

We would like to point out that even if programmers restrict themselves in pure safe Rust, their programs may still implicitly rely on FFI through dependencies. In fact, we find that more than 72% of packages on the official Rust package registry (crates.io) depend on at least one unsafe FFI-bindings package, as shown in Figure 1. The data is crawled by reading the metadata of reverse dependencies[5] on crates.io. Among all the 76,894 packages, we start from all the packages that are of category "external-ffi-bindings" (900 packages). These packages contain direct Rust FFI bindings to libraries written in other languages, often denoted by a "`-sys`" suffix. Then we collect all the reverse dependencies of them and repeat this process to get multi-level dependencies. As a result, the number of packages converges at the 10th level, with a total of 55,762 packages ($55,762/76,894 \approx 72.52\%$). Note that the "external-ffi-bindings" category by no means includes all the FFI binding libraries since many packages' categories are not tagged properly, hence the actual percentage can only be higher.

Since the manual memory management in C/C++ is naively unsafe, in this paper, we only consider the case where the heap memory is allocated in Rust and passed to C/C++. There are two ways of passing a heap-allocated object across FFI: (1) by *borrowing* the object as a reference, (2) by *moving* the ownership to the FFI. For *borrowing* as a reference, the ownership remains on the Rust side, so the ownership system is responsible for releasing the memory after it goes out of its scope. For *moving* the ownership, one can first "forget" it from the ownership

[5] As of February 14, 2022.

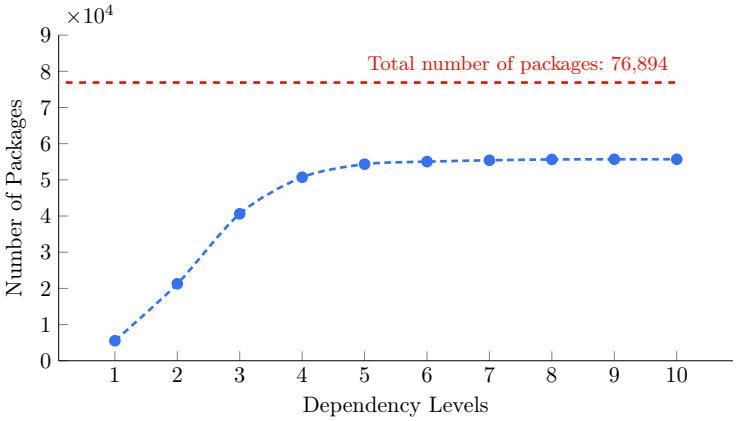

Fig. 1. Number of packages that depend on unsafe FFI.

system, then pass it to the FFI via a raw pointer. The Rust standard library provides several functions to "forget" an object, e.g., `std::mem::forget` and `Box::into_raw`. In this case, the responsibility of memory management returns back to the programmers, who have to take extra care because the ownership system no longer takes charge.

3 Security & Memory Management Issues via FFI

To explain why the memory management across the FFI boundaries may lead to security vulnerabilities and how the Rust ownership system gets involved, we give several bug examples detected by `FFIChecker`. We also categorize the vulnerabilities caused based on our observations: (1) common memory corruption, (2) exception safety, and (3) undefined behavior caused by mixing memory management mechanisms.

3.1 Memory Corruption

When heap memory is passed across the FFI boundaries, the ownership system cannot guarantee its safety. Therefore the responsibility of memory management returns back to the programmers, meaning that all kinds of common memory corruption bugs that happen in C, like use-after-free, double free, and memory leak, still exist. Listing 1 shows a memory leak found in package `emd`[6]. In Rust, `Box` is a smart pointer type used to securely manage heap memory. The developer uses `Box::into_raw` to expose the raw pointer of the heap memory managed by the `Box` in order to pass it to the FFI. However, after using `Box::into_raw`, the ownership system will "forget" the memory and hence will not automatically reclaim it. Instead, the developer is responsible for releasing the memory previously managed by the `Box`. Otherwise, there will be a memory leak.

[6] https://crates.io/crates/emd

```
let mut cost = Vec::with_capacity(X.rows());
for x in X.outer_iter() {
  let mut cost_i = Vec::with_capacity(Y.rows());  // Allocate a vector
  for y in Y.outer_iter() {
    cost_i.push(distance(&x, &y) as c_double);
  }
  // Forget the memory using `Box::into_raw`
  cost.push(Box::into_raw(cost_i.into_boxed_slice()) as *const c_double);
}

// Call FFI function
let d = unsafe { emd(X.rows(), weight_x.as_ptr(), Y.rows(), weight_y.as_ptr(), cost.as_ptr(),
↪  null()) };
```

Listing 1: `Box::into_raw` leaks memory but it is not released by the developer.

3.2 Exception Safety

Unlike many other programming languages, Rust does not support the `try-catch` statement for catching "exceptions". Instead, Rust provides a more reliable error handling mechanism: All *recoverable* errors must be handled or propagated back to the caller function, and all *unrecoverable* errors are handled by terminating the execution and unwinding the stack. All the stack objects' destructors will be called during the stack unwinding to prevent resource leakage. However, when passing heap memory across the FFI boundaries and cooperating with external code, developers usually have to transiently create unsound states via unsafe code (e.g., creating temporarily uninitialized data). Then after the external code finishes, developers manually clean up the states. If some error happens in between, the execution stops and the stack is unwound, so the clean-up procedure will not be executed. The remaining unsound state may cause security issues.

Listing 2 gives an example found in package `libtaos`[7]. At line 2, variable `params` is initialized by allocating heap memory. The memory is passed to FFI in the following `unsafe` block in lines 3 – 8. Note that the question mark operator (`?`) at lines 5 and 7 means that if the operation fails, the function returns early and propagates the error to the caller function. Therefore, the memory may be leaked if the function returns early and hence the `free` function at line 10 will not be called.

3.3 Mixing Memory Management Mechanisms

It is common that some C libraries provide functions for constructing/destructing data structures (usually implemented through `malloc` and `free`). To reuse these libraries, Rust developers usually implement Rust wrappers to handle these C APIs. One possible error is mixing different memory allocation/deallocation procedures provided by different languages. For example, it is illegal to allocate memory on the Rust side using `Box` and release it on the C side using `free`. Mixing different memory management mechanisms is undefined behavior, because

[7] https://crates.io/crates/libtaos

```
pub fn bind(&mut self, params: impl IntoParams) -> Result<(), TaosError> {
  let params = params.into_params();
  unsafe {
    let res = taos_stmt_bind_param(self.stmt, params.as_ptr() as _);
    self.err_or(res)?;
    let res = taos_stmt_add_batch(self.stmt);
    self.err_or(res)?;
  }
  for mut param in params {
    unsafe { param.free() };
  }
  Ok(())
}
```

Listing 2: When errors happen, `bind` returns before calling `free`.

(1) Rust and C may use different memory allocators. (E.g., on Linux, Rust can be configured to use *jemalloc*, while C uses *ptmalloc* by default.) (2) Rust and C have totally different memory management mechanisms and they operate on different levels. Specifically, Rust calls the constructors/destructors for constructed objects while C only deals with raw memory.

Listing 3 shows an example of mixing the memory management mechanisms of Rust and C, found in package `jyt`[8]. At line 5, a string is constructed through `CString::new`, which internally allocates memory on the heap using Rust's own memory allocator. Then at line 7, the string is explicitly leaked by `mem::forget`, and a raw pointer that points to the string is returned (line 8). Finally, at line 16, the heap memory is freed by the standard C function `free`. Note that the heap memory is obtained through Rust's allocator but freed on the C side through function `free`. This may lead to allocator corruption since the Rust code is compiled as a library and may be used in multiple projects with different memory allocators. Even if this may "work" in practice, it is undefined behavior and hence it is not guaranteed to work on other machines or on newer compilers.

3.4 Our Methodology

Based on the above motivating examples, we propose to use static analysis to detect these bugs because static analysis can examine every control flow path in a program and catch all potential bugs. It is especially appropriate for catching defects in exceptional situations because they are hard to be triggered with normal execution paths. At a high level, our approach does the following: We first compile both the Rust and C/C++ code into LLVM IR. Then we perform static analysis on the LLVM IR and keep track of the states of all the heap memory allocations, i.e., while the heap memory is propagated among the control flow graph, we determine whether it is *borrowed* or *moved*. Finally, if any heap memory is passed across the FFI boundaries, we continue to analyze whether it is freed in the external code. Depending on its state, we can find out whether the memory is incorrectly managed and generate diagnostic messages accordingly.

[8] https://crates.io/crates/jyt

```
// Rust code:
pub unsafe extern "C" fn to_json(from: ext::Ext, text: *const c_char) -> *const c_char {
  ... ...
  // CString internally allocates heap memory
  let output = CString::new(ext::json::serialize(&value.unwrap()).unwrap()).unwrap();
  let ptr = output.as_ptr();
  mem::forget(output);  // Memory is "forgotten" by the ownership system
  ptr  // The raw pointer will be passed across the FFI boundary
}

// C code:
int main() {
  ... ...
  const char* output = to_json(Yaml, input);
  ... ...
  free((char*)output);  // Memory allocated in Rust is freed by free()
  return 0;
}
```

Listing 3: Memory allocated on the Rust side but is freed on the C side.

4 System Design

In this section, we show the high-level architecture of FFIChecker and elaborate on the functionality of each component. The workflow of FFIChecker is depicted in Figure 2. The whole system consists of three parts: (1) the user interface and the driver program, (2) the entry point and foreign function collector, and (3) the static analyzer and bug detector.

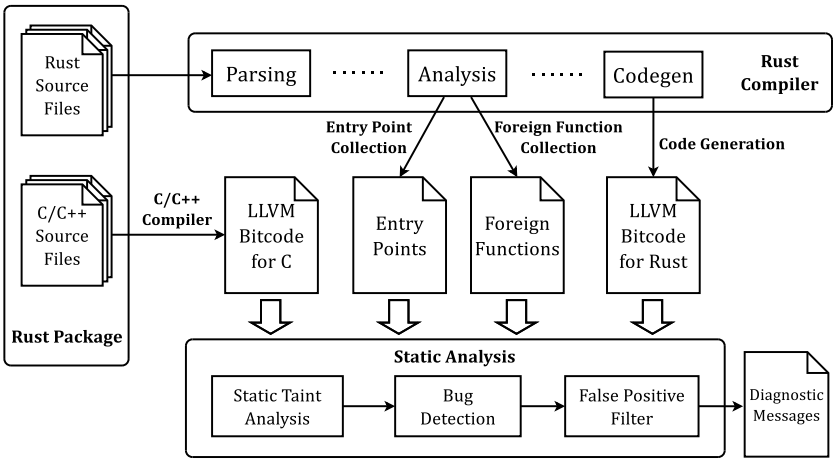

Fig. 2. The architecture of FFIChecker.

4.1 User Interface

The goal of the user interface is to get a Rust package being analyzed from the user and prepare all the ingredients that the static analyzer requires, such as the LLVM bitcode and a set of appropriate entry points. Then it works as a driver program that delegates the remaining procedures to other components. The user interface takes a Rust package as input, which contains one or many Rust *crates* and C/C++ source files (if they exist). A *crate* is a unit of compilation and linking for the Rust compiler. It contains one or many Rust source files and may depend on other *crates*. We leverage the official build system `Cargo` to resolve dependencies and download all the dependent *crates*. Then different source files are dispatched to either the Rust compiler or the C/C++ compiler, and both the compilers are configured to generate LLVM bitcode. The `Cargo` integration provides a user-friendly interface similar to many existing tools used by Rust developers, such as `Clippy`, so that users can easily integrate `FFIChecker` into their daily development workflow.

4.2 Entry Point and Foreign Function Collection

Performing static analysis requires an appropriate function as the entry point. We focus on public functions/methods for a Rust program because they are visible to attackers and hence may be exploited. Also, since we care about the cross-language scenario, we want to distinguish whether a function is written in Rust or C/C++. The entry point/foreign function collector is designed to collect all of the information we need. Specifically, after the user interface downloads all the dependencies, the collector is invoked to process each of these crates and collects: (1) a list of public functions/methods, and (2) a list of C/C++ functions called in the Rust program. The collector is implemented as a customized callback function of the Rust compiler, so that it can access the internal data structures inside the compiler. It first goes through the Rust High-level Intermediate Representation (HIR) generated by the Rust compiler, which contains required information such as the function names, visibility, and whether it is implemented in Rust or C/C++. Then it extracts the required function names and passes them to the static analyzer.

4.3 Static Analysis and Bug Detection

The LLVM bitcode, entry points, and foreign functions are sent to the static analyzer as input. The static analyzer performs analysis by traversing the control flow graph (CFG) provided by the LLVM bitcode. The details of the algorithms will be discussed in Section 6. Once the static analysis finishes, a bug detection module reads the analysis results and generates diagnostic messages. The messages are filtered by user-specified rules in order to suppress false positives, and then printed to users (§ 6.3). According to the diagnostic messages, users can manually inspect the source code and pinpoint potential bugs in their programs.

5 Abstract Interpretation

In this section, we present the definition of our abstract domain and transfer functions based on the language model of LLVM IR.

5.1 LLVM IR, Abstract Values and Abstract Domain

In LLVM IR, a single function is modeled as a Control Flow Graph (CFG), where each node is a basic block containing one or more instructions without any jumps. At the end of each basic block, there is one terminator, a special instruction representing a jump among the control flow. Static analysis models the program execution in a certain *abstract domain*, and each element of the domain represents a certain execution state, which is referred to as an *abstract state*. It first assigns *abstract states* to each variable and basic block, then traverses the CFG and updates these states according to the semantics of each instruction. The *abstract domain* varies depending on different purposes. We design our abstract domain as follows in order to capture the ownership state of heap memory. Note that our design is derived from the classical Abstract Interpretation literature [26,29].

For each CFG, we denote the set of all the variables that appear in the CFG as **Var**, and the set of all basic blocks in the CFG as **Block**. To distinguish whether a variable stores heap memory and its state in the ownership system (e.g., whether it is *borrowed* or *moved*), we define the state **MState** as a *lattice* with 5 elements, a partial ordering relation $\sqsubseteq$ and a join operator $\sqcup$, as shown in Figure 3. Intuitively, the bottom element ($\bot$) is the default value for all variables. When a variable is initialized by a heap memory allocation procedure, we mark it as *Alloc*. Note that a heap memory can be passed to FFI by either taking a reference (*borrow*) or forgetting its ownership (*move*). We distinguish them by the corresponding states $Borrowed$ and $Moved$. To be conservative, when the state cannot be determined, we set it as the top element ($\top$).

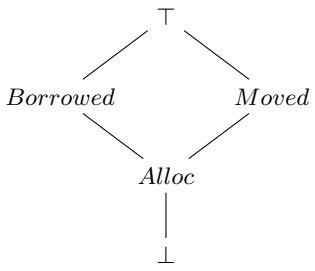

Fig. 3. **MState** lattice used by FFIChecker.

To keep track of the abstract values for each basic block, we maintain a lookup table σ_b : **Var** $\rightarrow$ **MState** for each basic block b. The abstract state **AState** is defined as a map lattice consisting of all the mappings from **Var** to **MState**.

Intuitively, an element in **AState** is a lookup table, which depicts the abstract memory state for each variable after executing the current basic block of the program. **AState** is still a lattice and the partial ordering is defined as:

$$\text{For } \sigma_1, \sigma_2 \in \mathbf{AState}, \sigma_1 \sqsubseteq \sigma_2 \iff \forall a \in \mathbf{Var}, f(a) \sqsubseteq g(a).$$

And the $\sqcup$ operator is defined pointwise in terms of the operators from **MState**:

$$\forall \sigma_1, \sigma_2 \in \mathbf{AState}, \sigma_1 \sqcup \sigma_2 = \{(a, \sigma_1(a) \sqcup \sigma_2(a)) : \forall a \in \mathbf{Var}\}.$$

Finally, the *abstract domain* is defined as a mapping from all basic blocks **Block** to **AState**. Equivalently, it is defined as the powerset of **AState**, i.e., $\mathbf{Domain} = 2^{\mathbf{AState}}$.

5.2 Transfer Functions

In static analysis, transfer functions are used to extract information from the program semantics and update the abstract states. Since FFIChecker runs on LLVM IR, we assign a transfer function to each LLVM instruction according to its semantics. Specifically, we focus on the following instructions: (1) Instructions that affect the data flow such as `load`, `store`, and `GetElementPtr`, because we need to propagation the abstract states. (2) Instructions that call other functions, such as `Call` and `Invoke`, through which we perform context-sensitive interprocedural analysis (§ 6.2). For details, please refer to our implementation.

6 Algorithms

In this section, we present the main algorithms used in FFIChecker. The algorithms consist of three parts: (1) A fixed-point algorithm that traverses a CFG and executes transfer functions until a fixed point is reached. (2) An algorithm that achieves context-sensitive interprocedural analysis. (3) A bug detection algorithm used to determine whether there are any potential bugs.

6.1 Fixed-Point Algorithm

Similar to most static analysis tools, FFIChecker traverses a given CFG and iteratively runs transfer functions to update the abstract state until it reaches a fixed point. The fixed-point algorithm is formulated in the Appendix (Algorithm 1). We implement the classical *worklist* algorithm [26,29], where the worklist W is a set initialized to contain all the basic blocks in the CFG. Then the algorithm chooses a basic block b from W and analyzes it by executing the transfer functions of its instructions. The state is updated by joining the states of all the predecessors of b. If the state changes, all the successors of b will be inserted into the worklist, waiting for a re-analysis. This procedure is repeated until the worklist W becomes empty. The algorithm terminates because either the state goes "up" in the lattice (because of the join operator), or the length of W decreases. Since the lattice we defined has finite height, W will eventually be depleted.

6.2 Analyzing Function Calls

When analyzing instructions that call other functions, such as `Call` and `Invoke`, FFICHECKER performs interprocedural analysis. Different functions need different treatments, therefore we categorize functions into different types: (1) Functions that allocate heap memory, e.g., `exchange_malloc`. These are the "taint sources" of our algorithm, indicating that the resulting variable stores heap memory, so we can mark its abstract state into *Alloc*. (2) Functions that *borrow* a reference (e.g., `Vec::as_mut_ptr`) or *move* the ownership (e.g., `Box::into_raw`). These functions change the abstract state of heap memory into either *Borrowed* or *Moved*. (3) Foreign functions called through FFI. These are the potentially vulnerable functions that FFICHECKER cares about. FFICHECKER will analyze these functions and see whether there are any bugs (§ 6.3). (4) LLVM intrinsic functions and the Rust standard library functions. The former are implemented by the compiler backend so their implementations do not even exist in LLVM IR. The latter are commonly used but usually hard to be analyzed because of their complexity and heavy abstraction. These functions are also not FFICHECKER's targets because our goal is to find bugs in third-party libraries instead of in the Rust compiler or the standard library. Therefore, we provide some special handlers that work as the model of these functions by resembling their behaviors. FFICHECKER internally maintains a map between such functions and their handlers, and will execute the handler instead of launching a new function analysis. (4) For all other functions, FFICHECKER launches context-sensitive interprocedural analysis by initializing a new fixed-point algorithm instance for this function. The algorithm is formulated in the Appendix (Algorithm 2).

6.3 Bug Detection and False Positive Suppression

After the fixed-point algorithm terminates, FFICHECKER checks whether there are any variables that store heap memory but are passed to FFI. If this is the case, some heap memory leaks into the external code, which may lead to potential vulnerabilities. To further determine the bug type, FFICHECKER launches a new function analysis instance for all foreign functions to which some heap memory is passed, and checks whether the heap memory is freed or not in the external code. Then it generates warnings according to the ownership state of the heap memory. For example, suppose a variable is *moved* across FFI by Rust and freed in C. In that case, this is an undefined behavior caused by mixing memory management mechanisms (§ 3.3). The rules of warning generation are summarized in Table 1.

As shown in the table, we also tag a confidence level on each generated warning depending on how much information we can leverage during the analysis. For example, the LLVM IR of a foreign function is not always available because it may come from a dynamically linked C library. Or it may be called via a function pointer, so FFICHECKER cannot statically know which function is called. In this case, FFICHECKER cannot further analyze the foreign function, so it generates warnings with lower confidence. This design helps us to suppress false alarms. We implement a precision filter to determine what level of warning messages

is reported to users. Only warnings with a confidence level higher than the filter's threshold will be issued. Users can pass command-line options to the user interface to override the default filter configuration.

Table 1. Rules of warning generation. The reported warnings include use-after-free (UAF), double free (DF), undefined behavior (UB), and memory leak (LEAK). SAFE means no warning is issued. The confidence levels (high, medium, or low) are enclosed in parentheses.

	C Code is Unavailable	**C Code is Available**	
		Freed	Not Freed
Borrowed	UAF/DF (Low)	UAF/DF (High)	SAFE
Moved	UB/LEAK (Mid)	UB (High)	LEAK (Mid)

7 Implementation

FFICHECKER is written in Rust (2,468 lines of code) and has three binaries, which are the user interface, entry point/foreign function collector, and static analyzer. The user interface is implemented as a `cargo` sub-command, which tightly integrates with the official build system. Users can easily integrate FFICHECKER in their daily workflow and check their packages by a single command: `cargo ffi-checker`. The entry point/foreign function collector is implemented as a customized Rust compiler, in which we insert the collector routine as a callback function. The callback function is invoked automatically after the compiler gathers all the information of the source code. Thus it can access the internal compiler data structures such as HIR. The static analyzer is a standalone binary configurable through the user interface. Users can specify the precision filter, which determines whether to issue a warning message according to its priority. We also provide several Python scripts for downloading packages on the official package registry and running evaluations.

8 Evaluation

In order to evaluate FFICHECKER in terms of its effectiveness and performance, we collect Rust packages as test cases on the official package registry crates.io. Since we care about the cross-language scenario and focus on external code written in C/C++, we only crawl packages that heavily use the FFI between Rust and C/C++. Specifically, we download packages that are of category "external-ffi-bindings", or depend on other packages that assist the use of FFI, such as `cc`, `bindgen`, or `cbindgen`. Finally, we collect a total of 987 packages as our analysis targets, which contain 3,232,574 lines of Rust and 46,321,573 lines of C/C++.

All the experiments were done on a machine with a 3.70 GHz Intel Xeon E5-1630 v4 CPU and 16GB RAM, running Gentoo Linux (kernel 5.15.32).

8.1 Effectiveness and Performance of FFIChecker

We run FFIChecker on our dataset, and it generates 222 warnings. Then we manually inspect the output at a rate of about 100 reports per person-hour. Finally, 34 bugs (19 memory leaks, 3 exception-related bugs, 12 undefined behaviors) in 12 packages are confirmed. The statistical details are listed in Table 2, where columns "# of Bugs" and "Reports" show the number of true positives we confirmed and the number of warnings in the emitted diagnostic messages with different confidence levels. We have reported all the bugs to the package maintainers. At the time of writing, 15 bugs were confirmed and fixed. For more details, we refer readers to our GitHub repository[9].

We further measure the execution time and memory usage of FFIChecker for all the 987 packages. We run the evaluation in 8 parallel threads, and FFIChecker can finish all the analysis in 5.2 hours with at most 4.1 GB memory consumption. On average, FFIChecker can analyze a package in 116.9 CPU seconds with 1,056.6 MB memory consumption. Note that the execution time and memory usage do not correlate to the lines of code or the number of interfaces, because the convergence of the fixed-point algorithm mainly depends on the structure of the CFG. Overall, FFIChecker is scalable enough to analyze real-world Rust packages with a reasonable amount of computational resources.

Table 2. Bugs detected by FFIChecker. The types of bugs include memory leak (LEAK), exception safety (EXC), and undefined behavior (UB). "N/A" means that the foreign functions are from shared libraries instead of the Rust package.

Package	# of Bugs	Reports			Bug Type	Elapsed Time (s)	Memory Usage (MB)	# of Entries	# of FFIs	LoC	
		High	Mid	Low						Rust	C/C++
arma-rs	3	0	1	0	LEAK	38.67	1040.85	29	4	1686	N/A
cobyla	1	0	1	0	LEAK	48.14	1979.54	2	1	225	1635
emd	1	0	1	0	LEAK	7.21	237.75	4	1	87	541
impersonate	1	0	1	0	LEAK	19.11	767.54	6	1	117	61
iredismodule	11	0	0	10	EXC, LEAK	78.15	1958.46	364	230	3761	777
jyt	6	0	0	1	UB	97.25	2711.75	3	6	450	N/A
liboj	1	0	0	3	LEAK	108.58	3109.21	86	38	1342	N/A
libtaos	1	0	0	1	EXC	99.23	1724.13	461	50	5491	N/A
moonfire-ffmpeg	1	0	0	1	UB	7.83	228.78	53	92	1513	231
pdb_wrapper	1	0	0	1	EXC	68.04	2530.41	20	14	499	375
snap7-rs	2	0	1	4	LEAK	8.97	203.77	387	276	6110	14085
triangle-rs	5	0	1	0	UB	47.46	1095.58	34	2	681	15050

8.2 Understanding False Positives and False Negatives

FFIChecker reports numbers of false positives. After inspecting the reported warnings, we summarize two reasons that lead to the false alarms: (1) It is common that Rust calls foreign functions from dynamically linked shared libraries. Therefore the LLVM IR of the foreign code is not available. In this case, FFIChecker cannot further analyze the foreign function, so it generates

[9] https://github.com/lizhuohua/rust-ffi-checker/tree/master/trophy-case

imprecise results. (2) FFICHECKER cannot always distinguish whether a variable is *borrowed* or *moved* via LLVM IR because the *borrowing*/*moving* operations may be optimized away by the Rust compiler.

During the manual inspection, we also observe some bugs in functions with generic type parameters but they are not reported by FFICHECKER. The reason is that the Rust compiler will not generate code for generic functions unless they are monomorphized, meaning that FFICHECKER cannot find the LLVM IR for generic functions that are only implemented in the package but not used.

Nevertheless, as presented in Section 6.3, FFICHECKER generates warnings with different confidence levels. Users can configure the precision filter through command-line options to only output warnings with high confidence. Even if all the warnings are issued, users can still filter out false alarms quickly during the manual inspection with the help of the confidence levels attached to them.

9 Discussion

Thoughts about Rust's security guarantees. As shown in Table 2, most bugs we found are memory leaks. We interpret this as a limitation of Rust's security guarantees: memory leak is considered *safe* in Rust [16]. The reason behind this design choice is that leaking resources is possible in pure safe Rust (consider creating a cycle of reference-counted pointers using interior mutability). Therefore, the authors of the Rust standard library decide not to mark functions that leak memory as unsafe, such as `mem::forget`. As a result, the Rust compiler will not give any warnings when inexperienced programmers misuse these functions and cause memory leaks, leading to denial of service attacks or information leakage.

Future work. Although we focus on Rust combined with C/C++, the idea of FFICHECKER and the threat model can be extended to other cross-language scenarios. Especially, the static analyzer is designed to be an individual binary that operates on LLVM IR. Therefore by changing the Rust-specific part of the system, our approach can be adapted to analyze other FFIs, as long as they support the LLVM backend for code generation, e.g., languages such as Haskell, Julia, and Swift.

10 Related Work

10.1 Static Analysis for Rust

Many existing studies extend off-the-shelf static analysis engines to perform bug detection on LLVM IR generated by the Rust compiler. Lindner *et al.* [23] use the symbolic execution engine KLEE [5] to verify whether a program is panic-free. SMACK [32,3] translates LLVM IR into the Boogie intermediate verification language [11]. Rust2Viper [14] and Prusti [1] utilize user-provided specifications and the Viper [28] symbolic execution engine to verify functional correctness

properties. CRUST [38] translates functions that contain unsafe code to C, then it generates tests and checks them by the CBMC [6] model checker.

There are also many tools that work on Rust's own intermediate representation. Qin *et al.* [31] build two bug detectors for use-after-free and double-lock bugs according to their empirical studies on Rust security issues. SafeDrop [10] focuses on the deallocation of heap memory and detects memory corruptions by performing alias analysis and taint analysis on Rust MIR. MIRAI [25] is a formal verification tool that performs symbolic execution on Rust MIR. It enables users to add annotations and utilizes the SMT solver Z3 [27] to prove the correctness of Rust programs. MirChecker [22] collects both the numerical and symbolic information from Rust MIR, and detects runtime panics and memory-safety issues without the need for annotations. Rudra [2] uses both Rust MIR and HIR, and detects potential memory safety bugs in unsafe Rust.

10.2 Cross-Language Bug Detection and Prevention

It is well-known that developing software using multiple languages may interfere with each other and lead to subtle bugs. Mergendahl *et al.* [24] propose a threat model to reason about cross-language attacks. They also demonstrate these attacks on Rust and Go. Kondoh *et al.* [17] use static analysis to detect common mistakes and bad programming practices when using Java Native Interface (JNI). Tan *et al.* [35] apply static analysis and carry out an empirical security study on a portion of the native code in Sun's Java Development Kit (JDK). JET [20,19] is a static analysis tool that enforces exception checking and reports bugs on Java exceptions raised in native code through JNI. Jinn [18] is a compiler and virtual machine independent bug detection tool for both JNI and Python/C. Galeed [33] and PKRU-Safe [15] isolate heap memory at runtime using Intel Memory Protection Keys (MPK), such that unsafe (external) code cannot corrupt memory used exclusively by the safe-language components.

Unlike these existing efforts, our work focuses on the memory management issues between Rust and C/C++. The new pattern of bugs introduced by the interaction between the Rust ownership system and C/C++ is out of the scope of all the existing detection or prevention efforts.

11 Conclusion

Rust leverages FFI to invoke external C/C++ code, making incremental software development convenient and efficient. In this paper, we showed that there could be security issues since programmers may make mistakes when using FFI. To secure the use of FFI, we designed and implemented FFIChecker, an automated static analysis tool based on augmented taint analysis, which captures the state transitions of heap allocations when they are passed to external code through FFI. It can detect potential memory management issues across the FFI boundaries. We evaluated it by analyzing 987 real-world Rust packages. It successfully revealed 34 bugs in 12 packages that were unknown previously. Finally, we open-sourced FFIChecker with various examples and test scripts.

Acknowledgments The work of Zhuohua Li, Jincheng Wang, and John C.S. Lui were supported in part by the RGC's RIF R4032-18.

References

1. Astrauskas, V., Müller, P., Poli, F., Summers, A.J.: Leveraging Rust Types for Modular Specification and Verification. Proceedings of the ACM on Programming Languages **3**(OOPSLA), 1–30 (2019)
2. Bae, Y., Kim, Y., Askar, A., Lim, J., Kim, T.: Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale. In: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. p. 84–99. SOSP '21 (2021)
3. Baranowski, M., He, S., Rakamaric, Z.: Verifying Rust Programs with SMACK. In: Proceedings of the 16th International Symposium on Automated Technology for Verification and Analysis. pp. 528–535. ATVA '18 (2018)
4. Bushev, D.: Language details of the Firefox repo (2022), https://4e6.github.io/firefox-lang-stats/
5. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. pp. 209–224. OSDI '08 (2008)
6. Clarke, E., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 168–176. TACAS '04 (2004)
7. Cousot, P., Cousot, R.: Static Determination of Dynamic Properties of Programs. In: Proceedings of the 2nd International Symposium on Programming. pp. 106–130. ISOP '76 (1976)
8. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 238–252. POPL '77 (1977)
9. Cousot, P., Cousot, R.: Systematic Design of Program Analysis Frameworks. In: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 269–282. POPL '79 (1979)
10. Cui, M., Chen, C., Xu, H., Zhou, Y.: SafeDrop: Detecting Memory Deallocation Bugs of Rust Programs via Static Data-Flow Analysis (2021)
11. DeLine, R., Leino, R.: BoogiePL: A Typed Procedural Language for Checking Object-Oriented Programs. Tech. Rep. MSR-TR-2005-70 (2005)
12. Evans, A.N., Campbell, B., Soffa, M.L.: Is Rust Used Safely by Software Developers? In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. pp. 246–257. ICSE '20 (2020)
13. Girard, J.Y.: Linear Logic: Its Syntax and Semantics. In: Proceedings of the Workshop on Advances in Linear Logic. pp. 1–42 (1995)
14. Hahn, F.: Rust2Viper: Building a Static Verifier for Rust. Master's thesis, ETH Zürich (2016)
15. Kirth, P., Dickerson, M., Crane, S., Larsen, P., Dabrowski, A., Gens, D., Na, Y., Volckaert, S., Franz, M.: Pkru-safe: Automatically locking down the heap between safe and unsafe languages. In: Proceedings of the Seventeenth European Conference on Computer Systems. p. 132–148. EuroSys '22 (2022)

16. Klabnik, S., Nichols, C.: The Rust Programming Language. No Starch Press, USA (2018)
17. Kondoh, G., Onodera, T.: Finding Bugs in Java Native Interface Programs. In: Proceedings of the 2008 International Symposium on Software Testing and Analysis. p. 109–118. ISSTA '08 (2008)
18. Lee, B., Wiedermann, B., Hirzel, M., Grimm, R., McKinley, K.S.: Jinn: Synthesizing Dynamic Bug Detectors for Foreign Language Interfaces. In: Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 36–49. PLDI '10 (2010)
19. Li, S., Tan, G.: Finding Bugs in Exceptional Situations of JNI Programs. In: Proceedings of the 16th ACM Conference on Computer and Communications Security. p. 442–452. CCS '09 (2009)
20. Li, S., Tan, G.: JET: Exception Checking in the Java Native Interface. In: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications. p. 345–358. OOPSLA '11 (2011)
21. Li, Z., Wang, J., Sun, M., Lui, J.C.: Securing the Device Drivers of Your Embedded Systems: Framework and Prototype. In: Proceedings of the 14th International Conference on Availability, Reliability and Security. pp. 1–10. ARES '19 (2019)
22. Li, Z., Wang, J., Sun, M., Lui, J.C.: MirChecker: Detecting Bugs in Rust Programs via Static Analysis. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. p. 2183–2196. CCS '21 (2021)
23. Lindner, M., Aparicius, J., Lindgren, P.: No Panic! Verification of Rust Programs by Symbolic Execution. In: 2018 IEEE 16th International Conference on Industrial Informatics. pp. 108–114. INDIN '18 (2018)
24. Mergendahl, S., Burow, N., Okhravi, H.: Cross-Language Attacks. In: Proceedings of the Network and Distributed System Security Symposium (NDSS'22) (2022)
25. MIRAI Contributors: MIRAI: Rust mid-level IR Abstract Interpreter (2022), https://github.com/facebookexperimental/MIRAI
26. Møller, A., Schwartzbach, M.I.: Static Program Analysis (2018)
27. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. TACAS '08 (2008)
28. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A Verification Infrastructure for Permission-Based Reasoning. In: Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583. pp. 41–62. VMCAI '16 (2016)
29. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer Publishing Company, Incorporated (2010)
30. Ojeda, M.: Rust for Linux (2022), https://github.com/Rust-for-Linux
31. Qin, B., Chen, Y., Yu, Z., Song, L., Zhang, Y.: Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 763–779. PLDI '20 (2020)
32. Rakamaric, Z., Emmi, M.: SMACK: Decoupling Source Language Details from Verifier Implementations. In: Proceedings of the 26th International Conference on Computer Aided Verification. pp. 106–113. CAV '14 (2014)
33. Rivera, E., Mergendahl, S., Shrobe, H., Okhravi, H., Burow, N.: Keeping safe rust safe with galeed. In: Annual Computer Security Applications Conference. p. 824–836. ACSAC '21 (2021)
34. Secure Rust Guidelines Contributors: Secure Rust Guidelines (2022), https://anssi-fr.github.io/rust-guide/

35. Tan, G., Croft, J.: An Empirical Security Study of the Native Code in the JDK. In: 17th USENIX Security Symposium (USENIX Security 08) (Jul 2008)
36. The Rust FFI Omnibus Contributors: The Rust FFI Omnibus (2022), http://jakegoulding.com/rust-ffi-omnibus/
37. The Rustonomicon Contributors: The Rustonomicon (2022), https://doc.rust-lang.org/nomicon/
38. Toman, J., Pernsteiner, S., Torlak, E.: Crust: A Bounded Verifier for Rust. In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering. pp. 75–80. ASE '15 (2015)
39. Unsafe Code Guidelines Working Group: Rust's Unsafe Code Guidelines Reference (2022), https://rust-lang.github.io/unsafe-code-guidelines/
40. Wadler, P.: Linear Types Can Change the World! In: Programming Concepts and Methods (1990)
41. Xu, H., Chen, Z., Sun, M., Zhou, Y., Lyu, M.R.: Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs. ACM Transactions on Software Engineering and Methodology **31**(1) (sep 2021)

Appendix

A Fixed-Point Algorithm

Algorithm 1: Fixed-point algorithm for FFIChecker

Input: Control Flow Graph: CFG
Output: Abstract State: $State$
Initialization: $State[n] \leftarrow \bot$ for all n
Function `FixedPoint`(CFG, $State$)**:**
  $W \leftarrow CFG.basicblocks$
  while $W \neq \emptyset$ **do**
    $b \leftarrow W$.`remove()`
    foreach $instr \in b.instructions$ **do**
      `Transfer`($State[b], instr$)
    `Transfer`($State[b], b.terminator$)
    $new\_state \leftarrow \bigsqcup_{n \in \texttt{Predecessors}(b)} State[n]$
    if $new\_state \not\sqsubseteq State[b]$ **then**
      $State[b] \leftarrow new\_state$
      foreach $v \in$ `Successors`(b) **do**
        W.`insert`(v)
  return $State$

B Context-Sensitive Interprocedural Analysis

To avoid duplicated analysis for the same function, we also implement the classical summary-based method [26,29]. It caches previously computed results (i.e., summaries) in a lookup table $cache : ((f, in\_state), out\_state)$ that maps a calling context (f, in_state) to an output out_state. (f is a function, in_state is the abstract state of its input, and out_state is the corresponding output.) Before analyzing a function, we first check whether there is an existing summary

that has been computed. If it is the case, the fixed-point algorithm is skipped and the result is directly returned. If not, the fixed-point algorithm is performed and the analysis result is cached in the lookup table.

Algorithm 2: Interprocedural analysis algorithm for FFIChecker

Input: Function: f, Arguments: $args$, Destination: $dest$,
State of the current basic block: σ, Summary Cache: $cache$
Output: Updated State: σ

```
1  begin
2      switch FunctionType(f) do
3          case Heap Allocation do
4              σ[dest] ← Alloc
5          case Borrow Arguments do
6              σ[arguments that are borrowed] ← Borrowed
7          case Move Arguments do
8              σ[arguments that are moved] ← Moved
9          case FFI do
10             Run AnalyzeFunction and generate warnings if necessary
11         case LLVM Intrinsic or Standard Library do
12             Handle it through function models
13         otherwise do
14             AnalyzeFunction(f, args, dest, σ)
   // Subroutines
15 Function AnalyzeFunction(f, args, dest, σ):
16     in_state ← state generated by args
17     summary ← GetFunctionSummary(f, in_state)
       // Set the state of the return value
18     σ[dest] ← summary.ret_state
       // Propagate the state of parameters
19     foreach (caller_arg, callee_arg) do
20         σ[caller_arg] = σ[callee_arg]
21 Function GetFunctionSummary(f, in_state):
       // If the summary has been computed, directly return it
22     if (f, in_state) in cache then
23         return cache[(f, in_state)]
       // Initialize initial state for the fixed-point algorithm
24     forall n do
25         State[n] ← ⊥
26     foreach (state, param) in zip(in_state, f.parameters) do
27         State[param] ← state
       // Compute the summary and cache it
28     out_state ← FixedPoint(f.CFG, State)
29     cache[(f, in_state)] ← out_state
30     return out_state
```