École Polytechnique Fédérale de Lausanne

Characterization of interface definition bugs for Rust projects

by Andrija Jelenković

Master Project Report

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

The External Reviewer
External Expert

Sánchez Marín Andrés
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

January 23, 2024

They don’t know me son

- David Goggins

Abstract

The FooSystem tool enables analysis of a Rust project's foreign function interface calls.

According to Microsoft (reference), around 70% of all Common Vulnerabilities and Exposures (CVEs) are memory safety errors. Rust, as a memory safe language, completely prevents these classes of errors. However, because of existing tooling and software, it still needs to maintain interoperability with unsafe programming languages such as C. For this to happen, Rust has to enable the use of "raw pointers" in "unsafe" code blocks, which relax the ownership and borrow-checking rules. As a result, it is still possible to have memory safety issues in "unsafe Rust", both through misuse of raw pointers and through memory safety issues in foreign function calls.

Since the vast majority of non-trivial Rust projects use C libraries (e.g. libc, bzip2, winapi etc.), it is not possible to guarantee memory safety. Currently, there is some research in this area, but it is focused at defining the Rust memory model (Stacked borrows) and determining the memory safety of Rust's unsafe blocks (Miri). Therefore, the FooSystem tool was developed to analyise the foreign function calls of Rust projects by modifying the (Cargo Geiger project).

Contents

Chapter 1

Introduction

Rust is a memory safe general purpose language without garbage collection. It is supposed to address the shortcomings of C++ and, ideally, replace it. In order to accomplish this, the writers of Rust have allowed interop with other programming languages (notably C) through foreign function interfaces. This allows to reuse already implemented functionality and to continue improving it. However, this is a double-edged sword, as the Rust compiler obviously cannot enforce ownership and borrowing rules for code written in other programming languages. As a result, it is possible to introduce memory safety errors in correct Rust programs.

The aim of the FooSystem tool is to analyse the function calls through FFI. This analysis can be used for determining potential bugs in Rust programs which use externally defined functions.

There is a lot of current work on unsafe Rust, however, this research is mostly focused on detecting undefined behaviour in unsafe Rust and not on FFI.

The core contribution of this document is a tool for the analysis of FFI and to point out that the large and ubiquitous use of externally defined functions in Rust programs, opening otherwise safe Rust programs to vulnerabilities.

Chapter 2

Background

Most memory-safe languages use a garbage collector for memory management. In these languages the runtime tracks which variables are referencing which memory locations, and, it periodically stops execution of the program to determine which memory locations are unreachable in order to free some memory. This approach guarantees that it is not possible to make memory safety bugs, however, it comes at a performance cost due to the necessary runtime checks. If you can be sure that your program is correct, you must still pay this price.

There are performance-critical systems that cannot justify the drop in performance. For a long time, these systems were forced to trade memory safety for speed as there was no alternative approach. This is what Rust's type system promises: a memory-safe program with performance on par with C/C++ programs.

This is possible because of Rust's unique "ownership model". Every variable uniquely "owns" the containing memory and it is not possible to have 2 different variables owning the same memory. By default, when assigning variables, the ownership is passed, therefore, it is not possible to use the "old" variable after the assignment. Because Rust will automatically free the memory once an "owning" variable goes out of scope, this rule prevents the double-free error.

```
let a = String::from("hello");
let b = a;
// Compile error, because ownership of string
// "hello" was passed to b
a.len();
```

Figure 2.1: Ownership

There is another mechanism which completes the memory safety - borrows (references). An

immutable borrow allows shared immutable access to memory, while a mutable borrow allows unique mutable access. In practice, this means that you can have multiple immutable borrows at the same time, but, the moment you borrow mutably, it is only possible to access a variable through this mutable borrow.

```
let mut a = 10;
let b = &a; //Immutable (shared) borrow
let c = &b; //Immutable (shared) borrow
println!("{b} {c}"); // Allowed
let d = &mut a; //Mutable borrow, b and c invalidated
println!("{b} {c}"); // Not allowed, because d is mutably borrowed
println!("{d}");
```

Figure 2.2: Borrowing

References enable easier reuse, however, the compiler still has to track that the lifetime of the reference does not outlive the lifetime of the memory it references.

```
let mut out = String::from("outer");
let mut reference = &mut out;
{
    let mut a = String::from("a");
    reference = &mut a;
    // a implicitly dropped at the end of the scope
}

// reference is now a dangling pointer
// dereferencing it would produce undefined behaviour
reference.push('b');
```

Figure 2.3: Dangling pointer

In the figure 2.3, it is unknown what would happen when the `reference.push('b')` is called. It is possible that the program continues to work as expected, but it is also possible that the program terminates due to a segmentation fault. In any case, referencing memory which was freed beforehand is an error. For this reason, it is necessary that the lifetime of the reference does not exceed the lifetime of the value it references.

Every "borrow" in Rust has a lifetime, but, in most cases, the compiler can infer these lifetimes, which are therefore left out. In the figure 2.4, the two `foo` function declarations are equivalent, because the compiler can infer that the function output has to be a reference of the value the input references. Consequently, the lifetime of the input has to be at least as long as the lifetime of the output. As a result, the programmer does not have to specify lifetimes explicitly. This feature of

Rust's compiler is also known as "lifetime elision".

```
fn foo(arg: &String) -> &String {
    ...
}

// 'a is a lifetime specifier, this means that the arg reference has to have
// a lifetime which is not shorter than the lifetime of the function return
fn foo<'a>(arg: &'a String) -> &'a String {
    ...
}
```

Figure 2.4: Lifetime elision

Unfortunately, there are circumstances when the compiler cannot infer the lifetimes. In such cases, the programmer has to manually provide the needed lifetimes. In the figure 2.5, the compiler is not able to infer the lifetime, because two arguments are references to a string and the output is a reference to a string as well. The compiler does not do an in-depth analysis of the code and cannot infer that the lifetime of the function return depends on the lifetime of the first argument. As a consequence, the programmer has to manually provide a lifetime.

```
// In this case, the return value of the function is a reference
// to the value arg1 references, therefore, arg1's lifetime
// must not be shorter than the lifetime of the function's return value
fn bar<'a>(arg1: &'a String, arg2: &String) -> &'a String {
    return arg1
}
```

Figure 2.5: Lifetimes

With these rules in place, the compiler will prevent the programmer from making mistakes. However, because of the complexity and the Halting problem, the compiler will sometimes reject perfectly valid programs. In fact, there are parts of the standard library (e.g. `Vec`, `HashMap`) which are extremely hard to implement with the aforementioned rules. Fortunately, there is a workaround for this problem - the `unsafe` keyword.

Inside an `unsafe` block, there are no rules. The compiler will not intervene and the programmer is responsible for all the chaos that ensues. The main idea is to provide the freedom to express any program, but to limit this freedom to small portions of code which can be trivially checked for correctness. With this approach, the programmer has to be extra careful not to shoot themselves in the foot, because the compiler will gladly allow it. Therefore, it still makes sense to follow the rules even in unsafe code, but the programmer will have to apply them manually.

The programmer is also allowed to use raw pointers and call functions defined in another

programming language (such as C) through a foreign function interface (FFI).

The figure 2.6 represents a code snippet from the Rustonomicon showing how to call a function through FFI:

```
use libc::size_t;

#[link(name = "snappy")]
extern {
    fn snappy_max_compressed_length(source_length: size_t) -> size_t;
}

fn main() {
    let x = unsafe { snappy_max_compressed_length(100) };
    println!("max compressed length of a 100 byte buffer: {}", x);
}
```

Figure 2.6: Rustonomicon FFI

The function `snappy_max_compressed_length` is defined as an external function available in the dynamically linked C library "snappy". It has to be called from an unsafe block, because the compiler cannot guarantee that the function is memory safe.

While Rust is a memory-safe language, because it is young compared to established programming languages like C and C++, there are not as many libraries written in it. As a consequence, almost all non-trivial Rust programs call into external libraries written in another languages (usually C). Therefore, it is still possible to have memory safety errors. In the coming chapters, there will be more talk about the use of external functions in Rust programs.

Chapter 3

Design

The FooSystem tool works at the abstract syntax tree level. This choice was made for two reasons: performance and difficulty of implementation.

In general, compiling Rust is very slow compared to other programming languages (excluding maybe C++). This is because during compilation, Rust programs go through multiple representations until they are compiled to assembly:

1. High-Level Intermediate Representation (HIR)
2. Mid-level Intermediate Representation (MIR)
3. LLVM Intermediate Representation

"Lowering" code into each of these representations is costly and it should be avoided unless absolutely necessary. Thankfully, for this use case, this kind of processing is not needed, as all the information about external functions and calls is available at the abstract syntax tree (AST) level.

Also, the author of this document was not familiar with Rust before the project and it would be very time-consuming and inefficient to understand the HIR and MIR. Especially considering that these representations do not offer meaningful additional information compared to an AST representation.

The tool leverages cargo - Rust's package manager to resolve needed dependencies, download the necessary source code, and invoke build scripts. Cargo allows programmatic invocation of the provided functions, however, there is not a lot of documentation available for this use case online.

Once all the source files are generated, each file is converted to an AST representation for easier analysis. The analysis consists of two phases:

1. First pass over all of the file ASTs to collect all the functions defined in `extern` blocks
2. Second pass to analyze each function call and determine whether it is calling a function defined through a foreign function interface

The analysis includes information such as the file, line, column, and package where an external function is called. Also, for each external function, the file, line, column, and package of the definitions are included, alongside the functions return type and arguments.

Chapter 4

Implementation

The tool FooSystem is an adaptation of Cargo Geiger. Cargo Geiger is a tool which analyses Rust programs and reports unsafe usage. It is fairly limited and performs only a surface level analysis of Rust programs. Geiger reports the number of unsafe blocks, expressions, functions etc. in a given Rust program and its dependencies. It is good to get a feel of the size of the attack surface of a Rust program, but, otherwise, it does not provide meaningful analysis.

It works by programmatically invoking cargo to resolve the dependency tree of a Rust program. Then, it uses the syn crate to construct the abstract syntax tree and crawl it. This process can be parallelised, as all of the files can be analysed independently.

The "syn" crate includes a `Visit` trait which is used to implement an AST visitor. This trait has default implementation of the visit method for each node in the AST. The programmer only has to implement the methods necessary for the analysis they want to perform.

The figure 4.1 shows how to implement an AST visitor which prints the name of all defined functions in the "foo.rs" file (taken from the "syn" crate's documentation):

```rust
use syn::visit::{self, Visit};
use syn::{File, ItemFn};

struct FnVisitor;

impl<'ast> Visit<'ast> for FnVisitor {
    fn visit_item_fn(&mut self, node: &'ast ItemFn) {
        println!("Function with name={}", node.sig.ident);

        // Delegate to the default impl to visit any nested functions.
        visit::visit_item_fn(self, node);
    }
}

fn main() {
    let syntax_tree = syn::parse_file("foo.rs").unwrap();
    FnVisitor.visit_file(&syntax_tree);
}
```

Figure 4.1: Visit implementation

In order for the Geiger tool to perform the desired analysis of foreign functions, two custom visitors had to be implemented. The first visitor collected all the functions defined in `extern` blocks. Then, the second visitor visited function call sites and checked whether the site called a function defined through FFI. Finally, all the data was accumulated and output in JSON form.

There are two limitations for this approach however. Geiger uses a lot of memory to keep some cargo metadata structures. This, combined with the need to keep all of the external function definition information and calls leads to a high memory footprint. While the former can probably be optimised, the latter unfortunately cannot, because of the way Rust programs are written. There are a lot of external crates (such as libc, winapi etc.) which just contain `extern` blocks. Consequently, the callsite of the functions defined in these crates are in different files. Because of this separation, the function definition information has to be kept during the entirety of the analysis, as the function may be called from multiple different files.

Another limitation of this implementation is the lack of macro expansion. The "syn" crate was intended for use in procedural macros, therefore, it does not perform macro expansion when it constructs the AST. As a result, the analysis performed in this document does not include statistics from macros. However, this should not be an issue for the analysis, as using unsafe blocks in macros is rare in practice.

Chapter 5

Evaluation

The FooSystem tool was used to evaluate some of the most popular Rust crates, namely:

- base64
- byteorder
- bytes
- bzip2
- hyper
- regex
- tokio

5.1 base64

The "base64" crates has 1986 calls to externally defined functions, most of which are calls to WASM functions. The functions with most calls are shown in figure 5.1.

Out of these functions, 11% contain pointer arguments, with the most popular arguments listed in the figure 5.2.

Finally, the dependency graph of the external calls is shown in figure 5.3.

Name	Callsites
check()	297
f()	176
drop()	141
test()	73
load()	68
scope()	57
compile()	56
channel()	27
error()	26
parse()	24

Figure 5.1: base64: Most called functions

Type	Count
usize	26
&str	22
i32	18
&JsValue	17
size_t	17
u32	17
c_int	16
JsValue	16
Rules	13
&Symbol	13

Figure 5.2: base64: Most frequent arguments

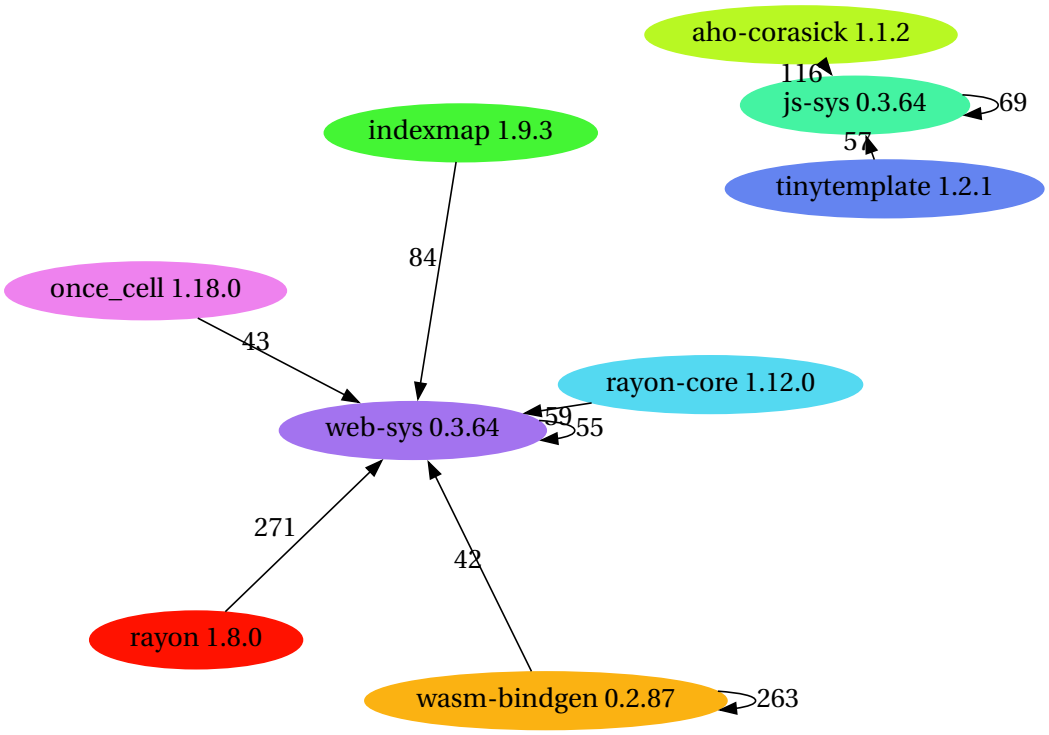

Figure 5.3: base64: Dependency graph

Name	Callsites
getrandom()	8
random()	4
___errno()	2

Figure 5.4: byteorder: Most called functions

5.2 byteorder

The "byteorder" crate uses a low number of externally defined functions, totalling to 45 calls. The most popular ones are `getrandom()`, `random()` and `___errno()`, shown in figure 5.4.

Out of these functions, 79% contain pointer arguments, with the most popular arguments listed in the figure 5.5.

Finally, the dependency graph of the external calls is shown in figure 5.6.

Type	Count
size_t	16
*mut u8	6
c_int	6
team_id	6
u32	6
usize	5
*mut i32	5
*mut c_void	3
*mut c_char	3

Figure 5.5: byteorder: Most frequent arguments

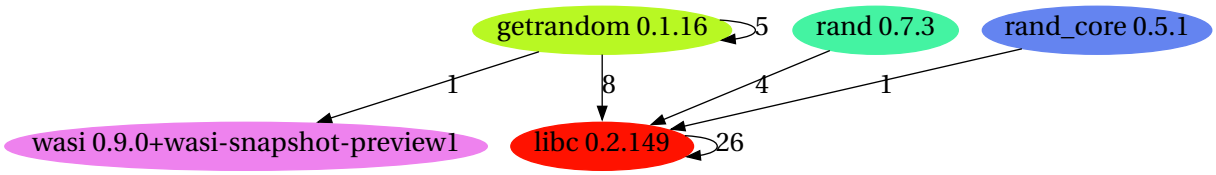

Figure 5.6: byteorder: Dependency graph

5.3 bytes

The "bytes" crate calls 6414 externally defined functions. The most popular ones are shown in figure 5.7.

Out of these functions, 28% contain pointer arguments, with the most popular arguments listed in the figure 5.8.

Name	Callsites
prefetch()	6
GetLastError()	5
GetErrorInfo()	4
memchr()	4
SysFreeString()	3
GetProcessHeap()	3
SysStringLen()	3
DPA_InsertPtr()	3
EncodePointer()	3
RegQueryValueExW()	3

Figure 5.7: bytes: Most called functions

Type	Count
DWORD	2797
LPCWSTR	1076
c_int	905
HANDLE	838
ULONG	776
LPDWORD	758
LPCSTR	717
UINT	573
HWND	462
LPWSTR	447

Figure 5.8: bytes: Most frequent arguments

Finally, the dependency graph of the external calls is shown in figure 5.9.

5.4 bzip2

The "bzip2" crate calls 320 externally defined functions. The most popular ones are shown in figure 5.10.

Out of these functions, 56% contain pointer arguments, with the most popular arguments listed in the figure 5.11.

Finally, the dependency graph of the external calls is shown in figure 5.12.

5.5 hyper

The "hyper" crate calls 726 externally defined functions. The most popular ones are shown in figure 5.13.

Out of these functions, 45% contain pointer arguments, with the most popular arguments listed in the figure 5.14.

Finally, the dependency graph of the external calls is shown in figure 5.15.

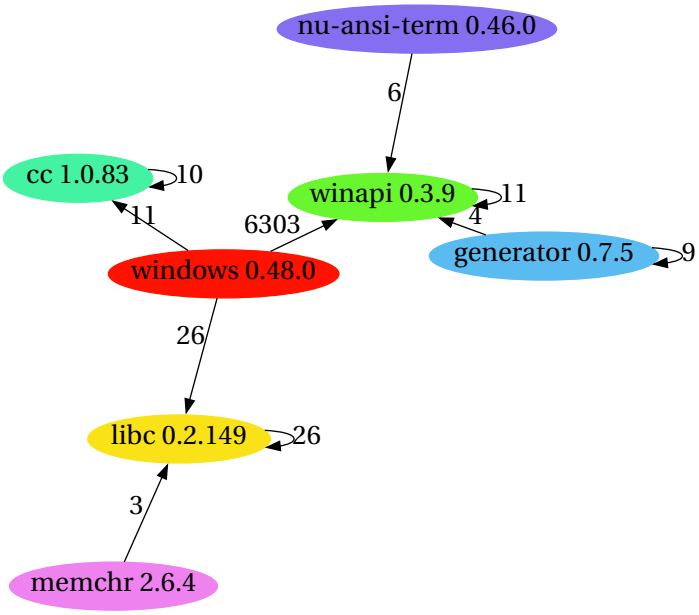

Figure 5.9: bytes: Dependency graph

Name	Callsites
spawn()	34
sleep()	10
write()	9
read()	7
shutdown()	6
abort()	5
getrandom()	5
GetProcAddress()	5
send()	4
random()	4

Figure 5.10: bzip2: Most called functions

Type	Count
usize	56
fd	30
DWORD	29
i32	24
HANDLE	20
size_t	17
LPDWORD	15
*const u8	14
u32	13
c_int	13

Figure 5.11: bzip2: Most frequent arguments

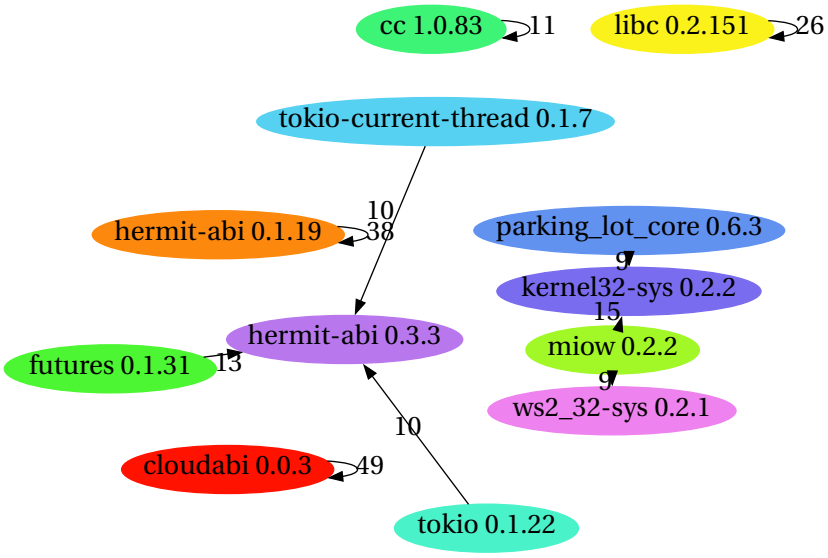

Figure 5.12: bzip2: Dependency graph

Name	Callsites
setsockopt()	142
getsockopt()	96
spawn()	91
connect()	49
write()	20
memset()	19
socketpair()	18
signal()	15
getauxval()	9
send_signal()	9

Figure 5.13: hyper: Most called functions

Type	Count
c_int	68
DWORD	38
HANDLE	35
size_t	27
c_int	17
*const c_char	10
c_uint	9
SOCKET	8
u32	7
*mut c_void	6

Figure 5.14: hyper: Most frequent arguments

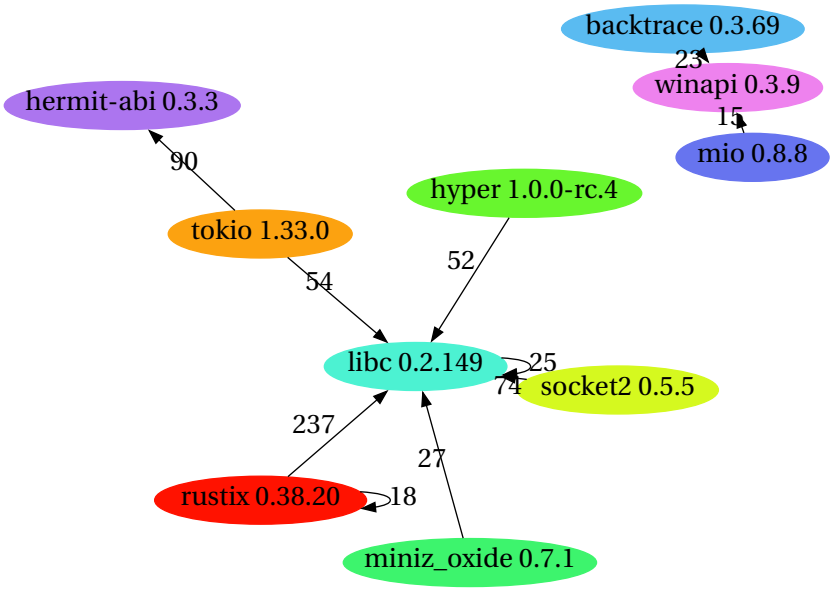

Figure 5.15: hyper: Dependency graph

Name	Callsites
memchr()	9
write()	8
random()	4
CloseHandle()	4
memrchr()	3
CreateFileMappingW()	3
getrandom()	3
VirtualProtect()	2
UnmapViewOfFile()	2
MapViewOfFile()	2

Figure 5.16: regex: Most called functions

Type	Count
usize	24
size_t	20
c_int	16
HANDLE	14
DOWRD	13
u32	10
i32	10
*const c_void	10
*mut u8	8
team_id	6

Figure 5.17: regex: Most frequent arguments

5.6 regex

The "regex" crate calls 144 externally defined functions. The most popular ones are shown in figure 5.16.

Out of these functions, 49% contain pointer arguments, with the most popular arguments listed in the figure 5.17.

Finally, the dependency graph of the external calls is shown in figure 5.18.

5.7 tokio

The "tokio" crate calls 12041 externally defined functions. The most popular ones are shown in figure 5.19.

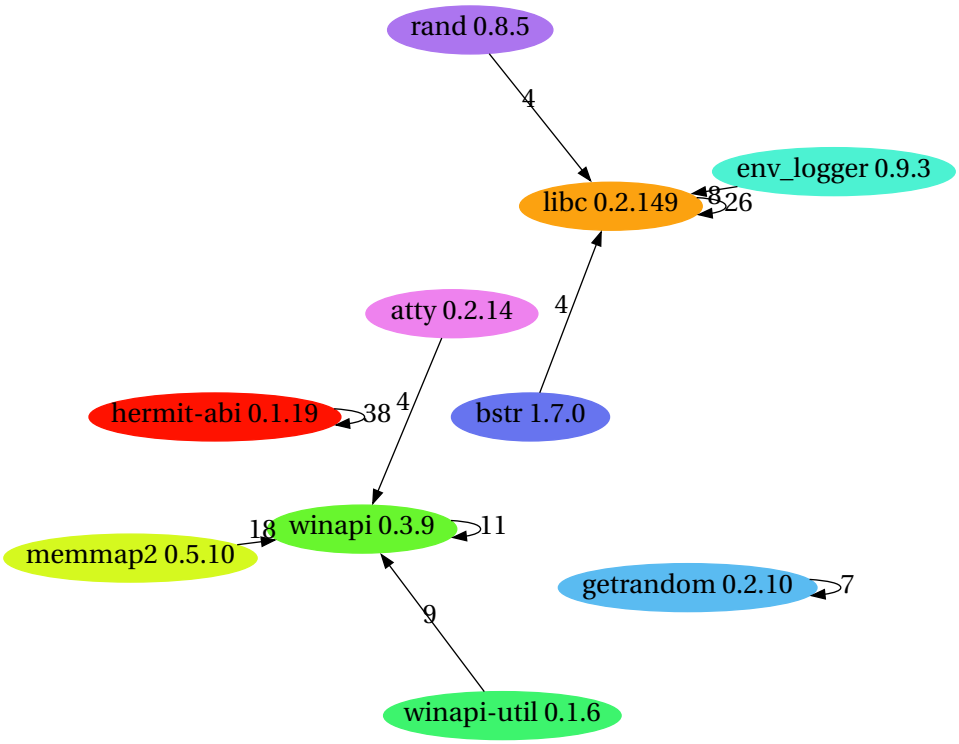

Figure 5.18: regex: Dependency graph

Name	Callsites
drop()	855
f()	476
check()	243
setsockopt()	224
close()	164
getsockopt()	133
spawn()	123
socket()	118
test()	98
channel()	89

Figure 5.19: tokio: Most called functions

Type	Count
DWORD	2797
LPCWSTR	1076
c_int	910
HANDLE	841
ULONG	778
LPDWORD	758
LPCSTR	717
UINT	573
HWND	462
LPWSTR	447

Figure 5.20: tokio: Most frequent arguments

Type	Count
*mut DWORD	190
*const GUID	184
*mut c_void	102
*const BYTE	67
*mut BYTE	65
*mut LPBYTE	53
*const RECT	50
*const c_char	47
*mut c_int	43
*mut ULONG	43

Figure 5.21: tokio: Most frequent pointer arguments

Out of these functions, 27% contain pointer arguments, with the most popular arguments listed in the figure 5.20.

The most popular pointer arguments are shown in figure 5.21.

Finally, the dependency graph of the external calls is shown in figure 5.12.

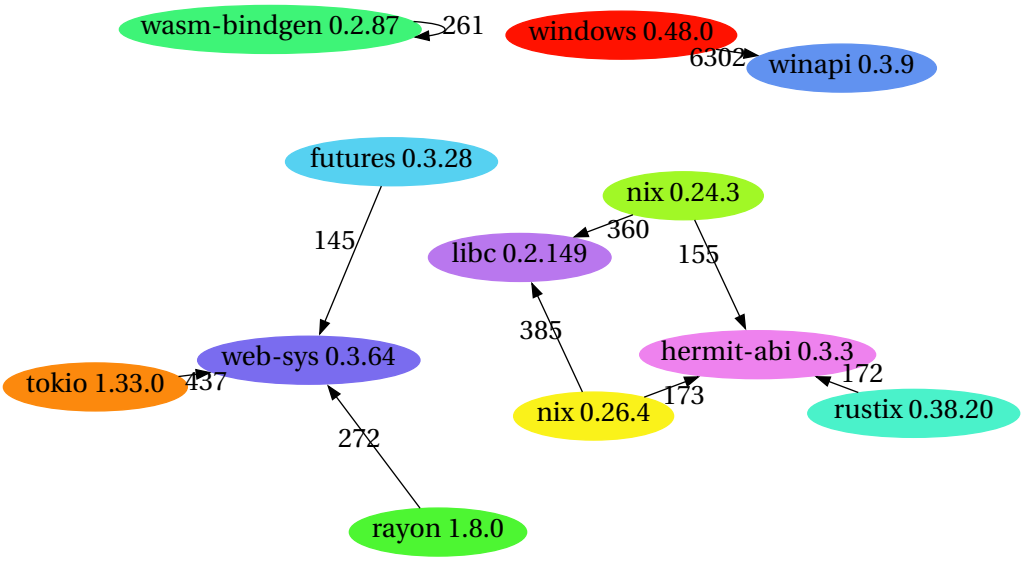

Figure 5.22: tokio: Dependency graph

Chapter 6

Related Work

There is a lot of ongoing research into unsafe Rust. Notably, the Stacked Borrows paper which is trying to precisely define Rust's memory model. The authors of the paper are trying to apply the same ownership and borrowing rules from safe Rust on unsafe code. The final product of this research is "miri" (Mid-level Intermediate Representation Interpreter). The goal of miri is to detect undefined behaviour in Rust programs during runtime.

Also, the Geiger tool was instrumental in developing the FooSystem tool, as it was a basis for the modifications required to run the desired analysis. Unfortunately, Geiger is very limited and provides only a surface level analysis of unsafe Rust code. It does not carry out analysis of externally defined functions, as these definitions are not unsafe. However, when calling such a function, one must wrap the call in an unsafe block. The FooSystem tool uses a more sophisticated analysis and is a noticeable improvement compared to Geiger.

Chapter 7

Conclusion

To conclude, the FooSystem tool enables analysis of calls to externally defined functions. The results of analysis suggest that all non-trivial Rust projects contain such functions. The larger the project, the more calls to these function are required. The byteorder crate contains around 3000 lines of code and only 45 calls to externally defined functions. On the other hand, the tokio crate, which contains around 150 000 lines of code contains 12 000 calls to FFI functions. Therefore, it is important to understand both how these function are called from Rust as well as their implementations in other languages (mostly C) in order to minimize introducing memory safety errors.

While Rust promises memory safety in theory, in practice, it is difficult to write programs only in Rust, therefore it is still possible to introduce memory safety errors even when all Rust code is correct.