Tooling to Detect Unwanted Thread Exits in Rust

D. A. van Cuilenborg
B. T. J. van Schaick
F. P. Stelmach
A. S. Zwaan

Tooling to Detect Unwanted Thread Exits in Rust

by

D. A. van Cuilenborg
B. T. J. van Schaick
F. P. Stelmach
A. S. Zwaan

in partial fulfillment of the requirements for the degree of

Bachelor of Science
in Computer Science & Engineering

at the Delft University of Technology,
to be presented on Monday July 2, 2018 at 12:00 AM.

Project Duration:	April 23, 2018 — July 2, 2018	
Thesis committee:	Dr. R. J. Krebbers,	TU Delft, Supervisor
	Ir. O. W. Visser,	TU Delft, Coordinator
	Ir. E. Gribnau,	Technolution, Client

Preface

With this report we conclude our bachelor project, and (most of us) will thereby receive our bachelor degree in Computer Science and Engineering at the University of Technology Delft. A total of 10 full-time weeks were spent at *Technolution*; a technology integrator that brings business, technology, and knowledge together. In this report we provide the reader with not just an explanation of the final product, but also the road that led us there.

We would like to thank the various people that supported us during those project. First of all, we would like to thank *Technolution* for the opportunity to carry out this project and providing us with an office, whiteboards and coffee for the duration of our project. Furthermore we would like to thank Erwin Gribnau, our direct supervisor at *Technolution*, who supported us with useful feedback, to stir the project into the right direction. Finally, we would like to thank Robbert Krebbers, for his continuous effort in providing us with thorough feedback and his excellent mentoring.

D. A. van Cuilenborg
B. T. J. van Schaick
F. P. Stelmach
A. S. Zwaan
Gouda, June 2018

Summary

Technolution is a company that specializes in building embedded and information systems, in which software plays a key role. Recently, *Technolution* is transitioning from the use of C in embedded systems, to Rust, a relatively new programming language developed by *Mozilla*. By design, Rust provides the programmer with higher security and reliability guarantees, such as memory safety, type safety and the absence of race conditions. These guarantees are ensured by means of an expressive ownership-based type system. However, it is impossible for the Rust type system to detect all errors statically. Hence, there are still many operations that contain dynamic checks to test for erroneous conditions. When such a check fails, an unrecoverable problem has been encountered and the current thread exits, this is called a *panic* in Rust. A *panic* causes the program to terminate, leading to a decrease in availability of the system. To avoid situations causing *panics*, *Technolution* wants tooling that detects possible ways a program could *panic*.

For this purpose, we developed a static analysis tool: *Rustig*. When given a program, *Rustig* notifies the user of all the operations that either directly, or indirectly via another library, may cause a *panic*. The tools performs the analysis of *panic* calls in two stages. First, it builds a call graph from the executable of a Rust program, modelling functions as nodes and function calls as directed edges. Secondly, it performs an analysis on the call graph to determine which functions could cause *panic*.

As part of the development of *Rustig*, we devised two new approaches. We have developed an approach to construct call graphs taking into account dynamic dispatch calls. This is based upon the assumption once a function address is loaded, it will also be called during execution. Furthermore, in order to efficiently analyze the call graph, a simplification of the all paths problem is proposed. In contrast with the all paths problem, the simplification is solvable in polynomial time. The approach involves finding the shortest path for every crossing edge on a graph cut.

Contents

1 Introduction

Code for low-level systems is usually written by the select few who are able to code using low-level languages, keeping in mind memory management, data representation and concurrency. Rust, a fairly new programming language by *Mozilla*, breaks these barriers by having a very strict compiler. This compiler ensures memory safety and data race freedom by means of an expressive ownership-based type system [1]. This allows for easier low-level systems programming, while the performance is on par with C, the lingua franca for low-level code. Companies are gradually learning about the potential of Rust and some of them are currently shifting towards using this new programming language. One of such companies is *Technolution*.

1.1. Technolution

Technolution creates information and embedded systems for their clients, realizing innovative ideas in the domains of transportation, defense and public security. These systems are subject to high security standards, to ensure confidential data is securely processed. Furthermore, the systems have to meet high availability and reliability standards, leaving no room for mistakes affecting its uptime. Moreover, fixing bugs after deployment is difficult and should be avoided if possible, since updating embedded systems can be very costly, because of the need for physical access to update the systems.

While Rust provides *Technolution* with most of the necessary safety guarantees, availability is not guaranteed, since execution may still fail at some point. This occurs when an unrecoverable problem is encountered, the program enters a state which the programmer did not account for [2], and terminates immediately. In Rust, this is called a *panic*. Despite the safety advantages of Rust, a panicking program could still be a great burden to the operations of *Technolution* due to the loss of availability. Therefore, they want to be able to analyze their programs to find execution paths in their code potentially leading to a *panic*.

1.2. Contribution

Our contribution to the Rust community is *Rustig*, a static analysis tool which provides the programmer with paths in their code that could potentially lead to a *panic*. This is done by first creating a call graph from a binary file and then analyzing the graph to find code paths which lead to a premature termination.

The significance of *Rustig* is that the programs written in Rust can be of higher quality, with less *panics*. This would help Rust programmers, and *Technolution*, since they would be able to achieve a higher availability and reliability for their software. Based on the output of *Rustig*, the programmer could refactor their program to be less prone to *panic* during runtime. To further help Rust programmers, *Rustig* allows them to ignore certain functions by means of whitelisting. This can be used to reduce false-positives or errors the programmer does not care about.

Our scientific contribution is twofold. First of all, a strategy was devised to find all relevant paths in a graph in a situation where finding all paths was too complex from a computational perspective. The proposed approach can be expressed in terms of a well-known graph algorithm, namely breadth-first search. The second contribution is that, to the extend of our knowledge, we are the first to use loaded function addresses instead of invoked function pointers to approximate dynamic dispatch in a static call graph.

1.3. Outline

This report presents the development of *Rustig*. In Chapter 2 the problem is described in greater detail. This problem is then further analyzed in Chapter 3. This software development project required various design choices, these are discussed in Chapter 4. A description of the workings behind *Rustig* is provided in Chapter 5. Next, The development process and the final product are evaluated in Chapter 6 and Chapter 7 respectively. In Chapter 8 the ethical implications of *Rustig* are discussed. During the development of the tool we ran into various issues, which are discussed in Chapter 9. Finally, conclusions and recommendations are provided in Chapter 10.

2

Problem Definition

High availability systems are characterized by their guarantees on operational performance, usually uptime; therefore, these systems should never crash. Furthermore, updating embedded system may not be feasible due to the need for physical access to the system. Patching the bugs would likely be postponed or not done at all, due to high costs of such operation. Therefore, programs written for such systems should be developed and checked carefully. Currently, the systems written in the Rust programming language cannot be properly analyzed for unwanted thread exits, since no analysis tools exist that are capable of performing such analysis.

Our goal is to develop such analysis tools. In order to do so, in this chapter, the definition of the problem is presented. The safety guarantees of the Rust programming language are first be described in Section 2.1. Afterwards, the handling of unwanted program state in Rust is explained in Section 2.2. This explanation consist of a description of idiomatic Rust error handling, analysis of *hidden panic* calls and clarification of the complexity of the problem.

2.1. Rust Safety Guarantees

Technolution uses various programming languages for the different solutions that they create for their clients. The company has also started to use the Rust programming language in situations where high performance, protection against exploits and robustness are crucial. Before Rust, C was usually used in these situations and so the Rust programming language is used as a replacement for the C programming language.

The advantages of Rust include, but are not limited to, memory and type safety. Many erroneous constructs that cause undefined behavior in C, are prevented by the Rust compiler. Common examples of such constructs in C are dangling pointers, which may lead to errors such as double free and use after free. Double free is the situation where a memory block is freed more than once, and use after free is the situation where some memory is used after being freed. Both errors cause undefined behavior and may cause vulnerabilities. Special tools are often necessary in order to find the before mentioned errors [3].

By design, Rust does not allow the existence of dangling pointers, due to its single ownership data model. In this model, data only has one owner, who can use the data freely. The data can be shared by a mutable reference `&mut T` or aliasing `&T`, but not both. Moreover, "The basic ownership discipline enforced by Rust's type system is a simple one: If ownership of an object (of type `T`) is shared between multiple aliases ("shared references" of type `&T`), then none of them can be used to directly mutate it" [4].

Even though Rust solves a lot of programming pitfalls originating from bad memory management, Rust programs can still end up in an unwanted state. Such unwanted states are characterized by the fact that they cannot be detected during compilation time.

2.2. Handling of Unwanted Program State in Rust

Many unwanted system states are handled by the Rust language through error propagation. Error propagation is often enforced by the Rust language by the use of *Result* type. An example of such propagation of errors can be seen in the standard library function responsible for opening of files. The signature of the function `std::io::open` is:

```
pub fn open<P: AsRef<Path>>(path: P) -> io::Result<File>
```

It is important to note that the return type is: `io::Result<File>`. The general **`Result`** type can either contain the wrapped value, or some error if the function has failed to execute properly, which in this example can happen if the given file does not exist. In the case of an error, the programmer may try to resolve the problem themselves and prevent the thread from crashing.

Unfortunately, not all functions in Rust use explicit error propagation. For example, array indexing will cause a runtime error when the index exceeds the size of the array. When such unwanted program state is reached, the **`panic!`** *macro* will be called directly, which in turn, will cause the thread to terminate immediately. In some cases, the programmer may be well aware of the fact that their code may *panic* when some specific program state is reached. They may, for example, call the *panic macro* themselves in order to crash the thread when some unwanted state is reached. However, if such *panic* call is made within some library code, the programmer will not have a chance to resolve the problem and prevent the thread from crashing. They will also not know whether the library code even contains calls to the *panic macro*, unless they analyze the libraries source code very well. In this report, calls to *panic* in the library code that the programmer is not informed of, are referred to as *hidden panic* calls.

2.2.1. Hidden Panic Calls

In some cases, the programmer may not be aware of the fact that some code that they use might *panic*, since Rust does not inform the programmer of specific occurrences of *panic* calls. One case where such *panic* calls occur, is the code from the Rust standard library used for thread spawning. An example using the thread spawn is shown in Listing 2.1. Notice that simply looking at the source code of the program in Listing 2.1 does not present us with any indication whether the code might *panic* or not. Therefore, we present a manual approach that can be used for finding out whether a function can *panic* or not.

Listing 2.1: Render an image in a separate thread

```
use std::thread;

let child = thread::spawn(move || {
    render_image()
});

let image = child.join();
```

A quick glance at the source code of the `std::thread::spawn` gives more clarity about what happens when the `std::thread::spawn` function is called.

Listing 2.2: Source code of `std::thread::spawn`[5]

```
#[stable(feature = "rust1", since = "1.0.0")]
pub fn spawn<F, T>(f: F) -> JoinHandle<T> where
    F: FnOnce() -> T, F: Send + 'static, T: Send + 'static
{
    Builder::new().spawn(f).unwrap()
}
```

Notice that the *unwrap*() function is called within the `std::thread::spawn` function. The *unwrap*() function looks as follows:

Listing 2.3: Source code of `std::result::unwrap`[6]

```
#[inline]
#[stable(feature = "rust1", since = "1.0.0")]
pub fn unwrap(self) -> T {
    match self {
        Ok(t) => t,
        Err(e) => unwrap_failed(
            "called `Result::unwrap()` on an `Err` value", e),
    }
}
```

Finally, the source code of `unwrap_failed()` looks as follows:

Listing 2.4: Source code of `std::result::unwrap_failed`[7]

```
#[inline(never)]
#[cold]
fn unwrap_failed<E: fmt::Debug>(msg: &str, error: E) -> ! {
    panic!("{}: {:?}", msg, error)
}
```

We have now shown that the thread spawn function contains a *hidden panic* call, which will cause the program to crash when the thread fails to spawn. The Rust documentation offers help on how to prevent the *panic* call. The description of `spawn()` tells us that it: "Panics if the OS fails to create a thread; use Builder::spawn to recover from such errors."[8]. Following this instruction, we obtain the non-panicking version of thread spawning, shown in Listing 2.5. Note that only one *panic* call has been prevented now. There are still other *panic* calls within this code snippet, but for the sake of simplicity, in this example, they are not addressed.

Listing 2.5: Non-panicking version of thread spawn

```
fn main() {
    let builder = thread::Builder::new();

    let child = builder.spawn(|| {
        render_image();
    });

    if child.is_err() {
        // Handle error case
    }
    else {
        // Since we know that child is not an error,
        // we can safely unwrap
        let image = child.unwrap().join();
    }
}
```

Noticeably, the **`panic!`** call may have gone unnoticed if the programmer that used thread spawning did not read the source code or the documentation of the `std::thread::spawn`. Moreover, if the *panic* call occurred in an external library instead of the Rust standard library, the programmer may not even be informed of the *panic* call through documentation.

2.2.2. Complexity due to Unreachable Code

The Rust language does not offer the functionality to automatically detect *panic* calls within the source code. One might expect that such functionality could be implemented by simply looking at `panic` occurrences in the source code. This is not the case, since some branches of the program may not be reachable at all. These branches can thus not cause a *panic* during the execution of the program and so they can be safely ignored. An example is presented to illustrate this. In Section 2.2.1 it has been shown that calls to `unwrap`() can *panic*, however, that is not always the case.

Listing 2.6: Correct use of unwrap

```
1 fn max_array_10(arr: [i64; 10]) -> i64 {
2     *arr.iter()
3         .max()
4         .unwrap()
5 }
```

Consider the case in Listing 2.6, where the function `max_array_10()` is a function that finds the maximal value of an array of 10 64-bit integers and returns it. In line 2, the iterator over the array is created. Afterwards, in line 3, the `max()` function of the iterator is called. Notice that the `max()` function has the following signature:

```
fn max(self) -> Option<Self::Item>
```

The type **`Self`**`::Item` is in the case of this example set to **`i64`**, since `arr` is an array that contains 10 values of type **`i64`**. The **`Option`** enum is similar to the **`Result`** enum. It wraps either some value or no value at all. Similarly to **`Result`**, `unwrap`() will *panic* if it has been called when the **`Option`** did not wrap any value.

It is important to note that the function `max()` only returns a **`None`** if the iterator is empty. It is clearly not the case in the function `max_array_10`, since the input array has a fixed length of 10. In this case, the **`Option`** returned by `max()` function can be unwrapped, as seen in line 4, and that **`Option`** will always contain some value. Therefore, no *panic* will occur due to the *unwrap* call.

In many cases, it is not possible to avoid writing code without any dead branches. Such branches are, however, irrelevant to the programmer, as they are never executed during the runtime of the program and so they will not cause an unwanted thread exit. Therefore, branches such as the *unwrap* call in Listing 2.6 should be ignored by the analysis tool, since if they are not ignored, the tool may produce inferior results that is bloated with false positives.

3 Problem Analysis

In this chapter the problem described in the previous chapter is analyzed and the various sub-problems are defined. First, the definition of the *panic macro*, and the detection of potential execution paths paths that lead to its execution are analyzed. As an intermediate result, it is concluded that a call graph is a good data structure to solve the problem (Section 3.1). Therefore, the problem is divided into two main steps: building the call graph (Section 3.2), and analyzing it (Section 3.3). For both sub-problems, several aspects of the problem are discussed, and a justification for several implementation choices is given.

3.1. Decomposing the Problem

As explained in Chapter 2, it is not easy to determine whether an arbitrary function call may lead to a **`panic!`** invocation. However, it is important to be able to detect these calls, because only then a warning to the programmers that these indirect invocations of **`panic!`** can be given. The first step to gain a better understanding of this problem is to analyze what code the *macro* expands to, so that it can be recognized at program representations below the source code level (More on different Rust program representations in Section 3.2). In Listing 3.1, the *macro* definition for Rust 1.26.1 is shown [9]. This definition shows that a call to the `panic` *macro* is translated into a function call to `core::panicking::panic()` or `core::panicking::panic_fmt()`. Therefore, the analysis can be done at source-code level by detecting calls to the `panic` *macro*, and at lower compilation levels by detecting calls to one of the aforementioned functions.

Listing 3.1: **`panic!`** macro definition

```
macro_rules! panic {
    () => (
        panic!("explicit panic")
    );
    ($msg:expr) => ({
        $crate::panicking::panic(&($msg, file!(), line!(),
            __rust_unstable_column!()))
    });
    ($msg:expr,) => (
        panic!($msg)
    );
    ($fmt:expr, $($arg:tt)+) => ({
        $crate::panicking::panic_fmt(format_args!($fmt, $($arg)*),
            &(file!(), line!(), __rust_unstable_column!()))
    });
}
```

Although being able to recognize direct calls to **`panic!`** is important, it does not solve the complete problem. A call to **`panic!`** may be in a library, which cannot be easily patched by the programmer. Furthermore, it may be possible that the function that invokes **`panic!`** is never executed. This can be due to the fact that compiler optimization has not taken place, or because the compiler was not able to prove that a certain branch is never executed. More information about the conditions under which **`panic!`** is invoked must be given, in order to judge if the invocation is a real threat to program robustness.

An effective representation of a program for the purpose of **`panic!`** call analysis is a call graph. This is a graph, where each node f represents a function or subroutine, and each directed edge (f, g) represents a call from subroutine f to subroutine g. Metadata can be stored on both the nodes and the edges. This makes a call graph a flexible, high level representation of a program that can be used for many different forms of program analysis [10–12].

When a program is represented as a call graph, it is possible to determine for an arbitrary subroutine (f) if it may (indirectly) lead to an invocation of **`panic!`** (p) by finding a path from f to p. More information about the cause and severity of the risk of that call path can be derived from the metadata. For example, we can define the metadata attribute *user_code* on the nodes (defining whether a subroutine is defined by the programmer, or imported from an 3rd party dependency). If there is an arbitrary node f that belongs to the user code in the graph, and an edge (f, p) exists (where p is a **`panic!`** node), we can conclude that the programmer used the `panic` *macro* directly. This example illustrates that we can use metadata $(user\_code)$ to retrieve more information about a *panic* call (In this case, the fact that it is a direct use of the *macro*).

Based on these results, it can be concluded that a call graph is an appropriate data structure to serve as a basis to analyze *panic* calls in a program. However, before being able to perform this analysis, a call graph must first be built from the program that is to be analyzed. The following two sections discuss the details of those two subproblems. Section 3.2 covers problems concerning the building of a call graph. Finally, some issues encountered when analyzing this call graph are discussed in Section 3.3.

3.2. Call Graph

In the previous section, it was explained that a call graph is a data structure that fits our purpose. In this section, several considerations about building that call graph are made. First, the question regarding which requirements the algorithm should satisfy are discussed. Secondly it is discussed whether a static or a dynamic call graph fits our problem best. Thirdly, the representation of the program that is used as input data is determined.

3.2.1. Callgraph Requirements

In an ideal situation, the algorithm should return a call graph that contains all edges that occur during runtime, but no more. This is called an exact call graph. However, determining the exact call graph is an undecidable problem [13]. Therefore, one of the following two options can be chosen. The first possibility is to have a potentially under-represented graph, for which it is certain that all edges can actually occur during runtime, but it is not certain that all actual invocations are represented. The other option is to build an over-represented graph, for which it is certain that all possible invocations are present, but it cannot be assured that all individual invocations will actually occur during program execution. Regarding the problem of finding execution paths to **`panic!`**, it is better to report too many errors, and let the programmer filter out the relevant ones. Therefore, the first requirement of our call graph is that all possible invocations are present.

Although it should be certain that no possible paths to *panic* are not discovered by *Rustig*, as few false positives as possible should be reported. Here, *Rustig* might benefit from compiler optimization. As illustrated in Section 2.2.2, dead branches are often unavoidable on the source code level. The Rust compiler performs dead code elimination in order to remove unreachable branches from the code so that they do not pollute the final executable. This elimination also includes unreachable *panic* paths. Therefore, it is desirable to be able to apply compiler optimization on the program before building the call graph.

The third requirement is given by the fact that Rust is a young and fast evolving language. New stable versions are released every 6 weeks, while the most progressive channel, nightly, is updated every night [14]. However, it is undesirable that the call graph algorithm needs to be adapted to every individual version of Rust, because maintenance would be too costly. Therefore, the third requirement to our call graph algorithm is that it should not depend on the exact Rust version of the target program.

Summarized, the requirements for the call graph building algorithm are as follows:

1. It should contain all possible edges (thus be over-represented)
2. It should represent an optimized program
3. It should work correctly independent of the Rust version

In the next sections, we will explore how to design an algorithm matching these requirements.

3.2.2. Analysis Strategy

The first choice that is explained concerns the approach to building the call graph. In general, there are two types of call graph builders. The first type are static call graph builders. These builders do not require a program to be executed, but instead build a call graph from the source of a program, or from an executable. This builder type is usually implemented in compilers, like *gcc* or *LLVM*, that use the graphs for optimization purposes [13]. For example, dead code elimination can be implemented using call graphs [15], by removing all functions that are not in the same connected component as the main function.

The second type of call graph builders are dynamic call graph builders, which require the program to be executed [16]. They usually record jump instructions, out of which a call graph is built. This call graph type is usually implemented in profilers, that record profiling information on the call graph. For example, *gprof* does record invocation count and runtime on an dynamic call graph [17].

For us, as we did explain in Section 3.2.1, it is important that all possible edges are found. Therefore, dynamic call graph builders are not suitable for our purpose, since they only record actually occurring calls. When a certain possible call is not hit during execution, it will not be present in the graph. In fact, it will almost never be the case that all branches are hit. Therefore, the requirement that the graph should be over-represented cannot be guaranteed to be met by dynamic call graphs.

When the dynamic approach is related to the problem that is solved (finding all paths to **`panic!`**), it can be seen that it is required to hit all *panic*s. If a **`panic!`** would not be hit, it would also not be present in the call graph. However, if it would be possible to device an algorithm that ensures for an arbitrary binary that all *panic*s are hit, it would not be necessary anymore to create a call graph, and find all **`panic!`** calls again, since they were already found by the call graph builder. Therefore, it can be concluded that doing *panic* trace analysis *based on dynamic call graphs* does not add any value.

In contrast, static call graphs, like those used by compilers, are generally over-represented. For an illustration, consider the example regarding the dead code analysis. If an edge would be missing, it can occur that a function will be eliminated while it was still invoked. This is something compilers generally would like to avoid. Therefore, static call graphs, and particularly those created by compilers, are useful for our purpose.

3.2.3. Input

A program that builds a static call graph generally takes a representation of the target program as input, and gives a call graph as output. This input representation can be in many different forms. In order to choose the correct form to take as input, the Rust compilation process and the intermediate output forms it can produce are analyzed in this section. After that, the possibilities of all these output forms are compared with the requirements stated in Section 3.2.1 and the best input format for our problem is chosen.

Compilation Process

Rust is a compiled language, meaning that a program written in Rust undergoes a compilation process before it can be executed on the target machine. During the compilation process, source code is translated into machine code. This translation is not a single-step procedure. Instead, the process contains many intermediate steps between the source-level representation and the machine code representation. An overview of the important steps of the translation process is shown in Figure 3.1.

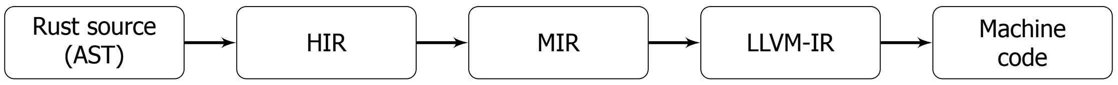

Figure 3.1: Rust compilation process overview [18]

Note that the overview in Figure 3.1 is not complete. More intermediate steps of compilation exist, but the additions that these steps offer are not relevant to our project, so they are not listed.

During the compilation process, the source code is first converted into an intermediate phase: *higher-level intermediate representation (HIR)*. HIR is the representation of a program, roughly equivalent to an abstract syntax tree. During this step, the source code is parsed and some constructs are expanded into more fundamental constructs [19], which is called desugaring.

Secondly, the translation from HIR to MIR, the *mid-level intermediate representation*, takes place. Desugaring has taken place in the steps before and so MIR is an intermediate representation that is in a much simpler form compared to Rust's actual syntax, since MIR consists of a relatively small amount of primitive operations. During the MIR phase, type checking is executed [20].

Thirdly, the transition from MIR to LLVM-IR, the low-level intermediate representation, occurs. During this transition the checks that provide the memory safety of Rust are executed [18]. Furthermore, some Rust-specific optimizations are performed. LLVM-IR is an intermediate language which is described by the LLVM Language reference manual [21].

Finally, the LLVM-IR code is translated into machine code. This is done using LLVM from The LLVM Compiler Infrastructure [22]. When compiled for the Linux platform, this binary is created in the *ELF* format.

Optimal Input Format

It turns out that all intermediate stages below HIR can be dumped from the compiler infrastructure, using the `--emit` flag on *rustc* (the Rust compiler). For the source code (*AST*) and HIR levels, Rust offers an unstable compiler plugin API. Therefore, the accessibility of the format is not a limitation a priori. For that reason, it is possible to create a callgraph from any of these representations.

On the other hand, not all representations fit the requirements from Section 3.2.1. The *AST*, HIR and MIR formats are unstable, and might have breaking changes every night. Furthermore, it is not possible to apply LLVM optimizations to these formats. Therefore, these formats do comply with our requirements of stability and optimization.

These objections do not apply to the LLVM-IR representation. This representation is stable, and can be optimized with the LLVM optimizer. Furthermore, LLVM has an option to generate a call graph. A possible command sequence to perform this is shown in Listing 3.2. However, the call graphs emitted by this procedure are not very accurate. As an example, a simplified example of the output of a hello world program is shown in Figure 3.2.

Listing 3.2: Callgraph creation command

```
rustc main.rs --emit=llvm-ir -o main.ll
opt main.ll -O3 | opt -dot-callgraph
```

When examining this graph in more detail, several things are remarkable. Firstly, there are 2 'virtual' nodes, External node and Sink. These account for all invocations to and from external symbols. The fact that LLVM uses these nodes implies that the actual invocation target is not known. Secondly, from an

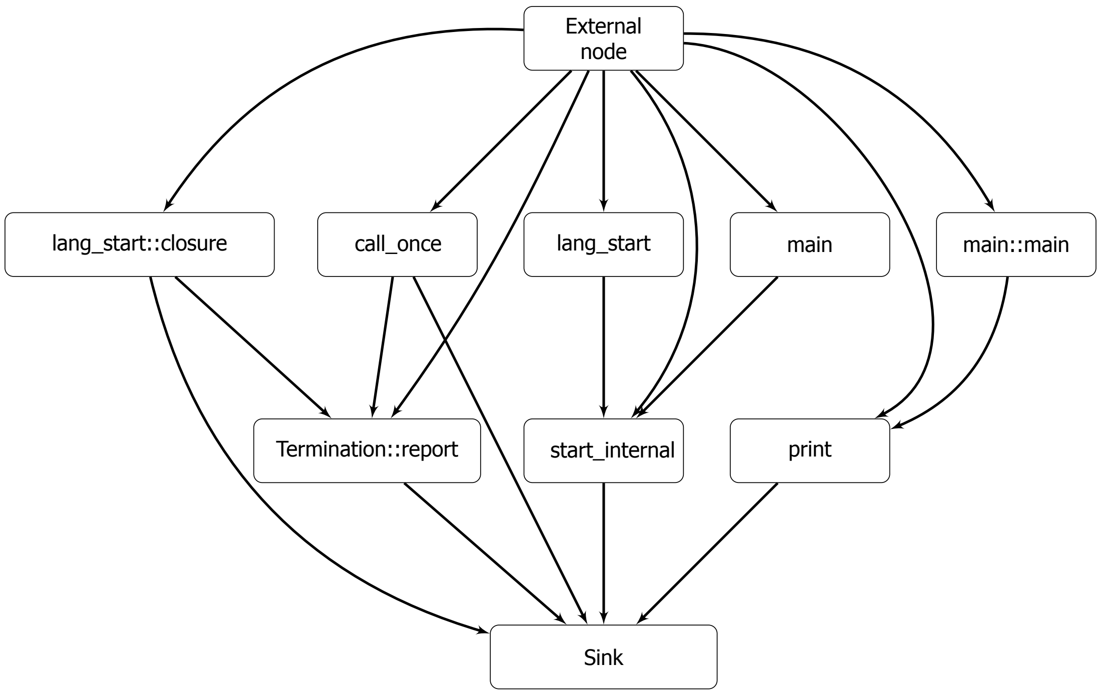

Figure 3.2: Example LLVM call graph

arbitrary Rust backtrace it can be derived that `main` calls `lang_start` and `lang_start::closure` calls `main::main`. All these invocations are not represented in the graph. Thirdly, it is remarkable that there is no node for the *panic* handler, while `print` can lead to a *panic*. Fourthly, there are no nodes for functions in external compilation units. This last problem could be solved by generating LLVM-IR code for all modules, use the LLVM linker to create one combined LLVM intermediate module, and generate a call graph from that. However, this would require duplication of the Rust build procedure, but instead with LLVM output. This process is very tedious and prone to error.

Based on these results, it can be concluded that the call graphs generated by LLVM do not match the requirements from Section 3.2.1. As shown in the previous paragraph, some edges are missing, which violate the requirement that the call graph should not be under-represented. Furthermore, when analyzing an LLVM-IR call graph for a single module, not all calls `panic` appear. Altogether, this makes the LLVM call graphs not suitable for our purpose.

Finally, the final compiled and linked binary can be used for the analysis. On the Linux platform, these binaries are in *ELF* format, with program metadata for debugging purposes in *DWARFv4* format, and x86_64 bytecode. All these formats are standardized, and may therefore be expected to remain stable for a considerable amount of time. Furthermore, Rust compiles all dependencies into one binary (there is no dynamic linking by default). This implies that all Rust functions that might be called are present in the binary. Moreover, it is possible to make an approximation of all function calls, which are further explained in Chapter 5. Finally, the final executable has been optimized by the compiler. An additional advantage of analyzing executables is the fact the source code is not required to be available. Therefore, we can conclude that the final binary is the most appropriate input format for our analysis.

3.3. Analysis

In the previous section, it was explained that it is possible to build an accurate call graph from a Rust binary program. In this section, several types of analysis that can be applied on such a call graph are explored. First, a simple approach to investigate whether a function calls **`panic!`** is described. Afterwards, it is explained how we can derive the last function in the trace that is written by the creator of the program. That is useful, since that is often the best point to fix such a situation. Then, some ideas on how to extract useful statistics from the call graph are discussed. Finally, a summary of the problem analysis, and possible solutions are provided.

3.3.1. Finding Panic Paths

After having build a call graph, the most important task is to identify traces that lead to a **`panic!`**. This can be implemented by first identifying nodes that represent *panic* handlers, and identifying entry points (nodes where we assume that the program starts executing). After determining these nodes, we can find paths between all pairs of entry points and *panic* handlers.

In an optimal situation, it would be desirable to find all possible paths. However, finding all possible paths between two nodes in a graph is exponentially bound to the number of nodes and edges in the graph. Therefore, reporting all possible paths to **`panic!`** is not feasible. In Chapter 5, we explain in detail the final algorithm we use to find the most relevant paths.

3.3.2. Reporting Last User-Defined Function

After having found a certain number of paths to **`panic!`**, it is useful for the programmer to know the last function in the trace that is in their own codebase, because that is most often the best place it fix the issue. Using *DWARF* debug info (more information on *DWARF* can be found in Appendix C), it is possible to determine the *crate* (Rust software package) the function was defined in. When the user can specify what *crates* are under their control, it is trivial to compare these *crate* names to the *crate* names if the functions in the trace, and report the last one. Moreover, using *DWARF* debug data, other useful information, like file and line number, can be given as well. This helps the programmer to find and fix issues more efficiently.

3.3.3. Statistics

Finally, it can useful to have a summarized report of the data. There can be many calls to **`panic!`**, especially in an industry-sized projects. Just reporting all these individually can overwhelm the user. In that case, *Rustig* does not help the programmer to achieve its goal of writing robust software. When the data is summarized properly, *Rustig* can aid in identifying host spots and commonly used patterns.

3.4. Conclusions

After having seen several aspects of the problem on how to find **`panic!`** calls in Rust projects, the following conclusions can be derived. Firstly, the program can be solved by representing a problem as an over-represented call graph. The call graph data structure is a flexible instrument to carry out **`panic!`** trace analysis, as well as deriving meta-information. Secondly, Rust binary files are the most appropriate input sources for creating a call graph, because they contain all necessary information, and their format is stable. Thirdly, it is not possible to report all possible paths to **`panic!`** due to the $\mathcal{NP}$-hard nature of the problem, so an algorithm to find the most useful paths needs to be designed. Fourthly, reporting meta-information and statistics is possible, and could be convenient for the user.

In summary, the problem seems to be quite well solvable, although some aspects are algorithmically complex. In the next chapter, the design of the product that was derived from this analysis is presented. In Chapter 5, the final implementation is discussed. In that chapter, we evaluate how the implementation reflects the results of the problem analysis as well.

4 Design

In this chapter the design of *Rustig* is made clear. In order to make a software engineering project run smoothly, the design of the tool should be defined. Various design goals should be agreed upon before implementing the tool. This allows for people new to the project to understand the underlying design and for the people already working on the project to have a goal to work towards. An important aspect of the design are the requirements which the tool should adhere to. These requirements are provided in Section 4.1. The design goals for this project are discussed in Section 4.2. Furthermore, various goals for test design are defined in Section 4.3. Finally, the architecture of the tool is provided in Section 4.4

4.1. Requirements

To have a clear picture of the goal of *Rustig*, various success criteria need to be defined. These criteria are defined by means of requirements, which were decided upon during the first week of the project and are presented and prioritized using the MoSCoW method [23]. Functional and non-functional requirements are discussed in Section 4.1.1 and Section 4.1.2 respectively.

4.1.1. Functional Requirements

This section provides the functional requirements as agreed upon with *Technolution*. These are the requirements of the functionality the final product should contain.

Must have

- The tool must operate from the command line.
- The tool must internally build a call graph from a Rust *ELF* binary.

Should have

- The tool should be able to detect, whether code paths exist that may lead to a panic call (which may reside in an external *crate*)
- The tool should print errors if the previously mentioned thread exits occur to the standard output. The format if the output will be:
  ```
  ”Method ’<method name> in <file name>:<line number>’
      calls ’<crate>::<method name>’”.
  ```
  Here the first mentioned method name will be the last method in backtrace which is user code, while the second mentioned is the first method in an external library that is called.
- The tool should have exit code 0 if no errors were found.
- The tool should have exit code 1 if unwanted thread exits were found
- The tool should have exit code 101 if an internal error occurred.

Could have

- The tool could be configurable to ignore particular patterns.
- It could be possible to ignore single warnings by placing a comment in the code.
- The tool could propose suggestions to fix found problems.
- The tool could support CI-server (Jenkins) compatible output.
- The tool could support XML as output format.
- The tool could support JSON as output format.
- The tool could support HTML as output format.
- The tool could be able to print the call graph in DOT as output format.
- The tool could have a verbose output mode, where full backtraces of unwanted thread exits are printed.

Won't have

- The tool will not be a compiler plugin
- The tool will not fix found issues in the source code itself.
- The tool will not detect unwanted thread exits originating from unsafe Rust code.

4.1.2. Non-Functional Requirements

This section defines requirements which is not part of the expected functionality of our tool, but rather define the way of operating and the design constraints.

Must have

- The tool must be executable within a Linux environment.
- The tool's source code should have no compilation errors under standard compilation conditions.

Should have

- The tool should be written in the Rust programming language.
- The tool's source code should have each public, non-trivial code element (method, struct, trait, etc.) documented. Documentation should indicate the purpose of the code element, and particularities or non-trivial properties of the implementation.
- The tool should have a short user guide, explaining all the implemented options, the output, and configuration as part of its source code.
- The tool should have a help-option, that prints a small guide on how to invoke the tool.
- The tool should have no compilation warnings.
- The source code should adhere to the Rust Style Guide [24]
- For each function, all usage patterns that can be reasonably expected should have a test case. For these test cases, it should be documented what use case is tested.

Could have

- The tool could be able to run on a Windows environment, so that developers who want to work on Windows, and cross-compile their code, can still use the tool.
- The tool could support the ARM architecture.
- The tool could be supplied with an architecture design, that explains which academic principles and structures were used in developing the tool.

Won't have

- The tool will not be supported by the original development team after project termination.

4.2. Design Goals

In this section various goals which *Rustig* should adhere to are described. These goals are to ensure that the final product is of high quality. In Section 7.1 we reflect on these goals and see how our tool managed to accomplish them.

Extensibility

It should be easy to extend *Rustig* with a new feature, without requiring major changes to the rest of the program architecture to ensure that the tool will be able to grow and evolve. For example, new ways of analyzing call graphs or the ability to add support for more computer architectures should be a straightforward process.

Modularity

Rustig should be divided into various modules, this would allow for easier testing of these independent components before adding them to the pipeline. Furthermore, it allows the user to easily swap out modules for their own implementations if they would like a different kind of functionality. Furthermore, it should also be possible to use the modules by themselves. For example, it could be useful to create a call graph of a program without using any of the modules for analysis.

Usability

It should be clear how to use *Rustig*. While the tool will be a simple command-line interface, there should be no confusion on how to invoke the tool. Therefore a clear help menu is needed to tell the user how to use the tool. Furthermore, sensible defaults should be chosen when a user fails to provide optional input parameters.

Scalability

The performance of *Rustig* should adapt well to increasing amounts of data. Programs with a lot of external dependencies should not be an issue for the tool and it should finish within reasonable time. The goal of *Rustig* is to constantly help the programmer and should therefore not take more than 2 minutes to create a report of a 100MB *ELF* file.

4.3. Test Design

One of the necessary steps to ensure that a program works correctly is the testing of that program. It was decided that for this tool, the combination of unit tests, integration tests and regression tests would be necessary to ensure the quality of the program. The coverage of these tests should be summarized in the code coverage report. Finally, manual performance tests are required to ensure that *Rustig* is able to perform analysis on realistic, and possibly large scale, projects as well.

4.3.1. Unit Tests

Unit testing is an important measure that programmers can take in order to ensure that the program building blocks are implemented properly. The Rust language offers native support for unit testing and so no external frameworks are necessary. Furthermore, these unit tests should be testing one isolated module at a time, in accord with the rust guidelines: "Unit tests are small and more focused, testing one module in isolation at a time, and can test private interfaces."[25] The aim of this project is to have most, if not all, public methods covered by unit tests.

4.3.2. Integration Tests

Whereas unit tests focus on isolated modules, integration tests ensure that multiple modules interact correctly with each other. The Rust language offers native support for integration tests as well, and so, the idiomatic way of implementing integration tests in Rust will be used. Noticeably, the Rust language does not differentiate between integration tests and systems tests. Therefore, systems tests will be implemented as integration tests.

4.3.3. Regression Tests
For most programs, a new compiler version of *rustc* would mean that the programs functionality may break. Regression tests should ensure that the developed software works with newly released *rustc* versions. Since *Rustig* relies on the analysis of *ELF* binaries, it is necessary to ensure that *ELF* binaries generated by new *rustc* version are still properly analyzed. Regression tests should thus not only ensure that the tool still functions with the new *rustc* version, but they should also ensure that programs generated with the new compiler can still be analyzed. This means that the test suite must be able to create binaries with the current *rustc*, which can then be used as resources for the tests.

4.3.4. Code Coverage
Technolution does not yet provide any specific coverage requirements for their software written in Rust, since the coverage tooling is not mature enough. Even though *Rustig* will be written in Rust, and so it will have inherent memory and type safety, some measurable data would be useful to ensure the quality of the program. For this goal, one of the two *crates*: *cargo-kcov* or *Tarpaulin* will be used in order to calculate the code coverage of the previously mentioned test methods.

4.3.5. Performance Tests
In order to make sure that *Technolution* is able to use *Rustig* to analyze their existing programs, the tool will be manually ran on already existing projects in order to ensure that time and space requirements are in order.

Firstly, *Rustig* will be tested on programs made by *Technolution*. It should be possible to run the analysis tool on all of the currently existing programs made by *Technolution* with the Rust programming language. Secondly, *Rustig* should be able to analyze itself. Finally, *Rustig* will be used to analyze a large project, namely The Servo Browser Engine. This project has been chosen as it is one of the largest open source projects available that are written in Rust. The source code of the browser contains 300 thousand lines of Rust code and the executable is 1GB. The assumption is that if such large program can be analyzed with *Rustig*, then the tool is suited to analyze most of Rust programs.

4.4. Architecture
After having established the design goals and some generic choices, we established a more concrete and complete technical design of our project. That design is presented in this section. First, in Section 4.4.1, the different components of the project are discussed. After that, Section 4.4.2 explains in more detail how these components interact.

4.4.1. Component Design
In this subsection, it is explained how the project is split up into different autonomous components. A UML component diagram of the design is presented in Figure 4.1.

As explained in Chapter 3, the problem is solved in two different steps: First, a call graph is build. Subsequently, the call graph is analyzed, and traces to `panic!` are extracted and reported. This separation is translated into two different modules (*Callgraph Builder* and *Panic Analyzer* respectively), performing these respective tasks. We choose to implement both modules as static libraries, so it is possible for other projects to use our output as well.

The output of these libraries is consumed by two executable wrappers: *CLI* and *Cargo plugin*. The difference between those is that the former can be executed on any Rust binary, while the latter can only be executed in a *Cargo* workspace. The advantage of the *Cargo* plugin is that certain parameters (like the binary to analyze) can be determined automatically. It will therefore require less command line parameters, which is more convenient for the user.

In order to share the output functionality, a separate library for printing *panic* traces (*Panic Output*) is created. This component depends on *Panic Analyzer* for the data definition of the *panic* call traces, and offers an interface to print collections of these traces. It is used by *CLI* and *Cargo plugin* component to print the traces that are returned from the *Panic Analyzer* component.

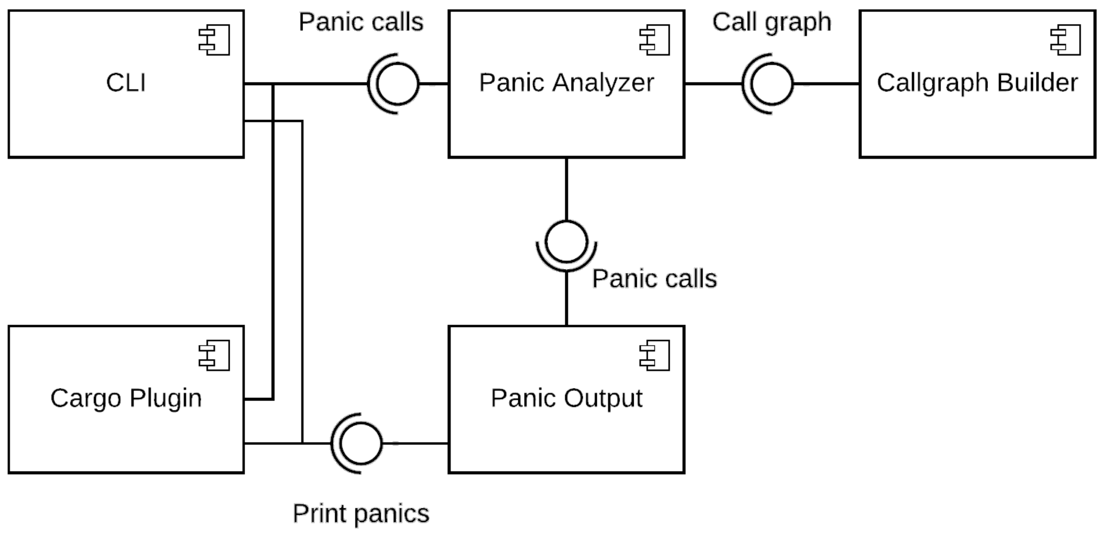

Figure 4.1: Component diagram

4.4.2. Control Flow

In the previous section, the static structure of the project was explained. In this section, the way *Rustig* behaves dynamically is explored. Based on the sequence diagram in Figure 4.2, the steps *Rustig* runs through, and the data that is passed are discussed. Note that this sequence diagram is simplified. The objects represent our components, where all self calls are usually implemented as calls to submodules in the same component. For reasons of brevity, these calls to submodules are not depicted explicitly.

It can be seen that the *CLI* handles over control to *Panic Analyzer*. Then, the *Panic Analyzer* component ensures the target binary exists, and requests the *Callgraph Builder* module to build a call graph. The *Callgraph Builder* module parses the file and builds a call graph, which is returned back to the *Panic Analyzer*. The analyzer subsequently calculates some metadata attributes on the call graph nodes and edges, and filters out irrelevant ones. After that, the actual *panic* traces are found, and some metadata for them is calculated. This result is passed back to the *CLI* component, which in turn uses the *Panic Output* component to format the output, and print it to the desired output destination.

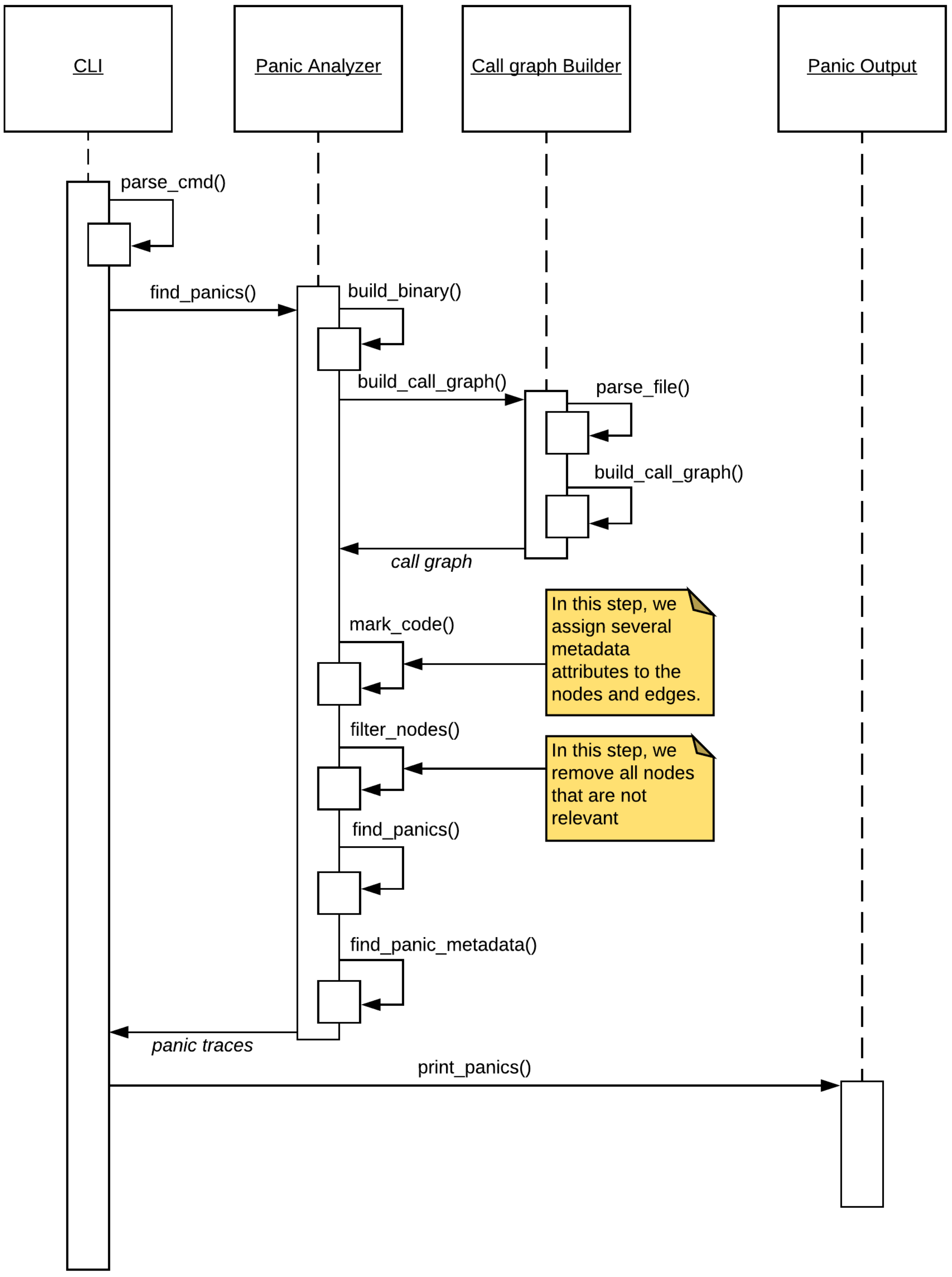

Figure 4.2: Component diagram

5 Implementations

In this chapter, we provide details on the implementation of the various modules of the final tool. First, the pipeline of *Rustig* is discussed in Section 5.1. In the subsequent sections the individual components of the pipeline are explained. Finally, the framework which was built to allow for integration and regression testing is described in Section 5.7.

5.1. Pipeline

In this section, we give a broad outline of the structure of *Rustig*. *Rustig* performs its analysis in a number of sequential steps. These steps are usually implemented in separate modules. An overview of this pipeline is provided in Table 5.1, displaying the steps in consecutive order, with a brief explanation of each step. In Section 5.2 to Section 5.6, most of these steps are explained in detail. However, not all steps are described explicitly, this is due to the fact that these steps are trivial or are handled by external *crates*.

Table 5.1: Tool pipeline

Step	**Component**	**Module**	**Description**	**Section**
Build binary	*Panic Analysis*	*binary*	Ensures the target binary exists.	-
Read binary	*Callgraph Builder*	*binary_read*	Reads the target binary into a byte vector.	-
Parse binary	*Callgraph Builder*	*parse*	Parses the file content to usable formats, like *ELF* and *DWARF* information.	-
Build call graph	*Callgraph Builder*	*callgraph*	Creates a static call graph from the binary.	Section 5.2
Marking	*Panic Analysis*	*marker*	Sets various metadata attributes on the nodes and edges of the call graph.	Section 5.3.1
Filtering	*Panic Analysis*	*filter*	Removes redundant nodes and edges.	Section 5.3.2
Find *panic* traces	*Panic Analysis*	*panic_calls*	Builds traces for all *panic* calls in the graph.	Section 5.4
Find patterns	*Panic Analysis*	*patterns*	Assigns patterns to the *panic* traces.	Section 5.5
Print output	*Panic Output*	*panic_calls_output*	Prints the *panic* traces to the standard output.	Section 5.6

5.2. Call Graph

The first step in our pipeline is building a call graph from the input binary. As explained in Section 3.2, the call graph is an important asset in the analysis. In Section 5.2.1 the way call graph nodes are built is explained. After that, it is explained how static and dynamic invocations are found in section Section 5.2.2 and section Section 5.2.3 respectively.

For the graph data structure, the *petgraph crate* is used [26]. This *crate* implements a generic `Graph` structure, which allows for custom node and edge weights. We use custom defined structs to store information of the `Procedures` and `Invocations` of the nodes and edges, respectively. This is summarized in Figure 5.1.

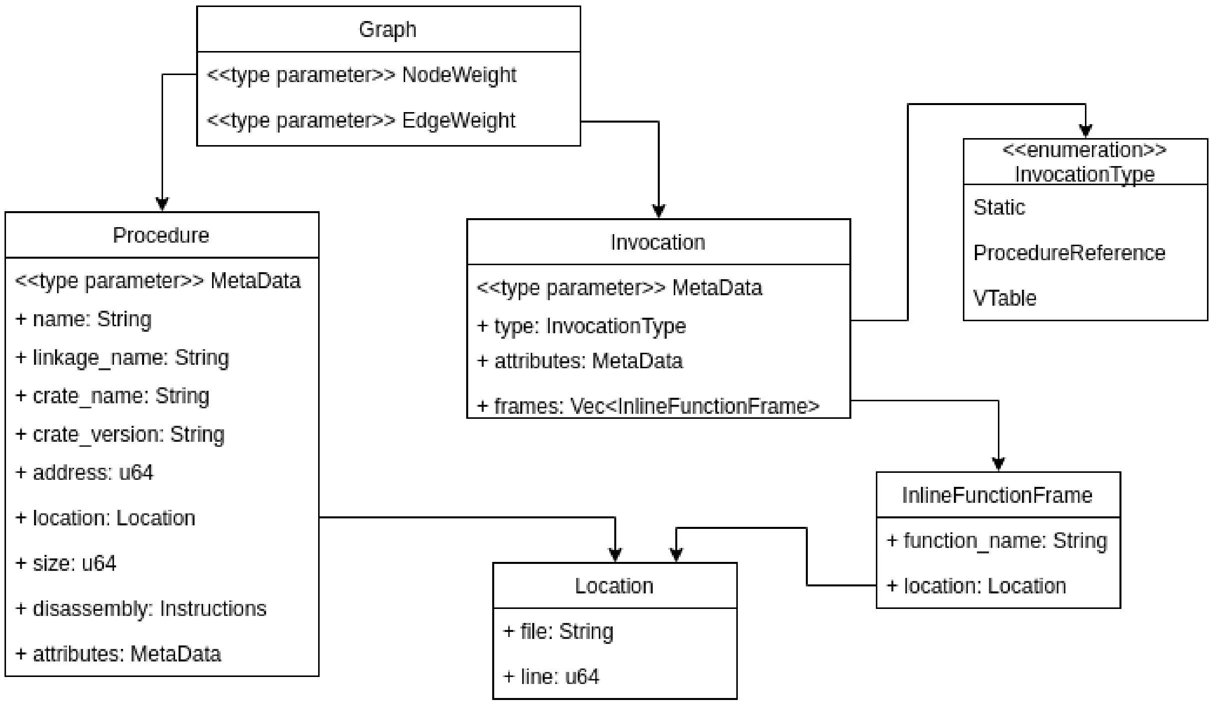

Figure 5.1: Call graph data structure

5.2.1. Call Graph Nodes

The first step of building a call graph is to insert nodes for all procedures. In order to find all these functions, all the compilation units in the *DWARF* information (see Appendix C for more information in *DWARF*) are iterated. When the compilation unit was written in the Rust language (which can be determined by the `DW_AT_LANG` attribute), all subprogram entries are iterated. For all subprograms, a node in the graph is created. All information on the procedure (see Figure 5.1, struct `Procedure`) can be derived from the debug attributes.

There are four important peculiarities to take notice of. Firstly, sometimes a function is duplicated. In that case, multiple *DWARF* entries are made as well. However, not all information is duplicated on these entries. Instead, an 'abstract origin' entry is made, containing all information. The other entries have a `DW_AT_abstract_origin` attribute, that points to the origin entry. Our algorithm creates a distinct nodes for all these duplicated procedures, but it also follows these indirections, to determine the attribute values for all these duplications.

The next relevant implementation detail considers the `disassembly` field. We use the `address` (`DW_AT_low_pc` attribute) and procedure **`size`** (`DW_AT_high_pc` attribute) to determine the machine code bytes corresponding to this procedure. We use the *crate capstone* to disassemble the machine code, and store the disassembled code on the `Procedure` [27].

Another relevant aspect of the node generation is the way the algorithm deals with inlining. Since *DWARF* entries for inlined functions do not include the necessary metadata, no nodes are generated for inlined functions. Instead, we maintain a list of inlined functions on the edges of the call graph.

For example, assume there are three functions, `foo()`, `bar()` and `baz()`. `foo()` invokes `bar()`, and `bar()` invokes `baz()`. When the compiler inlines `bar()` into `foo()`, the call graph builder will create nodes for `foo()` and `baz()`, and an edge between them. On that edge, an `InlineFunctionFrame` for `bar()` is stored. This is illustrated in Figure 5.2.

The last aspect of the node generation that is explained is the address index. In the call graph, we can easily find an address for a given procedure. However, it turned out that it is often useful to find the procedure for an given address. Therefore we maintain an index (implemented as **`HashMap`** that stores the `Procedure` for a given address. This enables $O(1)$ lookup of a procedure by address.

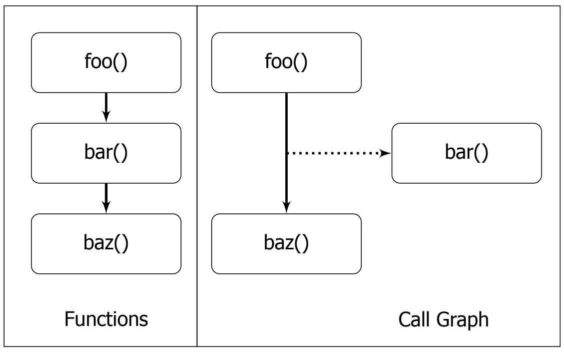

Figure 5.2: Example call graph

Another way to find all the procedure nodes could be by iterating the symbol table, instead of iterating *DWARF* information. This approach is more robust, since it does not require debug information to be available. We chose not to implement this approach, since we ultimately need the *DWARF* info anyway to determine the *crate* details, and the file location.

5.2.2. Static Invocations

Now that nodes are generated for all procedures, edges for function calls can be inserted. Static invocations consider the cases where the called function is known at compile time. In these cases, call instructions with a hard-coded address are created. A simple example of a static call, and the corresponding assembly code, can be seen in Listing 5.1 and Listing 5.2, respectively.

Listing 5.1: Code with static call

```
1 fn main() {
2     foo();
3 }
4
5 fn foo() {
6     println!(”Hello world!”);
7 }
```

Listing 5.2: Assembly with static call

```
0000000000007490 <static::main>:
    7490:       50                      push   %rax
    7491:       e8 0a 00 00 00          call   0xa(%rip) <static::foo>
    7496:       58                      pop    %rax
    7497:       c3                      ret
00000000000074a0 <static::foo>:
...
```

In this example, it can be seen that the function call in line 2 of Listing 5.1 translates to a call to an address that is relative to `rip`. The absolute address can be calculated by adding the offset to the address of the *next* instruction. In this case, this is $0x7496 + 0xa = 0x74a0$. It can also occur that the generated call instruction already has an absolute address ($0x74a0$) as argument.

In order to add edges for all calls that follow this pattern, the graph builder iterates over the instructions of all procedures. If an instruction is a **call** instruction, and the argument is either absolute or relative to rip, an edge from the function in which the call instruction is defined to the procedure at the argument address is made. If there is no node for the callee, a call to a non-Rust function is assumed, and therefore the edge is ignored.

5.2.3. Dynamic Invocations

In the previous section, it was discussed how edges for calls, with known destinations at compile time, are added to the call graph. However, it is not the case that for any function call, the called function is known at compile time. For example, if a function pointer to function f is passed as parameter to another function g, and g invokes that function f, it cannot be known at compile time in the context of g what function is pointed to by f.

A similar situation occurs when a function h has a formal parameter with a trait type. A trait is a Rust construct that defines an interface, without defining an implementation. Traits can be implemented by concrete data types. They are comparable to interfaces in object-oriented languages. When a function parameter with a trait type is defined, it is unknown which struct implementing that trait will be passed to it. Still, Rust allows to call functions defined on that trait, since that is what traits are actually useful for. Similar to the previously mentioned function pointer example, the actual implementation of the trait function is unknown at compile time.

Since call target information is not known at compile time, it can be concluded that it is passed at runtime. We explore how this is done using a simple example (see Listing 5.3, and the corresponding disassembly snippet in Listing 5.4) with a trait invocation.

Listing 5.3: Code with dynamic call

```
trait T {
    fn f(&self);
}

struct S;

impl T for S {
    fn f(&self) { println!("Hello from T implementation on S!") }
}

fn main() {
    let s: &T = &S;
    foo(s);
}

fn foo(t: &T) {
    t.f()
}
```

Listing 5.4: Assembly with dynamic call

```
1  26d350 40740000 00000000 00000000 00000000
2  26d360 01000000 00000000 d0740000 00000000
3
4  00000000000074d0 <<dynamic::S as dynamic::T>::f>:
5      ...
6
7  0000000000007520 <dynamic::main>:
8      7520:   push   %rax
9      7521:   lea    0x4f180(%rip),%rdi         # 566a8
10     7528:   lea    0x265e21(%rip),%rsi        # 26d350
11     ...
12     7535:   call   7540 <dynamic::foo>
13     ...
14
15 0000000000007540 <dynamic::foo>:
16     7540:   push   %rax
17     7541:   call   *0x18(%rsi)
18     7544:   pop    %rax
19     7545:   ret
```

It is noticeable that the destination of the **call** instruction in line 17 cannot be exactly determined, since it is determined by the memory value `rsi` is pointing to. In this case, we can determine that value, by looking at the **lea** instruction in line 10. It can be seen that, in line 17, `rsi` will have value $0x26d350$. Therefore, `0x18(%rsi)` will resolve to $0x26d368$, and dereferencing that with `*` resolves to $0x74d0$, as can be seen in the second half of line 2.

In order to understand what is going on here, we need to know how Rust passes pointers to objects having a trait type. Internally, Rust passes 2 pointers to `foo`, a pointer to the data of `s`, and a pointer to the `vtable` of `s`. This vtable is a construct that contains data about the implementation of the trait. The first field contains a pointer to the destructor of the object that is passed. After that, the size and alignment of the data are given. Finally, for each function defined on the trait, a pointer to its implementation is given. The Rust compiler generates such vtables as read-only data for any trait implementation.

For the implementation of trait `T` for struct `S`, the generated vtable object is displayed in Figure 5.3. It can be seen that the `f` field of the vtable refers to the implementation of `T::f` for `S`. This object is derived from the first 2 lines of Listing 5.4. There we see the `destructor` field (at address $0x26d350$) has value $0x7440$ (`40740000 00000000` corrected for endianness), the **size** field has value 0, because `S` has no fields, the `align` field has value 1, and the `f` field has value $0x74d0$. This is the address of the implementation of `T::f` for `S`, as can be seen in line 4.

vtable_S : VTable_T

destructor: *fn(&self) = &S::drop_in_place
size: usize = 0
align: usize = 1
f: *fn(&self) = &S::f

Figure 5.3: Example call graph

For this simple example, the call target (`<S` **as** `T>::f`) can ultimately be detected manually. However, function references can be arbitrarily nested in other data structures, or passed in other ways. Therefore, by just examining the **call** instructions, it cannot be guaranteed that all possible function calls can be determined. Therefore, in order to approximate these dynamic calls, *Rustig* does not look at **call** instructions, but rather at `lea` instructions. Although the called function reference cannot be found programmatically in general, we know that for a function to be called dynamically, its address, or the corresponding vtable address should be loaded. To the extend of our knowledge, this loading always occurs by means of **`lea`** instructions. Accordingly, *Rustig* analyzes **`lea`** instruction to find possible dynamic invocations.

The algorithm that creates edges for these dynamic invocations is implemented as follows. First the instructions of all procedures are iterated. For all **`lea`** instructions, the algorithm tries to determine the argument value. If that value is an address of a procedure, an edge between the procedure the instruction is defined in, and the procedure that is pointed to is added. If the loaded address is the address of a vtable, edges from the procedure, where the **`lea`** instruction is defined in, to all functions in the vtable are added.

While this approach guarantees that all dynamic invocations are found, there are some repercussions. First of all, we assume that all loaded functions *are* actually invoked, which of course is not necessarily true. If one of the created edges leads to a *panic*, but is never invoked, false positives may occur. Secondly, the edges created in this way have the procedure in which the **`lea`** instruction was defined as caller. In fact this function will almost never be the actual caller of the loaded function. However, for the purpose of finding paths to *panic* this is only a minor problem. Although the given path might be incorrect, the fact that there exists a path in this call graph implies that a path in the binary exists as well and could thus lead to a *panic*.

5.3. Markers and Filters

During the analysis of the call graph, two operations that are used extensively are the marking operation and the filtering operation. Marking nodes in the call graph is used to put metadata on some particular nodes into the graph, whereas filtering has been used to remove nodes which were deemed unnecessary. In this section, the marking operation and uses thereof is first be explained in Section 5.3.1. Afterwards, the filter operation, together with its uses, particularly whitelisting, is be discussed in Section 5.3.2

5.3.1. Marking Operation

The marking of particular nodes is used extensively in *Rustig*. Marking simply means putting metadata on some particular node in the call graph. This information would be put in the `attributes` field of procedures that are contained in the call graph (see: Figure 5.1). Four different attributes of the nodes are marked, they are briefly discussed.

Internal Nodes

Throughout the rest of this section, a distinction between types of *crates* is made. Two terms are introduced for this purpose. External *crates* refers to *crates* which are not part of the codebase that the programmer is analyzing, and external functions are functions that are a part of such external *crate*. One of the characteristics of external *crates* is that the programmer cannot modify the code in these *crates* directly, unless they fork those *crates* or submit a patch to their repository. On the other hand, internal *crates* are *crates* which the programmer can modify directly. Internal functions are functions that belong to the internal *crate*.

The distinction between marking a node internal or external is made based on the *crate* that the function is in. In turn, the *crate* of a function is derived from its compilation directory. Compilation directories of functions can be found in *DWARF*. Usually, that directory is in the following format: `/path/to/checkout/<crate-name>-<version>`. From this format, we can derive the *crate* name to be: `<crate-name>`. Noticeably, an exception to this format occurs. The Rust standard library has a compilation directory of `/checkout/src/`.

The nodes in the call graph that the programmer can edit are thus marked as internal nodes. The significance of the distinction between internal and external nodes is further explained in Section 5.4.

Main Entry Point

The second attribute of the nodes which is sought after by the markers is the main entry point attribute. Only one node is marked with this attribute, namely the main function. It is important to note that the main function does not necessarily equal the entry point of the executable, as Rust programs usually specify Rust-specific setup functions as the entry point, which in turn call the main function specified in the source code of the program. This function can still be found with the proper *DWARF* tags.

The main function is important to know, as it is used to determine the name of the internal *crate*. The assumption is that the main function is always placed in the internal *crate*.

Panic Endpoints

In Rust, there are many different ways to end up in a *panic* state. The stack trace should always end with a *panic* call. As mentioned in Section 3.1, the functions `std::panicking::begin_panic` and `std::panicking::begin_panic_fmt` are currently the only two *panic* end points. Notably, both the functions that indicate a *panic* end point start with `std::panicking::begin_panic`. This information is used for the classification of the *panic* end points.

Whitelisted Functions

A programmer may want to whitelist some of the functions that cause panicking, as they may be sure that such function would never call *panic*, or, even if such *panic* would occur, they would not care. In order to do so, the programmer may specify the names of the procedures, or invocations which they would want to whitelist. In turn, the procedures or invocations with matching names will be marked as whitelisted in the call graph. This is further used in filtering, described in Section 5.3.2.

5.3.2. Filter Operation

Some nodes of the generated call graph may be redundant. An example of such nodes are nodes that do not lead to a *panic* call. Figure 5.4 shows a call graph in which such nodes are present, namely `baz()` and `buz()`. These nodes are removed from the call graph during the filtering phase.

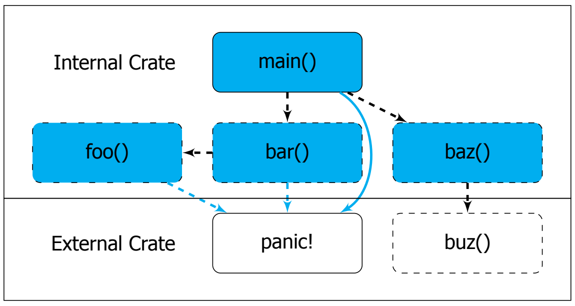

Figure 5.4: Call graph with filtered nodes

Another type of nodes that may be removed during this phase are the whitelisted nodes, this is represented in the call graph by the node `bar()`. Whitelisted nodes should produce no *panic* stack traces, however, it is arguable whether nodes reachable from whitelisted nodes, such as `foo()` should produce *panic* traces. By default, *Rustig* assumes that since `foo()` is unreachable, it does not produce any *panics*. However, the tool also supports a `full-crate-analysis` mode, where the whitelisting of `bar()` does not have an effect on the node `foo()`. Nodes that are marked whitelisted, as described in Section 5.3.1, are thus removed.

5.4. Find Panics

During the process of finding *panics*, a full stack trace is made. This trace begins with function which is in the analyzed *crate* and ends with a *panic* call. It is not plausible to find all paths in the call graph that lead to a *panic*, because this problem is $\mathcal{NP}$-hard, as shown in Section 3.3.1. Therefore, some simplifications have been imposed on characteristics of paths that are marked as relevant. The criteria for marking a *panic* as relevant are explained in Section 5.4.1. Implementation details of the algorithm for finding relevant *panics* are presented in Section 5.4.2

5.4.1. Relevant Panic Paths

The problem that is being solved in this step, is the problem of finding paths from nodes with some code to nodes that start the execution of *panic* (*panic* nodes). In order to keep the amount of *panic* paths manageable, only relevant paths are sought. Relevant *panic* paths are defined as shortest paths to *panic!* from each unique pair of nodes on the internal/external *crate* boundary. The pair has to contain one node of the internal *crate* and one node of the external *crate*. Due to the directionality of the graph, the relevant *panic* paths only contain one node from the internal *crate* and may contain more than one node from the external *crate*. These rules are explained with an example, using the call graph in Figure 5.5.

The blue nodes represent functions in internal *crates* and white nodes represent functions in external *crates*. From the perspective of a programmer, `foo()` is the function that is problematic, as it calls two other functions that may *panic*: `quz()` and `bar()`. The programmer, therefore, has two choices on how to remove *panic* calls from the program: either get rid of the function `foo()`, or fix `foo()` such that it calls neither `quz()` nor `bar()`. Note that if it was possible to change `buz()` directly so that it would not *panic*, then the call graph would have no calls to *panic*. This is not the case, since `buz()` is in an external *crate*. Even though many stack traces can be found that lead to panic,

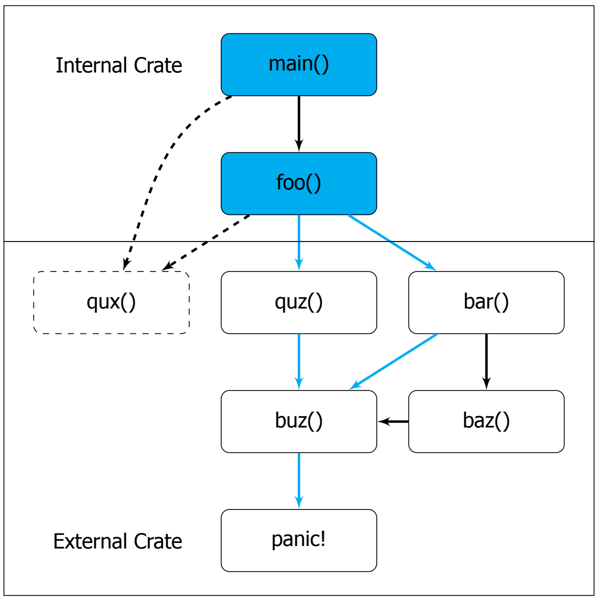

Figure 5.5: Example call graph

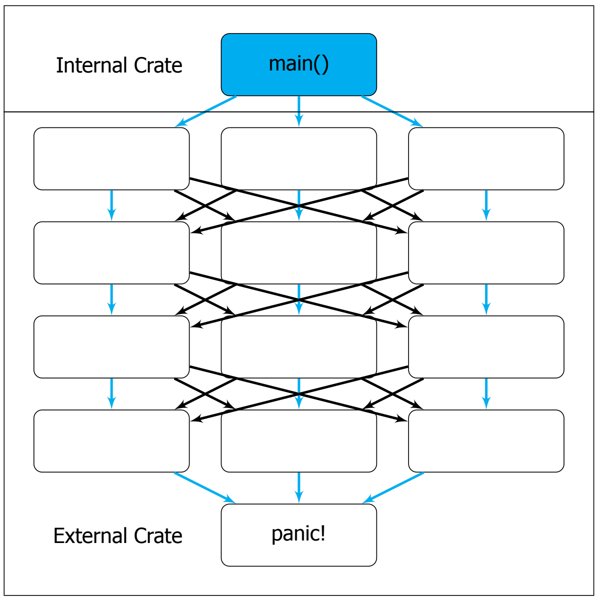

Figure 5.6: Example call graph with high branching factor

not all of them comply with the relevancy criteria defined before. Therefore, only two paths in this example are marked as relevant. These two paths are: `foo() - quz() - buz() -` **`panic!`** and `foo() - bar() - buz() -` **`panic!`**. Edges of these paths are marked blue. The two paths do not include `main()`, since for the *panic* trace, it is not relevant how the internal *crate* function, that has outgoing calls to *panic*, is called. Moreover, the path `foo() - bar() - baz() - buz() -` **`panic!`** is skipped, since it it not the shortest path from `bar()`. It is clear that the amount of *relevant panic* traces always equals the amount of outgoing calls from internal *crate* nodes to external *crate* nodes that *panic*. Traces containing `qux()` are not calculated, since `qux()` does not call *panic* and is thus filtered out from the call graph during the filtering phase.

Notice in Figure 5.6 that when the external *crate* code has a high branching factor, the total amount of paths from `main()` to **`panic!`** may increase exponentially with the size of the graph. On the the contrary, the amount of relevant paths is linearly bounded by the total amount of edges. A total of 81 paths can be found in the presented call graph, which contains 33 edges. Only 3 of those paths are useful by the criteria of relevancy defined before.

5.4.2. Implementation Details

The process of finding *panic* call traces in the call graph consists of two main stages, which can be summarized as follows. In the first stage, the shortest path to a *panic* node is calculated for each node in the external *crates*. In the second stage, for each edge between an internal node and an external node, a the stack trace is made from a *panic* up until that node in the internal *crate*. Note that each of the edges between an internal node and an external node leads to a *panic*, since paths that do not lead to a *panic* have been removed in previous step, described in Section 5.3.2.

Determination of Shortest Path

The call graph used for the representation of the program is unweighted. Therefore, breadth first search algorithm is used for the determination of shortest paths from *panic* calls to other nodes in the external *crates*. Note that the algorithm begins with the `panic` call and continues upwards, in the direction opposite to that of the edges of the graph. The algorithm visits each node at most once, and so the runtime complexity of this step is linearly bounded by the size of the graph. During this step, the partial traces to the closest *panic* are saved for each external node. The partial stack traces for the call graph in Figure 5.5 are shown in Table 5.2. It is possible for a call graph to have more than one *panic* node

Table 5.2: Traces to closest panic

Function	qux()	quz()	bar()	baz()	buz()
Stack trace	-	buz(), panic!	buz(), panic!	buz(), panic!	panic!

endpoint. In this case, a virtual node is added to the call graph, which is connected to all the *panic* endpoint nodes. The algorithm is executed in the same way, but the virtual node is excluded from the eventual stack traces. This is visualized in Figure 5.7.

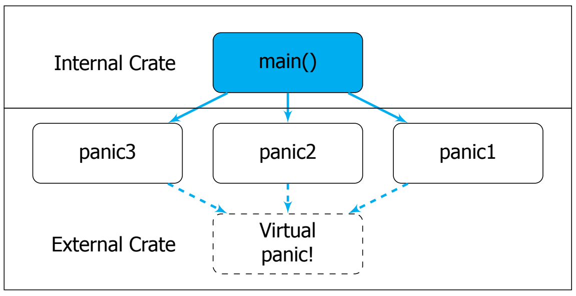

Figure 5.7: Example call graph with a virtual panic node

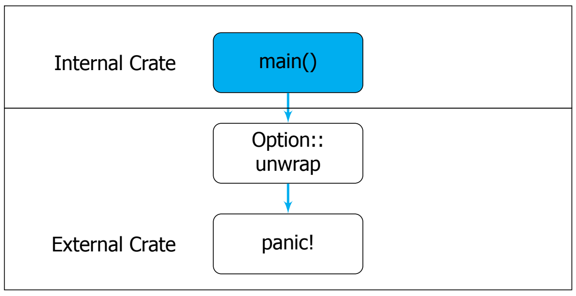

Figure 5.8: Call graph leading to unwrap

Collection of Stack Traces

During the second stage, the proper stack traces are collected. First, all edges that cross the boundary between internal and external *crate* are sought. For all of those edges, both endpoints are added to the stack trace. Then, the stack trace of the external node, which was calculated in the previous step, is added as well. This creates a complete stack trace of a relevant *panic* call.

5.5. Patterns

After finding the full stack traces of relevant *panics*, it is important to realize that some patterns can be recognized in those stack traces. Those patterns can provide insights as to what led to some specific *panic* call.

The categorization of different patterns can be made on the basis of the *panic* stack trace. As an example, Figure 5.8 shows a call graph of a function which leads to *panic*. Notice that the intermediate node is called `Option::unwrap` and it is therefore reasonable to assume that the *panic* occurs due to a call to *unwrap* on an *Option*.

The differentiation between *panic* stack trace patterns can be useful for the programmer to decide on the severity of some *panic* call. *Rustig* supports the detection of two types of *panic* calls. The first type, caused by `unwrap()` call, has already been described. The second type that is recognized, is the direct *panic* call. This type of *panic* occurs when the *panic macro* is called directly from the internal *crate*.

5.6. Output

The last step of the pipeline is providing the user of *Rustig* with the information, which is obtained in the phases mentioned in the previous sections. This information is presented in different output formats, each aiding the user in some way. The main output is given through the command line and consists of a list of found *panic* calls. This type of output is covered in Section 5.6.1. Furthermore, *Rustig* is able to write its inner graph representations to file. This is covered in Section 5.6.2.

5.6.1. Print Panic Calls

When providing feedback through a command line interface, it is important to determine what information is printed and how this information is structured. The essential information needed to fix a call to **`panic!`**, is the exact location in the source code of the function call that is found to *panic*. Providing more information, like the full stack trace or a matching pattern, could be useful. On the other hand, the output could become cluttered and might overwhelm the user. This consideration is accounted for by implementing two printing modes: *simple* and *verbose*.

Simple is the standard output format of *Rustig*. As the name suggests, simple contains concise information; the found pattern and full stack trace are left out. Instead, only the external function that was found to *panic* and its internal caller are printed, as well as the *crates* and files they were defined in. The format of simple can be seen in Listing 5.5. In this listing the output is split up over two lines, where normally the output would be printed on one line.

Listing 5.5: Entry in simple output format (i=internal, e=external)

```
<i-crate>::<i-function> calls <e-function> (<e-crate>)
  at <file name>:<line number>
```

When more information is required, the *verbose* output format can be chosen by using the `-v` or `--verbose` flag. Unlike the simple output format, verbose does show the found pattern and full stack trace. Furthermore, whenever a stack trace contains a dynamic invocation and is therefore uncertain (as explained in Section 5.2.3), a notification is added to the output of this trace. The exact format of verbose can be seen in Listing 5.6, lines 3 and 4 are repeated for every call in the stack trace.

Listing 5.6: Entry in verbose output format

```
1 --#<trace count> --Pattern: <Pattern> [--dynamic invocation]
2
3 [<entry number>: <crate><crate version>::<function name>}
4     at <file name>:<line number>] {number of entries in trace}
```

5.6.2. Call Graph

Rustig can output the content of its internal graph representations to a *DOT* file (a graph description language). This feature is optional and can be triggered by supplying the `-g` or `--callgraph` flag. Furthermore, the type of graph that the user would like to be returned should be specified. This is done by adding either `full` or `filtered` as an argument to the flag. A *full* call graph contains all the procedures and invocations contained in the executable that was analyzed. The *Filtered* type does not, this DOT file only contains the procedures and invocations that form a path to *panic!*.

5.7. Testing

The Rust programming language distinguishes two types of tests: unit tests and integration tests. The input to *Rustig* is a binary executable, hence, such binary executables are needed as test resources for both unit and integration tests. In this section, the generation of the resources is first explained in Section 5.7.1. Secondly, the types of generated resources are discussed in Section 5.7.2. Finally, the build script used for generation of these resources is investigated in Section 5.7.3.

5.7.1. Test Resources Generation

Integration tests are an important consideration for *Rustig*, since they require a full binary executable to perform analysis on. This binary could be included either as a pre-compiled binary or as Rust source code that would need to be compiled together with the tool when testing takes place. The final choice for implementation of the test suite was the compilation during testing. While this approach would generally increase the runtime of the testing phase, the benefits of this solution outweigh the costs. Those benefits are explained in the following paragraphs.

The first advantage of generating binary test resources during the compilation phase is the transparency of *Rustig*. One of the intentions for the tool was to release it as *FOSS (Free Open Source Software)*. Generally, software classified as *FOSS* is not prohibited from using pre-compiled binaries, as long as the source code for them is available. According to the Open Source Initiative: "Where some form of a product is not distributed with source code, there must be a well-publicized means of obtaining the source code for no more than a reasonable reproduction cost, preferably downloading via the Internet without charge."[28] The users might, however, be suspicious of such pre-compiled binaries in the sources of the tool.

The second advantage of the compilation during testing phase is the reduced size of the sources. The *ELF* binaries created by Rust are quite large, a simple "Hello World" application has a size of 5.1MB, while the source code of such application has a size of just 8KB. Including multiple binaries in the source code of *Rustig* would result in an unnecessarily large project.

The third advantage of the chosen approach is that *Rustig* can be tested more thoroughly, if both the program as well as the test resource are compiled by the same compiler. The change of version of *rustc* poses two challenges. On one hand, using versions of *rustc* newer than the version in which the tool was written may cause unwanted changes to the behavior of the program. On the other hand, using newer versions of *rustc* can also change the binary format slightly. This would mean that the input to the tool also changes and that should be tested. Note that when necessary, a fixed version of the compiler can still be used to compile some test resources. This functionality is used for tests that expect a specific binary as input.

5.7.2. Test Resource Types

The test resources needed for proper testing of *Rustig* are executable files compiled with either some fixed *rustc* version or the same version that *Rustig* itself is built with. In the source files of the tool, the directory `test_subjects` represents test resources that are compiled with the same version as the tool itself. These resources are used for most integration tests. The projects in directory `test_subjects_stable_rustc` are compiled with a fixed *rustc* version, namely version `1.26.0`. This version has been chosen as it was the latest available version of stable *rustc* during the initial phase of the development of *Rustig*. This type of test resources is mostly used for unit tests of the binary parsing stage.

5.7.3. Build Script

The test resources are generated during the compilation of *Rustig*. This process is possible due to a custom build script. The build script is used to build two separate projects that contain test resources: `test_subjects` and `test_subjects_stable_rustc`. Both projects are virtual *cargo* manifests, which means that they contain multiple *cargo* projects in them. When the virtual manifest is compiled, all the projects contained in it are compiled as well.

If the virtual manifests were not simple test resources but actual projects, the compilation would be done by running the `cargo build` command in the appropriate directory. The custom build script does exactly the same, it calls the `cargo build` command in the appropriate directory. For the `test_subjects` directory which simply uses the same version of *rustc* compiler as the project itself, the snippet of the build script can be seen in Listing 5.7

Listing 5.7: Snippet of the build script, showing the `cargo build` command

```rust
use std::process::Command;

Command::new("cargo")
    .current_dir(&test_subjects_dir)
    .arg("build")
    .status()
    .expect("Building of test subjects did not produce any output")
```

The build script for `test_subjects_stable_rustc` is similar, but since this directory should always be built with a specific, fixed version of *rustc*, this version is enforced by the build script. This can be seen in Listing 5.8.

Listing 5.8: Snippet of the build script, showing the `cargo build` command with fixed *rustc* version

```
1 use std::process::Command;
2
3 Command::new("cargo")
4     .current_dir(&test_subjects_dir)
5     .env("RUSTUP_TOOLCHAIN", "stable-2018-05-10")
6     .arg("build")
7     .status()
8     .expect("Building of test subjects did not produce any output")
```

Notice that the 5th line in Listing 5.8 sets the environment variable `RUSTUP_TOOLCHAIN` to the value of `stable-2018-05-10`, this value corresponds to the Rust version of `1.26.0`. This variable is only used when *rustup* is installed. Since *rustup* is the default toolchain manager of the Rust programming language, the assumption that it is installed is reasonable.

6 Process Evaluation

This chapter discusses the development process of *Rustig*. To ensure a flexible development process, it was decided to use Scrum as the development methodology for this project. Our take on Scrum is described in Section 6.1. Furthermore, our process used various tools to encourage a streamlined development process, these tools are discussed in Section 6.2.

6.1. Development Methodology

The project made use of the Scrum methodology, which is ”an agile approach for developing innovative products and services” [29]. Scrum uses short development cycles called sprints, which usually last between one week and one month. For this project, it was decided to use sprints with a length of one week each. These short sprints allowed us to be flexible in our development process and it would allow us to change the design if it were to be requested by *Technolution*.

During a sprint, several features would be scheduled to be implemented. These features would have to be fully implemented, tested and documented before they would be considered as done. To ensure that we would be able to meet our sprint deadlines, we would have daily stand-up meetings. During these meetings the product owner, who is a representative of *Technolution*, the scrum master, who is a project member who we appointed to be the scrum master, and the development team would be present. They would then discuss the individual progress that was made the day before, what each team member would accomplishing today, and if they foresaw any problems which could potentially prevent them from accomplishing their goal. If it seemed that a feature would not be finished on time we would be able to shift our priority towards that specific feature, or to reschedule it to a later sprint.

6.2. Development Tools

During the development process various tools were used. In this section it is explained how we used these tools to help development. The main tools for this project were Continuous Integration and Static analysis, these are explained in Section 6.2.1 and Section 6.2.2 respectively.

6.2.1. Continuous Integration

To ensure that problems were caught early, we would build and test our code pushes on a continuous integration server. For this project we got access to the Jenkins server of *Technolution*. This server was hosted by *Technolution* and provided us with very fast build times. On this server we ran static analysis, which will be discussed in the next section, and our tests on every commit. While it was great that Jenkins worked well, there was no integration with Gitlab. This made pull-based development slightly more inconvenient, since we had to check the status of our builds manually.

6.2.2. Static Analysis

To ensure that our code was of high quality, various static analysis tools were used. While the Rust eco-system is stil very young, various tools already exist for static analysis.

- **Clippy** is a static analysis tool used to catch common mistakes and to improve Rust code [30]. Clippy was ran on our continuous integration server to ensure a commit did not introduce any code smells. While this proved to be very useful during a project, we did not use Clippy as well as we could have. We decided not to run Clippy locally, Because it is very unstable, since new versions of Rust often break Clippy. Instead, we opted to run Clippy exclusively on the continuous integration server. This caused some failing builds, since there was no way to check Clippy warnings without pushing.
- **rustfmt** provides automatic formatting according to the Rust style guidelines [31]. During our developing process we used git hooks, which ran rustfmt before every commit. This ensured that we would only commit correctly formatted code. Despite the git hook, we could have improved our rustfmt usage. While rustfmt can be run on a continuous integration server, we did not use this functionality. If we did run it on our continuous integration server, we would be certain that our code would always be correctly formatted.

7 Final Product Evaluation

In this chapter, we discuss how well the product solved our initial problem. This is done by verification and validation of the final tool. Verification is used to check whether the final product is built according to the design goals and requirements, this is discussed in Section 7.1 and Section 7.2 respectively. Validation is done by ensuring that the built tool is solving the problem that *Technolution* had, this is discussed in Section 7.3. Finally we discuss the feedback we received from the Software Improvement Group in Section 7.4

7.1. Verification of Design Goals

In this section the design goals, which were first introduced in Section 4.2, are reflected upon.

Extensibility

Rustig is written to allow for easy extensibility. This is done by the use of traits, which can easily be used to implement a new feature without making major changes in other parts of the code. This adheres to the 'inversion of control' design principle [32]. Furthermore we made use of a creational design inspired be the factory design pattern. Using this creational pattern, the tool is able to return a specific implementation of a trait for the use case at hand. A programmer could easily extend this by adding an implementation to the factory.

Modularity

Rustig is built using various modules. As explained in Section 5.1, the design allows for easy removal and addition of modules steps in the pipeline. Furthermore, every module was built to allow it to be tested without strongly depending on code in other modules. This ensures that a module would work by themselves, as long as the correct input is provided.

Usability

During the project, we had various meetings with representatives of *Technolution*. This allowed us to thoroughly focus on usability of *Rustig*, since we constantly received feedback. One of such points of feedback was that our tool was providing an overwhelming amount of information. It was suggested to filter the results based on what would be interesting to the programmer. After implementing these features, the output of the tool became a lot more manageable. To give an indication of the size of the output, the total number of paths to *panic!* found for different Rust programs can be found in Section 9.3.

Scalability

Scalability was a major factor for this project. During the project we had to keep the size of the programs which we would be analyzing in mind. This influenced decisions on the various algorithms we implemented. While some other algorithms might have provided a slightly better result, their runtime and memory usage would be too large. These considerations proved to be beneficial, since we were able to analyze the executable of the Servo Browser Engine. This was a challenge, since the codebase of

Servo is quite large. It consists of approximately 300.000 lines of Rust code, and builds, with optimization enabled, to a binary of almost 1GB. The analysis has been performed successfully and took less than a minute, using 16.5GB of memory.

7.2. Verification of Requirements

In Section 4.1 various requirements were given. *Rustig* was built in accordance with these requirements and adheres to all of the *must have* and *should have* goals. However, *Rustig* does not adhere to all *could have* requirements. Therefore, it is interesting to discuss the *could haves* and why these could, or could not, be implemented. The functional *could haves* are found in Table 7.1. The non-functional *could haves* are found in Table 7.2

Table 7.1: Explanation of the functional could have requirements.

Functional Could Have Requirements		
Requirement	**Finished**	**Explanation**
The tool could be configurable to ignore particular patterns.	Partially	Rather than ignoring patterns, the user is able to ignore functions as explained in Section 5.3.1. This can be used to approximate pattern whitelisting.
It could be possible to ignore single warnings by placing a comment in the code.	Partially	While the user is unable to ignore single warnings, the user could whitelist an entire function as explained in Section 5.3.1
The tool could propose suggestions to fix found problems.	No	While it could have been possible to implement this, it was decided the gained benefits were not worth the time. The programmer is very well capable of solving the problem based on our current output.
The tool could support XML, JSON, Jenkins and HTML as output format	No	After some discussion with *Technolution* it was found that this had very low priority. They told us that, if necessary, they could implement this themselves. Therefore, it ended up at the bottom of our backlog and was not implemented due to a lack of time.
The tool could be able to print the call graph in DOT as output format.	Yes	Our tool is able to dump the call graph in DOT format, as explained in Section 5.6.2
The tool could have a verbose output mode, where full backtraces of unwanted thread exits are printed.	Yes	Our tool has both a simple mode and a verbose mode as explained in Section 5.6.1

7.3. Validation

Technolution provided us with regular feedback during the development of *Rustig*. This ensured that the tool would indeed be solving their problem. The output contained all the information they were looking for; however, the size of the output was quite overwhelming. Nonetheless, it did bring awareness within *Technolution* of the scale of the problem, since they were able to see how many distinct paths to *panic* there were in their code base.

To fix the problem of the results being too overwhelming, we made two changes for the tool to be usable by *Technolution*. The first change is whitelisting. Whitelisting allows the user to get rid of functions they do not care about. This allows the user to focus on the functions they do care about. The second change is clickable file names in the console output of your IDE. This change allows the user to immediately jump to a function that causes a *panic*. After applying these changes, *Technolution* was very satisfied with the workings of our tool.

Table 7.2: Explanation of the non-functional could have requirements.

Non-Functional Could Have Requirements		
Requirement	**Finished**	**Explanation**
The tool could be able to run on a Windows environment, so that developers who want to work on Windows, and cross-compile their code, can still use the tool.	Yes	Our tool does not use any system calls which are exclusive to Unix. Furthermore, it has been tested to run correctly on Windows.
The tool could support the ARM architecture.	No	Several parts of our code are x86 specific. While the code could quite easily be adapted to support ARM, *Technolution* told us this was not a priority for them.
The tool could be supplied with an architecture design, that explains which academic principles and structures were used in developing the tool.	Yes	The architecture design is provided in Section 4.4

7.4. SIG

As part of the process, our code was assessed by the SIG *(Software Improvement Group)*. They provided us with two points of improvements to better the quality of our code. A copy of the feedback by SIG can be found in Appendix E. In this section we discuss how we incorporated their feedback in our code.

The first point of feedback was that some of our units were too complex. This was caused by various long functions in our code. These long functions cause unnecessary complex code and they should be split up to allow for easier testing and a better structure. We fixed this by examining the tasks performed by a function, and distributing these tasks over different functions. This improved the readability of our code, since it made functions more compact. However, the runtime of the program, when compiled with optimizations enabled, increased by 4.6 seconds (8%) on average.

This benchmark was performed by running *Rustig* on Servo 10 times before the refactoring and 10 times after the refactoring. We suspect the runtime decrease happens because function parameters are passed on the stack in Rust when optimization is disabled. This is not as efficient as passing through registers. Some refactored functions were called relatively often, and therefore the newly introduced functions may cause significant overhead. When optimizations are enabled, this overhead is not completely optimized away. Since the runtime of the tool in absolute values is still acceptable, we ought this increase of readability to outweigh the performance loss. However, it shows that improving one metric can deteriorate another. Therefore, it is always needed to compare a set of metrics to obtain optimal code quality [33].

Secondly, SIG commented on the fact that we did not have any unit tests; however, we had more than 100 tests at that time. After some back-and-forth mailing, we learned that their tool did not detect tests in the same file as source code. They suggested us to put the test cases in a separate folder. However, Rust guidelines state that unit tests should be in the same file as the method they are testing [34]. Therefore, SIG decided to withdraw this suggestion and told us to follow the Rust guidelines.

8 Ethics

In this chapter ethical concerns regarding *Rustig* are discussed. Just like other static analysis tools, this software has been created with the incentive to support Rust programmers in writing robust code, but it could also be used by people with malicious intent, as described in Section 8.1. Furthermore, the documentation of *Rustig* should be clear about what the tool is capable of. Section 8.2 covers false assumptions users might have, resulting in users overestimating the quality of their software.

8.1. Control Flow Analysis

For us as the development team, and *Technolution* as a client, the objective of this project has always been evident: To aid developers in writing robust Rust code. In Section 3.2 we concluded that to obtain the information necessary to achieve this goal, the analysis must be performed on binary files. This approach provides the programmers with all the potential paths to **`panic!`**, enabling them to secure their program against unforeseen situations, that would lead to the system going down.

To perform analysis on binaries, these files first have to be abstracted to a higher level language (assembly) to make analysis less cumbersome and eventually provide feedback on a source code level to the programmer. This is achieved by reverse engineering the binary, a practice known to raise a lot of ethical and legal questions. Legally, copyright law protects the intellectual property of the developer (or company), prohibiting others from reproducing or making adaptions to the software. However, most countries allow reverse engineering in some cases. For instance, regulation in the United States and the European Union allows decompilation and disassembly of software that has been legitimately obtained, for the purpose of interoperability and error correction (the objective of *Rustig*) [35, 36]. On the ethical side, the intention of the user determines whether reverse engineering is considered right or wrong.

In the case of our tool, most use cases will be aiding the goal of writing fail-safe code. However, *Rustig* might also be used by people with malicious intentions, trying to expose vulnerabilities of a program. The tool allows for control flow analysis on an executable as input, which may lead to the exposure and exploitation of a vulnerability. This may cause systems running the software to stop or slow down execution, due to a handling the **`panic!`** that was caused.

The outlined problem will probably be hard to take advantage of, because the output of *Rustig* gives little to no indication about how to exploit the found *panic* calls. Furthermore, for the tool to be able to analyze an executable, it has to be compiled with debug information. Generally when a program is not open source, the executable does not contain this information, and is therefore not susceptible to an attack that makes use of the tool. When the source code is available, numerous other methods exist that are better at finding exploits in code. Therefore, we deem it unlikely that *Rustig* will be used for these practices.

8.2. False Assumptions

When using static analysis tools, it is important to keep in mind how they function and what aspect of the code they are trying to improve. In the case of this tool, it should be used by developers to assess how their own programs might fail. It is important that users do not have false expectations of *Rustig*, or how limiting the number of paths to *panic* helps them. A piece of code might be perfectly valid Rust, and contain no calls to **`panic!`**, but be full of bugs. Users of the tool might not realize this and falsely assume that their programs do not contain bugs.

Ethically, the big concern here is that *Rustig* becomes some kind of measurement on the reliability of software, directly associating the number of paths to *panic* with software quality. While a correlation between these two most definitely exists, one cannot simply deduce one from another. This wrong assumptions between a metric and software quality is not specific to this tool, but are a problem in general with the use of analysis tools. Bouwers, Visser and van Deursen characterize this mistake as ”Treating the metric”, where changes to software do not improve its quality, but are merely to improve the value of a metric [33].

We as the developers of this tool have a responsibility to make users aware of how *Rustig* functions and its possible shortcomings. We cannot expect the user to not associate less output of the tool with higher software quality, because it is a reasonable assumption to make. The intention of *Rustig*: to make programmers aware of hidden *panic* calls in their code, should be clear from the documentation. This documentation should also contain warnings about the aforementioned wrong assumptions, giving the user a clear prospect of the tool.

9 Discussion

in this chapter, the various issues and findings that emerged during development are discussed. Some of these issues could be solved by using a different approach, these are discussed in Section 9.1. Other issues are still present in the final product, and restrict the domain of the tool. These limitations are discussed in Section 9.2. Finally, various findings, which were found by the use of *Rustig*, are provided in Section 9.3.

9.1. Failed Approaches

Some of the tried implementations were unfeasible for multitude of reasons. One of such implementations was symbolic execution, which is explained in Section 9.1.1. Another failed approach was trying to find every path to a *panic*, this is further explained in Section 9.1.2.

9.1.1. Symbolic Execution

In Section 5.2, it was explained how `lea` instructions are used to approximate dynamic invocations. However, this algorithm was not the first implementation of this dynamic invocation finder. Our initial attempt implemented symbolic execution to find these cases. For every call relative to a register value we attempted to combine the previous instructions with the *DWARF* formal arguments information to determine what parameter value is in that register. Subsequently, we applied symbolic execution on all callers of that function, in order to determine the parameter they passed to that function.

It turned out that, in most cases, not enough context information was available to reliably determine the passed function pointers. Moreover, when function pointers were hidden in more complex data structures, doing only partial symbolic execution did not suffice. Therefore, we do not recommend implementing algorithms that are prone to missing cases when it is not a problem for the call graph to be over-represented.

9.1.2. Find Panics

In earlier sections it was discussed how finding every path in a graph is an $\mathcal{NP}$-hard problem. We, however, failed to realize this initially and tried to implement an algorithm which saves a list of all paths to `panic!` on each node. As a result we were very surprised when our initial algorithm ran out of memory while analyzing very simple programs. This was due to the algorithm being exponential in space. After further research we realized that the problem was $\mathcal{NP}$-hard.

Consequently, we had to find an algorithm that would provide the user of *Rustig* with the expected results, but would not run in exponential time. We then came up with the algorithm as explained in Section 5.4. This algorithm provided the necessary results and ran in polynomial time. Therefore, this algorithm is a suitable replacement for our original idea.

9.2. Limitations

Finally, the final product that is delivered also has its limitations. As stated in Chapter 8 it is important to be aware of these.

The first important limitation is that we do not attempt to find premature thread exits that are not caused by `panic!` calls. While the Rust compiler normally guarantees that these cannot occur, it is possible to circumvent these compiler checks by using `unsafe` blocks. In these `unsafe` blocks, memory access violation errors can occur. Our tool does no attempt to find these.

A second limitation is the fact that false positives can be reported. As explained in Section 5.2, dynamic invocations are over-approximated. Moreover, it might occur that some code is not reachable, but the compiler fails to optimize it away. In these cases the user needs to be aware that the reported trace is not an actual issue.

A third limitation is the fact that *Rustig* is dependent on the fact that `#[inline(never)]` is defined on the *panic* handler functions. As explained in Section 5.2.1, no nodes are created for inline functions. If the `panic!` handler would be inlined, our tool will not detect calls to it.

The last important limitation is the fact that traces through embedded C code are not reported. This can occur if Rust functions are inlined in C functions when link-time optimization is applied, or when Rust functions are explicitly called from C functions.

9.3. Findings

In this section, some findings on the number of *panics* found by the tool are discussed. It turned out that many *panics* are not relevant in all situations, but some *panic* traces that were found can really lead to program terminations. The whitelisting functionality (see Section 5.3.1) is very useful to filter these situations. An indication of the ratio of *panic* traces found to the number of lines in production code is provided in Section 9.3.1. An overview on how many of the found traces require more analysis is given in Section 9.3.2.

9.3.1. Concentration

The results obtained by our tool show that the problem does indeed exist. We ran an analysis on optimized builds of *servo*, *cargo* and two often downloaded *cargo* plugins. The results are displayed in Table 9.1. The number of lines is measured using *tokei*, on production code only. It turns out that approximately for every 3 - 8 lines of code, a path to *panic* is made. This is significantly more than we initially expected.

Table 9.1: Tool output

Crate	**Lines of Code**	**Number of *panic* paths**
Servo	219806	50881
cargo	25892	9439
cargo-make	8243	1195
cargo-edit	671	237

9.3.2. Quantitative analysis

We performed some analysis on the details of the output for five open-source projects and five internal pre-production projects (numbered 1 - 5 in the table), that were provided by *Technolution*. This was done by judging the relevance of the outputted traces that were reported. In order to categorize the output, we used whitelisting to remove traces with a specific cause, like formatting or memory allocation. The results are shown in Table 9.2. In this table, the column **Total** denotes all the panic traces in the program. In the column **Project specific**, we whitelisted some functions that were explicitly meant to *panic* if something went wrong in the setup or teardown phase. In the subsequent columns **Format**, **Allocation** and **Indexing**, we disabled traces for string formatting, memory allocation and array indexing respectively. We whitelisted these because, based on our experience, they are very unlikely to *panic*. These whitelists are added on top of the project-specific whitelists. In the **All** column, we combined all these whitelisting configurations. The traces in this output did not have an easy to identify cause, and would require more investigation. In the last two columns, we identified how many of the traces in the **All** column were caused by an `unwrap` or by use of the *panic* macro itself.

Table 9.2: Tool output

Project		**Whitelisted**					**Type**	
No	**Total**	**Project specific**	**Format**	**Allocation**	**Indexing**	**All**	**Unwrap**	**Direct**
cargo	9439	7782	4833	5514	6758	2238	738	49
cargo-make	1195	1136	695	697	1030	148	71	8
cargo-add	388	283	201	209	236	95	17	4
cargo-rm	126	108	64	91	97	37	5	0
cargo-upgrade	415	298	204	231	267	112	16	2
Project 1	1989	294	191	269	259	164	38	1
Project 2	2015	1839	398	1316	1577	237	118	0
Project 3	257	93	69	80	80	48	15	0
Project 4	461	230	114	214	203	100	23	0
Project 5	263	93	3	78	82	50	16	0

Based on these results, we tried to estimate the impact of this output. It turned out that many panics were difficult to trigger, since they are very environment specific. However, in the *cargo-make* project, we were able to trigger 2 *panics* after trying for just half an hour. These occurred when the current user has no read permission on some of the input files for *cargo-make*, or when that file was not formatted correctly. This demonstrates it is possible to find bugs with the tool; however, it certainly requires some effort. To make the tool more useful, more research regarding the causes and frequency of certain *panics* is necessary. This would allow the tool to only report *panics* that could actually occur in practice.

10
Conclusion

This project began with a clear goal in mind: analyze Rust code to find ways that a program could potentially *panic*. Using this information, *Technolution* could write more robust Rust code to ensure the availability of their software. The initial research phase proved that the problem was complex, but solvable.

The final product is a tool that statically determines paths to *panic* for a given executable written in Rust. Internally, *Rustig* creates a call graph using *ELF* binary and the embedded *DWARF* information. This construction of the call graph is based on the function calls found in the assembly. While static calls will always be correctly detected by the tool, estimations had to be made to detect dynamic calls. We concluded that we should overestimate these calls, since we want to ensure that no potential path *panic* is left unseen. To accomplish this we assume that every function that is loaded, using Load Effective Address instructions, will later be invoked and thus an edge should exist towards this function.

Building the call graph was only part of the problem. The next step was to analyze the graph to provide the programmer with usable output. This analysis consists of several smaller steps to preprocess the graph and finally finding the code paths which lead to a *panic*. While finding every possible path in a graph was the original goal, this problem turned out to be $\mathcal{NP}$-hard. It was therefore decided to find at most a single path for every function call to an external *crate*. This single path could already be used to warn programmers that the external *crate* they were using could cause a *panic*. All these paths to *panic* through an external *crate* are then written to the console.

The output, as provided by *Rustig*, allows a programmer to find potential premature terminations. The programmer could then decide to refactor their code to be less prone to *panics* during runtime. Rust programmers would then be able to to achieve a high availability for their code, while having the luxuries of type and memory safety of Rust.

Nonetheless, we propose several recommendations concerning *Rustig*. Firstly, whitelisting of certain functions could be done by using source code annotations. Currently whitelisting of functions is done in an external configuration file. Annotations would greatly improve the ease of use of whitelisting, since the programmer would no longer have to update an external file.

Secondly, *Rustig* could be improved to recognize more *panic* patterns. There are many *panic* call types which can be categorized by their stack traces. The pattern recognition is useful for the programmer as it helps them to approximate the severity of the *panic* call. The two *panic* types that are already categorizable by the tool, are direct *panic* calls and *panic* calls due to *unwrap*. It is expected that patterns such as array indexing or *panics* due to overflow checks can be recognized as well.

Finally, an extensive usability analysis of the tool could not be performed during this project. However, it would be advantageous to get answers to the following questions: "How often does *Rustig* help to identify and solve causes of premature thread exits?" and "How can the results of the tool be categorized, prioritized and filtered to optimize relevance and convenience to the user?". Answers to these questions could be used to further improve the effectiveness and usability of *Rustig*.

In conclusion, we are very satisfied with the results achieved by our tool, since the core functionality has been built and can easily be extended upon. Furthermore, we believe that our tool provides the Rust community with the means necessary to write more robust and highly available software.

A
Project Description

This appendix contains the original project description as found on BEPSys (https://bepsys.ewi.tudelft.nl/)

A.1. Background Information

Technolution is working on various systems in the defense, public security, industry and transportation markets.

In general, these kinds of systems require software to meet high security, availability and reliability standards.

Mozilla's new programming language **Rust** is a language that, by design, gives higher security and reliability guarantees than languages like C and Java. However, in some cases the higher security can lead to a decrease in availability of the software. The system goes down. It requires careful programming to ensure this does not happen.

A.2. Goal

The goal of this assignment is to **create tooling** to verify (*prove*) that the programmer did not trade availability for security. We want both.

To do this, the students will dive deep into the ELF binary format for executables and DWARF debug information and derive a call graph. Using this call graph the students will check for constructs in Rust that lead to the system going down.

A.3. Research Questions

- Research what information is needed to generate a call graph from an ELF binary.
- Research which situations in the call graph signify an unwanted situation as described above.
- Research the available tools and libraries to generate the required information.
- Research the available methods to allow a software engineer to blacklist/whitelist certain paths that lead to unwanted situations.

A.4. Technologies Involved

- Rust
- Linux
- ELF binary executable format
- DWARF debugging info format

A.5. Future

Detecting unwanted thread exits is only one verification that can be done once we understand the way a Rust program gets compiled and optimized. Future research might try to find additional proofs or other methods to deduce the required information from the Rust compiler and executable.

B Project Plan

The project plan was written in week 1 and defines our initial approach of the project.

B.1. Introduction

The TU Delft Computer Science Bachelor Project is the final project carried out by Computer Science bachelor students. During this project, as a team of 4 students, we will produce a product that is commissioned by the client, Technolution. The project is also supervised by a TU Delft coach, Robbert Krebbers.

The product commissioned by the client is an analysis tool for unwanted thread exits in Rust programming language. Even though the Rust programming language provides memory and data-race safety, the programmer is still responsible for ensuring the availability and reliability of the program. Irresponsible error handling is an example of a situation that can lead to unwanted exits. Such situations should be detected programmatically and hint the user what part of the code-base contains the unwanted thread exits. For some known patterns, the programmer can also be hinted in how the unwanted exit could be prevented.

The issue of unwanted exits is crucial for high availability systems, for which the downtime should be well below minutes per year. These systems cannot afford to lose uptime due to programming errors. Examples of such systems are: carrier-grade telephony, health systems and banking systems.

During the initial project phase, it is important for the three parties (the team, the client and the supervisor) to reach an agreement on the content, goals and methods of the project. It is of utmost importance that the product delivered by the team is usable by the client and also contributes sufficiently to the scientific world. Moreover, the project must allow us, as a team, to carry out a full software development project with a research component.

In this document, a project plan is proposed, such that the requirements of the three parties are satisfied. Firstly, the purpose of the project is described. Secondly, the general process that we plan to use during the project are laid out. Finally, we present the planning for the project.

B.2. Purpose

In this section, we will describe in more detail what goals we want to achieve by the end of the project. First, we will explain what research questions we want to have answered. These questions will help us to sufficiently understand the problem to come to an optimal solution. After that, we will describe the goals for the product we want to deliver. These goals (requirements) will be prioritized using the MoSCoW method. Finally, we will give an overview of deliverables we want to provide at the end of the project.

B.2.1. Research goals

Regarding the research goals, two particular questions are of interest. In its raw form, binary files formats are difficult to analyze. As a first step, we should transform binary information into a call graph. This call graph denotes what functions call to which other functions. Additionally, some context information can be given to these calls. For us, this means we should research what information we should extract from the binary files, and how we should represent this in a graph format. The question that we want to answer is: "How can binary information be used in order to create a call graph?"

The second question that is relevant is how to analyze the call graph we build. From a call graph (or binary itself) it is easy to detect whether certain procedures get called. However, it is more interesting to detect if these functions will actually be hit during code execution. The question that we want to answer is: "How can the call graph be used in order to find unwanted thread executions?"

In order to be able to answer the questions posed in the previous paragraphs, we also need to know some details about how Rust code is transformed into binary code. These include how functions can call other functions (static and virtual), how debug information is encoded, and how external code is linked and called.

B.2.2. Product goals

Another goal of this project is to deliver a tool that can detect unwanted thread exits (calls to *panic*). In this subsection, we will state the functional and non-functional requirements for this tool. These will be categorized into four priority classes: must-have, should-have, could-have and won't-have (where appropriate). This prioritization is scoped to our project only. This means that, for example, won't have requirements will not be implemented *by us*, but might be implemented at a later stage, in a different project.

B.2.3. Functional Requirements

This section provides the functional requirements as agreed upon with *Technolution*. These are the requirements of the functionality the final product should contain.

Must have

- The tool must operate from the command line.
- The tool must internally build a call graph from a Rust ELF binary.

Should have

- The tool should be able to detect, whether code paths exist that may lead to a panic call (which may reside in an external crate)
- The tool should print errors if the previously mentioned thread exits occur to the standard output. The format if the output will be:
  ```
  "Method '<method name> in <file name>:<line number>'
      calls '<crate>::<method name>'".
  ```
  Here the first mentioned method name will be the last method in backtrace which is user code, while the second mentioned is the first method in an external library that is called.
- The tool should have exit code 0 if no errors were found.
- The tool should have exit code 1 if unwanted thread exits were found
- The tool should have exit code 101 if an internal error occurred.

Could have

- The tool could be configurable to ignore particular patterns.
- It could be possible to ignore single warnings by placing a comment in the code.
- The tool could propose suggestions to fix found problems.
- The tool could support CI-server (Jenkins) compatible output.
- The tool could support XML as output format.
- The tool could support JSON as output format.
- The tool could support HTML as output format.
- The tool could be able to print the call graph in DOT as output format.
- The tool could have a verbose output mode, where full backtraces of unwanted thread exits are printed.

Won't have

- The tool will not be a compiler plugin
- The tool will not fix found issues in the source code itself.
- The tool will not detect unwanted thread exits originating from unsafe Rust code.

B.2.4. Non-Functional Requirements

This section defines requirements which is not part of the expected functionality of our tool, but rather define the way of operating and the design constraints.

Must have

- The tool must be executable within a Linux environment.
- The tool's source code should have no compilation errors under standard compilation conditions.

Should have

- The tool should be written in the Rust programming language.
- The tool's source code should have each public, non-trivial code element (method, struct, trait, etc.) documented. Documentation should indicate the purpose of the code element, and particularities or non-trivial properties of the implementation.
- The tool should have a short user guide, explaining all the implemented options, the output, and configuration as part of its source code.
- The tool should have a help-option, that prints a small guide on how to invoke the tool.
- The tool should have no compilation warnings.
- The source code should adhere to the Rust Style Guide [24]
- For each function, all usage patterns that can be reasonably expected should have a test case. For these test cases, it should be documented what use case is tested.

Could have

- The tool could be able to run on a Windows environment, so that developers who want to work on Windows, and cross-compile their code, can still use the tool.
- The tool could support the ARM architecture.
- The tool could be supplied with an architecture design, that explains which academic principles and structures were used in developing the tool.

Won't have

- The tool will not be supported by the original development team after project termination.

B.2.5. Deliverables

In order to achieve the previously set goals, we will provide the following at the end of our project:

Product

We will deliver an executable that can be run on a Linux environment. Furthermore, the complete, commented, source code, including tests and development history, of the project will be available to the client and the TU Delft coach. Technolution will be the owner of the code. Technolution may decide to publish this code under the MIT license or the Apache-2.0 license.

Documentation

Furthermore, we will deliver the following documents in both PDF and LaTeX form: Firstly, a user guide, that explains the purpose of the tool, what command line parameters can be given. Secondly, we will provide a research report, in which the findings of our research will be summarized. Thirdly, we will provide an architecture overview, which will explain the technical design choices made. This document will refer back to the research report for scientific background of the design. Fourthly, we will deliver a final report, which will elaborate more on the full project.

B.3. Process

This section describes how we plan on achieving the previously mentioned research goals (Section B.2.1) and product goals (Section B.2.2), and which approach we will take to meet the requirements of the deliverables (Section B.2.5). The process description will be split up in two parts. First we will illustrate how we plan to collaborate as a team and communicate with our client and coach in Section B.3.1. Secondly we will describe what facilities we have at our disposal and which software will be used for implementation in Section B.3.2.

B.3.1. Communication

Our client provides us with a spacious room, from which we will be working for the next ten weeks. Since we all sit together, we will be able to communicate in person on a daily basis. The room is equipped with two whiteboards, of which one was covered immediately in post-it notes, functioning as our Scrum board. The Scrum framework will be used to provide fast feedback and to assure that a working version of the tool is available at all times. Due to the short project duration, sprints of a 1-week period are chosen. The first two weeks will deviate from the standard Scrum methodology, the university requires us to conduct research, preparatory to any product implementation. This also means that we will be starting with daily Scrums after the first two weeks.

Meetings with our client, Erwin Gribnau, are scheduled on Mondays and Thursdays at 10:00 am in our room at Technolution and will generally take half an hour. Furthermore the client has indicated to be available for questions during work hours and has provided us with his contact information. The meetings with our TU Delft coach, Robbert Krebbers, are scheduled on Fridays around 9:00 am, at his office in Delft, and will generally take half an hour. Due to holidays and the schedule of both client and coach, some meetings will be moved to another time.

To ensure a smooth project process, we agreed upon some rules for conflict resolution. First of all, to avoid confusion, agreements between group members or between group and client should be explicit, clearly formulated and well documented. If a group member or the group in general foresees that an intermediate deadline cannot be met, this should be communicated to the rest of the group and the client at least 24 hours before the respective deadline. A disagreement between group and client should first be discussed briefly between these two parties, before the TU Delft coach gets involved.

B.3.2. Means
The company provided us with four modern desktop PC's, on which most of the work during this project will be done. We had the freedom to choose a Linux distribution that we felt comfortable with. The client also required us to make use of the GitLab server of the company as version control. We will use this server with the following five rules:

1. No pushes to master
2. Each feature is made on a separate branch
3. Each branch is peer reviewed before it is merged into master
4. A merge is approved only when documentation and test requirements are met
5. Only code that has successfully been built on our CI server, is able to be merged into Master.

Early on, we, as a group decided to write our tool mainly in Rust. This choice was made for a number of reasons: First and foremost the code base should be maintainable within the company, after the project has ended. Furthermore, Rust provides methods for binary parsing and easy access to C-functions. Writing the problems in Rust also gets us acquainted with the problems we are trying to resolve. At last, after the project has ended the code will be made open-source, at which point we hope the Rust community will start using it. Writing the code in Rust will facilitate this process.

The code we produce will be sent twice to the Software Improvement Group (SIG) during this project. The feedback we get from SIG during the first code submission will be used to improve the code base for the second and final code evaluation.

B.4. Risks
In this section, in order to maximize the chances of this project succeeding, we identify some risks that may possibly negatively influence the progress of our project. Furthermore, we describe the methods of dealing with them in order to minimize their negative effects.

B.4.1. Leave of a team member
The risk here is that unexpectedly, a member of the team would not proceed with the project. In this case, the solution will depend on the time-frame when this happens. If this situation occurs at the end of the project, we expect to continue the project as planned. If, however, this happens at the beginning of the project, some changes to the expected goal of the project will be applied. Due to lowered amount of team members, we expect that the main goal would be the generation of the call graph. We still expect to identify some thread panic calls, but the research on this topic will be less sophisticated.

B.4.2. Expected design is infeasible
There exists a risk that the program design that is expected from us is not feasible. For this risk, we identified two scenario's of how this could happen. It is either not possible to make a full call graph with information in ELF/DWARF formats or it is not possible to reliably find the thread panic calls within this graph.

The call graph cannot be reliably made with ELF/DWARF information
In the situation where the full call graph cannot be reliably made with only ELF/DWARF information, we could extend our approach to include more information sources, such as source code or compiler information. While these methods may not be preferred, it may be necessary to use them if no other method will prove to be useful.

The thread panic calls cannot be reliably identified from the call graph
We have seen that at least some thread panic calls can be identified from the call graphs. If, however, more in-depth research will reveal that not all of these thread panic calls can be identified, we plan to continue our research and classify cases where the thread panic calls cannot be identified properly. Using this approach, we can specify what guarantees our tool can provide for program analysis.

B.5. Planning

The planning for our project is as follows, we would like to submit the project plan within the first week of our project, this project plan has to be approved by both our client and our TU Delft coach, we can then start our research phase. We then have to submit our research report 2 weeks into our project. Due to absence of our client during the second week, we propose to submit the report on the 7th of May. The next deadline is the code submission to SIG on the 1st of July. Furthermore, we have a second mandatory deadline to submit the code to SIG on the 25th of July. The final mandatory deadline is our final presentation, which will be either on the 2nd, 3rd or 4th of July. Table B.1 provides an overview of all the deadlines provided by the university. Finally we want to propose a few dates where we can submit our report to our TU Delft coach for review. These dates will be discussed during our first meeting.

We plan on making a more in-depth planning after our research phase. We want to set ourselves a clear road-map, however before doing so we need more research to make a judgment about the time-span of all the software components.

Table B.1: Deadlines

Week	Date	Description
Week 1	April 26	Project Plan due
Week 3	May 7	Research Report due
Week 6	June 1	Submit code to SIG
Week 10	June 22	Submit code to SIG
Week 10	June 25	Submit final Report
Week 11	July 2, 3 or 4	Final Presentation.

C Research Report

Our research project was originally written during the first weeks of the project. Most of our original Research Report has been integrated into the main report and therefore it was decided to not add the entire report as appendix. However, since the chapter on *ELF* and *DWARF* was not included in the main report, it was decided to add it as an appendix.

C.1. ELF and DWARF

To accomplish our goal of finding potential software exits, we analyze executables which are stored using the Executable and Linkable Format (ELF). Using the data found in the ELF we create a callgraph which can be used to analyze the program. Furthermore ELF contains a DWARF subsection. DWARF can be used to store debug data about a compiled program. Using DWARF, a debugger can find the relevant information in the source code when given a compiled program. In Section C.1.1 we give a short overview on the structure of ELF and DWARF. In Section C.1.2 we provide a comparison between various methods of parsing ELF in Rust.

C.1.1. Structure

A file using ELF consists of various headers [37]. The first of these headers is the ELF header, which describes the structure of the ELF file and general system information, for example, for which operating system it was compiled for and the endianness of the file. Furthermore, it describes at which byte the program headers and the section headers start. The program header contains information to allow the operating system to prepare for the execution of the file. A section header contains information on a specific section, most importantly the offset of a section and the type of the section. The following sections are the most valuable to us:

- .text, which contains machine code that can be disassembled.
- .data, which contains various data, but most importantly the virtual method tables which can be used to trace method invocations.
- .debug, contains the DWARF debug information.

DWARF, is meant ”to facilitate source level debugging” [38]. DWARF represents the debug information as a tree, where a node of the tree is the debug information related to that entry, an entry could be a type, variable or function, and the children of this node are debug information owned by that node. Furthermore, DWARF contains a mapping from variables and function in the .text section of an ELF file to the original source code. Using this mapping we can trace patterns in the assembly (e.g. Panics) back to its original declaration in the source code.

Using ELF and DWARF we can analyze the compiled machine code and find patterns which could potentially lead to a Panic.

C.1.2. Parsing

In order to be able to access all the information in the binary files in our tool, we need to parse these files to a format that can easily be used and interpreted. As a first step, we will parse the ELF file, in order to access the DWARF information, machine instructions, symbol tables and data sections. As a second step, we will need to parse the DWARF information and disassemble the machine code.

There are many Rust crates available for ELF parsing. The first option is the *object* crate, which has a simple API, but does not offer detailed header information. Since we need to extract, for example, the section header offset, in order to disassemble correctly, this crate might not be a good choice for us. The second option is the *elf* crate. This crate gives us enough information, but does not represent headers as enums, which makes them more difficult to read for us. The *xmas-elf* crate, on the other hand, does represent headers as enums. The disadvantage of that crate is that it applies a very non-intuitive construction to dispatch different byte-sizes. All struct types take a P32 or P64 type parameter, which are returned in an enum that encapsulates both types. The *elrond* crate provides a very detailed, but still intuitive api. It also performs the flag parsing which, e.g. the *elf* crate did not do. The last option we found is *goblin*, which also has a good API, all needed header information available, and even checks the type of entries of the symbol table.

We see that for our use case, only the *object* crate is not a good option. This makes sure that we can, if needed, use the same crate as a potential other dependency does, in order to speed up running time (not parsing the elf twice) and binary size (not including superfluous dependencies).

Regarding the DWARF parsing, we found only the *gimli* crate that can perform this task. When examining the documentation, we found that it is quite feature-complete. It can handle different DWARF versions, use abbreviations, and much more. Also, this crate has been used to implement an *addr2line* clone in Rust, and for creating a tool that generates backtraces at runtime (*backtrace*). The *addr2line* crate might even be useful for us, to map call destinations to function names and source lines.

For disassembling the code, we found 2 possible options. The first one is the *capstone-rs* crate, that provides bindings to the C *capstone* library. We could get that library to work in an example. A disadvantage of this API is that only a text representation of the instruction is given. If we want to extract this information, we will need to parse these instructions manually. Another option to decompile is using *radare2* [39], and the *r2pipe* crate. This option also has the disadvantage of string output, but in addition to that requires a full *radare2* installation, which introduces a lot of overhead.

D
Test Report

Coverage Report

Command:			
Date:	2018-06-22 14:54:26	**Instrumented lines:**	4040
Code covered:	91.3%	**Executed lines:**	3689

Filename	Coverage percent	Covered lines	Uncovered lines	Executable lines
./lib/callgraph/src/lib.rs	67.6%	94	45	139
./lib/panic_analysis/src/graph_output/mod.rs	78.0%	64	18	82
./lib/panic_analysis/src/marker/mod.rs	82.6%	38	8	46
./lib/callgraph/src/callgraph/static_calls.rs	87.1%	101	15	116
./lib/panic_analysis/tests/function_whitelist.rs	87.4%	146	21	167
./lib/panic_analysis/src/lib.rs	88.0%	491	67	558
./lib/panic_analysis/tests/recognize_unwrap.rs	88.6%	31	4	35
./lib/panic_analysis/src/filter/mod.rs	90.6%	48	5	53
./lib/panic_analysis/src/marker/function_whitelist.rs	90.9%	130	13	143
./lib/panic_calls_output/src/lib.rs	91.1%	205	20	225
./lib/panic_analysis/src/patterns/direct_panic.rs	91.6%	141	13	154
./lib/panic_analysis/src/marker/analysis_target.rs	91.7%	111	10	121
./lib/panic_analysis/src/binary/mod.rs	92.6%	25	2	27
./lib/panic_analysis/src/marker/entry_point.rs	93.0%	80	6	86
./lib/panic_analysis/src/filter/panic_filter.rs	93.0%	66	5	71
./lib/callgraph/src/dwarf_utils.rs	93.1%	54	4	58
./lib/test_common/src/lib.rs	93.4%	57	4	61
./lib/callgraph/src/parse/mod.rs	94.1%	112	7	119
./lib/panic_analysis/src/patterns/unwrap_panic.rs	94.3%	148	9	157
./lib/panic_analysis/src/filter/whitelist_filter.rs	94.3%	116	7	123
./lib/panic_analysis/src/panic_calls/mod.rs	94.5%	445	26	471
./lib/panic_analysis/src/marker/panic.rs	94.7%	249	14	263
./lib/callgraph/src/callgraph/lea_dynamic_calls.rs	95.9%	301	13	314
./lib/panic_analysis/tests/libcalls.rs	96.4%	53	2	55
./lib/callgraph/src/callgraph/mod.rs	97.0%	318	10	328
./lib/rtti-derive/src/lib.rs	100.0%	1	0	1
./bin/cli/src/main.rs	100.0%	15	0	15
./bin/cli/src/cmd_args.rs	100.0%	3	0	3
./lib/callgraph/src/binary_read/mod.rs	100.0%	16	0	16
./lib/panic_analysis/src/patterns/mod.rs	100.0%	29	0	29
./lib/rtti/src/lib.rs	100.0%	1	0	1

Generated by: Kcov

E
SIG Evaluation

This appendix contains the evaluation by the Software Improvement Group. Note that the feedback was provided to us in Dutch.

E.1. Feedback on first SIG submission

De code van het systeem scoort 3.5 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code marktgemiddeld onderhoudbaar is. We zien Unit Complexity vanwege de lagere deelscore als mogelijk verbeterpunt.

Voor Unit Complexity wordt er gekeken naar het percentage code dat bovengemiddeld complex is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, makkelijker te testen is en daardoor eenvoudiger te onderhouden wordt. Door elk van de functionaliteiten onder te brengen in een aparte methode met een descriptieve naam kan elk van de onderdelen apart getest worden en wordt de overall flow van de methode makkelijker te begrijpen.

In jullie project is LEABasedDynamicInvocationFinder() in lea_dynamic_calls.rs een goed voorbeeld van een complexe methode, waarbij het opvalt dat het merendeel van jullie code uit vrij kleine methodes bestaat. Dit is dus een uitschieter, waarbij een grote hoeveelheid functionaliteit in één stuk wordt uitgeschreven. Er zijn pogingen gedaan om door middel van commentaar structuur en duidelijkheid aan te brengen. Dat is in principe goed, maar idealiter zou je die structuur in de code zelf willen aanbrengen.

Als laatste nog de opmerking dat er geen (unit)test-code is gevonden in de code-upload. Het is sterk aan te raden om in ieder geval voor de belangrijkste delen van de functionaliteit automatische tests gedefinieerd te hebben om ervoor te zorgen dat eventuele aanpassingen niet voor ongewenst gedrag zorgen.

Over het algemeen is er dus nog wat verbetering mogelijk, hopelijk lukt het om dit tijdens de rest van de ontwikkelfase te realiseren.

F
Project Reflection

This chapter contains the reflections on the project of individual team members. In this evaluation we reflect on our personal contributions to the project and the communication between the various external parties.

F.1. Dominique van Cuilenborg

During the initial research phase of the program, I invested most of my time in researching *ELF* binaries and the *DWARF* information they contained. This was very important to our project, since we had to know whether it would be feasible to build a call graph from a binary. After the research phase we came to the conclusion that this would be feasible.

During the project I was heavily invested in the algorithmic part of the tool. This part consisted mostly of algorithms to extract the necessary data from the call graph. However, the issue is coming up with an actual algorithm that provides you with the correct answer. Whenever we came up with a possible solution, I found a lot of enjoyment in coming up with a counter-example, as to why our algorithm would fail. This prevented us from wasting time on implementing algorithms that would not provide the correct result.

However, there is more to an algorithm than just providing the correct answer; Algorithms have to be fast. I found it very interesting to optimize our algorithms as much as possible. Optimization was very important due to the fact that the tool should be able to analyze big binaries, without a major increase in runtime. A slight error in one of our algorithms could greatly increase the runtime of our program, which we wanted to avoid at all costs. Therefore, we often had to come up with a simplification of our problem that would still return the results necessary, but would be a major speedup compared to our original algorithm.

Nevertheless, we would not have been able to do any of this without the great cooperation within our team. Every member was able to put their stamp on the project. Furthermore, whenever someone would get stuck working on their idea (or was stuck fighting the Rust compiler...), the team made sure to help them as soon as possible. What contributed immensely to the cooperation within the team was the fact that we were working full-time in an office provided to us by *Technolution*.

Other than the cooperation within the team, we had to cooperate with the client and the TU coach. I do not have any remarks on this aspect of our project, since we had weekly meetings with our coach and almost daily meetings with the client. These meetings always provided us with in-depth and useful feedback on not just the workings of the tool, but also the final paper we would write. This ensured that *Technolution* would be satisfied with our program and that there was a scientific purpose to our project.

When looking back at the product we created, I feel very satisfied. We were able to create a tool that helps the programmer build a more robust Rust program. However, I also feel like our tool reveals the dark-side of rust; A language that is meant for low-level programming should not be able to crash this easily. Therefore, I hope that our tool can stir some conversation in the Rust community on this topic.

F.2. Bart van Schaick

Looking back at the past three months and assessing the final product, I think we as a team can all agree that the project has been a success. The client and TU Delft coach also seem pleased with the way things went and the achievements we made. Luckily, I already had been in contact with *Technolution* during a previous project. The positive impression I got back then, contributed to the decision of picking this project over other interesting alternatives.

The impression proved to be correct, because the client has been very helpful and understanding during the past months. It really helped that the client had previous experiences with student projects, and also that, as a software company, the client understood what we were doing. The room and other facilities that they provided us with, really contributed the project and its outcome. Furthermore, the engineers at *Technolution* have provided us with helpful insights and alternative solution to problems that we were facing.

During the research phase, when we were analyzing the problem and possible solutions, together with Fabian, I was assigned to look into how the call graph should best be built. This proved to be a one-man-job, so I had to come up with another interesting research topic. Those two weeks I spent most of my time investigating how other static and dynamic analysis tools worked, and whether they could be used to serve our purposes. Furthermore, I was involved in testing a number of crates (libraries), which might come in handy during the realization phase of the project.

For the remainder of the project I was responsible for the output of the tool. My task was to gather all the information acquired in previous stages and to provide this information in useful way to the user. Because of continuous improvements that led to the tool being able to draw more and more information from the binaries we analyzed, the output was bound to change every once in a while. This could be quite frustrating, because every new feature and structural change to the code, could force the output to change as well.

Another factor problematic to the implementation of the output and the tool in general, is the problem turning out to be bigger than expected. The number of ways to panic we found in some projects is staggering, especially when you consider that we already dismissed a number of paths. No developer in his right mind would be willing to analyze thousands of stack traces leading to a panic, just to find the few that could really cause hazard. I think that the people that will continue the development of this tool should be focusing on the classification of treats, something we already started by implementing white-listing and pattern recognition.

F.3. Fabian Stelmach

In the first phase of the project, the research phase, I divided my attention between two topics: the Rust compilation process and building of a call graph. The former topic gave us insight about the process which is responsible for the creation of executables from Rust source code. The latter topic has helped me develop knowledge about the structure of a call graph and allowed me to develop ideas about algorithms that could be used on the call graph to help us solve the problem, some of which have been implemented in *Rustig*.

One of the areas where I have been active during the realization phase of the project was the algorithm design. I have presented many ideas that I had developed during both the research phase and the realization phase. The most important design decisions, that I helped develop, were the bottom-up approach of the call graph analysis and the definition of relevancy of a path in a call graph. These two topics are quite closely related to each other, since the combination of the two has improved the usability of the program by reducing the (overwhelming) amount of output that *Rustig* produces.

An important task that I had been assigned during the project was that of creating a full test suite that would allow for integration, and regression testing of the tool. Updates to the Rust compiler could possibly change the workings of the program, but the updates could also change the input of the program. Such changes should be caught by simply running the test suite of the tool. This concept has been developed quite well, and so I expect that in the future, *Rustig* will be well maintainable.

In general, I think that the project has been a success. The tool that we have developed could be used productively in order to reduce the occurrences of *panic* calls. The client, *Technolution*, has already shown interest in reducing the amount of *panic* calls in their Rust programs. My hope is that the Rust community will also take a notice of the value that *Rustig* provides, since that could lead to an increase of quality in the open source crates.

The success of the project can be credited a combination of many factors. One of such factors is the working environment. The client, *Technolution*, has provided us with a spacious office that we have used for the whole duration of the project. The key to maximizing the usefulness of that office, in my opinion, was the communication process between the team members. I was really glad that the others actively helped maintain the open-minded atmosphere, where bringing up new ideas was welcome and discussion of design choices occurred frequently.

Finally, the success of the project can also be partially credited to the collaboration with our coach, Robbert Krebbers and our direct supervisor at *Technolution*, Erwin Gribnau. Their guidance consisted of continuous critical feedback on the *Rustig* design ideas, implementations and written work. With this guidance, they have both greatly helped us to complete the project successfully.

F.4. Aron Zwaan

My role in this project, besides developing, was that of scrum master. Since the project was too short to really get the scrum-train going, we only did the daily stand up meetings, which were led by me. I found the stand-up meetings very useful, because we used them to synchronize ideas, report progress, and decide on future directions.

During the project, I also contributed a lot in discussions regarding design details. For many parts of our project, it was very important to specify the implementation very precise, since minor difference in implementation could often lead to superfluous, missing or incorrect results, significantly higher runtime or problems in implementing other planned features.

Regarding the research, I was mainly responsible for the technical details regarding the binaries we analyzed. This included finding out what information was contained in an ELF file, and the corresponding DWARF information. Moreover, I researched how different Rust constructs translated to assembly instructions and how we could use these to build a call graph.

Based on this knowledge, a lot of my implementation work went in to the dynamic invocations. I came up with the idea to implement dynamic invocations using loaded functions reference. I was surprised by how well it worked, despite the risk of false positives.

Another important part of the project I contributed was the overall structure. Based on the knowledge of the problem after 2 weeks, I designed the pipeline and most important data structures. Although this structure was later adapted to the new requirements and insights that came up during the project, the overall idea worked quite well.

Looking back on the cooperation within the team, I think we did a good job. Every member was able to have his say, and most often the combined insights gave the best results. Moreover, details of the problems and different solutions were often difficult to comprehend, most often regarding the different graph analysis options. Discussing these characteristics in depth together often led to valuable insights.

In addition to that, the cooperation with the client and TU supervisor was very good. Both the functional aspects of the tool, and the scientific contributions were guarded well. This made sure we, as a team, kept hold of the bigger picture, instead of focusing only on the technical details of our problem only. The feedback provided on our tool (by the client) and the report (by the TU supervisor) was always detailed, in-depth, and useful. We learned a lot their comments.

When reflecting on the final product, I think we did a great job. The suspicion that there were many panics in Rust code and library could be confirmed and quantified. I hope that the Rust community will take the message we will bring out serious, and try to greatly reduce the amount of panics. I am definately looking forward to new developments in this field.

To conclude, I think the project definitely succeeded. We delivered a useful tool, demonstrated a problem regarding the Rust software stability, and developed several new concepts regarding call graph building, and graph analysis.

G
Info Sheet

On the next page a copy of the info sheet can be found. The purpose of this info sheet is to provide a short overview of the project.

Title of the Project: Tooling to Detect Unwanted Thread Exits in Rust Software
Name of the Client Organisation: *Technolution*
Date of the Final Presentation: July 2, 2018

Description

Technolution is a company that realizes complex software systems and electronics. Some of these systems are implemented using the Rust programming language. Despite inherent safety guarantees of the language, unforeseen situations in Rust can cause the program to end up in state it cannot recover from, forcing program termination and the system going down. This is highly undesirable for *Technolution*, due to the high availability requirements for their systems.

For the purpose of detecting such unwanted thread exits, we created a static analysis tool: *Rustig*. *Rustig* provides the user with stack-traces to show how their program could prematurely terminate.

During the initial research phase we analyzed whether it would be feasible to do our static analysis on the binary of a program. To do this, we had to delve deep into *ELF* binaries to extract the necessary data. We concluded that there was a sufficient amount of data in these binaries to successfully analyze the control flow of a program.

The challenge of this project was to build a call graph using the machine code of a compiled Rust program. One particular problem we came across during the project was how to deal with function calls that are unknown at compile time (dynamic dispatch). We solved this by implementing an advanced over-approximation of those function calls.

Furthermore, we had to find a way to analyze the graph to provide the programmer with execution paths that could potentially lead to a thread exit. An unexpected challenge that we faced, during the call graph analysis, was the amount of execution paths that could be found. This required us to come up with a way to only extract the paths necessary to warn the programmer.

Rustig will be released as an open-source project. The intention behind this is that the tool could be further developed and maintained by the Rust community. Furthermore, the results generated by the program could spark a discussion within the community about the large amount of unwanted thread exits in Rust. For the future development, we have recommended the client to expand the ability of the tooling to categorize the types of unwanted thread exits. Doing this would allow programmers to further determine the severity of the output *Rustig* provides.

Members of the Project Team

Dominique van Cuilenborg — d.vancuilenborg@hotmail.com
Interests: Algorithm Design, Programming Languages and Operating Systems
Contribution: Call Graph Analysis, Panic Pattern Detection and Metadata Extraction

Bart van Schaick — bvschaick@gmail.com
Interests: Embedded Systems, Artificial Intelligence and Software Design
Contribution: CI Setup and Output Formatting

Fabian Stelmach — fabianstelmach@gmail.com
Interests: Algorithm Design, Programming Languages and Software Design
Contribution: Call Graph Analysis and Test Suite

Aron Zwaan — aronzwaan@gmail.com
Interests: Software Design, Software Analysis, Programming Languages and Embedded Systems
Contribution: Call Graph Building and Executable Analysis

Contact Information

Client	Erwin Gribnau	erwin.gribnau@technolution.nl
Coach	Robbert Krebbers	r.j.krebbers@tudelft.nl

Glossary

AST Abstract syntax tree. A tree representation of the abstract syntactic structure of source code written in a programming language. 10

cargo Rust's package manager. 16, 29, 40

crate Rust software package. 12, 13, 16, 19–21, 24–28, 43

DWARF A standardized debugging data format. 11, 12, 19–21, 24, 25, 39, 43, 53, 59

ELF Executable and Linking Format. A common standard file format for executable files, object code, shared libraries, and core dumps. 10, 11, 13, 15, 16, 19, 29, 43, 53, 59, 64

FOSS Free Open Source Software. Software that can be classified as both free software and open-source software. That is, anyone is freely licensed to use, copy, study, and change the software in any way, and the source code is openly shared so that people are encouraged to voluntarily improve the design of the software. 29

gcc GNU C compiler. 9

gprof GNU profiler. 9

hidden panic Calls to *panic* in library code, of which the user of this library code is not aware. 3–5

LLVM Low Level Virtual Machine. A collection of modular and reusable compiler and toolchain technologies used to develop compiler front ends and back ends. 9

macro Language construct that expands to regular code before actual compilation happens. They can be recognized by the trailing `!`. In contrast to functions, they are not compiled to lower level representations. They are also more flexible (having variable arguments) than regular Rust functions. 4, 7, 8, 27, 65

Option A type representing an optional value, an Option is either `Some` and contains a value, or `None` and does not. 6, 27, 65

panic Rust's way of saying that the program ended up in a unrecoverable state, which leads to a premature thread or program exit. v, 1, 4–9, 11, 12, 16, 17, 19, 24–28, 34, 37–41, 43, 61, 65

panic! The *macro* that forms the entry point for *panic* in Rust threads. 4, 5, 7–9, 11, 12, 16, 25, 28, 33, 37–40

Result A type that is either an `Ok` representing success and containing a value, or `Err` representing an error and containing an error message. 4, 6, 65

rustc The Rust compiler. 10, 16, 29, 30

rustup Rust toolchain installer. 30

unsafe Rust keyword to disable Rust's safety checks in particular code blocks. 40

unwrap Function that returns the value contained in an *Option* or *Result* if it represented `Some` or `Ok`, respectively. Unwrap *panics* whenever this is not the case and either `None` or `Err` is encountered. 4, 6, 27, 41, 43

Bibliography

[1] N. Matsakis and A. Turon, *The Rust Programming Language*, 2nd ed. Rust Lang., 2017. [Online]. Available: https://doc.rust-lang.org/book/second-edition/index.html

[2] "Error Handling," 2017. [Online]. Available: https://doc.rust-lang.org/book/second-edition/ch09-00-error-handling.html

[3] J. Caballero, G. Grieco, M. Marron, and A. Nappa, "Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities," *Proceedings of the 2012 International Symposium on Software Testing and Analysis - ISSTA 2012*, 2012.

[4] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "RustBelt: Securing the Foundations of the Rust Programming Language," in *ACM on Programming Languages, Vol. 2, No. POPL, Article 66*, 1 2018.

[5] "Rust 1.26.1 std::thread::spawn definition," 2018. [Online]. Available: https://github.com/rust-lang/rust/blob/master/src/libstd/thread/mod.rs#L535

[6] "Rust 1.26.1 std::result::unwrap definition," 2018. [Online]. Available: https://github.com/rust-lang/rust/blob/1.26.1/src/libcore/result.rs#L777

[7] "Rust 1.26.1 std::result::unwrap_failed definition," 2018. [Online]. Available: https://github.com/rust-lang/rust/blob/1.26.1/src/libcore/result.rs#L944

[8] "Rust 1.26.1 std::thread::spawn description," 2018. [Online]. Available: https://github.com/rust-lang/rust/blob/master/src/libstd/thread/mod.rs#L467

[9] "Rust 1.26.1 panic macro definition," 2018. [Online]. Available: https://github.com/rust-lang/rust/blob/1.26.1/src/libcore/macros.rs#L15

[10] M. W. Hall and K. Kennedy, "Efficient call graph analysis," *ACM Letters on Programming Languages and Systems (LOPLAS)*, vol. 1, no. 3, pp. 227–242, 1992.

[11] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[12] J. Law and G. Rothermel, "Whole program path-based dynamic impact analysis," in *Proceedings of the 25th International Conference on Software Engineering*. IEEE Computer Society, 2003, pp. 308–318.

[13] G. C. Murphy, D. Notkin, W. G. Griswold, and E. S. Lan, "An empirical study of static call graph extractors," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 7, no. 2, pp. 158–191, 1998.

[14] "Stability as a Deliverable," 2014. [Online]. Available: https://blog.rust-lang.org/2014/10/30/Stability.html

[15] Y.-F. Chen, E. R. Gansner, and E. Koutsofios, "A C++ data model supporting reachability analysis and dead code detection," *IEEE Transactions on Software Engineering*, vol. 24, no. 9, pp. 682–694, 1998.

[16] R. Jalan and A. Kejariwal, "Trin-Trin: Who's Calling? A Pin-Based Dynamic Call Graph Extraction Framework," *International Journal of Parallel Programming*, vol. 40, no. 4, pp. 410–442, Aug 2012. [Online]. Available: https://doi.org/10.1007/s10766-012-0193-x

[17] S. L. Graham, P. B. Kessler, and M. K. Mckusick, “Gprof: A call graph execution profiler,” in *ACM Sigplan Notices*, vol. 17, no. 6. ACM, 1982, pp. 120–126.

[18] N. Matsakis, “Introducing MIR,” 4 2016. [Online]. Available: https://blog.rust-lang.org/2016/04/19/MIR.html

[19] “Macro RFC,” 2018. [Online]. Available: https://github.com/rust-lang/rfcs/blob/master/text/1566-proc-macros.md

[20] “MIR RFC,” 2018. [Online]. Available: https://github.com/rust-lang/rfcs/blob/master/text/1211-mir.md

[21] “LLVM Language Reference Manual.” [Online]. Available: https://llvm.org/docs/LangRef.html#runtime-preemption-specifiers

[22] C. Lattner and V. Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation,” in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO’04)*, Palo Alto, California, 3 2004.

[23] International Institute of Business Analysis, *A Guide to the Business Analysis Body of Knowledge (BABOK Guide), Version 2.0*. International Institute of Business Analysis, 2009.

[24] “Rust Style Guide,” 2017. [Online]. Available: https://github.com/Rust-lang-nursery/fmt-rfcs/blob/master/guide/guide.md

[25] N. Matsakis and A. Turon, *The Rust Programming Language*, 2nd ed. Rust Lang., 2017. [Online]. Available: https://doc.rust-lang.org/book/second-edition/ch11-03-test-organization.html

[26] Bluss, “petgraph,” 2018. [Online]. Available: https://crates.io/crates/petgraph

[27] N. A. Quynh, m4b, R. Healey, and T. Finkenauer, “capstone,” 2018. [Online]. Available: https://crates.io/crates/capstone

[28] “The Open Source Definition.” [Online]. Available: https://opensource.org/osd

[29] K. S. Rubin, *Essential Scrum: A Practical Guide to the Most Popular Agile Process*, 1st ed. Addison-Wesley Professional, 2012.

[30] A. Bogus, G. Brandl, M. Goregaokar, M. Carton, and O. Schneider, “rust-clippy,” 2018. [Online]. Available: https://crates.io/crates/clippy

[31] N. Cameron, “rustfmt-nightly,” 2018. [Online]. Available: https://crates.io/crates/rustfmt-nightly

[32] M. Fowler, “Inversion of control containers and the dependency injection pattern,” 2004.

[33] E. Bouwers, J. Visser, and A. v. Deursen, “Getting What You Measure,” *Acm Queue*, 2012.

[34] “Test Organization,” 2017. [Online]. Available: https://doc.rust-lang.org/book/second-edition/ch11-03-test-organization.html

[35] United States Congress, “17 U.S.C. 1201 - Circumvention of copyright protection systems,” 1998. [Online]. Available: https://www.gpo.gov/fdsys/granule/USCODE-2011-title17/USCODE-2011-title17-chap12-sec1201

[36] European Parliament, Council of the European Union, “Directive 2009/24/EC,” 2009. [Online]. Available: https://eur-lex.europa.eu/legal-content/EN/ALL/?uri=CELEX:32009L0024

[37] J. R. van der Werven, *elf(5) Linux User’s Manual*, 09 2017. [Online]. Available: https://linux.die.net/man/5/elf

[38] DWARF Debugging Information Format Committee, *DWARF Debugging Information Format Version 4*, 6 2010. [Online]. Available: http://www.dwarfstd.org/doc/DWARF4.pdf

[39] S. Alvarez, “radare.” [Online]. Available: https://rada.re/r/