GhidRust: Rust decompiler plugin for Ghidra

Dhruv Maroo

Abstract

In this report, I outline my development process for the **GhidRust** plugin, a Rust decompiler plugin for Ghidra. I discuss the motivation for the project, the design of the plugin, and the challenges I faced during development. I also discuss the future of the project and the work that needs to be done to make it production ready. The development process required dabbling in multiple topics of computer science, including reverse engineering, compiler design, and software design. I also had to learn how to use the Ghidra API. I hope that this report will be useful to anyone who is interested in building their own plugin for Ghidra, or anything related. The end result of this project is a beta plugin for Ghidra, which has been open sourced and will be maintained and improved by me (and hopefully the community) in the future.

Motivation

Ghidra is a reverse engineering tool developed by the *National Security Agency* (*NSA*). It is a powerful tool that can be used to analyze binaries, disassemble and decompile code. It is also extensible, allowing users to write their own plugins to extend its functionality. The Ghidra team has already written a number of plugins, including a decompiler for C and C++ code. However, there is no decompiler for Rust code. This is a problem because Rust is a popular programming language, and it is used in many security-critical applications. The lack of a decompiler for Rust code makes it difficult to analyze Rust binaries. This is especially true for beginners, who may not be familiar with the Rust language.

Also, the strains of malware built using Rust are slowly increasing. This has already happened for other languages like *Go* and *Nim*. But at the same time, there are a lot of tools being developed to reverse engineer Go binaries. Ghidra plugins like *gotools* and *ghostrings* help in analyzing Go binaries. But there hasn't been any such development for Rust binaries. In fact, throughout my research phase, I am yet to find aa Ghidra plugin targeted towards Rust binaries. The lack of a reversing ecosystem for Rust may lead to more threat actors using languages like Rust to build hard-to-reverse malware.

Apart from this, Rust is becoming an increasingly common language, and is no longer limited to systems level applications. In fact, it has consistently been the most loved language according to Stack Overflow's developer survey. This means that there will be more and more important applications being written in Rust, which makes it imperative to have an equally flourishing reversing ecosystem for Rust binaries as well.

Goal

The goal is to have an integrated and user friendly Ghidra plugin which can be used out-of-the-box to analyze Ghidra binaries and decompile them to Rust. Apart from decompilation, various other analyses can also be performed on the Rust binaries to help in reverse engineering. Alongside this, it should ideally also expose an API which can be used by other tools to analyze Rust binaries. Many of these goals are long-term and will take long to complete, however, in my UGRC, I have been able to build a pretty solid base, upon which I can continue my work.

Features

I have concisely listed below the features that the plugin provides. Please also go through the Design section for each of these features to know more about the inner working, details and any "gotchas".

Auto-detection

The plugin can detect whether the binary is a Rust binary or not. This is more of a quick handy utility rather than a full-fledged feature. The user can navigate to `GhidRust -> Check if Rust binary` in the main toolbar. A popup will appear indicating whether the binary is a Rust binary or not.

This is also used to automatically trigger the following Rust analyzer, which runs only on Rust binaries.

Function Signature Analysis

This analysis is used to identify the functions in the binary using the bytes/instructions in the function, and then apply the saved function signatures corresponding to that function. This is done using Ghidra's **Function ID** database. The plugin matches the function instructions in the binary with the instructions of the functions in Rust's `std` package (packaged as a `libstd.so`). If it finds any matches, the signature of that function is applied to the binary.

This is extremely useful when analyzing a binary stripped of any debug symbols or function signatures. Without the Rust detection feature and this analysis, the user would not even guess that it could be a Rust binary. And even if they know it's a Rust binary, they will have to manually identify the functions in the binary and apply the function signatures.

This feature vastly improves the quality of decompilation as well, since the decompiler can make use of the function signatures to decompile the functions more precisely. Plus, this feature is easily extensible. One can add their own `.fidb` database and apply those signatures during analysis. This way, the plugin is not limited to a single architecture or binary version, since the users can just generate their own `.fidb` database and use it.

Rust Decompilation

This is the main goal of the plugin. The plugin uses the decompiled C code from Ghidra, and transpiles it to Rust code, and displays it in a new panel. If anything gets updated in Ghidra's decompiler window, the same updates also get reflected in the Rust decompiler panel. This makes the complete decompilation process very smooth and the seamless.

At this point, having decompiled Rust code is more of a syntactic sugar, since the transpiler is not very mature. But in later stages, having decompiled Rust could make the complete analysis process much easier. For example, the user can use the decompiled Rust code to search for certain patterns, or to identify certain functions. Combined with the analyzer, this forms the staple of Rust binary analysis.

Macro Building

This feature is a work-in-progress right now, however, it is noteworthy. This allows the plugin to go through the decompiled Rust code and look for patterns which match Rust macro expansions. Rust macros are very common. In fact, the Rust *"Hello, world!"* contains the macro `println!`. The goal is to identify the macro expansions and replace them with the macro (which are more often used). This will make the decompiled Rust code much more readable and understandable.

Design

Below, I outline design choices and decisions that went behind implementing and integrating the features mentioned above.

Auto-detection

This requires detecting Rust binaries using the information in them. Binary detection techniques differ from language to language (and even compiler to compiler). For example, Go binaries have a `.gopclntab` section in their binary which can be used to detect them. Similarly, binaries may also have compiler information which can be used to deduce the language used to program them.

There is no standard way to detect Rust binaries, however, there are a few common patterns that can be used to detect Rust binaries. One pattern used by Mandiant's capa (malware detection tool) is to look for strings in the `.data` section of the binary. Specifically look for:

- `"run with ‘RUST_BACKTRACE=1‘ environment variable"`
- `"called ‘Option::unwrap()‘ on a ‘None‘ value"`
- `"called ‘Result::unwrap()‘ on an ‘Err‘ value"`

I have also used this method. This method is fairly reliable since Rust binaries are coupled with `libstd` and the above strings are part of the data used by `libstd`. Plus, since we are analyzing the `.data` section, it doesn't matter if the binary is stripped. Therefore the method even works for stripped binaries. The only way to bypass this check is to use some data packer and modify `libstd` to use that packer. This method can have false positives in the cases where the binary is not a Rust binary, but still has these strings in its `.data` section. This is very unlikely to happen without intention, however.

Function Signatures Analysis

Function Signatures

Storing function signatures is a very common technique in reverse engineering. Function signatures are stored and then later applied where required so as to save time and get precise signatures. Different reverse engineering tools have different methods to store function signatures. For example, Ghidra uses a `.fidb` database to store function signatures. IDA Pro uses F.L.I.R.T. function signatures. Radare2 (and Rizin, Cutter, Iaito) uses a `.sdb` database to store function signatures.

F.L.I.R.T.

Out of these, **F.L.I.R.T.** is the most sophisticated. It stands for *Fast Library Identification and Recognition Technology*. It takes nto account instruction reordering, different registers and a lot of other factors, therefore making it's pattern matching more extensive. Therefore, it is allowed to detect functions even if some instructions have different registers, or they have been moved around. There are also scripts which allow using F.L.I.R.T. signatures in Ghidra. There is only one drawback, however. The F.L.I.R.T. format is proprietary. Even though it has been reversed, dealing with it requires access to proprietary tools. This is the exact reason why it is not a good choice, because this would make it harder for other people to extend the function signature database by adding their own signatures or creating an entirely new one from using a different library.

Function ID

This is why we stick to Ghidra's Function ID. Function ID is a rudimentary format compared to F.L.I.R.T. But it has seamless integration with Ghidra, since it is a native format. Plus it's also open source, and very easy to generate and manipulate. Therefore, it is a good choice for this project. The only drawback is that it is too rigid. It cannot match patterns where some instructions are moved around or some registers are different. This reduces the number of functions it is able to detect. Plus, this also makes the detection compiler and arch dependent. A function ID database for one architecture and compiler may not work for a different architecture and compiler. This is exactly why it it important to give users the ability to use their own databases.

`RustStdAnalyzer`

This analysis is done by the `RustStdAnalyzer`, which comes with the plugin. It is an analyzer, which is supposed to be run when the binary is loaded. The job of this analyzer is to add the Rust library (`libstd`) functions to the function ID database. This analyzer also checks whether the binary is a Rust binary (using the same logic as auto-detection), and is only activated for Rust binaries. It is enabled by default for Rust binaries.

The Function IDs are managed using the `FidFileManager` class in Ghidra API. Users can add more `.fidb` files, which would then be added for Function ID analysis when they use the extension.

Generating a Function ID database is fairly easy, and can be done using the Function ID plugin. I have provided the Function ID database for Rust's `libstd` on x86-64 for Rust version 1.58.0 in the project. If users want, they can annotate functions signatures and generate a Function ID database which can then be later used to restore those annotations.

One small improvement here could be to provide a script which takes in an object file and generated the function ID database. This sort of automation would make the Function ID database generation process even easier.

Rust Decompilation

Idea

The general idea is to somehow capture the decompiled C code and then transpile it to Rust code and display it. This has to be well integrated so that it feels more like a built-in feature rather than a cranky extension.

Extending the Decompiler

I initially intended to use the Ghidra's decompiler and show Rust code instead of teh C code generated by Ghidra. However, this was not possible. The decompiler is not exposed in Ghidra's API, and therefore cannot be used/modified directly by extensions. The second option was to inherit the `DecompilePlugin` class, along with the other provider and component classes, and override the decompilation callback to return Rust code instead of C code. But such sort of inheritance is not possible, because many of the required fields needed to override functions are private to Ghidra's decompiler package. The only way to access them is to use very hacky and shabby reflection, which is definitely not a good idea. Figuring out the best way to do this took a lot of time, since the Ghidra's Decompiler source is not very well documented since it was never meant to be exposed to the users. After a lot of trying and failing, I had to rule out the above alternatives.

Separate Decompiler

In the end, I went ahead with a simple `JTextArea` showing the Rust decompiled output. This is not ideal, but it is a good balance between what is "allowed" and what is desired. The decompiler window integrates very well with the UI. Also any updates in the decompiled C code also get reflected in the Rust decompiled code. It is as close to native as it can get. The only drawback is that it is not a part of the decompiler, and therefore cannot be used to annotate/modify the decompiled Rust code like it is possible with the C code. It also lacks syntax highlighting. Adding syntax highlighting would either require writing a custom highlighter or packaging libraries which are pretty heavyweight for the required task. This is something that can be added in the future.

`DecompInterface`

The above decompilation is done using the `DecompInterface` class exposed by the Ghidra API. It spawns a decompiler process and makes sure to respawn the decompile process in case of any crashes. We use this interface to get the decompiler C code, which can then be used by the transpiler to get the decompiled Rust code.

Grammar

The transpiler consists of a C grammar, which is used to parse the decompiled C code and subsequently convert it to Rust code. The C grammar used is very extensive in regards to the decompilation output which Ghidra usually generated, but that isn't an issue.

Parser

Now once we have the grammar, we need to choose a parser generator. Two popular choices are `JavaCC` and `ANTLR4`, but `ANTLR4` is very bloated compared to `JavaCC`, hence I went ahead with `JavaCC`. Now we need to generate the parse tree so that it can be used to visit all the nodes and convert them to Rust code. Initially I went ahead with JTB by UCLA. I later realized that it is not a very widely known tool. This is problematic since it would increase the barrier for other people to contribute and improve the plugin. I also got to know that `JavaCC` comes along with a tree generator known as `JJTree`. Hence, removing JTB would result in lesser dependencies but the same functionality.

Transpiler

Once the parser and code for the tree (with all it's nodes) has been generated, we can use the *visitor pattern* to visit all the nodes and convert emit equivalent Rust code at every node. This requires having a type conversion mapping between both the languages which would be used to convert C types to equivalent Rust types. As of now, the type mapping is very trivial and only exists for primitive types. This can be improved in the future.

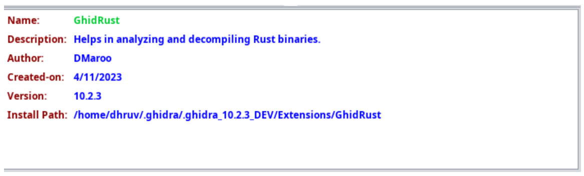
Also, the generated Rust code should prioritize readability over preciseness. It should be semantically correct, but being smart about the code conversion can help generate pretty Rust code. Right now, there are no specific techniques being employed for this, but this is a goal for the future.

Macro Building

This is not a trivial process. In fact, extracting macros is an undecidable problem since the mapping from the macro inputs to macro output is many-to-one. However, trivial macro extraction is feasible. The idea is to extract the macro definition and the macro call, and then replace the macro call with the macro definition. This requires, again, using a parser to parse the generated Rust code, see where a macro pattern occurs, and then replace it accordingly. The hard part here is to make it general and easily extensible. It is impractical to write code for every new macro which needs to be added. There needs to be a mechanism in place to generate macro identification code just from he macro expansion. I am yet to find a good way to this, however, it is indeed possible (and has been done in the project for `println!` macro as a proof-of-concept).

Plugin In Action

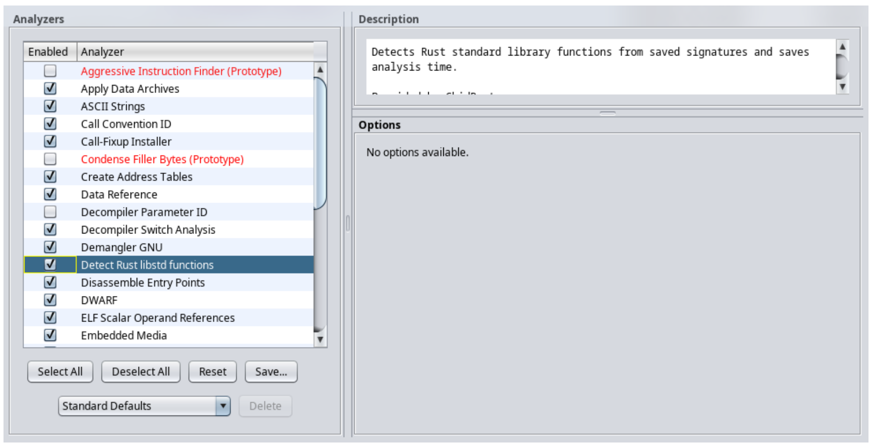
Plugin Description



Name:	GhidRust
Description:	Helps in analyzing and decompiling Rust binaries.
Author:	DMaroo
Created-on:	4/11/2023
Version:	10.2.3
Install Path:	/home/dhruv/.ghidra/.ghidra_10.2.3_DEV/Extensions/GhidRust

Figure 1: GhidRust in extension menu

Analyzer Menu



Analyzers

Enabled	Analyzer
☐	Aggressive Instruction Finder (Prototype)
✓	Apply Data Archives
✓	ASCII Strings
✓	Call Convention ID
✓	Call-Fixup Installer
☐	Condense Filler Bytes (Prototype)
✓	Create Address Tables
✓	Data Reference
☐	Decompiler Parameter ID
✓	Decompiler Switch Analysis
✓	Demangler GNU
✓	Detect Rust libstd functions
✓	Disassemble Entry Points
✓	DWARF
✓	ELF Scalar Operand References
✓	Embedded Media

Description

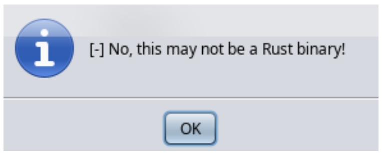
```
Detects Rust standard library functions from saved signatures and saves
analysis time.
```

Options

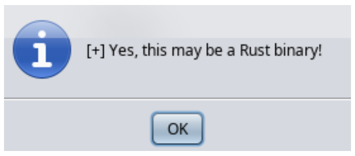
No options available.

Figure 2: GhidRust in analyzer menu

Auto-detection

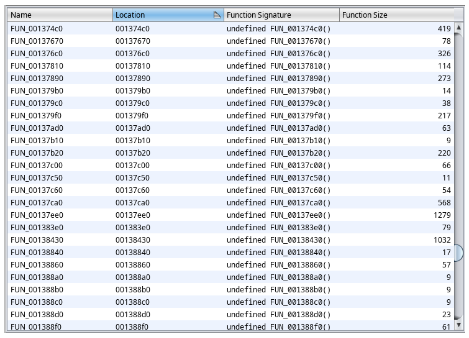

[-] No, this may not be a Rust binary!

(a) Not a Rust binary


[+] Yes, this may be a Rust binary!

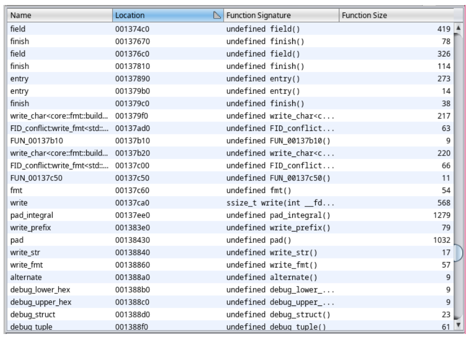
(b) Rust binary detected

Function ID



Name	Location	Function Signature	Function Size
FUN_001374c0	001374c0	undefined FUN_001374c0()	419
FUN_00137670	00137670	undefined FUN_00137670()	78
FUN_001376c0	001376c0	undefined FUN_001376c0()	326
FUN_00137810	00137810	undefined FUN_00137810()	114
FUN_00137890	00137890	undefined FUN_00137890()	273
FUN_001379b0	001379b0	undefined FUN_001379b0()	14
FUN_001379c0	001379c0	undefined FUN_001379c0()	38
FUN_001379f0	001379f0	undefined FUN_001379f0()	217
FUN_00137ad0	00137ad0	undefined FUN_00137ad0()	63
FUN_00137b10	00137b10	undefined FUN_00137b10()	9
FUN_00137b20	00137b20	undefined FUN_00137b20()	220
FUN_00137c00	00137c00	undefined FUN_00137c00()	66
FUN_00137c50	00137c50	undefined FUN_00137c50()	11
FUN_00137c60	00137c60	undefined FUN_00137c60()	54
FUN_00137ca0	00137ca0	undefined FUN_00137ca0()	568
FUN_00137ee0	00137ee0	undefined FUN_00137ee0()	1279
FUN_001383e0	001383e0	undefined FUN_001383e0()	79
FUN_00138430	00138430	undefined FUN_00138430()	1032
FUN_00138840	00138840	undefined FUN_00138840()	17
FUN_00138860	00138860	undefined FUN_00138860()	57
FUN_001388a0	001388a0	undefined FUN_001388a0()	9
FUN_001388b0	001388b0	undefined FUN_001388b0()	9
FUN_001388c0	001388c0	undefined FUN_001388c0()	9
FUN_001388d0	001388d0	undefined FUN_001388d0()	23
FUN_001388f0	001388f0	undefined FUN_001388f0()	61

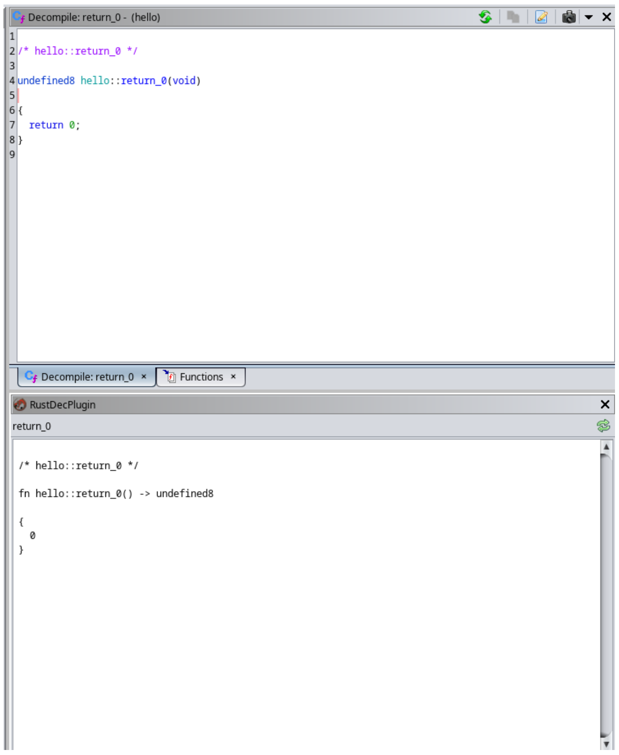
(a) Without using Function ID database



Name	Location	Function Signature	Function Size
field	001374c0	undefined field()	419
finish	00137670	undefined finish()	78
field	001376c0	undefined field()	326
finish	00137810	undefined finish()	114
entry	00137890	undefined entry()	273
entry	001379b0	undefined entry()	14
finish	001379c0	undefined finish()	38
write_char<core::fmt::build...	001379f0	undefined write_char<c...	217
FID_conflict:write_fmt<std::...	00137ad0	undefined FID_conflict...	63
FUN_00137b10	00137b10	undefined FUN_00137b10()	9
write_char<core::fmt::build...	00137b20	undefined write_char<c...	220
FID_conflict:write_fmt<std::...	00137c00	undefined FID_conflict...	66
FUN_00137c50	00137c50	undefined FUN_00137c50()	11
fmt	00137c60	undefined fmt()	54
write	00137ca0	ssize_t write(int __fd...	568
pad_integral	00137ee0	undefined pad_integral()	1279
write_prefix	001383e0	undefined write_prefix()	79
pad	00138430	undefined pad()	1032
write_str	00138840	undefined write_str()	17
write_fmt	00138860	undefined write_fmt()	57
alternate	001388a0	undefined alternate()	9
debug_lower_hex	001388b0	undefined debug_lower_...	9
debug_upper_hex	001388c0	undefined debug_upper_...	9
debug_struct	001388d0	undefined debug_struct()	23
debug_tuple	001388f0	undefined debug_tuple()	61

(b) With Function ID database

Decompiler Plugin



```

/* hello::return_0 */

undefined8 hello::return_0(void)

{
  return 0;
}

```

```
/* hello::return_0 */

fn hello::return_0() -> undefined8

{
  0
}
```

Figure 5: Decompiler plugin with proof-of-concept function

Challenges

No Prior Work

There are only a handful of proper Ghidra plugins which can be used as a reference for building a Ghidra plugin. Adding onto this, there are no plugins which directly interact with/extend the Decompiler. And to make it even more difficult, there are no Rust plugins for Ghidra. This is a challenge in itself, since it requires a lot of trial and error to figure out how to do things. Even if we ignore the decompilation and the Rust part of the plugin, still getting it to work was a task at first. And finding the solutions to problems is hard because there aren't enough people developing Ghidra plugins.

Decompiler

As explained above, the decompiler internals are not very well documented and no API is exposed. Therefore all my efforts of trying to extend it were futile and cost me a lot of time. In the end, I found a middle ground with `DecompileInterface`, but it took a fair amount of time to reach this solution.

Switching from JTB to `JJTree`

I was familiar with JTB because I had seen it being used when I was learning compilers. However, I later realized that `JJTree` is a better choice. But getting familiar with `JJTree` wa time consuming (and frankly somewhat frustrating). The documentation is *extremely* sparse. But I was able to get it to work in the end, which was a relief.

Rust Macros

Finding out the correct patterns to match for Rust macros is not as simple as just seeing the macros' source code. This is because Rust macros are very complex and their source cannot be directly used to deduce the pattern used for macro detection. One has to manually convert every macro to a pattern. Plus, having no official specification for Rust's grammar does not help either.

Other Relevant Work

Rust Demangler

A **demanlger** extracts function names (and signature) from mangled symbols.

I implemented a Rust demangler for *Rizin*. I have been a long-time contributor to the project, and it did not have a Rust demangler. So I went ahead and implemented it. The relevant commit is: `rizinorg/rz-libdemangle@11b3f28`.

The reason why I did not integrate this in Ghidra is because Ghidra's current demangler is good enough and works well for Rust binaries. However, it is definitely a good idea to integrate this in Ghidra in the future.

Learnings

Ghidra Internals

I got to learn a lot of things about how Ghidra has been designed. The design is very modular and extensible. I learnt about P-code, SLEIGH, analyzers, plugins, Ghidra scripts and so on. Ghidra's API and ecosystem is very rich and is a good example of object-oriented programming done right.

Rust Macros

I also got to learn a lot of new things about Rust macros, how they are processed, what is allowed, what isn't, and so on.

`JJTree`

After putting in a lot of effort, I got to learn how to `JJTree` to generate trees (and optionally visitors), and how to then iterate through those trees using the visit-accept calls.

Future Plans

Speeding up the Decompiler

One of the most common issues while testing was to wait for Ghidra to finish the decompilation every time when analyzing a Rust binary. The reason why Ghidra takes up to minutes to analyze Ghidra binaries is because they are bundled with functions from Rust's `std`. To reduce the decompilation time, we can skip the functions which belong to `libstd`. This would speed up the analysis immensely. I spent a good amount of time trying to figure out how to do this, but there seems to be no trivial way to do so. A concept similar to Function IDs can be incorporated, but it would require significant work. This is something I would like to work on in the future.

Better Decompilation

Currently, the decompiler possesses the capability to decompile only simple language constructs. There is no support for compound types like Rust `struct`, `union`, `enum`, etc. There is also no support for `traits` and other functional programming and objective programming constructs. This is something which would be desired, but it would require tons of work to get it right.

Macro Extraction

Macro extraction is a very important feature which is currently not mature at all. This is essential, and although difficult, it is feasible. After macro extraction the Rust decompiled code would look prettier than Ghidra's C++ decompiled code, which is a great motivator for going forward with this.

Improving UI/UX

The decompiler window isn't very user-friendly or pretty. It is quite static and there is no way to interact with the code. It would be nice to have a better UI for the decompiler. This would make it easier to use and would also make it more appealing to the users.

Open-source

The project is currently on GitHub, which means people can contribute to it. But as of now, it is not completely ready for handling any contributions. There is no continuous integration and continuous delivery system in place. There is very sparse documentation. There are no contributor guides. I plan to make it into a decent piece of open source software, which can be used by everyone and contributed to by everyone. One of the main motivations for the project was that it wouldn't just be a one-time thing, but would be a long-term project which would be useful to other people as well.

The goal is for GhidRust to be the standard go-to tool for any sort of Rust reversing/decompilation. This would take time and effort, but once it is done, it would be a great achievement. This projects has prepared the base for it, now we just need to build upon this.

Code Repository

All the code for the project can be found at DMaroo/GhidRust. The code is licensed under the MIT license. I plan to keep on working on this project even after the UGRC ends.

Acknowledgement

I would like to thank my mentor, prof. Chester Rebeiro, for his guidance and feedback on my ideas. His feedback, support and advice has been invaluable. I would also like to thank my faculty advisor, prof. Narayanaswamy, for accepting my UGRC proposal and allowing me to work on this project as my UGRC.

Relevant links

- The proposal for building the Rust decompiler as my undergraduate research project.
- The source code can be found on GitHub at DMaroo/GhidRust.
- Ghidra's source and website.
- The Rust programming language's website.