Macro-Driven Virtualization of Rust Binaries

by

Ian Tatum

A thesis presented to the Honors College of Middle Tennessee State University in partial fulfillment of the requirements for graduation from the University Honors College

Fall 2024

Thesis Committee:

Dr. Arpan Sainju, Thesis Director

Dr. Philip Phillips, Thesis Committee Chair

Macro-Driven Virtualization of Rust Binaries

by Ian Tatum

APPROVED:

Dr. Arpan Sainju, Thesis Director

Assistant Professor, Computer Science

Dr. Philip Phillips, Thesis Committee Chair

Associate Dean, University Honors College

Abstract

This thesis presents a proof-of-concept macro-driven virtualization-based obfuscator for the Rust programming language. The implementation demonstrates how macro systems can be leveraged to achieve code obfuscation through virtualization transformations, providing a novel approach to software protection specifically targetting Rust. In addition to the working prototype, this thesis proposes an advanced obfuscation design that, while currently constrained by Rust's compiler, outlines a path for future implementation once the compiler evolves to support the necessary features. The research contributes both a practical demonstration of Rust obfuscation techniques and a theoretical framework for more sophiscated protection mechanisms, providing a blueprint for future work.

Table of Contents

List of Figures

List of Terms

Reverse Engineering The process of recovering the design and function of a product through deductive reasoning by analyzing code or program behavior.

Source Code A human readable piece of text written in a programming language that specifies the behavior of a computer.

Compile-Time The point at which a piece of source code is turned into machine code that a computer can understand.

Instruction Set Architecture The "language" of a processor, the interface between hardware and software.

Abstract Syntax Tree A tree representation of a programming language's syntax that is mainly used in compilers.

Intermediate Representation A data structure used internally by a compiler to represent source code in a generic way.

Virtual Machine Software-based emulation of a computer system that runs programs as if they were executing on actual hardware.

Primitive Type A type built-in to the language.

Foreign Type A type composed of primitive types, not built-in to the language.

Trait A trait defines the functionality a particular type has and can share with other types.

1 Introduction

With the rise of reverse engineering and piracy efforts, the software industry has become increasingly concerned with protecting their property. Consequently, programmers have turned to obfuscation as a mitigation and deterrent against such threats. *Obfuscation* describes the process of altering a program with various transformations to thwart and complicate program analysis. These transformations affect segments or the entirety of a program, reshaping it into a convoluted and less intelligible form, all while preserving its input-output behavior [1]. Such transformations come at a cost of increased program runtime and binary size. As such, obfuscation is often only applied to critical sections of a program. Despite the overhead, obfuscation is actively used in various domains, such as intellectual property protection and digital rights management. For instance, it can commonly be found in license verification components of commercial software to deter potential attackers. One of the most promising approaches to obfuscation is Virtual Machine (VM)-based transformations, which embed an interpreter with a *custom* instruction set architecture (ISA) into the binary [2].

Obfuscation can occur through either source-to-source or binary-to-binary transformations. Source-to-source transformations, formally known as *transpilers*, convert source code into equivalent code in the same or different language. Binary-to-binary transformations involve modifying and rewriting existing binaries.

In this paper, we introduce a novel approach to code obfuscation through source-to-source transformations through the use of *macros*-a powerful mechanism that many programming languages provide for source-to-source transformations. They work by taking in source code and producing modified or expanded source code, a technique known as *metaprogramming*. Our implementation leverages Rust, a programming language renowned for its powerful macro system and accompanying libraries that empower metaprogramming. By leveraging Rust's existing tooling, we are able to perform virtual machine-based obfuscation at compile-time.

2 Technical Background

2.1 Rust

Rust is a systems programming language designed for performance, memory safety, and concurrency. It was first developed in 2006 by Graydon Hoare at Mozilla Research and officially released in 2015. [3] Rust's primary goals are to eliminate common programming bugs such as memory corruption and data races while maintaining performance similar to C and C++. It achieves this through a unique ownership and borrowing model, termed the 'borrow checker', which ensures memory safety without a garbage collector.

In recent years, memory safety has become a topic of increasing concern, as nearly seventy percent of software bugs are attributed to memory safety issues. [4] These bugs often result in severe security vulnerabilities, such as buffer overflows, use-after-free errors, and data races, which can lead to system crashes or exploits. In light of these concerns,

The White House has highlighted the importance of memory-safe programming languages for the future of secure and reliable software, endorsing the use of languages like Rust for future development. [5]

Rust addresses these critical issues by enforcing strict rules on ownership, borrowing, and lifetimes at compile time. This ensures that memory is automatically managed without the need for a garbage collector, eliminating the runtime overhead typically seen in other languages. Through these guarantees, Rust prevents many common types of memory-related errors, making it an ideal choice for building secure high-performance software.

As a result of its emphasis on safety, speed, and reliability, Rust has gained significant traction in both the open-source community and industry. Major companies like Microsoft, Cloudflare, and Amazon have adopted Rust for various projects, particularly where performance and security are paramount. [3] For our paper, we focus on three features of Rust: compile-time evaluation, macros, and the trait-system.

2.1.1 Constant/Compile-Time Evaluation

Constant evaluation refers to the process of evaluating an expression during the compilation phase rather than at runtime. [6] In Rust, this is achieved through the const keyword, which allows developers to define values and even functions that are computed during compilation, before the program starts executing.

In Rust, the `const` keyword is used to create values or expressions that are evaluated at compile time. [6] These constants can be simple values, such as numbers or strings, but they can also be more complex, involving function calls or algorithmic computations.

Rust allows `const` functions, which are functions that can be invoked during compile-time evaluation to calculate a value.

In our work, we leverage constant evaluation to generate bytecode and entirely unique virtual machines at compile time. Instead of computing these values or constructing a virtual machine dynamically at runtime, which reveals to an attacker the original code, we perform all the necessary computations during the build process. By doing so, we replace the original unobfuscated code with bytecode that only the virtual machine understands at compile-time, thereby making it obfuscated.

2.1.2 Macros

In our work, Rust's macro system plays a critical role in enabling us to modify and generate bytecode at compile-time. Rust offers two primary types of macros: declarative macros and procedural macros, each with distinct capabilities and varying levels of flexibility.

Declarative macros are used to define patterns that can be expanded at compile time, reducing redundancy and making the code more concise. [7] These macros rely on pattern matching to generate code based on the input they receive. A common example of a declarative macro is the `println!` macro, which simplifies printing formatted text by automatically expanding into the appropriate code to handle output. [7] By allowing developers to write patterns that are expanded at compile time, declarative macros make repetitive tasks much more manageable.

Procedural macros, in contrast, offer greater power and flexibility. They allow developers to define custom syntax transformations by writing functions that accept a token stream as input and produce new code as output. Procedural macros can be used for a variety of purposes, including the automatic generation of code. One popular example of procedural macros is derive macros, which automatically implement common traits for structs or enums, eliminating the need for developers to write boilerplate code.

For our specific implementation, we leverage procedural macros to obfuscate functions simply by annotating them, as demonstrated in Figure 1. This approach enables us to seamlessly modify code without turning to a custom compiler or build system. By utilizing macros, we can achieve powerful obfuscation transformations while maintaining full compatibility with Rust's ecosystem, allowing us to better garner adoption.

```
#[virtualize]
fn add_together(a: i32, b: i32) -> i32
```

Figure 1: An example of an annotated function

2.2 Obfuscation

Obfuscation is a technique used to make software difficult to understand or reverse-engineer, often employed as a security measure to protect intellectual property, prevent tampering, or thwart unauthorized analysis. The core principle behind obfuscation is the transformation of code into a form that still functions the same way but is extremely hard to interpret or analyze. This can be achieved through a variety of methods, from simple

code restructuring to more complex techniques like packing [8], control flow obfuscation [9], and the use of virtual machines [1], [2]. The goal is to obscure the underlying logic of the code, making it more difficult for attackers, reverse engineers, or unauthorized users to comprehend or modify it.

2.2.1 VM-based Obfuscation

Virtual machine-based obfuscation [1], [2], [10], [11], [12], often referred to as virtualization, can be conceptually explained using the analogy of cooking recipes. In this analogy, the recipe is like code, as both consist of a sequence of instructions designed to achieve a particular result, such as a finished dish or a computational output.

For instance, imagine that we are the creators of a recipe. The recipe contains a list of instructions that must be followed in sequence to produce the dish (or output). To prevent others from easily understanding or copying our recipe, we develop a new language that only we understand. This language acts as an obfuscation tool, similar to how virtualization disguises the original code.

Once the new language is developed, we translate our original recipe into this secret language. The obfuscated recipe is now harder to understand on its own, much like how obfuscated code cannot be easily interpreted by an outsider. We then distribute this obfuscated recipe to our customers, much as virtualized code is distributed to users.

However, our customers cannot make sense of the obfuscated recipe without the right tool. Similar to how a chef would need an interpreter to understand the obscure recipe, customers need an interpreter that can read and execute the obfuscated instructions. The

interpreter is packaged along with the recipe to ensure that customers can follow the instructions and achieve the desired result, just as the virtual machine allows obfuscated code to execute.

If an attacker wanted to reconstruct the original recipe, they would face the difficult task of reverse-engineering both the obfuscated recipe and the interpreter. This process is highly complex and time-consuming, making it challenging for attackers to access the original code, just as they would struggle to recreate the original recipe without the interpreter.

Figure 2 below shows a visual representation of the translation process, illustrating how the original recipe is obfuscated into a language that only the creator understands. Similarly, Figure 3 provides a visual explanation of how the obfuscated recipe is distributed to customers, along with the interpreter, to enable them to execute the recipe and obtain the final result.

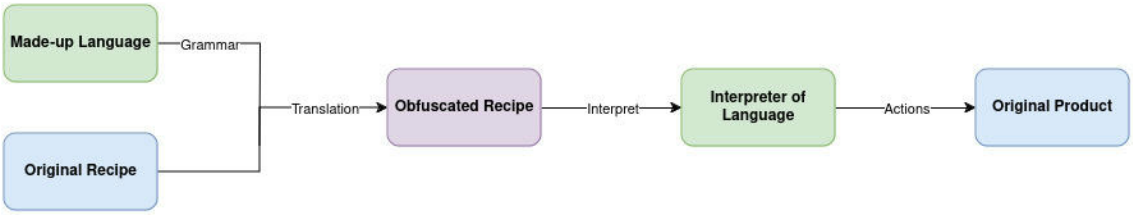

Figure 2: A visual explanation of the translation process

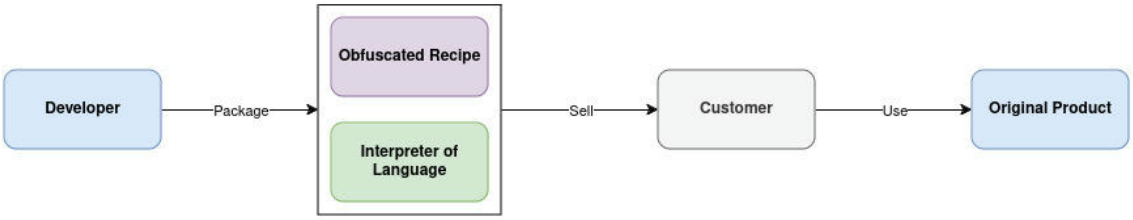

Figure 3: A visual explanation of the selling process

2.3 State of the Art

The current state-of-the-art virtual-machine based obfuscators that perform source-to-source transformations often rely on compiler frameworks like Low Level Virtual-Machine (LLVM) or develop their own custom compilers [13], [14]. The reasoning for this lies in the leverage afforded by the compiler's intermediate representation (IR) of source code. This intermediate representation allows for the creation of generic obfuscation passes, operating on the intermediate representation rather than the source code itself [13]. Furthermore, it facilitates the use of a wide-breadth of obfuscation techniques, not limited to virtual machine-based transformations.

However, when solely looking at the prospect of writing a virtual machine-based obfuscator, this level of abstraction is unnecessary. Our further proposed work could perform the same transformations while eliminating the need for a custom toolchain, potentially increasing usability and adoption among the Rust community. Until now, there has not been any obfuscator for any programming language that has taken our approach.

3 Design

3.1 Core Concepts

For our design, we present one practical demonstration to prove the underlying core concepts of our approach and one theoretical design for future work where the compiler is more capable. The proof-of-concept obfuscator was designed to prove three core concepts: the macro-based traversal and modification of Rust AST, the compile-time generation of

bytecode, and the execution of said bytecode at run-time in a virtual machine. These three concepts are the basis of our proposed future work and gives credence to our theoretical design.

3.2 Architectural Overview

The core architecture of our obfuscator consists of three key components:

- **Virtual Machine**: This module is responsible for representing the possible operations able to be interpreted and dictates the structure of our bytecode. It is expected to efficiently execute bytecode and produce the expected program result.
- **Compile-time Bytecode Generator**: This module is responsible for generating the respective bytecode of our virtual machine at compile-time and exposes an interface for the AST transformer to generate bytecode that preserves the semantics of the original program.
- **AST Transformer**: This module is responsible for traversing and modifying the Rust AST during the compilation process. It uses Rust's macro system to hook into the compilation process and make the necessary transformations to the program's structure. The transformations performed are designed to translate a subset of Rust syntax into calls to our bytecode generation module while retaining original program logic and functionality.

3.3 Proof-of-Concept and Limitations

In an effort to demonstrate the feasability of our core architectural concepts, we designed our proof-of-concept obfuscator to be a simple stack-based virtual machine.

Moreover, we chose to restrict the virtual machine to only support a handful of operations which simply allocate and add numbers together. The rationale for this was to focus on proving the underlying technology of our theoretical design, compile-time bytecode generation and proper AST transformation. The virtual machine itself is interchangable and not limited by constant evaluation and is seen as a trivial problem for the purpose of this paper.

Our main design goal was to provide an interface for generating bytecode to our AST transformer. However, in exploration of designs, a glaring limitation was spotted in the rust compiler - the lack of compile-time evaluated traits. Traits are used by the Rust compiler to implement mathematical and logical operations between primitive and foreign types, so it is only natural to try and extend this to our own design. By leveraging traits, we could push off type-checking to the compiler through trait bounds and allow the compiler to generate the correct bytecode for an operation rather than manually determining it.

Despite, this limitation it is possible to emulate traits with constant evaluated functions, but it comes with boilerplate and one limitation. The boilerplate highlights why we chose to restrict the set of operations in our virtual machine, mainly in the interest of our reader. As for the limitation, it requires that any obfuscated function must annotate any variables with their type as we cannot off-load it to the compiler and we must manually determine the right function. This requires our design to generate different functions for each type. To illustrate the boilerplate and limitation, we provide an example of allocating multiple primitive types in our virtual machine comparing both approaches.

```rust
// Emulating trait-like functionality
const fn valloc_u8(v: u8) -> [Opcode; 1] { // implementation }
const fn valloc_u16(v: u16) -> [Opcode; 2] { // implementation }
const fn valloc_u32(v: u32) -> [Opcode; 3] { // implementation }

// Trait-based solution
trait VAlloc {
  type OpcodeArray;
  fn valloc(self) -> OpcodeArray;
}
impl VAlloc for u8 {
  type OpcodeArray = [Opcode; 1];
  fn valloc(self) -> Self::OpcodeArray { // implementation }
}
```

Beyond the boilerplate and limitations to emulate a trait-based solution, we also designed a simple macro to transform our annotated functions into bytecode. This was accomplished by utilizing *syn* and *quote* which provide ways to traverse, manipulate, and create Rust AST on the fly easily [15], [16].

3.4 Theoretical Design

Building upon what we already discussed, our theoretical design could leverage compile-time evaluated traits to achieve a better solution. This design would call for all operations in the virtual machine to be implemented as a trait. For instance, there would be a trait for addition, subtraction, exclusive or, and even comparisons. By utilizing a trait-

based approach, it does not have the added limitation of explicitly typing variables. The compiler will infer types and make subsequent inference on which trait definition to call based on the inferred type(s).

The following example illustrates the granular level of control that a trait-based solution provides. This approach is not exclusively limited to primitive types, but could just as easily be implemented on foreign types created by others. However, there is a minor limitation with this approach, any type implementing these operations must be copyable. This is a limitation imposed by the Rust compile-time interpreter and could be lifted in the future.

```
// The exact syntax of compile-time evaluated traits has not been settled
and this is subject to change.
trait VAdd<Rhs : Copy = Self> : Copy {
  type OpcodeArray;
  const fn vadd(self, rhs: Rhs) -> OpcodeArray;
}
impl VAdd for u32 { // implementation }
impl VAdd<u8> for u32 { // implementation }
fn main() {
    1u32.vadd(2u32); // valid
    1u32.vadd(2u8); // valid
    2u8.vadd(1u32); // invalid as VAdd<u32> is not implemented for u8
}
```

4 Implementation

```
const fn concat<const N: usize, const M: usize>(
    lhs: [Opcode; N],
    rhs: [Opcode; M],
) -> [Opcode; N + M] {
    let mut result = [Opcode::NOP; N + M];
    let mut i = 0;
    while i < N {
        result[i] = lhs[i];
        i += 1;
    }
    i = 0;
    while i < M {
        result[N + i] = rhs[i];
        i += 1;
    }
    result
}
```

For our proof-of-concept implementation, we aimed to keep it simple and concise. We introduced a singular attribute macro called `virtualize` that operates on a singular function. It is responsible for all AST traversal and modification, it walks the AST of the function and translates any relevant operations. In this case, the relevant operations were

just variable declarations and addition operations between variables. However, this logic can be expanded to any operation if supported by the underlying virtual machine.

As for our compile-time bytecode generation, we utilized the Rust compiler's constant generics and evaluation to add arrays of bytecode together as seen in our `concat` function. This requires some size annotations for our emulated-trait solution, but this could be inferred away with a trait-based solution further reducing the complexity.

5 Discussion

Our proposed approach to VM-based obfuscation using Rust macros presents a promising future for enhancing Rust software security. The ability to perform transformations at compile-time through function annotation allows developers to incorporate obfuscation seamlessly into their workflows without the need for extensive modifications to existing toolchains. This not only promotes usability but also increases the adoption of obfuscation techniques across the Rust community.

One of the key benefits of our method is the flexibility provided by Rust's macro system, which enables the creation of powerful metaprogramming constructs. By leveraging macros, we can traverse and modify the Rust AST to generate bytecode at compile-time. This capability allows us to maintain the integrity of the original program while transforming it into a less recognizable form.

However, several challenges and limitations remain. The boilerplate associated with emulating trait-like functionality is a significant barrier, particularly as it complicates the

AST transformer and restricts the usable types to those implemented. Additionally, the current limitation of requiring explicit type annotations for obfuscated variables introduces extra complexity at the burden of the user.

Future work should focus on addressing these limitations, particularly by exploring the full implementation of compile-time evaluated traits. If successfully integrated, this would allow for type inference and reduce boilerplate, making our obfuscator more powerful and user-friendly.

Moreover, there is further room for exploration into a register-based virtual machine which we did not explore in this paper. The compile-time generation of bytecode easily lends itself to statically determining the amount of registers needed, potentially introducing a new avenue for obfuscation. Additionally, by expanding the set of supported operations and integrating virtual machine hardening techniques, we could see an increase in the robustness of our obfuscation.

6 Conclusion

In conclusion, our novel approach to VM-based obfuscation utilizing Rust's powerful macro system offers a fresh perspective compared to other research-based obfuscators. By demonstrating the feasibility of compile-time transformations through a proof-of-concept obfuscator, we lay the groundwork for a more comprehensive solution that leverages the future capabilities of Rust. As the software landscape continues to evolve, it is imperative that we develop innovative methods to protect intellectual property and deter malicious

actors. Our proposed work not only contributes to this goal but also opens the door for future research and advancements in code obfuscation techniques.

Bibliography

[1] M. Schloegel *et al.*, “Loki: Hardening Code Obfuscation Against Automated Attacks,” *CoRR*, 2021, [Online]. Available: https://arxiv.org/abs/2106.08913

[2] H. Fang, Y. Wu, S. Wang, and Y. Huang, “Multi-stage Binary Code Obfuscation Using Improved Virtual Machine,” 2011, pp. 168–181. doi: 10.1007/978-3-642-24861-0_12.

[3] W. Foundation, “Rust (programming language).” [Online]. Available: https://en.wikipedia.org/wiki/Rust_(programming_language)

[4] Microsoft, “70 percent of all security bugs are memory safety issues,” *ZDNET*, Feb. 2019, [Online]. Available: https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/

[5] T. W. House, “Back to the Building Blocks: A Path Toward Secure and Measurable Software.” [Online]. Available: https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf

[6] S. Klabnik and C. Nichols, [Online]. Available: https://doc.rust-lang.org/reference/const_eval.html

[7] S. Klabnik and C. Nichols, [Online]. Available: https://doc.rust-lang.org/book/ch19-06-macros.html

[8] J. F. Reiser and M. F. X. J. Oberhumer, “UPX.” 1998.

[9] H. Xu, Y. Zhou, J. Ming, and M. Lyu, “Layered obfuscation: a taxonomy of software obfuscation techniques for layered security,” *Cybersecurity*, vol. 3, no. 1, p. 9, Apr. 2020, doi: 10.1186/s42400-020-00049-3.

[10] R. Rolles, “Unpacking virtualization obfuscators,” in *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, in WOOT'09. Montreal, Canada: USENIX Association, 2009, p. 1.

[11] M. Schloegel *et al.*, “USENIX Security ’22 - Loki: Hardening Code Obfuscation Against Automated Attacks.” [Online]. Available: https://www.youtube.com/watch?v=vurLTQVMWnE

[12] T. Blazytko, “Analysis of Virtualization-based Obfuscation.” [Online]. Available: https://www.youtube.com/watch?v=b6udPT79itk

[13] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, “Obfuscator-LLVM – Software Protection for the Masses,” in *2015 IEEE/ACM 1st International Workshop on Software Protection*, 2015, pp. 3–9. doi: 10.1109/SPRO.2015.10.

[14] C. Collberg, “Tigress: A Source-to-Source-ish Obfuscation Tool. Home.” Dec. 03, 2018.

[15] D. Tolnay, “Dtolnay/syn: Parser for rust source code.” [Online]. Available: https://github.com/dtolnay/syn

[16] D. Tolnay, “Dtolnay/quote: Rust quasi-quoting.” [Online]. Available: https://github.com/dtolnay/quasi-quoting