Computer Security Symposium 2024
22-25 October 2024

Rustで生成されたマルウェアを解析するための Ghidra拡張機能の開発

川越 謙宏[1,a)]　小林 良太郎[1]

概要：近年，脅威アクターはマルウェアの作成に，Go Lang，Rust，Nim 等といった言語を使うことがある．特に Rust は実行速度，並行処理の容易さ，メモリに関するバグが少なさから，マルウェアの作成に使われることがある．一般的に静的解析で使用されるリバースエンジニアリングツールはほとんどがデコンパイル機能を持っている．しかし，現状デコンパイラは C 言語へのデコンパイルのみで，Rust へのデコンパイルができないので，Rust で生成されたバイナリをデコンパイルすると，あまり参考にならない結果が返ってくることがあり，現状解析が困難である．今回，National Security Agency (NSA) が開発したリバースエンジニアリングツールである Ghidra の拡張機能を作成し，Rust の静的解析を容易にする手法を提案する．

キーワード：静的解析，マルウェア，Ghidra 拡張機能，Rust

Development of Ghidra Extension to Analyze Malware Generated by Rust

Akihiro Kawagoe[1,a)]　Ryotaro Kobayashi[1]

Abstract: In recent years, threat actors have been using languages such as Go Lang, Rust, and Nim to create malware. In general, most reverse engineering tools used for static analysis have decompilation capabilities. However, currently only decompilers to C exist, and decompiling binaries generated by Rust to C sometimes returns unhelpful results, making analysis difficult. In this study, we propose an extension to Ghidra, a reverse engineering tool developed by National Security Agency (NSA), to facilitate static analysis of other languages.

Keywords: Static Analysis, Malware, Ghidra Extension. Rust

1. はじめに

近年，脅威アクターはマルウェアの作成に，Go Lang, Rust, Nim 等, 比較的新しい言語を使用することがある [1]. これは解析を困難にさせることとシグネチャベースの検出を回避することが主な要因である．Rust は実行速度，並行処理の容易さ，メモリに関するバグが少ないことが特徴の言語である．最近では Linux のカーネルに用いられるなど人気のある言語であると同時に，脅威アクターからも注目されている．Loader では BuerLoader, SSLoad, ランサムウェアでは BlackCat (ALPHV), Agenda などに Rust が用いられている [2].

マルウェアの解析は対策の役に立つ．解析の手法は大きいく分けて表層解析，静的解析，動的解析の 3 種類ある．静的解析は検体を実行せずにアセンブリなどの情報から挙動を推測する解析手法である．静的解析には Hex-rays 社が開発した IDA pro [3], Vector 35 社が開発した Binary ninja [4], NSA が開発した Ghidra [5] など専用のリバースエンジニアリングツールを用いる．これらのツールには，バイナリをアセンブリに変換するディスアセンブラ，実行

[1] 工学院大学
Kogakuin University, Shinjuku, Tokyo 163-8677, Japan
[a)] j121095@ns.kogakuin.ac.jp

ファイルから元のソースコードを推測し C 言語へと変換するデコンパイラ，関数呼び出しやアセンブリのグラフによる可視化など，解析の役に立つ機能がある．静的解析はこれらの機能を使い挙動を推測するため，静的解析にかかる時間もこれらの機能の性能に大きく影響する．特にバイナリから高級言語を推測するデコンパイラは解析への影響度が高い．一般的に静的解析で使用されるリバースエンジニアリングツールはほとんどが逆コンパイル機能を持っている．しかし，デコンパイラは基本的に C 相当の言語にしか変換できないため，他言語で生成されたマルウェアの解析は困難である．

Ghidra は，無償ながらも非常に多くの機能が利用できるリバースエンジニアリングツールである．本論文では，Ghidra で Rust のバイナリを解析した際の特徴や問題点，またそれらを解決するための拡張機能を提案する．

2. 関連研究

本論文ではデコンパイラの改善するための様々な手法を提案する．本章では，デコンパイラの開発に関する関連研究を紹介する．

2.1 機械学習を用いた変数名の推測

リバースエンジニアリングに機械学習を活用したものを紹介する．

Chen らは，デコンパイラが出力するコードは変数名を “uVar1” のように意味のない名前であることを指摘し，Transformer を用いた変数名と型を推測するためのモデル DIRTY (**D**ecomp**I**led variable **R**e**TY**per) を提案した [6]．また Cao らは DIRTY が Hex-rays のデコンパイラを使用していたため，オープンソースの Ghidra で DIRTY アーキテクチャの検証を行った [7]．

他には，Binary ninja の Plugin である Sidekick [8] は LLM を用いて，構造体の推測，関数や変数のリネームができる．

2.2 既存のデコンパイラ

本論文では Rust のデコンパイラの改善に関して述べているため，ここでは既存のデコンパイラの仕組みを一部紹介する．

Avast 社が開発したデコンパイラである RetDec [9] は，バイナリを C 言語に変換する機能のみを提供する CLI ツールである．RetDec はバイナリをアセンブリ，LLVM-IR，C 言語の順に変換する．RetDec の特徴は，デコンパイラにコンパイラの基盤である LLVM を使用していることである．

Binary ninja が提供するデコンパイラは BNIL という独自の中間言語を使用している [10]．バイナリをアセンブリに変換し，Low Level Intermediate Language, Middle Level Intermediate Language, High Level Intermediate Language, C 言語の順に変換する．Ghidra も同様に PCode という独自の中間言語をデコンパイル時に使用している．

このようにデコンパイルをする際に中間言語を使用することがある．

2.3 Project OxA11C

AI を活用したセキュリティサービスを提供する SentinelOne 社の研究機関である SentinelLabs は，過去に Go 言語で作られたバイナリを解析するための IDA の拡張機能である AlphaGolang を開発した [11]．Black Hat USA 2024 では SentinelLabs と Intezer 社が Rust のバイナリが解析しづらい問題を解決するために，プロジェクト OxA11C を発表した [12]．OxA11C は Yara ルールを用いて Rust のバイナリを検知する．またコンパイラのバージョンの特定や，Slice と文字列のリキャストなどができる．このプロジェクトは IDA Python を用いた Script である．

3. Rust の特徴と Ghidra の改善点

本章では Ghidra で Rust のバイナリを解析した際の特徴と Ghidra の改善点を述べる．

3.1 Rust の解析困難性

以下は 0 から 99 までの総和を求め出力するプログラムを Rust で作成したものである．

```
fn main() {
    let mut sum = 0;
    for i in 0..100 {
        sum += i;
    }
    println!("sum: {}", sum);
}
```

図 1 0 から 99 までの総和を求めるプログラム
Fig. 1 Program to Calculate the Sum from 0 to 99

Rust では変数が可変かどうかを明示するために mut キーワードを使用する．for 文でループ処理を行い，sum に i を加算し，最後に println! マクロを用いて sum を出力している．このプログラムから関数や変数に関する symbol を消さずに生成したバイナリを Ghidra でデコンパイルした結果は以下である．

```
Decompile: FUN_00108db0 - (rust_sample_dec0_stripped)
1
2 void FUN_00108db0 (void)
3
4 {
5   undefined4 extraout_EDX;
6   int extraout_EDX_00;
7   int local_6c;
8   undefined4 local_68;
9   undefined4 local_64;
10  undefined4 local_60;
11  undefined4 local_5c;
12  int local_58;
13  int local_54;
14  undefined local_50 [48];
15  int *local_20;
16  code *local_18;
17  int *local_10;
18  code *local_8;
19
20  local_6c = 0;
21  local_68 = 0;
22  local_64 = 100;
23  local_60 = FUN_00108d40(0,100);
24  local_5c = extraout_EDX;
25  while( true ) {
26    local_58 = FUN_00108d20(&local_60);
27    local_54 = extraout_EDX_00;
28    if (local_58 == 0) break;
29    local_6c = extraout_EDX_00 + local_6c;
30  }
31  local_20 = &local_6c;
32  local_8 = FUN_00147910;
33  local_18 = FUN_00147910;
34  local_10 = local_20;
35  FUN_00108bb0(local_50,&DAT_0015b128,2,&local_20,1);
36  FUN_0010bc80(local_50);
37  return;
38 }
```

図 2 Ghidra でのデコンパイル結果
Fig. 2 Decompiled Result in Ghidra

まず local_60 が元のソースコードの i に対応している．Rust では for 文にイテレーターを指定するのでイテレーターを作る関数 into_iter(0) が呼び出されている．while 内にある next 関数ではイテレーターを次の値にするための関数が呼び出されている．この関数の戻り値は次の値と，末尾かどうかを確認するためのフラグが返される．if 文の中で sum += i に該当する演算が行われている．

図 3は図 1と同じ内容を C 言語で作成し，Ghidra で解析した結果である．図からわかるように，C 言語であればデコンパイラが出力する結果は非常に単純である．一方で Rust は複雑なバイナリを出力するため，デコンパイラが出力する結果の意味を理解すること自体が難しい．

3.2 Symbol の復元

外部から関数を呼び出す方法は主に動的リンクと静的リンクである．静的リンクは実行ファイルに直接関数のバイナリを埋め込む方法である．この手法はバイナリサイズが大きくなるが，余計な情報が増えるため解析が困難に

```
Decompile: main - (c_sample_dec)
1
2 undefined8 main(void)
3
4 {
5   int local_10;
6   int local_c;
7
8   local_10 = 0;
9   for (local_c = 0; local_c < 100; local_c = local_c + 1) {
10    local_10 = local_10 + local_c;
11  }
12  printf("sum: %d\n",local_10);
13  return 0;
14 }
```

図 3 C 言語を Ghidra で解析した結果
Fig. 3 Result of Analyzing C Code in Ghidra

なる．動的リンクは必要な関数を実行時に呼び出す方法である．この手法は関数のバイナリが実行ファイルに埋め込まれないため，バイナリサイズが小さくなる．Rust は標準ライブラリ関数を静的リンクで呼び出している．図 2は関数名を表す symbol が残っているので into_iter や std::io::stdio::_print などの情報が一目でわかる．しかし，symbol が削除された Rust のバイナリは解析がより困難になることがわかる．この問題を解決するための機能として Ghidra には Function ID という機能がある．この機能は関数名とハッシュ値のデータベースを作ることができ，そのデータベースを用いて strip されたバイナリの関数名を復元できる．言語に標準で用意されている関数は基本的に中身がほとんど同じなのでこの機能で関数名を復元できる．

```
20  local_6c = 0;
21  local_68 = 0;
22  local_64 = 100;
23  local_60 = FUN_00108d40(0,100);
24  local_5c = extraout_EDX;
25  while( true ) {
26    local_58 = FUN_00108d20(&local_60);
27    local_54 = extraout_EDX_00;
28    if (local_58 == 0) break;
29    local_6c = extraout_EDX_00 + local_6c;
30  }
31  local_20 = &local_6c;
32  local_8 = FUN_00147910;
33  local_18 = FUN_00147910;
34  local_10 = local_20;
35  FUN_00108bb0(local_50,&DAT_0015b128,2,&local_20,1);
36  FUN_0010bc80(local_50);
37  return;
```

図 4 symbol が削除された Rust のバイナリ
Fig. 4 Rust Binary with Symbols Stripped

図 4は symbol が削除された Rust のバイナリである．このバイナリを Function ID を用いて解析した結果が図 5である．

```
25  local_6c = 0;
26  local_68 = 0;
27  local_64 = 100;
28  local_60 = FUN_00108d40(0,100);
29  local_5c = extraout_EDX;
30  while( true ) {
31    local_58 = next(&local_60);
32    local_54 = extraout_EDX_00;
33    if (local_58 == 0) break;
34    local_6c = extraout_EDX_00 + local_6c;
35  }
36  local_20 = &local_6c;
37  local_8 = fmt;
38  local_18 = fmt;
39  local_10 = local_20;
40  new_v1(local_50,&DAT_0015b128,2,&local_20,1);
41  _print(local_50);
42  return;
```

図 5 Function ID を用いて関数名を復元した Rust のバイナリ
Fig. 5 Rust Binary with Function Names Restored Using Function ID

Function ID にはいくつかの問題がある．まず，自身でデータベースを用意する必要があることだ．Ghidra にはデフォルトで Visual C++関連のデータベースは存在するが，Rust のデータベースは存在しない．Hex-rays のリバースエンジニアリングツールである IDA には Lumina server という既知の関数のメタ情報を登録し，それを用いて関数名を復元する機能がある．Ghidra の Function ID との大きな違いは，サーバがあるのでデータベースを自身で用意する必要がないことである．現状 Rust のバージョンは strip されたバイナリにも残るので，バージョンを特定し必要なデータベースを動的に用意できないか検討中である．

第 2 に，Function ID は関数名とハッシュ値でデータベースを作るため，少しでも中身が異なると，関数名を復元できない．現状 so ファイルからデータベースの作成を行っているが，ハッシュ値が異なり復元ができないことがある．ただ，この問題は BSim という既存の拡張機能で解決できる．BSim は関数名とデコンパイル結果，Ghidra に存在する Pcode という中間言語でデータベースを作り，作成したデータベースを用いて，コサイン類似度が一定以上の関数を出力する．BSim は類似した関数を探すことを目的とした機能のため，実行の速さやデータベースの大きさなどの問題点がある．

Function ID のもう 1 つの問題点は，namespace の補完ができないことである．namespace は名前の衝突を防ぐための機能であると同時に，core::inter::range::_<>::next が namespace からイテレータに関する関数であることがわかるように，関数の挙動を推測するための手掛かりになるため重要な情報である．

4. デコンパイラの開発

本章では現状存在する Rust のデコンパイラについて説明し，デコンパイラを開発する際に懸念

4.1 存在する Rust のデコンパイラ

現状，Ghidra には GhidRust [13] という拡張機能がある．この拡張機能は Rust のバイナリかどうかを判定する機能と，Rust のデコンパイラを提供している．しかし，デコンパイラが動作しないことが多く，またこのプロジェクトの開発は停止状態のため，今後にも期待できない．

GhidRust のデコンパイラの仕組みは，Ghidra が生成した，デコンパイル結果を用いて Rust のコードへトランスパイルするというものである．従って，Ghidra のデコンパイラの性能に大きく依存する．

4.2 Rust のコンパイラの仕組み

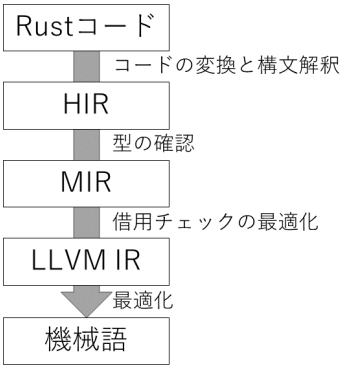

図 6 Rust のコンパイラの仕組み
Fig. 6 Rust compiler system

Rust のコンパイラの仕組みは図 6のようになっている．HIR は High-level Intermediate Representation，MIR は Middle-level Intermediate Representation，LLVM IR は Low Level Virtual Machine Intermediate Representation であり，どれも中間言語である．下に行くほど抽象度が低く，人間には理解しにくくなる．

```
#[prelude_import]
use ::std::prelude::rust_2015::*;
#[macro_use]
extern crate std;
fn main() {
  let mut sum = 0;
  {
    let _t =
      match #[lang = "into_iter"](#[lang = "Range"]{
        start: 0,
        end: 100,}) {
        mut iter => loop {
          match #[lang = "next"](&mut iter) {
            #[lang = "None"] {} => break,
            #[lang = "Some"] { 0: i } => { sum += i; }
            }
          },
        };
      _t
    };
  {
    ::std::io::_print(format_arguments::new_v1(
      &["sum: ", "\n"],
      &[format_argument::new_display(&sum)]));
  };
}
```

図 7 HIR に変換された Rust のコード
Fig. 7 Rust Code Converted to HIR

図 7は図 1から生成した HIR である．図から Rust コードが HIR に変換される段階でマクロが展開され，for 文が loop 文などコードの変換が行われていることがわかる．

```
fn main() -> () {
    let mut _0: ();
    let mut _1: i32;
    let mut _2: std::ops::Range<i32>;
    let mut _3: std::ops::Range<i32>;
    let mut _5: std::option::Option<i32>;
    let mut _6: &mut std::ops::Range<i32>;
    let mut _7: isize;
    let mut _9: (i32, bool);
    let _10: ();
    let mut _11: std::fmt::Arguments<'_>;
    let mut _12: &[&str];
    let mut _13: &[core::fmt::rt::Argument<'_>];
    let _14: &[core::fmt::rt::Argument<'_>; 1];
    let _15: [core::fmt::rt::Argument<'_>; 1];
    let mut _16: core::fmt::rt::Argument<'_>;
    let _17: &i32;
    scope 1 {
        debug sum => _1;
        let mut _4: std::ops::Range<i32>;
        let mut _18: &[&str; 2];
        scope 2 {
            debug iter => _4;
            let _8: i32;
            scope 3 {
                debug i => _8;
            }
        }
    }
    bb0: {
        _1 = const 0_i32;
        _3 = std::ops::Range::<i32> {
          start: const 0_i32,
          end: const 100_i32 };
        _2 = <std::ops::Range<i32> as IntoIterator>...
    }
(省略)
```

図 8 MIR に変換された Rust のコード
Fig. 8 Rust Code Converted to MIR

図 8が生成された MIR である．HIR が MIR に変換される段階で型の確認の他に，ローカル変数の暗黙的追加，処理のブロック化などが行われている．このブロックは LLVM-IR に変換する際の前処理である．アンダースコアから始まる変数がコンパイラが自動的に追加した変数である．MIR は LLVM-IR に変換する段階で借用チェックの最適化が行われ，LLVM-IR からマシンコードする際に最適化処理が行われる．

4.3 マクロによる影響

Rust にはマクロという機能がある．マクロはコードを書くためのメタプログラミング機能である．コンパイルの直前にマクロは展開され，別のコードへと変換される．Rust の関数は可変長の引数をとることができないため，マクロを用いて実装する必要がある．println! マクロや vec! マクロのように標準で使用できるマクロも存在するため，他の

言語に比べマクロを使用する頻度が高い．vec! マクロは動的にサイズを変えることのできる配列である．vec! マクロは vec![0,1,2] のように使用し，このマクロの図 9のように定義されている．

```
#![allow(unused)]
fn main() {
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

図 9 vec!マクロの定義
Fig. 9 Definition of vec! Macro

これは Vec 型の変数 temp_vec を用意し，引数がなくなるまで temp_vec.push を行い，最後に temp_vec を返すものだ．これは関数ではなくマクロなので，コンパイル時にマクロの定義に従ったコードが注入される．これにより，デコンパイラ開発をする際はマクロを復元するプロセスが必要になる可能性がある．

4.4 デコンパイラの開発

デコンパイラの開発にはルールベースの変換とディープラーニングを用いる方法が考えられる．ルールベースによる変換は確実であるが，開発に時間がかかる．ルールベースで開発する際は，1 つの手法としてアセンブリを LLVM-IR に変換し，MIR，HIR，Rust コードの順に出力する方法が考えられる．ディープラーニングを用いる場合，多くのデータベースを用意する必要がある．特に最適化オプションやコンパイラのバージョンなどで出力されるバイナリが変化するので膨大なデータが必要になる可能性がある．Rust は開発が活発で nightly 版は毎日更新される．ディープラーニングを用いる利点は，ルールベースのデコンパイラよりも変化に対応しやすい点だ．

5. 結論と今後の方針

本論文では，Rust のバイナリを解析する際の問題点と，それを解決するための手法を提案した．Rust はコンパイル時に複数の中間言語に変換するため，バイナリが複雑になり，現状解析を支援する機能がないため，解析が困難になる．Ghidra で Rust のバイナリ解析をするための前段階として Strip された関数名の復元，マクロによって展開されたコードの追跡が必要である．また，Rust のコンパイラシステムの仕組みを調査し，デコンパイラの開発について述べた．

今後の方針は，まず Function ID で Strip された標準ライブラリ関数の関数名を復元できるようにする．そして，使用された Rust のコンパイラから動的にデータベースを用意する機能を実装することである．

謝辞 本研究の一部は、JSPS 科研費 23H03396 の支援により行った．

参考文献

[1] BlackBerry Ltd., “BlackBerry 2022 Threat Report,” https://www.blackberry.com/ (Accessed 2024-8-20).
[2] M. Praveen and W. Almobaideen, “The Current State of Research on Malware Written in the Rust Programming Language,” Proceeding of the 11th International Conference on Information Technology (ICIT), pp. 266-270, 2023.
[3] Hex-Rays, “IDA Pro,” https://hex-rays.com/ida-pro/ (Accessed 2024-8-19).
[4] Vector 35 Inc., “Binary Ninja,” https://binary.ninja/ (Accessed 2024-8-19).
[5] National Security Agency, “Ghidra”, https://ghidra-sre.org/ (Accessed 2024-8-19).
[6] Q. Chen, J. Lacomis, E.J. Schuwartz, C.L. Goues, G. Neubig and B. Vasilescu, “Augmenting Decompiler Output with Learned Variable Names and Types,” Proceeding of the 31st Usenix Security Symposium, 2022.
[7] K. Cao and K. Leach, “Revisiting Deep Learning for Variable Type Recovery,” Proceeding of the 31st International Conference on Program Comprehension (ICPC), pp. 275-279, 2023.
[8] Vector 35 Inc., “Sidekick,” https://sidekick.binary.ninja/ (Accessed 2024-8-19).
[9] Avast, “RetDec,” https://github.com/avast/retdec (Accessed 2024-8-19).
[10] Vector 35 Inc., “BNIL,” https://docs.binary.ninja/dev/bnil-overview.html (Accessed 2024-8-19).
[11] SentinelOne, “AlphaGolang,” https://github.com/SentineLabs/AlphaGolang (Accessed 2024-8-19).
[12] N. Fishbein and J.A. Guerrero-Saade, “Project 0xA11C: Deoxidizing the Rust Malware Ecosystem,” Black hat USA, 2024.
[13] D. Maroo, “GhidRust,” https://github.com/DMaroo/GhidRust (Accessed 2024-8-19).