情報システム実験Ｂ

バイナリプログラムの解析
最終レポート

1　はじめに

このレポートは情報システム実験 B K-15「バイナリプログラムの解析」の最終レポートである．実験の期間中に開発したバイナリ解析プログラムについて説明する．ソースコードは https://github.com/Alignof/binary_program_analysis から参照できる．

2　機能の説明

各機能ごとに詳細を説明をする．なお機能の一覧は以下のコマンドで確認できる．

```
$ ./unlibit --help
unlibit 0.1.0

USAGE:
    unlibit [OPTIONS] <filename>

ARGS:
    <filename>    target file path

OPTIONS:
    -h, --header          Show header
    -p, --program         Show all segments
    -s, --section         Show all sections
    -d, --disasem         Disassemble ELF/PE
    -a, --analyze         Analyze target binaly file
        --dump            Dump binary file
        --diff <other>    Take a diff of the binary files
        --histogram       Show byte histogram
        --help            Print help information
    -V, --version         Print version information
```

2.1 ELF と PE を解析する機能

まず，ELF と PE の構造を理解するためそれらを解析する機能を付けた．32bit 向けの ELF ファイルを解析するプログラムは書いたことがあったのと授業課題で PE ファイルの解析が出たので ELF と PE 双方で 32bit/64bit の両方をサポートすることにした．

ELF では ELF ヘッダ，セクションヘッダ，プログラムヘッダが表示可能である．PE では MS-DOS ヘッダ，NT ヘッダ，セクションヘッダが表示可能である．

今回は ELF と PE を同じように処理できるように trait で抽象化を行った．ELF と PE のローダを抽象化した trait，Loader と ELF での実装例をリスト 1 に示す．

Listing 1: src/loader.rs

```rust
pub trait Loader {
    fn mem_data(&self) -> &[u8];
    fn header_show(&self);
    fn show_segment(&self);
    fn show_section(&self);
    fn disassemble(&self);
    fn show_all_header(&self);
    fn analysis(&self);
}
impl Loader for ElfLoader {
    fn mem_data(&self) -> &[u8] {
        &self.mem_data
    }

    fn header_show(&self) {
        self.elf_header.show();
    }

    fn show_segment(&self) {
        for (id, prog) in self.prog_headers.iter().enumerate() {
            prog.show(id);
            println!("\n\n");
        }
    }

    fn show_section(&self) {
        for (id, sect) in self.sect_headers.iter().enumerate() {
```

```
28              sect.show(id);
29              println!("\n\n");
30          }
31      }
32
33      fn disassemble(&self) {
34          for (id, sect) in self.sect_headers.iter().enumerate() {
35              sect.show(id);
36              sect.dump(&self.mem_data);
37              println!("\n\n");
38          }
39      }
40
41      fn show_all_header(&self) {
42          self.elf_header.show();
43
44          println!("\n\n");
45
46          for (id, prog) in self.prog_headers.iter().enumerate() {
47              prog.show(id);
48          }
49
50          println!("\n\n");
51
52          for (id, sect) in self.sect_headers.iter().enumerate() {
53              sect.show(id);
54          }
55      }
56
57      fn analysis(&self) {
58          let mut inst_list_overall = HashMap::new();
59          for func in self.functions.iter() {
60              let mut inst_list = HashMap::new();
61              let call_addrs = func.inst_analysis(&mut inst_list,
    ↪ &self.mem_data);
62
63              for (name, count) in inst_list.clone() {
64                  *inst_list_overall.entry(name).or_insert(0) +=
    ↪ count;
65              }
66
```

```
            let mut inst_list =
↪ inst_list.iter().collect::<Vec<(&String, &i32)>>();
            inst_list.sort_by(|a, b| (-(a.1)).cmp(&(-(b.1))));
            for t in inst_list.iter() {
                println!("{}: {}", t.0, t.1);
            }

            if !call_addrs.is_empty() {
                print!("calling functions: ");
                for call_addr in call_addrs {
                    let call_func = self
                        .functions
                        .iter()
                        .find_map(|f| {
                            if f.addr == call_addr {
                                Some(f.name.clone())
                            } else {
                                None
                            }
                        })
                        .unwrap_or(format!("{call_addr}"));
                    print!("func_{call_func} ");
                }
                println!();
            }
            println!();
        }

        println!("======================");
        let mut inst_count = 0;
        let mut inst_list_overall =
↪ inst_list_overall.iter().collect::<Vec<(&String, &i32)>>();
        inst_list_overall.sort_by(|a, b| (-(a.1)).cmp(&(-(b.1))));
        for t in inst_list_overall.iter() {
            inst_count += t.1;
            println!("{}: {}", t.0, t.1);
        }
        println!("-----");
        println!("instructions: {inst_count}");
        println!("======================");
    }
```

```
106 }
```

リスト 1: Loader trait

Loader trait では,

- 構造体内部にあるメモリマップのデータを貸し出す mem_data
- ヘッダを表示する header_show
- セグメントを表示する show_segment
- セクションを表示する show_section
- ディスアセンブルを表示する disaassemble
- すべてのヘッダを表示する show_all_header
- データを元に命令を解析する analysis

の 7 つを共通の振る舞いとして定めた.

mem_data は中身を貸し出すだけなので必要無さそうだが，trait はメンバにアクセスするために実装先のオブジェクトを指定する必要があるので trait の宣言に共通して書くことができない．PE ではセグメントという概念が存在しないのでセクションヘッダを代わりに表示するようになっている.

また今回は解析に必要なのでディスアセンブルの機能も付けた．ただし，x86_64 のデコードを手で実装するのは大変なので iced_x86[1] というライブラリを使用した．これにより簡単に x86_64 のデコードや命令の表示が可能になった．今回は x86_64 にのみ対応したが，必要になれば RISC-V 向けの手書きのデコーダが手元にあるのでそれが使える他，Arm なども簡単に扱えるツール [2] があるため必要になり次第対応しようと思う.

2.2 バイナリの hex dump をヒートマップで表示できる機能

バイナリをヒートマップで色付けした hex dump で表示できる機能を実装した．実際の出力を図 1 に示す.

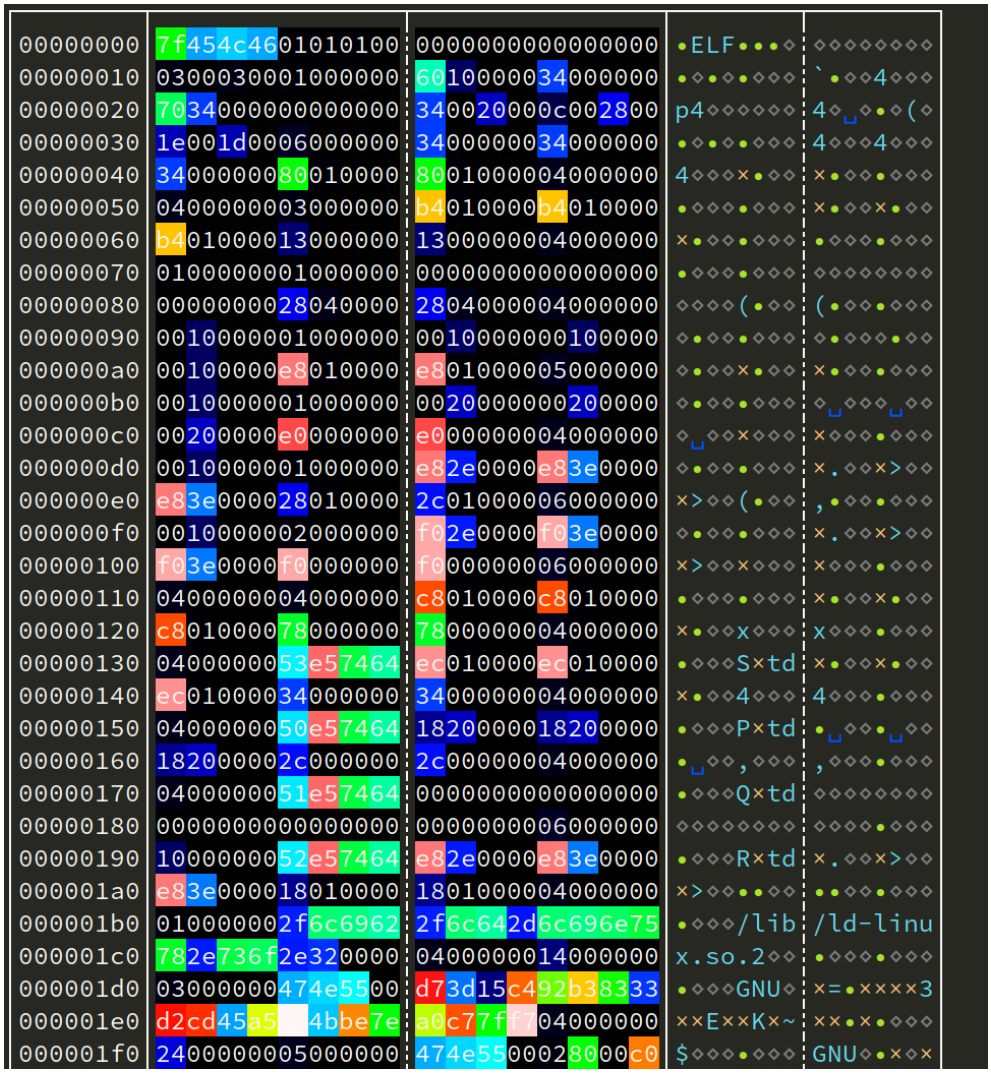

図 1: ヒートマップ表示された hex dump

各バイトの数値によって数字を割り当てて，セクションの境界や性質を直感的に感じることができる．数値の色の対応づけは 0x00 に黒，0xff に白を割り当て数が大きくなるごとに明るい色になるようにしている．具体的には黒→青→水色→緑→黄色→赤→白と段階的に変わっていく．

数値と色の具体的な対応を図 2 に示す．

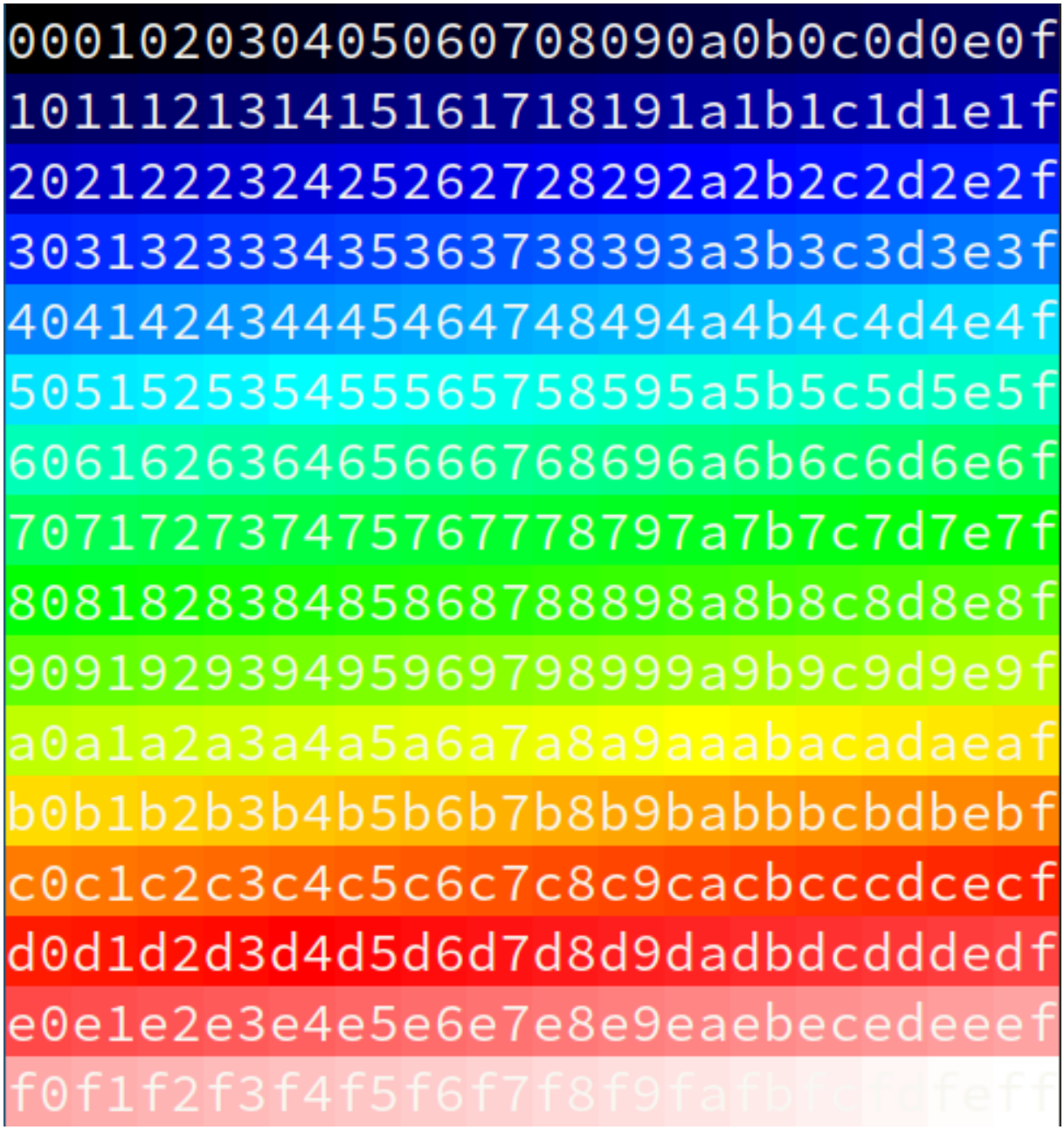

図 2: 数値の色の具体的な対応

この配色の問題点は緑色が占める部分の多いことと ascii 範囲である 0x70 0x7f と 0x80 0x8f の見分けが付きづらいということが挙げられるが dump の横には後述する ascii 表示がある上，仮に隣り合っているときにはそもそも ascii かどうかの区別は不要な場合（ascii 領域に非 ascii のバイトが来るとは考えづらい）であるのでこのまま採用した．

もし仮に改善するならば水色の次を紫にして 0x80 からを緑にする方法がある．ただしこれはルールに一貫性が無いことを考慮しなくてはならない．

配色の実装を以下に示す.

Listing 2: visualize.rs

```rust
fn hex_to_color(hex: u8) -> (u8, u8, u8) {
    const STEP: u8 = 6;
    let step_up = |start: u8| (hex - start).saturating_mul(STEP);
    let step_down = |start: u8| 255_u8.saturating_sub((hex - start).saturating_mul(STEP));
    let red = match hex {
        0..=127 => 0,
        128..=169 => step_up(128),
        170..=255 => 255,
    };
    let green = match hex {
        0..=41 => 0,
        42..=83 => step_up(42),
        84..=169 => 255,
        170..=211 => step_down(170),
        212..=255 => step_up(212),
    };
    let blue = match hex {
        0..=41 => step_up(0),
        42..=83 => 255,
        84..=127 => step_down(84),
        128..=211 => 0,
        212..=255 => step_up(212),
    };
    (red, green, blue)
}
```

リスト 2: 数値と配色を対応付ける処理

色を黒 (0,0,0)，青 (0,0,1)，シアン (0,1,1)，緑 (0,1,0)，黄 (1,1,0)，赤 (1,0,0)，白 (1,1,1) の間の領域に分けてその間で段階的に増加・減少を行っている．この処理はもっときれいに書けそうなので反省点である．

次に dump の横に表示されている記号の表示について説明する．右側には dump のバイト数と同じだけの記号が置かれている．この表示は普段自分が使っている hexyl[3] という hex viewer を参考に実装したもので以下の意味を持つ．

これにより hex の大まかな種類とバイナリに埋め込まれている ascii 文字を読むことができる．ascii 文字を読めることは解析において非常に重要な情報である．

表 1: 記号と色の意味

表示	記号	意味
灰色のダイヤ	⌧	0x00
青色の記号	␣	空白文字 (0x20)
緑色の文字	a	ascii 表示可能なもの
水色の丸	•	0x7f より小さい ascii 表示外の数
黄色のバツ	×	0x80 以上の ascii 表示外の数

2.3 バイナリの diff を取る機能

バイナリの diff を取る必要があることが良くあるが，コマンドライン上で diff や delta のように手軽に diff を取れるツールは見つからない．普段は hexyl の出力を左右に並べて目で比較したり hexyl の出力を delta に与えて比較する方法（図 3）を取っている．

図 3: hexyl の出力を delta に与えて比較する方法

しかし出力してからテキストで比較する方法は元のツールの色の情報が失われてしまう上，1 行ごとの比較には向かない．そこで，今回は上の hex dump の機能を拡張する形でバイナリファイルから直接 diff を取れる機能を開発した．

実際に使用している際のスクリーンショットを図 4 に示す．

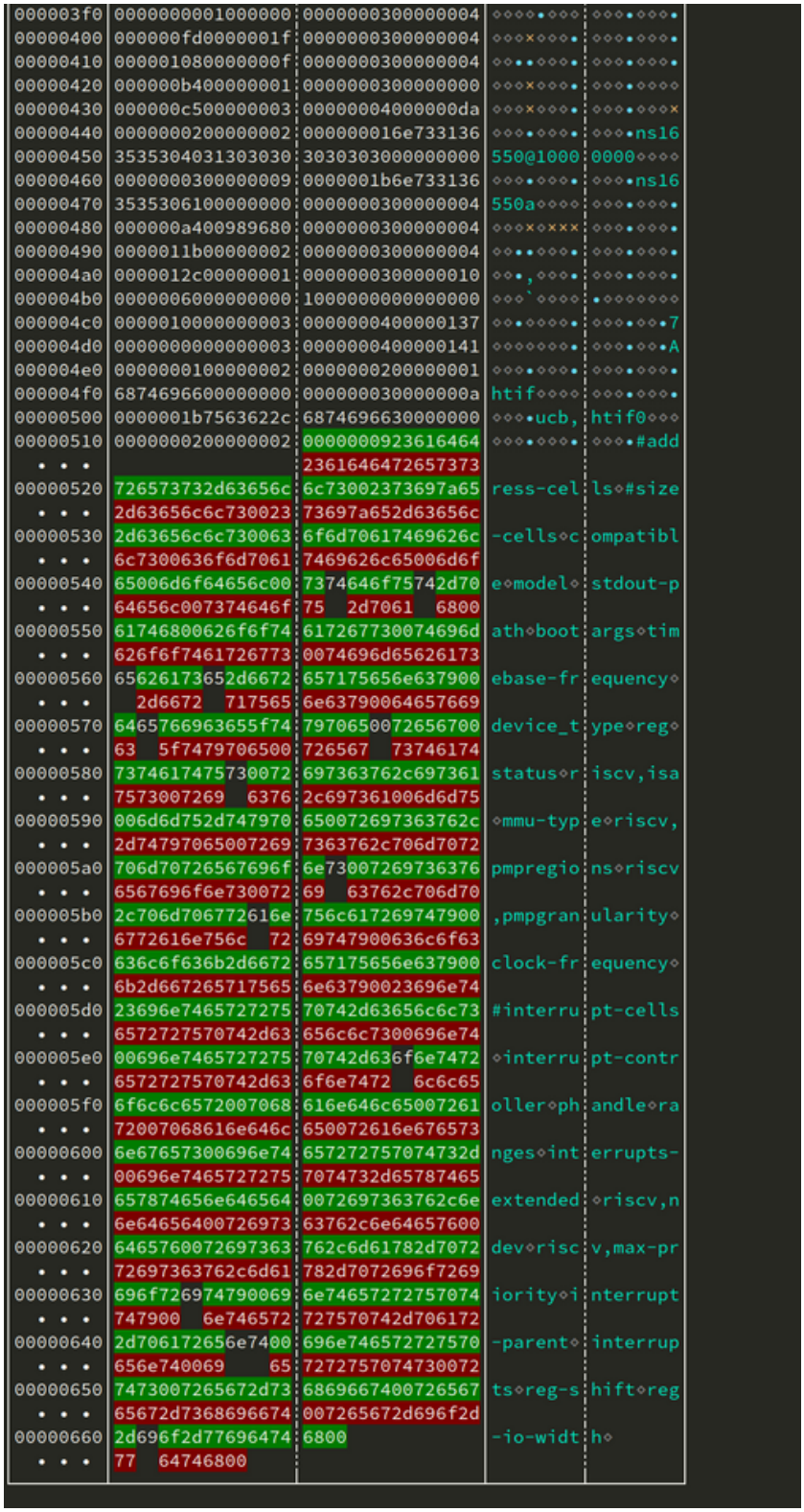

図 4: diff の機能のスクリーンショット

差分のある行のみ，下に比較対象の dump を表示している．これにより 1 行単位でどこが違うかを簡単に比較できる．また，差分のあるバイト列を比較元のファイルで

は緑に比較先のファイルでは赤にハイライト表示している．差分の無い部分は表示もハイライトもしてない．これにより一層差分のある部分が明快になった．

前述の通りこの機能は hex viewer の機能に追加して実装されたので ascii 表示はハイライトを保ったまま表示される．なお ascii 表示は比較元のもののみを表示している．どちらも表示するパターンやここにも diff をつけるパターンも試したが，見やすさを優先した結果現在の表示形式になった．

2.4 ヒストグラムを表示する機能

授業でバイナリヒストグラムを作成する課題があったのでこれを機能の一部として組み込むことにした．課題中では画像形式で出力をしていたが，これを CUI で表示するように変更した．表示には Unicode の Block Elements を使用した [4].

表示のための実装をリスト 3 に示す．

Listing 3: visualize.rs

```rust
pub fn create_byte_histogram(mem_data: &[u8]) {
    use colored::Colorize;

    let mut histogram = (0..=255)
        .collect::<Vec<u8>>()
        .iter()
        .map(|x| (*x, 0_u32))
        .collect::<HashMap<u8, u32>>();

    for m in mem_data.iter() {
        *histogram.entry(*m).or_insert(0) += 1;
    }
    let max_count: u32 = *histogram.iter().max_by(|a, b|
    ↪ a.1.cmp(b.1)).unwrap().1;

    // calc entropy
    let mut entropy: f64 = 0.0;
    for (_, count) in histogram.iter() {
        let p: f64 = (*count as f64) / (mem_data.len() as f64);
        if p != 0.0 {
            entropy -= p * p.log(2.0);
        }
    }
    println!("entropy: {entropy}");

```

```
25      const BAR: [&str; 8] = ["", "▏ ", "▎ ", "▍ ", "▌ ", "▋ ", "▊ ",
    ↪   "▉ "];
26      let mut histogram: Vec<(&u8, &u32)> =
    ↪   histogram.iter().collect();
27      histogram.sort_by(|a, b| (a.0).cmp(b.0));
28      for (hex, count) in histogram.iter() {
29          print!("{hex:02x}: ");
30          if **count < 512 {
31              (0..=(*count / 8)).for_each(|_| print!("▉"));
32              println!("{} {count}", BAR[(*count % 8) as usize]);
33          } else {
34              (0..=67).for_each(|_| print!("▉"));
35              println!("▒▒▒ {count}");
36          }
37      }
38      for (hex, count) in histogram.iter() {
39          let parcent = **count as f64 / max_count as f64;
40          let count = (parcent * 255.0) as u8;
41          let (r, g, b) = hex_to_color(count);
42          print!("{}", format!("{hex:02x}").on_truecolor(r, g, b));
43
44          if *hex % 16 == 15 {
45              println!();
46          }
47      }
48      println!();
49  }
```

リスト 3: ヒストグラムを表示するためのプログラム

HashMap を用意して分布を数え上げ，個数に応じて四角を出力している．分布に応じた四角が想定の画面の幅を上回りそうなときはフェードアウトの処理を入れ個数だけを表示する．この方法の問題点としてファイルサイズが非常に巨大な場合はすべての分布が幅を超えてしまうことが挙げられる．対策が必要な場合は全体のファイルサイズや分布に応じてスケールを変えることで対応する必要がある．

また，最大値を基準にその割合で表示を変える方法も検討したが，ファイルの傾向として 0x00 が極端に多いのでそれを基準にすると他の分布の表現力が極端に落ちてしまう可能性があったので採用しなかった．

ヒストグラムは縦に並べて表示されるがこれだと 256 行分の出力があるので一覧性に乏しい．そこで前述のヒートマップを使って簡単に概要を観ることができる機能も

付けた．

出力を図 5,6 に示す．

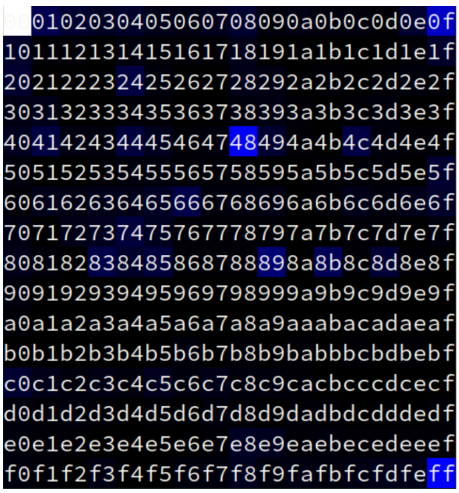

図 5: ELF のヒストグラム

図 6: 左の ELF を UPX にかけたもの

左の図 5 がある ELF ファイルのバイナリヒストグラムをヒートマップで表示したものである．前述の通り 0x00 が多いことが分かる．ついで 0xff や 0x48 が多い．

右の図 6 は左の図で使った ELF ファイルを UPX ににかけた後にバイナリヒストグラムを取ったものである．0x00 が多いのは変わらないが，黒がほとんどなく，色の分布がかなり広がっていることが視覚的に分かる．ただし図を見れば分かるようにまったく均一ではなく下位 4bit が 0 だったり 0xf だったりする左右の端に縦の帯が出来ている．また 0x40 の倍数の行は横に帯が出来ている．先頭 0x00 0x08 までが多いことも分かる．

このように簡単に分布を体感することができる．

3　おわりに

今回の実験期間を通してバイナリ解析に関わるいくつかの機能を備えたプログラムの開発を行った．ELF や PE の解析ツールの再発明だけでなく，自分の欲しい独自の機能を実装した

今後の展望としてまず，車輪の再発明として実装したバイナリ解析ツールの機能の充実化を図りたい．学習のために普段から触れている ELF と普段あまり触れない PE ファイルの両方に対応した解析ツールを書いたが，まだまだ足りない機能も多い．機能を実装すれば実装するほど詳しくなることができるので更に開発を進めていきたい．特に File Attributes のまわりは不足しているのできちんと勉強したい．

また，今回実装した機能のうちヒートマップ機能付きの hex viwer とバイナリの diff を取るツールは個人的に良いものが出来たと考えているのでレポジトリを別で作って名前を付けて開発をしようと思う．2 つとも小さな機能のプログラムなので hex viwer に diff の機能を追加する形で統合して 1 つのツールとして開発をしようと考えている．section 情報を併記するなどの機能を付けても面白いかもしれない．

今回の経験を活かして radare2 のようなもっと大きな解析ツールを開発することも考えている．実際に開発するなら今作っているような小さなプログラムから少しずつ機能を足してインクリメンタルに実装していくことになると思う．モダンで力のある解析ツールが GUI の ghidra の一強になっているのが個人的に気に入らないので，CUI から使える拡張性の高いモダンな解析ツールには意欲がある．今開発しているエミュレータに一区切りついたら作りたいものに加えておこうと思う．

参考文献

[1] iced_x86 - Rust. 2022 年 2 月 8 日閲覧.
https://docs.rs/iced-x86/latest/iced_x86/

[2] bad64 - Rust. 2022 年 2 月 8 日閲覧.
https://docs.rs/bad64/latest/bad64/index.html

[3] bad64 - Rust. 2022 年 2 月 8 日閲覧.
https://docs.rs/bad64/latest/bad64/index.html

[4] Unicode Block “Block Elements”. 2022 年 2 月 9 日閲覧.
https://www.compart.com/en/unicode/block/U+2580