Computer Security Symposium 2025
27-31 October 2025

エミュレーションベース文字列抽出ツールの出力への曖昧さの導入

大山 恵弘[1,a)]

概要：マルウェア解析ではしばしば，解析対象のプログラムが使うと推定される文字列が抽出される．文字列情報は挙動の理解やライブラリの特定などの様々な用途で有効である．文字列抽出手法の 1 つとして，プログラムの一部をエミュレーション実行する動的解析による手法がある．この手法には，実行時に生成される文字列も抽出できるという利点が存在する．しかし，エミュレーション実行は実際の実行とは異なるために，抽出される文字列情報が誤っていることや不完全であることがよく発生する．本研究では，エミュレーションベースの文字列抽出ツールの出力に曖昧さを含む注釈を導入し，正しいかどうかわからないが有用かもしれない情報を積極的にユーザに提示する手法を提案する．提案手法に基づくツールを用いた実験により，実際にバイナリプログラムから，曖昧さを含むが有用と思われる情報を抽出できたことを確認した．

キーワード：マルウェア，文字列，動的解析，エミュレーション，難読化

Introduction of Ambiguity into the Output of Emulation-Based String Extraction Tools

Yoshihiro Oyama[1,a)]

Abstract: Malware analysis often involves extracting strings presumed to be used by the program under analysis. String information is useful for various purposes, such as understanding behavior and identifying libraries. One string-extraction method is based on dynamic analysis, which emulates part of the program's execution. This method has the advantage of being able to extract strings generated at runtime. However, because emulation differs from actual execution, it is common for the extracted string information to be incorrect or incomplete. In this study, we propose a method that introduces annotations containing ambiguity into the output of an emulation-based string extraction tool, aggressively presenting users with potentially useful information even when its accuracy is uncertain. Experiments using a tool based on the proposed method confirmed that ambiguous but potentially useful information could actually be extracted from binary programs.

Keywords: Malware, character strings, dynamic analysis, emulation, obfuscation

1. はじめに

マルウェアが使用する文字列はマルウェアの挙動や内部構造を理解する上で最も重要な手がかりの 1 つである．マルウェア解析ではしばしば，マルウェアのバイナリプログラムから文字列が抽出されるが，そのための手法は多様である．簡便かつ広く用いられているのは strings コマンドを用いる静的手法であり，この手法ではプログラムのバイト列から一定以上の長さの印字可能（printable）文字の列を文字列として抽出する．静的手法による文字列抽出は，多くの文字列かもしれないデータを高速に抽出できる点で優れているが，プログラムのバイト列に出現しない文字列，

[1] 筑波大学
University of Tsukuba
[a)] oyama@cs.tsukuba.ac.jp

たとえば実行時にメモリ上に作成される文字列を必ずしも抽出できないという欠点を有している．今日，無視できない割合のマルウェアがパッカーでパックされているが，そのようなマルウェアでは使用する文字列のほとんどが暗号化されていることが多く，strings コマンドを使うだけのような単純な静的手法では，意味のある文字列はほとんど得られない．

実行時に作成される文字列を抽出するには，動的解析による手法が有効である．この手法では，解析対象のプログラムの状態を監視，制御しながら実行し，適切なタイミングでメモリの情報を読み取ってそこから文字列かもしれないデータを抽出する．たとえば FLOSS [9,12] はその手法に基づく文字列抽出ツールである．FLOSS では，対象プログラムの中の特に文字列処理に関係していると推測されるコード部分をエミュレーション実行する．実行の重要なポイントでメモリのスナップショットを取得し，スナップショットから文字列候補を抽出する．これにより，プログラムがスタック上に作る文字列や，暗号を復号してメモリに書かれた平文の文字列を抽出することができる．

静的解析でも動的手法でも当然ながら，抽出した文字列は対象プログラムが文字列として使用しているデータであるとは限らず，文字列にもみなせる他のデータである可能性がある．たとえば strings コマンドは本来文字列ではないデータを文字列として多くの場合に抽出することが知られている．動的解析によるツールでも，メモリやファイルのデータのうちどの部分が文字列であるかを常に正しく判断することは不可能であり，それがゆえに出力には不可避的に曖昧な情報が含まれる．また，動的解析では，プログラム全体を最初から最後まで実行することが時間やコストの面から難しいことがあり，プログラムの一部の実行を省略したり打ち切ったりすることがあるが，それに起因する曖昧さも文字列抽出結果に含まれることとなる．しかし，これまでの研究では我々の知る限り，動的解析で抽出された文字列情報のどの部分にどのような曖昧さが存在しうるかをユーザに示す手法はほとんど考えられてこなかった．

そこで本研究では，動的解析によりバイナリプログラムから文字列を抽出するツールにおいて，どの部分がどう曖昧であるかを明示して結果を出力する手法を提案する．具体的には，プログラムのエミュレーション実行に起因する曖昧さや，メモリ上のどの範囲が文字列であるかの判断に起因する曖昧さを明示する．これによりユーザは出力のうちのどの部分の信頼性が高いかや，信頼性が低い部分にどのような曖昧さが含まれるかをより良く把握することができる．本研究では文字列抽出ツール FLOSS を提案手法で拡張し，拡張ツールをいくつかのプログラムに適用した結果，抽出された文字列に内在する曖昧さに関して有用と考えられる情報が出力されたことを確認した．

本論文の構成は以下の通りである．2 章では動的解析による文字列抽出手法，特に FLOSS で用いられている手法を説明する．3 章では提案手法について述べる．4 章では提案手法に基づくツールをマルウェアに適用して得られた知見について報告する．5 章では関連研究を説明し，本研究との関係を明らかにする．6 章ではまとめと今後の課題を示す．

2. 動的解析による文字列抽出

動的解析による文字列抽出の基本的な手法は，プログラムを監視しながら実行して所定の場所やイベントでプログラムを止め，メモリなどから文字列かもしれないデータを抽出するというものである．先に述べた FLOSS もこの手法に基づくツールである．以下では FLOSS の処理を説明する．

FLOSS は The FLARE Obfuscated String Solver を短縮した名前であり，かつては FireEye Labs が開発しており現在では GitHub においてオープンソースで維持されている．FLOSS は Python で記述されており，少なくとも Windows と Linux で動作する．FLOSS のユーザは strings コマンドと同様にコマンドラインで，解析したいファイルを引数に与えて FLOSS を起動する．すると，FLOSS はそのファイルを解析し，抽出した文字列を表示する．FLOSS の特徴は難読化されている文字列を復元して抽出する能力が高いことである．FLOSS は以下の 4 種類の文字列を抽出する．

(1) static strings: ファイルのバイト列に含まれる ASCII および UTF-16LE コードの文字列
(2) stack strings: 実行時にスタックに作られる文字列
(3) decoded strings: 復号のための関数によって暗号データを復号して作られる文字列
(4) tight strings: stack strings と decoded strings の特徴を共に有する文字列であり，暗号データをスタック上に復号して作られる文字列．復号処理が tight loop で実現されることが多いことからこの名前になっている．

FLOSS は抽出した文字列を種類ごとに分けて表示する．static strings は strings コマンドが抽出するのと同様の文字列であるので，オプションなどによる細かい挙動の違いを考えなければ，FLOSS は strings コマンドを包含している．

stack strings，decoded strings，tight strings の抽出処理では基本的に以下の処理が実行される．

(1) 対象のバイナリプログラムを逆アセンブルして制御フローを解析し，関数や基本ブロックを認識する．
(2) 各関数に対して，復号用の関数らしさのスコアをヒューリスティクスによって付け，復号用の関数と思われる関数群を選び出す．
(3) 復号用の関数と思われる関数の呼び出しを含む各プログラム部分をエミュレーション実行する．これによ

り，復号用の関数を呼び出す際のありうる実行状態すべてを得る．

(4) 前項で得た各実行状態から復号用の関数をエミュレーション実行する．その際には，条件分岐命令に関して総当たりの形で命令列を実行し，プログラムの重要なポイントでの推定される実行状態すべてを得る．また，そのポイントでメモリのスナップショットを取得する．

(5) 復号用の関数と思われる関数の実行の前と後でメモリのスナップショットを取り，それらの間の差分を抽出する．

(6) スナップショットやメモリの差分から，一定数（デフォルトでは 4 文字以上 2048 文字以下）連続する印字可能文字の列を抽出する．

抽出する文字列の種類に応じて，この基本アルゴリズムを適切に変化させた処理が実行される．たとえば stack strings の抽出処理ではスタックのメモリのスナップショットが取られたり，decoded strings や tight strings の抽出処理では復号用の関数を高い精度で特定するためのヒューリスティクスが実行されたりする．エミュレーション実行は，FLOSS が利用している vivisect という Python ライブラリが x86 の命令列を解釈実行することによって実現される．

3. 提案手法

本研究ではエミュレーションベースの文字列抽出ツールからの出力に，曖昧さを示す情報や，曖昧だが有用かもしれない情報を注釈として付与する手法を提案する．本研究ではその手法で FLOSS を拡張したツール FLOTT を作成した．使用した FLOSS のバージョンは v3.1.0-50-ge4595b2 である．提案手法で付与される情報を，情報の種類ごとに以下で順に説明する．どれも strings コマンドの使用では生じ得ない曖昧さについての情報である．

以下で説明する実験はどれも，Ubuntu Linux 22.04.5 (x86_64) 上で行われた．コンパイラには Mingw32-gcc 10-win32 20220113 を使用し，Windows (x86 64bit) 用のバイナリプログラムを作成して FLOSS に与えた．

3.1 実行のスキップに伴う実行状態の曖昧さへの対処

エミュレーション実行でプログラムを最初から最後まで忠実に実行すると長大な時間や無限の時間がかかる可能性があるため，プログラムの一部のみを実行する必要や，一部の処理をスキップする必要がある．FLOSS では，復号処理を実行していると推測される関数を優先的に実行する．基本的には関数の最初から最後まで命令列を忠実に実行し，条件分岐命令では両場合を総当たりで実行する．しかし，関数呼び出しについては実行をスキップすることがある．関数呼び出しをどの程度エミュレーション実行するかは，解析精度と解析時間のトレードオフの問題となるが，解析を有限時間内に収めるために，関数実行を辿ることはいずれどこかで打ち切ることとなる．また，FLOSS は，対象のプログラム部分のエミュレーション実行の命令数が一定数を超えたときにも実行を打ち切る．これは無限ループに対処するために必要不可欠な措置である．実行の打ち切りにより，推定される実行状態には必然的に曖昧さが含まれる．

関数呼び出しをスキップする場合，関数の返り値としては（2025 年 8 月時点での実装では）必ず 0 が返される．スキップした関数が実行する処理や返り値としては様々なものが考えられるが，場合によっては，その関数が文字情報を返すこともある．その場合には，返り値を固定値 0（文字としては '\0'）であると仮定するのではなく，不明の値（ワイルドカード）とみなし，不明であることも含めて情報を提示するほうが，ユーザに有用な情報を提供できる可能性がある．

そのような例として，**図 1** のプログラムを考える．このプログラムは [7] の動画で文字列抽出を回避するとして説明されているプログラムを若干修正したものである．このプログラムを FLOSS および FLOTT で処理すると，stack string として**図 2** に示す文字列が出力される．このプログラムは `main` 関数の先頭でスタック上に無意味な言葉の文字列を作り，それを 1 文字ずつ別の文字で上書きしながら URL の文字列を作っていく．上書きする文字の一部は `plus1` という関数の呼び出しの返り値で与えられる．最後に URL の文字列を `printf` 関数で表示する．

FLOSS はこのプログラムに対しては `main` 関数をエミュレーション実行するが，`plus1` 関数の呼び出しはスキップする．`plus1` 関数の返り値は 0 であるとして関数呼び出し時点でのスタックのスナップショットを取得し，そこから文字列かもしれないデータを抽出する．0 は '\0' という文字列の終端を表す数であるため，返り値 0 で文字列を上書きするたびに，その書き込み場所で文字列が分割されていく．その結果，FLOSS の出力も短い部分文字列の集まりになり，このプログラムが実行時にどういうデータを作成しているのかが把握しにくくなる．

FLOTT では，スキップする関数の返り値としては 0 ではなくワイルドカードを示す文字（現在の実装のデフォルトでは *）を返す．これにより，本来分割されるはずではなかった文字列が分割されなくなるとともに，抽出した文字列のうち，どの文字は実際の実行で出現する可能性が高く，どの文字は解析では不明だったかをユーザに伝えることができる．少なくともこのプログラムでは，作成される文字列が URL であるとユーザが推定するのは，FLOSS の出力からよりも，FLOTT の出力からのほうがずっと容易であると考える．

当然ながら，関数の返り値は文字データとして使われる場合もあれば，そうでない場合もある．返り値は真偽値か

```
int plus1(int x)
{
  return x + 1;
}

int main(void)
{
  char str[] = "hooper-dooper-hoop!";
  *(str+1) = plus1('s'); // 't'
  // "htoper-dooper-hoop!"
  *(str+2) = plus1('s'); // 't'
  // "httper-dooper-hoop!"
  *(str+4) = 's';
  // "httpsr-dooper-hoop!"
  *(str+5) = plus1('9'); // ':';
  *(str+6) = '/';
  *(str+7) = '/';
  *(str+8) = '1';
  // "https://1oper-hoop!"
  *(str+9) = plus1('8'); // '9'
  *(str+10) = '2';
  // "https://192er-hoop!"
  *(str+11) = plus1('-'); // '.'
  // "https://192.r-hoop!"
  *(str+12) = '1';
  // "https://192.1-hoop!"
  *(str+13) = '6';
  // "https://192.16hoop!"
  *(str+14) = plus1('7'); // '8'
  // "https://192.168oop!"
  *(str+15) = '.';
  *(str+16) = '3';
  *(str+17) = '.';
  *(str+18) = '4';
  // "https://192.168.3.4"
  printf("%s\n", str);
  return 0;
}
```

図 1 関数の返り値を文字として使用して文字列を作成するプログラムの例

Fig. 1 Sample program that creates strings by using the return values of functions as characters.

FLOSS

```
hooper-dooper-hoop!
oper-dooper-hoop!
psr-dooper-hoop!
//1oper-hoop!
2er-hoop!
16hoop!
.3.4
```

FLOTT

```
hooper-dooper-hoop!
h*oper-dooper-hoop!
h**psr-dooper-hoop!
h**ps*//1oper-hoop!
h**ps*//1*2er-hoop!
h**ps*//1*2*16hoop!
h**ps*//1*2*16*.3.4
```

図 2 図 1 のプログラムから抽出された文字列

Fig. 2 Strings extracted from program in Fig. 1.

もしれないし，数値計算の結果かもしれない．返り値が文字データではない場合には特に，不明の返り値を 0 とするのがよいか，ワイルドカードを示す文字のコードにするのがよいか，その他の数（たとえばランダムな数）にするのがよいかは一概には判断できない．本研究では，そのような事情も理解した上で，返り値が文字として使用される場合を重視して，その場合に最適な出力を作成することを優先して手法を構築している．

また，現在の実装では抽出された文字列において文字データとしての * と，不明であることを示す特殊なデータとしての * を区別していない．ユーザはそれを了解してツールを使用する必要がある．不明な部分に使う文字はツール起動時のオプションで変えられるため，ユーザはそれを変えて 2 通りの出力を得た上でそれらを組み合わせれば，不明な部分についてある程度確からしい情報は得られる．

3.2 文字列範囲の曖昧さへの対処

パックされたマルウェアをはじめとする多くのマルウェアは，暗号データを実行時にメモリ上に復号し，できた平文の中にある文字列を使用する．FLOSS は対象プログラムの復号用の関数の前と後でメモリのスナップショットを取得し，それらの差分から文字列かもしれないデータを抽出する．ここで，対象プログラムが文字列として使用するのは，差分に含まれるデータだけにとどまらない可能性がある．たとえば，差分の前や後にあるバイト列を，抽出した文字列に連結したデータを文字列として使用している可能性がある．使用する文字列内の一部だけが暗号化されており，実行時にそこが復号されて完全な文字列を形成するプログラムでは，そのようなことが生じうる．

そのような例として，**図 3** のプログラムを考える．このプログラムを FLOSS および FLOTT で処理すると，decoded string として**図 4** に示す文字列が出力される．抽出された文字列群の冒頭には出自不明の文字列が並んでいるが，これらはライブラリなどの外部モジュール由来とみなして無視する．

このプログラムはまず，一部が暗号化されている文字列をデータ領域に `input1` として，スタック領域に `input2` として作る．次に，その両文字列について，暗号部分の復号結果と平文部分が連結された文字列（`result1`，`result2`）を作り，スタック上に確保されたバッファに書き込む．書き込まれた 2 つの文字列を `printf` 関数で表示する．復号の方法は各バイトに鍵（`key`）の値を足すという単純なものであり，`decrypt` 関数がそれを実行する．表示される文字列は，プログラムに書かれた文字列リテラルではなく，暗号部分を復号した後の `I say "Hello world," and you smile.` と `He said "Konnichiwa sekai," and she nodded.` である．

FLOSS による解析では `decrypt` が復号用の関数とみなされ，その実行の前後でのメモリのスナップショットの差分が作られる．さらに，その中から印字可能文字が一定数以上連続するバイト列が文字列として抽出される．このプログラムでは，その差分は復号対象部分のみである．しか

```
void decrypt(char *result, char *cipher,
             size_t len, int key)
{
  char *p = cipher;  char *q = result;
  while (len-- > 0) {
    *q++ = *p++ + key;
  }
}

char *input1 =
       "I say \"Ifmmp!xpsme,\" and you smile.";

int main(void)
{
  char input2[] =
    "He said \"Lpoojdijxb!tflbj,\" and she nodded.";
  char result1[256], result2[256];
  int cipher_offs1, cipher_offs2;
  int input_len1 = 37, input_len2 = 45;
  int cipher_len1, cipher_len2, key;

  memset(result1, '\0', 256);
  memset(result2, '\0', 256);
  memcpy(result1, input1, input_len1);
  memcpy(result2, input2, input_len2);

  cipher_offs1 =  7; cipher_offs2 =  9;
  cipher_len1  = 11; cipher_len2  = 16;
  key = -1;
  decrypt(result1 + cipher_offs1,
    input1 + cipher_offs1, cipher_len1, key);
  decrypt(result2 + cipher_offs2,
    input2 + cipher_offs2, cipher_len2, key);

  printf("%s\n", result1);
  printf("%s\n", result2);
  return 0;
}
```

図 3 各文字列の一部だけが暗号化されているプログラムの例
Fig. 3 Sample program in which only part of each string is encrypted.

```
FLOSS
015,23
Infinity
kpnJ
Konnichiwa sekai
Hello world

FLOTT
015,23
Infinity
kpnJ
Konnichiwa sekai
[He said "]Konnichiwa sekai[," and she nodded.]
Hello world
[I say "]Hello world[," and you smile.]
```

図 4 図 3 のプログラムから抽出された文字列
Fig. 4 Strings extracted from program in Fig. 3.

し，このプログラムが文字列として使用しているのは，その部分とその前後の平文部分を連結したバイト列である．一般に，暗号化を用いるプログラムの中には，自身が用いるデータの一部だけを暗号化したデータを保持し，実行時にそれを復号したデータと他の平文データを組み合わせて用いることがある．しかし，プログラムの意図や挙動を常に正しく把握することは不可能であり，復号した暗号データの前後に印字可能文字が並んでいたとしても，それらを連結するべきかどうかは自明ではない．文字列抽出ツールも，復号後文字列の前や後に別の文字列が続いているかもしれないし続いていないかもしれないという曖昧な判断しかできないことが多いと考えられる．

FLOTT では，スナップショットの差分から文字列の候補を取得すると共に，差分の前後のメモリ上にある印字可能文字の並びもあわせて取得する．それらの並びは，文字列の候補に連結されているかもしれないデータとして扱う．文字列をユーザに示す際には，それらの並びを角括弧（ブラケット）でくくり，色付きで表示する．これにより，対象プログラムが使う可能性が高い文字列のバリエーションをユーザに伝える．

3.3 データが文字列かどうかについての曖昧さへの対処

ファイルやメモリのデータから印字可能文字の列を抽出してそれらを文字列の候補とみなすのは，静的解析においても動的解析においても基本的かつ強力な手法である．しかし，抽出した印字可能文字の列が文字列なのか，文字列のように見える他のデータなのかについて常に正しく判断するのは難しく，抽出結果は必然的に不確実性を内包する．

そのような例として，**図 5** のプログラムを考える．このプログラムを FLOSS および FLOTT で処理すると，decoded string として**図 6** に示す文字列が出力される．ただし，(maybe pointed) は FLOTT を使用した場合にのみ出力される．このプログラムは符号無し 8 ビット整数の配列（x）と文字の配列（buf）のそれぞれに，0 から始まる整数列と A から始まるアルファベットの列を書き込む．その後，整数列に含まれる整数の和とアルファベットの文字列を printf 関数で表示して終了する．冒頭の出自不明の文字列群は先ほどの例と同様に無視する．次に出現する !"#$%&'()*+,-./0123456789:; は，整数の列を文字列と誤認識して抽出された文字列である．プログラムは整数の列を使っているが，それはメモリデータとしては印字可能文字の列と区別がつかない．一方，文字列として使っている ABCDEFGHIJKLMNOPQRSTUVWXYZ は正しく抽出されている．

FLOTT は FLOSS とは異なり，ABCDEFGHIJKLMNOPQRSTUVWXYZ に (maybe pointed) という色付きの注釈を付ける．この注釈は，メモリのどこかにその文字列の先頭アドレスが格納されている（すなわち，その文字列がメモリのどこかから指されている）ことを意味するものである．この注釈は，自身が付けられた文字列はプログラム中で実際に文字列として使われている可能性が高く，付けられていない文字列はその可能性が低いことをユーザに伝える．この注釈が付けられないが抽出される文字列の例としては，図 5 で示した整数列の他にも，実行時にメモリに書き込んだデータの途中部分を指して利

```c
void fill_numbers(uint8_t *x, uint8_t num, int cnt)
{
  for (int i = 0; i < cnt; i++) {
    x[i] = num++;
  }
}

void fill_alphabets(char *buf)
{
  for (int i = 0; i < 26; i++) {
    buf[i] = 'A' + i;
  }
}

int main(void)
{
  uint8_t x[64];
  char buf[64];
  char *b = buf;
  int sum = 0;
  memset(x, 0, sizeof(uint8_t) * 64);
  memset(b, 0, sizeof(char) * 64);
  fill_numbers(x, 0, 60);
  fill_alphabets(b);
  for (int i = 0; i < 60; i++) sum += x[i];
    printf("%d\n", sum);
    printf("%s\n", b);
    return 0;
}
```

図 5 文字列データとみなせる整数列が使用されるプログラムの例
Fig. 5 Sample program that uses a sequence of integers that can be regarded as string data.

```
015,23
Infinity
kpnJ
 !"#$%&'()*+,-./0123456789:;
ABCDEFGHIJKLMNOPQRSTUVWXYZ     (maybe pointed)
0771
```

図 6 図 5 のプログラムから抽出された文字列
Fig. 6 Strings extracted from program in Fig. 5.

用される文字列など様々なものがある．

FLOTT はこの注釈を付けるために，復号用の関数の実行前後のメモリのスナップショットの差分から文字列を抽出するたびに，実行後のスナップショットをスキャンし，その文字列の先頭アドレスが出現するかどうかを調べる．存在したらその文字列には注釈を付ける．

プログラムのメモリ使用量が多い場合にはスナップショットも大きくなるので，この注釈を付けるための処理には長い時間がかかる．FLOTT のユーザは自身にとってのこの注釈の重要性と処理時間のトレードオフを考慮して，この注釈のための処理を有効にも無効にもできる．

4. マルウェアへの適用

FLOSS と FLOTT を実際のマルウェアに適用して文字列を抽出する実験を行った．マルウェアの検体は Malpedia-FLOSSed という Web サイト [8] で提供されているものを用いた．Malpedia-FLOSSed では，Malpedia というマルウェア情報提供 Web サイトで扱っているマルウェアに対して FLOSS を適用した結果のデータセットを提供している．データセットを作成するのに用いた検体を入手することも可能である．Malpedia-FLOSSed には難読化を使用しない検体も含まれており，stack string や decoded string の文字列がまったく抽出されない検体も少なくない．ここでは，Malpedia-FLOSSed が扱っている検体に FLOTT を適用した結果のうち興味深いものを取り上げて説明する．

```
FLOSS
...
InitializeCriticalSection
EnterCritical
ection
...
ExitThread
teThre
DeleteCriticalSection
...
ExitProcess
oduleHandleA
MultiByteToWideChar
...
cmd.exe /c net stop AVP /y
cmd.exe /c net stop SQL
gent$SOPHOS /y
cmd.exe /c net stop MSSQL$SOPHOS /y
...

FLOTT
InitializeCriticalSection
EnterCritical
EnterCritical[Section]
ection
[EnterCriticalS]ection
...
ExitThread
Cre[ateThread]
teThre
[Crea]teThre[ad]
[CreateThrea]d
DeleteCriticalSection
...
ExitProcess
Get[ModuleHandleA]
oduleHandleA
[GetM]oduleHandleA
MultiByteToWideChar
...
cmd.exe /c net stop AVP /y
cmd.exe /c net stop SQL
cmd.exe /c net stop SQL[Agent$SOPHOS /y]
gent$SOPHOS /y
[cmd.exe /c net stop SQLA]gent$SOPHOS /y
cmd.exe /c net stop MSSQL$SOPHOS /y
...
```

図 7 FLOSS と FLOTT をマルウェア検体 0x004ede55... に適用した結果の抜粋
Fig. 7 Extracted results of applylication of FLOSS and FLOTT to malware sample 0x004ede55....

ハッシュ値 0x004ede55... の検体は難読化を用いていると推定され，FLOSS を用いると decoded string の文字列を 286 個抽出できる．しかし，それらの中には何を意味するのかが理解しにくいものも多く含まれる．FLOTT を用いるとその数が 407 個になり，さらに，理解しにくかった文字列を理解する手がかりを多く入手できる．**図 7**

```
FLOSS
LoadLibraryA
GetProcAddress
xLOZ

FLOTT
xLOZ
LoadLibraryA
GetProcAddress
C,R=****
G,R=****
```

図 8 FLOSS と FLOTT をマルウェア検体 0x01686d9c... に適用した結果の抜粋

Fig. 8 Extracted results of applylication of FLOSS and FLOTT to malware sample 0x01686d9c....

はそれらの文字列の一部を示したものである．FLOSS では `EnterCritical`，`ection`，`teThre`，`oduleHandleA` といった，意味が自明ではない文字列が抽出されているが，FLOTT の出力では，それらは API 名の部分文字列として使用されていることが容易にわかるようになっている．また，FLOSS では `cmd.exe /c net stop SQ` という文字列と `gent$SOPHOS /y` という文字列が出力されておりこれらの文字列の意図が掴みにくいが，FLOTT では全体として `cmd.exe /c net stop SQLAgent$SOPHOS /y` という SQL 文を使用しようとしていることが容易にわかる．

次に，ハッシュ値 0x01686d9c... の検体に FLOSS と FLOTT を適用した結果のうちの stack string の部分を**図 8** に示す．FLOSS では 3 個の文字列が抽出されているが，FLOTT ではそこに 2 個の文字列が加わっている．実行をスキップした関数の返り値を 0 ではなく文字 `*` の ASCII コードにしたことにより，`C,R=` と `G,R=` の文字列の後ろに 4 文字が書かれる可能性が認識されている．また，文字列が伸びたことにより，結果から除外されていたそれらの文字列が新たに結果に加えられることとなっている．

他には，ハッシュ値 0x6c759985... の検体については，FLOSS が抽出した stack string は 0 個だったが，FLOTT では `*tflower*` と `9EYeP` の 2 個の文字列を抽出した．なお, ワイルドカードを示す文字を `X` に切り替えて FLOTT を実行すると，`*tflower*` は `*tflowerX` に変わった．VirusTotal での検索結果によれば，このマルウェアには複数ベンダが TFlower という名称を付けている．`*tflower` は TFlower という名称が付いた理由と予想される象徴的な文字列であり，この文字列が抽出できることは，このマルウェアの特徴を迅速に把握する上で重要であると考える．

5. 関連研究

暗号化，パック，難読化されたマルウェアから元のデータを抽出するためのマルウェア解析手法は数多く提案されている．特に文字列に注目したものとしては，Mondon らによる研究 [10] がある．この研究ではマルウェアのバイナリコードを解析して文字列の難読化に関係しているコードをみつける検出器を生成するための手法を提案している．本研究が使用している FLOSS と彼らの研究は，バイナリコードの中から難読化に関係する部分を特定しようとする点で共通している．しかし，彼らの研究が静的解析による難読化関連コードの発見に主眼を置いているのに対して，本研究は動的解析で得た情報の質の向上に主眼を置いている．

パックされたファイルからパック前のファイルを復元する（パックされたファイルをアンパックする）技術も数多く提案されている [2, 3, 5, 11]．プログラムのファイルがパックされると，そのプログラムが使用する文字列などのデータを静的解析で得ることは難しくなるが，アンパックすることによってそれらのデータを得られるようになる．アンパックする技術と本研究で扱っている技術は，難読化された文字列を抽出するのに効果的であるという共通点はあるが，平文のコードとデータの復元を目指すか，実行状態からプログラムの使用するデータを抽出することを目指すかという違いがある．実際，4 章で示したプログラムはどれも，コードもデータも平文だが単純な静的手法での文字列抽出が難しいプログラムである．両技術は補完的なものであり，組み合わせて使用することでより多くの情報を得ることができる．

StringHound [4] は Android アプリケーションの Java バイトコードを対象に，難読化された文字列を抽出するツールである．StringHound は FLOSS と同様に，難読化された文字列に関する処理を実行していると推定されるコード部分を抜き出して実行し，平文の文字列を復元する．しかし，FLOSS とは対象とするコードの種類が異なるうえ，手法の詳細も異なる．本研究で扱ったような，動的解析において曖昧な形でしか得られない情報を積極的にユーザに提示する拡張は，StringHound など FLOSS 以外の文字列抽出ツール一般にも適用可能であり，その具体的な実現方法は興味深い研究テーマであると考える．

文献 [6] では，バイナリ解析ツール Binary Ninja およびそのプラグイン Sidekick により，FLOSS と似た処理を通じて難読化文字列から平文の文字列を得る手順が実例とともに説明されている．その手順は，復号用の関数の検出や復号用の関数のエミュレーション実行を LLM に依頼して実現している点に特徴がある．本研究では FLOSS をベースに手法を検討したが，今後は LLM を用いたツールをベースに，あるべき手法を検討することも必要になると思われる．

Chen らの研究 [1] では，LLM の存在を前提とした上でマルウェア分類における文字列情報の有用性を評価している．彼らの研究では FLOSS で抽出した文字列だけを使用した場合と，Falcon Sandbox によるマルウェアのサンド

ボックス実行で抽出した文字列も組み合わせて使用した場合とで分類精度がどう変わるかを調べている．彼らの研究と本研究は，FLOSS が出力する情報は十分ではなく，質の高いマルウェア解析をするには付加的な情報が必要であるという問題意識を共有している．彼らの研究では FLOSS とサンドボックス実行を組み合わせる手法や LLM を積極的に活用する手法を構築しており，それらの手法は重要な判断がブラックボックスの仕組みに依存するという特徴を有している．一方，本研究ではブラックボックスの仕組みの導入を避け，FLOSS の出力の質そのものを高める方向で有用な手法の構築を狙っている．

6. まとめと今後の課題

本論文では，エミュレーション実行によってバイナリプログラムから文字列を抽出するツールにおいて，曖昧さに関する情報を，そうであることがわかる形でユーザに提示して理解や判断を支援する手法を提案した．その手法で既存の文字列抽出ツールを拡張していくつかのプログラムから文字列を抽出したところ，拡張前に比べてより有用な情報を出力できたことを確認した．

今後の課題を以下に述べる．第一に，マルウェアを含む実際のプログラムに提案手法を適用した結果のデータおよびその評価が不足しているので，今後さらなる実験によりデータを収集する必要がある．4 章で述べたように，FLOSS によるマルウェアの解析結果については Malpedia-FLOSSed [8] というデータセットが公開されている．このデータセットおよびその作成に用いられた検体により，提案手法のさらなる機能拡張や評価がしやすくなると考える．提案手法で得られた情報に対する意見をユーザスタディなどにより収集することも重要である．第二に，提案手法では，エミュレーション実行中に得られる様々な情報のうち，活用されることなく捨てられているものがまだ多数あるので，それらも有用で使いやすい形でユーザに提示することが望ましいと考える．第三に，文字列以外のデータの難読化への対処に研究対象を広げることも有益であると考える．文字列は難読化されるデータのうちの 1 つでしかなく，コードや数値データなど他にも難読化されうるデータは多く存在する．それらのデータについても本研究と同様の考察が必要である．

謝辞 本研究に対しては富士通株式会社の小久保博崇氏から有益なコメントをいただいた．本研究は JSPS 科研費 JP23K11096 の助成を受けたものである．

参考文献

[1] Chen, Y., Wu, D., Zhong, J., Zhang, Z., Gao, D., Wang, S., Li, Y. and Liu, N.: Rethinking and Exploring String-Based Malware Family Classification in the Era of LLMs and RAG, arXiv, https://arxiv.org/abs/2507.04055 (2025).

[2] Cheng, B., Ming, J., Fu, J., Peng, G., Chen, T., Zhang, X. and Marion, J.-Y.: Towards Paving the Way for Large-Scale Windows Malware Analysis: Generic Binary Unpacking with Orders-of-Magnitude Performance Boost, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 395–411 (2018).

[3] Cheng, B., Ming, J., Leal, E. A., Zhang, H., Fu, J., Peng, G. and Marion, J.-Y.: Obfuscation-Resilient Executable Payload Extraction From Packed Malware, *Proceedings of the 30th USENIX Security Symposium*, pp. 3451–3468 (2021).

[4] Glanz, L., Müller, P., Baumgärtner, L., Reif, M., Amann, S., Anthonysamy, P. and Mezini., M.: Hidden in Plain Sight: Obfuscated Strings Threatening Your Privacy, *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pp. 694–707 (2020).

[5] Kang, M. G., Poosankam, P. and Yin, H.: Renovo: a Hidden Code Extractor for Packed Executables, *Proceedings of the 2007 ACM Workshop on Recurring Malcode*, pp. 46–53 (2007).

[6] Knudson, B.: Binary Ninja Blog: Sidekick in Action: Deobfuscating Strings in Amadey Malware, https://binary.ninja/2024/08/12/sidekick-in-action-deobfuscating-strings-in-amadey-malware.html (2024).

[7] L!NK: Stack Strings and Defeating the FLOSS tool, https://www.youtube.com/watch?v=DV4DKq7zTfE.

[8] Malpedia: Malpedia-FLOSSed, https://github.com/malpedia/malpedia-flossed.

[9] Mandiant: FLARE Obfuscated String Solver, https://github.com/mandiant/flare-floss.

[10] Mondon, P. and de Lemos, R.: Detecting Cryptographic Functions for String Obfuscation, *Proceedings of the 2024 IEEE International Conference on Cyber Security and Resilience*, pp. 315–320 (2024).

[11] Muralidharan, T., Cohen, A., Gerson, N. and Nissim, N.: File Packing from the Malware Perspective: Techniques, Analysis Approaches, and Directions for Enhancements, *ACM Computing Surveys*, Vol. 55, No. 5 (2022).

[12] Raabe, M. and Ballenthin, W.: Automatically Extracting Obfuscated Strings from Malware using the FireEye Labs Obfuscated String Solver (FLOSS), https://cloud.google.com/blog/topics/threat-intelligence/automatically-extracting-obfuscated-strings/ (2016).