難読化コードに対する暗号関数特定手法の提案

古川 凌也† 伊沢 亮一‡ 森井 昌克† 井上 大介‡ 中尾 康二‡

†神戸大学大学院工学研究科
657-8501 兵庫県神戸市灘区六甲台町 1-1
furukawa@stu.kobe-u.ac.jp,
mmorii@kobe-u.ac.jp

‡国立研究開発法人情報通信研究機構
184-8795 東京都小金井市貫井北町 4-2-1
{isawa,dai,ko-nakao}@nict.go.jp

あらまし 近年，多くのマルウェアは暗号技術を用いて通信の秘匿やパッキングを行い，解析を妨害する．このようなマルウェアは自身のバイナリ内に暗号アルゴリズムを持ち，暗号化及び復号を行う．マルウェアのバイナリから暗号アルゴリズムを有する暗号化/復号関数の位置を特定し，そこから鍵や平文などの情報を得る研究がなされている．暗号化/復号関数の位置を特定するためには，テイント解析などのデータフローを追跡する技術を用いた手法が有効である．しかし，マルウェアがデータの伝搬を意図的に隠蔽するような解析妨害を行う場合には，純粋なテイント解析のみを用いた手法では対応することができなくなる．本研究ではそのように隠蔽されたデータフローを補完し，暗号化/復号関数の位置を特定する手法を提案する．評価実験では，10 個のサンプルプログラムに対して本提案手法を適用し，そのうち 9 個に対して，復号関数の位置を一意に特定することができた．

Locating Cryptographic Functions against Obfuscated Code

Ryoya Furukawa† Ryoichi Isawa‡ Masakatu Morii† Daisuke Inoue‡
Koji Nakao‡

†Graduate School of Engineering, Kobe University.
1-1 Rokkodai-cho, Nada-ku, Kobe-shi, Hyogo 657-8501, JAPAN
furukawa@stu.kobe-u.ac.jp,mmorii@kobe-u.ac.jp

‡National Institute of Information and Communications Technology.
4-2-1, Nukui-Kitamachi, Koganei-shi, Tokyo 184-8795, JAPAN
{isawa,dai,ko-nakao}@nict.go.jp

Abstract More and more malwares are using cryptographic algorithms to protect themselves from being analyzed. They have cryptographic functions in their binary code. We focus on methods to automatically find cryptographic functions in binary code of malwares. Especially, methods using taint-tracking are effective. However, malware can evade such taint-based methods by using the obfuscated code(e.g., information transferring with under-tainting).

In this paper, we propose effective method that complements under-tainting and locates cryptographic functions against such the obfuscated code. With experiments using ten cryptographic programs, we confirmed that our method uniquely located the cryptographic function against nine programs.

1　はじめに

マルウェアを利用したサイバー攻撃への対策のためには，マルウェアを解析し，その挙動を把握することが重要である．しかし近年，多くのマルウェアは暗号技術を利用し，解析を妨害する．例えば，C&C サーバや攻撃者との通信を暗号化することで送受信されるデータを秘匿するなどである．

マルウェアの実行命令系列を取得し，そこから暗号アルゴリズムを含む関数の位置や暗号アルゴリズムの種類の特定，鍵や平文などのパラメータを取得する研究がなされている [1][2][3][4]．例えば ReFormat[3] や CipherXRay[4] は，暗号化や復号の対象となるデータの伝搬をメモリ上で追跡し，そのデータに対して実行された機械語命令の特徴やデータフローの特徴から暗号アルゴリズムらしい一連の処理や関数を推定する．

これらの手法は，高い粒度でデータの伝搬を追跡する必要があり，そのための手法としてテイント解析を用いている．テイント解析は，データフロー解析の技術として有効な手法であり，プログラムのデバッグや攻撃の検知など，さまざまな目的に応用されている [5]．しかしながら，マルウェアがテイント解析を回避する目的で難読化を行った場合，純粋なテイント解析のみではデータの伝搬を正確に追跡することが困難であることが指摘されている [6]．ここでの難読化とは，検知漏れを狙ったコードによるデータフローの隠蔽 (under-tainting) である．一般的なテイント解析では，機械語命令単位で発生する直接的なメモリの読み書きをもとにデータの伝搬を追跡するが，under-tainting は，条件分岐等を用いて間接的にデータを伝搬させることによって引き起こされる．このような隠蔽されたデータフローも含めてデータの追跡を行うためには，一般的なテイント解析に加え，何らかの手法を用いたデータフローの補完が必要になる．

本稿では，そういった隠蔽されたデータフローを補完した上で，復号関数の使用目的を問わず，それらの位置を特定する手法を提案する．本提案手法は，一般に利用されているテイント解析とは異なる方法で，データの伝搬を追跡する．具体的には，一般のテイント解析のように機械語命令単位で発生するデータの伝搬を追跡するのではなく，実行される関数間での入出力データの受け渡しを追跡する．このとき，追跡対象のデータを入力として読み込んだ関数がその後に出力したすべてのデータを追跡対象とすることで，under-tainting による追跡対象の検知漏れを防止する．

本提案手法はこのようにして得られたデータフローの中から，暗号文が入力され，その後平文が出力される部分を抽出し，その間にデータが通過した関数の中から復号処理を行った関数を特定する．そのためにまず，得られたデータフローから，平文のデータを含む入出力 (平文バッファ) の候補となる箇所を抽出する．我々は，入出力がやりとりされる頻度から各関数をそれらが属する処理ごとにグルーピングし，復号対象のデータを入力とする処理から外部の処理に対して出力されるデータを平文バッファの候補とした．これは，暗号処理内で生成された平文のデータは，その後必ず別の処理において利用されるという考えに基づく．最後に，平文バッファの候補を出力した関数を開始地点として，暗号文を入力とする関数が見つかるまでデータフローをさかのぼり，その間に含まれる関数の中から復号関数を特定する．復号関数の特定においては，共通鍵暗号方式であれば，暗号文と平文の長さは等しくなるという点に着目した．

評価実験では，10 個のサンプルプログラムに対して本提案手法を適用し，そのうち 9 個に対して，復号関数の位置を一意に特定することができた．また残りの 1 個のサンプルプログラムについても検知漏れはなく，わずかな誤検出を含むのみであった．本提案手法は，マルウェアによる under-tainting などの難読化処理に影響を受けない粒度の荒いテイント解析を用いて，復号関数の位置を特定することができる．

2　基礎知識

2.1　マルウェアの動作のモデル化

本稿では，マルウェアが行う処理を暗号処理とその他の処理で定義する．それぞれの処理は

プログラム上の一連の関数によって構成される．例えば，暗号化通信が行われる際の処理は主に以下の 4 段階に分けることができる [3].

(1) 暗号化パケットの受信および復号

(2) パケットの内容（要求）の解釈

(3) その要求に対する応答の作成

(4) 応答の暗号化および送信

処理 (1) において，受信した暗号化パケットは，マルウェアが動作するメモリ上の領域にデータとして保存される．保存されたデータは暗号アルゴリズムを含む復号関数によって読み込まれ，平文データに復号される．このとき生成された平文データもまたメモリ上の領域に保存される．そして，処理 (2) において保存された平文データは読み込まれ，平文データから解釈された要求に応じて処理 (3) が実行される．その後，処理 (3) で生成された応答データは，処理 (4) において平文データとして読み込まれ，暗号化される．このように，復号処理から出力された平文データは，その後に続く処理に入力される．また暗号化処理には，それ以前の処理によって出力された平文データが入力される．

2.2 関数と read/write 命令の定義

解析環境下でマルウェアを実行し，実行された命令をトレース (実行トレース) することで，関数の呼び出しとメモリ領域への読み書きを機械語命令単位で監視することができる．本稿では，メモリからレジスタにデータをロードする機械語命令を read 命令，メモリにデータをストアする機械語命令を write 命令と定義する．また，関数の呼び出しは call 命令によって行われ，関数からの復帰は ret 命令によって行われる．したがって本稿では，call 命令が実行された後に，はじめて ret 命令が実行されるまでの範囲をある関数が実行されている範囲であると定義する．そして, この範囲に実行された read/write 命令をその関数に対する入出力であると定義する．

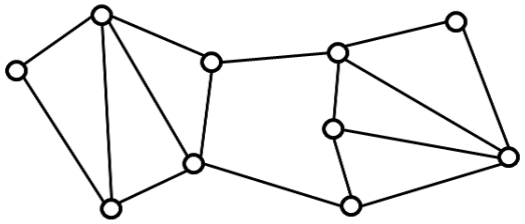

図 1: グラフの例

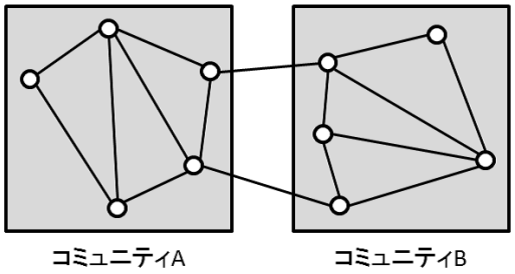

図 2: コミュニティ検出の例

2.3 グラフ

本提案手法では，各関数をそれらが属する処理ごとにグルーピングするために，グラフ理論におけるコミュニティ検出の手法を用いる．グラフとは頂点 (ノード) とそれを接続する辺 (リンク) の集合である．グラフの例を図 1 に示す．あるグラフ内の任意の 2 つのノードがリンクで接続される確率をリンク密度という．グラフのある部分集合 (サブグラフ) 内のリンク密度が相対的にそのサブグラフ外へのリンク密度よりも高いサブグラフをコミュニティという．コミュニティ検出とは，グラフ全体を複数のコミュニティに分割することを指す．図 1 のグラフは図 2 のように二つのコミュニティに分割することができる．

3 提案手法

3.1 提案手法の概要

本提案手法では，実行トレースで取得したログをもとに，実行された関数をノード，データ

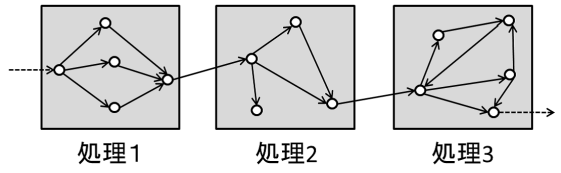

図 3: 処理ごとに形成されるコミュニティの例

の入出力をリンクとして，関数間での入出力のやり取りを表したグラフを考える．たとえば，関数 F_A が write 命令によって書き込んだメモリ領域に対して，関数 F_B が read 命令によって読み込みを行ったとき，関数 F_A と関数 F_B は入出力をやり取りしたものとしてリンクで接続される．このようにして生成したグラフを本稿では入出力グラフと呼ぶ．本提案手法では，入出力グラフ上でデータの追跡を行う．このとき，追跡対象のデータを読み込んだ関数のその後のすべての出力にデータが伝搬したものとして扱うことで，under-tainting による検知漏れを防止する．

入出力グラフ上で復号対象のデータを追跡し，復号対象のデータが入力される復号の開始位置から平文が出力される復号の終了位置までの部分を抽出できれば，その間にデータが通過した関数の中から復号関数を特定することができる．そのためにまず，復号の終了位置となる平文バッファの候補を抽出する．入出力グラフにおいて，我々はプログラム上で特定の処理に関わる一連のノードは図 3 のようにコミュニティを形成すると考えた．これは，同一の処理に含まれる関数は互いに出力したデータを入力として扱う確率が高く，リンク密度が高くなるであろうという考えに基づく．このとき，2.1 節のようなモデルを考えると，あるコミュニティの内部から外部のコミュニティのノードに対して接続されるリンクには，平文バッファが含まれていると考えられる．したがって，本提案手法では，生成した入出力グラフに対してコミュニティ検出を行い，検出された各コミュニティから外部のコミュニティへ接続されるリンクを平文バッファの候補とする．

最後に，平文バッファの候補を出力した関数を復号の終了位置と仮定して，暗号文を入力とする関数が見つかるまで入出力グラフをさかのぼり，その間に含まれる関数の中から復号関数を特定する．復号関数の特定においては，共通鍵暗号方式であれば，暗号文と平文の長さは等しくなるという点に着目した．平文バッファの候補と同じ長さであり，かつ復号対象のデータの部分列と一致するデータを入力とする関数を復号の開始位置とし，復号の終了位置までに入出力が経由するすべての関数を復号処理に関わる復号関数として出力する．

3.2 平文バッファ候補の抽出

本提案手法では，仮想環境内でマルウェアを実行し，その動作を機械語命令単位で記録した実行トレースを取得する．取得した実行トレースをもとに関数の呼び出しと復帰，書き出したメモリのアドレスと値，読み込んだメモリアドレスと値を時系列順に記録したログを生成する．このとき，実行された機械語命令をメモリへの書き出しを行う write 命令とメモリからの読み込みを行う read 命令に分け，メモリ上のどのアドレスにどのような値が書き出されたかを 1 バイト単位で記録していく．さらにログには，call 命令とそれによって呼び出される関数の開始地点のアドレス，ret 命令による関数からの復帰を記録する．

入出力グラフを生成するアルゴリズムを Algorithm1 に示す．Algorithm1 は，実行トレースをもとに生成したログを 1 行ずつ読み出し，その内容に応じた処理を行う．実行トレース中に呼び出された関数には，呼び出された順に割り振られた番号が id として与えられる．変数 j は現在着目している関数の id を保持している．読み出された行が write 命令であれば，メモリ上のアドレス $addr$ が現在着目している関数 f_j によって書き込まれたことを W_{addr} に記録する．読み出された行が read 命令であれば，読み出したアドレスに値が書き込まれているかをチェックし，すでに値が書き込まれていれば，書き込みを行った関数 W_{addr} と読み出した関数 f_j をリンクで接続する．読み出された行が call 命令または ret 命令であれば，j の値を増減させるこ

Algorithm 1 入出力グラフの生成

```
1: j ← 0
2: for all line_i do
3:    if line_i = write val in addr then
4:       w_addr ← f_j
5:       m_addr ← val
6:       add m_addr to outputs_j
7:    else if line_i = read val from addr then
8:       if m_addr is already written then
9:          connect w_addr to f_j
10:      end if
11:      m_addr ← val
12:      add m_addr to inputs_j
13:   else if line_i = call addr then
14:      j ← j + 1
15:      entry_j ← addr
16:   else if line_i = ret then
17:      j ← j − 1
18:   else
19:      pass
20:   end if
21: end for
```

とで，現在着目している関数を切り替える．またこのとき，それぞれの関数の入出力バッファや開始地点のアドレスといった情報もそれぞれ記録しておく．

入出力グラフの生成が完了した後，実行された関数を処理ごとに分割するためにコミュニティの検出を行う．本稿では，コミュニティの検出にReichardt らのアルゴリズム [7] を用いた．入出力グラフは複数のコミュニティに分割され，そのうちのいずれかに暗号処理が含まれている．その後あるコミュニティに含まれるノードから別のコミュニティに含まれるノードに接続するリンクをすべて抽出し，それらのリンクに対応するバッファを平文バッファの候補とする．

3.3 復号関数の特定

復号対象のデータを読み込んだコミュニティに対して Algorithm2 を適用し，復号関数の特定を行う．Algorithm2 について説明する．*cipher* は，コミュニティに読み込まれた復号対象のデータである．本稿では，復号対象のデータはネットワークまたはファイルに由来するデータを想定しており，これらのデータが読みだされたタイミングとメモリアドレスはファイルの読み出しやパケットの受信を行うシステムコールや API をフックすることで取得することができる．*out* はコミュニティから外部に接続されるリンクの集合であり，その全てについて以下の処理を行う．まず out_i の持つバッファに格納されているデータ *plain* が平文であると仮定する．そして，out_i の出力側の関数のノード f を暫定的に復号の終了位置 *end* とし，f に入力されるすべての

Algorithm 2 復号関数の特定

```
1: cipher ←ciphertext
2: out ← outgoing links from a community
3: for all out_i do
4:    plain ← a data in buffer of out_i
5:    f ← a tail node of out_i
6:    end ← f
7:    R ← incoming links to f
8:    for all R_j do
9:       tmp ← a data in buffer of R_j
10:      if tmp = substring of plain then
11:         f ← a tail node of R_j
12:         plain ← tmp
13:         goto 6
14:      end if
15:      if len(tmp) = len(plain) then
16:         if tmp = substring of cipher then
17:            add path end-f to decriptors
18:            i ← i + 1
19:            goto 4
20:         end if
21:      end if
22:      enqueue(tail node of R_j)
23:   end for
24:   if queue is not empty then
25:      f ←dequeue()
26:      goto 7
27:   end if
28: end for
```

表 1: 実験結果

サンプル	ライブラリ	暗号アルゴリズム
(1)	Beecrypt	AES
(2)	Gradman	AES
(3)	CryptPP	AES
(4)	CryptPP	Blowfish
(5)	CryptPP	DES
(6)	CryptPP	RC4
(7)	OpenSSL	AES
(8)	OpenSSL	Blowfish
(9)	OpenSSL	DES
(10)	OpenSSL	RC4

表 2: 平文バッファ候補の抽出結果

サンプル	総数	抽出数	削減率
(1)	2306	191	91.7
(2)	5705	426	92.5
(3)	9037	458	94.9
(4)	10926	6692	93.7
(5)	11200	681	93.9
(6)	3558	269	92.4
(7)	5624	353	93.7
(8)	1777	154	91.3
(9)	5652	358	93.7
(10)	4113	310	92.5

リンク R に対して次の処理を行う．R_j の持つバッファに格納されているデータ tmp について，$plain$ の部分列と一致する場合，メモリコピーに類する処理と判断し，復号の終了位置および着目する関数 f を R_j の出力側のノードに変更する．次に,$plain$ と tmp の長さが一致し，かつ tmp と $cipher$ の部分文字列が一致した場合，入出力に同じ長さの平文と暗号文のペアを持つことから，共通鍵暗号の特徴と一致したものとして，end から f までの探索において Algorithm2 がグラフ上で通過したパスに含まれるノードを復号処理に関わる一連の関数として出力する．tmp がどちらの条件も満たさなかった場合には，R_j の出力側のノードを探索キューに加える．f の持つすべての R_j に対して処理を完了した後，探索キューが空でなければ，f を探索キューの先頭のノードに変更し，引き続き処理を行う．以上が復号関数特定のためのアルゴリズムである．

4 評価実験

4.1 実験内容

提案手法の有効性を評価するために，複数の暗号ライブラリを用いて 10 個のサンプルプログラムを作成した．ここで作成したプログラムを用いるのは，復号関数が展開されるメモリ上の正しい位置を手動で確実に得るためである．暗号ライブラリとして Beecryfpt[11]，Brian Gladman による AES 暗号ライブラリ [9]，Crypto++[10]，OpenSSL[8] を用いた．それぞれのサンプルについて，用いたライブラリと実装した暗号アルゴリズムを表 1 にまとめる．これらのサンプルプログラムの動作は，受信した暗号化パケットを復号した後，標準出力に表示する，また入力されたデータを暗号化し，送信するというものである．実験においては，それぞれのサンプルについてパケットの送受信を数回行い，実行トレースを取得し，提案手法を用いて復号関数の特定を行った．なお本稿では，実行トレースの取得に QEMU[12] を利用した．

4.2 実験結果

まず，本提案手法により平文バッファの候補を抽出した結果を表 2 に示す．総数は生成したグラフのリンクに含まれるすべての入出力バッファであり，抽出数はあるコミュニティの内部から外部のコミュニティに対して出力されたバッファを表している．全てのサンプルにおいて 9 割以上のバッファが候補から除外されたが，いずれの場合にも抽出したリンクの中に平文が含まれていることが確認できた．

次に復号関数特定の結果を表 3 及び表 4 に示す．表 3 の総数は生成したグラフに含まれた関数のユニークな総数である．出力数は，Algo-

表 4: 提案手法の出力結果

サンプル	出力
(1)	0x6fecf320,0x69b02dab,0x6fe58230,0x7787f883,0x7778f1c0
(2)	0x1073f70
(3)	0xc035a4→0xc010c9→0xbffe5e
(4)	0xb4f9e0→0xb583bf→0xb4fdfa
(5)	0x1323710→0x12b6c26→0x1323500→0x12b2cb6→0x12b737e→0x12aeea4
(6)	0x1313dba
(7)	0x66f309c0
(8)	0x66e58f50
(9)	0x65a08760
(10)	0x66e55490

表 3: 提案手法による復号関数の特定結果

サンプル	総数	出力数	誤検出	検知漏
(1)	97	5	4	0
(2)	173	1	0	0
(3)	446	3	0	0
(4)	387	3	0	0
(5)	391	5	0	0
(6)	139	1	0	0
(7)	178	1	0	0
(8)	177	1	0	0
(9)	176	1	0	0
(10)	116	1	0	0

rithm2 によって出力されたパスに含まれる関数の個数である．誤検出は，出力された関数のうち，暗号処理に直接関係しなかった関数の個数であり，サンプル (1) を除いてすべて 0 であった．また検知漏れは，復号対象のデータに対して処理を行い，暗号アルゴリズムを有する関数のうち，出力されなかったものの個数である．こちらは，すべてのサンプルにおいて 0 であった．表 4 は Algorithm2 の出力結果である．出力の 16 進数はプログラム実行時の関数のメモリ上の位置を表すメモリアドレスである．サンプル (2) 及びサンプル (6)〜(7) については，ただ一つの関数が出力された．これは，ひとつの関数の内部ですべての復号処理が行われたためである．サンプル (3)〜(5) については複数の関数を含む一つのパスが出力された．これは復号処理を複数の関数に段階的に分割し，実装されていたためであったが，本提案手法はそれらすべての関数を検出することができた．

5 結論

本稿では，隠蔽されたデータフローを補完した上で復号関数を特定する手法を提案した．評価実験では，提案手法を用いることで，10 個のサンプルプログラムの内，9 個に対して復号関数を一意に特定することができた．

従来の手法では機械語命令単位でのテイント解析を用いて暗号化/復号関数の位置もしくは平文バッファの位置を特定している．しかしながら，under-tainting 等の難読化処理が施されたマルウェアには，この種類のテイント解析では対応できない．そこで本稿では，それよりも粒度の荒い関数単位でのテイント解析で暗号化/復号関数の位置特を定する手法を提案した．本提案手法は，関数を処理ごとのコミュニティに分類することで，データの伝搬の最低限の箇所にのみ着目するが，これは関数間の入出力データの受け渡しを追跡するのみで可能である．

参考文献

[1] F. Gröbert, C. Willems, and T. Holz, “Automated identification of cryptographic primitives in binary programs,” *In: Proc. Recent Advances in Intrusion Detection (RAID)*, pages 41-60. Springer, 2011.

[2] J.Calvet, J. M. Fernandez and J. Marion, “ Aligot: Cryptographic Function Identification in Obfuscated Binary Programs,” *CCS’ 12, October 16-18, 2012, Raleigh, North Carolina, USA*,

[3] Wang, Z., Jiang, X., Cui, W., Wang, X.,and Grace, M., “ReFormat: Automatic Reverse Engineering of Encrypted Messages,” *In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS*, vol. 5789, pp.200-215. Springer, Heidelberg (2009).

[4] Xin L., Xinyuan W., and Wentao C., “ CipherXRay: Exposing Cryptographic Operations and Transient Secrets from Monitored Binary Execution,” *IEEE transactions on dependable and secure computing*, vol. 11, no. 2, march/april 2014.

[5] G. Portokalidis, A. Slowinska, and H. Bos., “ Argos: an emulator for fingerprinting zero-day attacks,” *In EuroSys 2006*, April 2006.

[6] Cavallaro, L., Saxena, P., and Sekar, R., “ On the limits of information flow techniques for malware analysis and containment,” *In Detection of Intrusions and Malware, and Vulnerability Assessment*, (pp. 143-163). Springer Berlin Heidelberg (2008).

[7] Reichardt, J., and Bornholdt, S., “ Statistical mechanics of community detection,” *Physical Review E*, vol. 74, 016110 pp.1-14 (2006).

[8] OpenSSL: The Open Source toolkit for SSL/TLS, https://www.openssl.org/

[9] Brian Gladman’s Home Page http://www.gladman.me.uk/

[10] Crypto++ Library 5.6.2 http://www.cryptopp.com/

[11] Beecrypt http://beecrypt.sourceforge.net/

[12] F. Bellard. , “ QEMU, ” http://www.qemu.org/