シンボリック実行を利用したIOC変換

川古谷 裕平[1]　岩村 誠[1]　三好 潤[1]　森 達哉[2]

概要：Entity Matching は特定のパターンをデータから見つけ出す技術であり，侵入検知やアンチウィルスなどで応用される．本研究は，記述の自由度が高いプログラムとして表現された Entity Matching 用のシグネチャ (EM シグネチャ) を，多くのソフトウェア・機器が対応している静的な EM シグネチャに自動変換する手法を提案する．鍵となるアイデアはシンボリック実行を利用することにある．この研究ではシグネチャのフォーマット変換をプログラムのパス探索問題に置き換え，プログラムが検知と判断する箇所に到達するためのパス制約をシンボリック実行を用いて算出し，その制約を EM シグネチャへと変換する．ユースケースとして，Python で書かれた Cuckoo Sandbox のシグネチャを OpenIOC 形式に変換する．実験では変換前と変換後の検知結果が同等であることを示す．

キーワード：シンボリック実行，IOC，シグネチャ，マルウェア，フォーマット変換

IOC Conversion with Symbolic Execution

Yuhei Kawakoya[1]　Makoto Iwamura[1]　Jun Miyoshi[1]　Tatsuya Mori[2]

Abstract: We propose a technique to convert indicator-of-compromises (IOCs) for entity matching (EM) written in a program language into static ones using symbolic execution. We translate a problem of format conversion into one of path explore in program analysis field. We first automatically compute path conditions for an execution to reach a certain program point to alert a detection, and then translate the path conditions into an EM signature. This technique allows us to write a signature in a major program language and convert that into static one, which is able to deploy to many industrial appliances. As an use-case, we demonstrate a conversion of signatures used in Cuckoo Sandbox, which are written in Python, into OpenIOC formatted one, which is used in many forensics tools, such as Redline or OpenIOC Finder.

Keywords: Symbolic Execution, IOC, Signature, Malware, Format Conversion

1. はじめに

Entity Matching は予め定めておいた特定のパターンをデータから探し出す技術である．Entity Matching の応用例はネットワークトラフィックのパターン検査，バイトストリームのパターンマッチング，プログラムの挙動解析などであり，セキュリティ解析において広く利用されている [11][14][13][12][18]．この予め定める特定のパターンは Entity Matching シグネチャ（EM シグネチャ）と呼ばれ，過去の攻撃パターン等を元に作成される．Entity Matching の検知率はこの EM シグネチャの正確性や EM シグネチャの数に依存する部分が大きい．

この EM シグネチャを記述するために様々なフォーマットが利用されている．ここでは，それらをプログラムとして実行可能なもの（プログラム形式のシグネチャ）と実行できないもの（静的なシグネチャ）に分別する．表 1 にそれぞれのシグネチャの具体的な例を示す．例えば，Cuckoo Sandbox[11] では，マルウェアを解析した結果得られる挙動ログの中から，マルウェア特有の挙動を見つけ出す，Python で書かれたシグネチャを有している [4]．一方で，静的なシグネチャの例として，OpenIOC がある [7]．OpenIOC は Mandiant/FireEye 社によって提案された In-

[1] NTT セキュアプラットフォーム研究所
[2] 早稲田大学

dicator of Compromise（IOC）の一種で，Redline[6] や OpenIOC Finder[5] 等のフォレンジックスツールで使われている．

表 1 プログラム形式と静的シグネチャの例
Table 1 Examples of Programmable and Static Signatures

プログラム形式	静的
Cuckoo Sandbox[11]，Bro[13]，Suricata[12]，BPF[10]	Snort[14]，各種 IOC[7]，Yara[18]，iptables

これら様々なフォーマットで記述されたシグネチャ間のフォーマット変換をするため，ベンダや有志によるツール等が公開されており [17]，特定のフォーマットが特定のソフトウェア・機器に固有にならないよう努力が行なわれている．しかし，現在公開されているフォーマット変換ツールの多くは，静的なシグネチャを別の静的なシグネチャへの変換するものが主であり，プログラム形式のシグネチャを静的なシグネチャへ変換するツールはほぼ皆無である．このため，プログラム形式のシグネチャは，そのシグネチャが利用できるソフトウェア・機器が固定されやすく，シグネチャとして記述された知識が広く共有されない問題が生じている．

上述の問題の具体例として，前述の Python で記述された Cuckoo Sandbox のシグネチャがある．Cuckoo Sandbox のコミュニティでは，世界中のマルウェア解析者が作成した 400 程のシグネチャ（2018 年 8 月現在）があるにもかかわらず，これらの資産を他のソフトウェア・機器で直接利活用するための手段は存在しない．

そこで，本論文ではプログラム形式のシグネチャを静的なシグネチャへ自動的に変換する手法を提案する．具体的には Python で記述された Cuckoo Sandbox のシグネチャを OpenIOC 形式へと変換する．基本となるアイデアは，プログラム形式のシグネチャが，入力された Entity に対して検知と判断する条件をシンボリック実行で解析し，その検知と判断するプログラム行へのパス制約を算出する．また，当プログラム行までのパスが複数存在する場合を考慮し，マルチパス探索を行うことでこれらを網羅的に抽出する．これにより，プログラムとして記述されていた「検知と判断する条件」を入力 Entity に対する論理式の形に変換し，この論理式を静的シグネチャの特定のフォーマットへ変換する．この論理式は原始的な項の組み合わせのため，論理式の表現方法や理論（評価の方法）の違いを考慮する必要はあるが，基本的には直接的な変換が可能である．

本提案手法を利用することで，プログラム形式のシグネチャを静的なシグネチャへと自動的に変換できる．具体的には，例で挙げているように，Cukoo Sandbox のシグネチャを OpenIOC 形式へ自動的に変換できる．これにより，Cuckoo Sandbox でのみ利用可能であったプログラム形式のシグネチャを他の OpenIOC をサポートしているソフトウェア・機器，例えば OpenIOC Finder や Radline，で利活用することが可能になる．

本提案手法の有効性を示すため，以下の 2 つの実験を行った．1 つ目は，Cuckoo Sandbox のシグネチャを 1 つを選び（banker_cridex.py），これをケーススタディとして OpenIOC に変換されるまでの動作を詳細に記述し，正しく変換が行なわれていることを示した．2 つ目は，評価対象としている Entity の種類が異なる，5 つの Cuckoo Sandbox のシグネチャを選び，これらを提案手法を利用して OpenIOC 形式へ変換し，元のプログラム形式のシグネチャと検知結果を比較した．これらの実験を通して，プログラム形式のシグネチャを OpenIOC 形式に正しく変換できていること，また検知結果が変換前と後で同程度であることを示した．

以下にこの論文の貢献を示す．

- EM シグネチャをプログラム形式のものと静的なものという観点で捉え，これらの間で適切なフォーマット変換方法がない問題を指摘した．
- プログラム形式のシグネチャを，非プログラム形式の静的シグネチャへ自動的に変換する手法を提案した．
- 本提案手法を，Cuckoo Sandbox のシグネチャから OpenIOC 形式へ変換する問題へ適用し，シグネチャの変換が正しく行なわれていることを実験により示した．

2. 背景知識

ここでは，背景知識として Cuckoo Sandbox シグネチャと OpenIOC について説明する．またシンボリック実行についても説明する．

2.1 Cuckoo Sandbox シグネチャ

ソースコード 1 に Cuckoo Sandbox シグネチャの例を示す．Cuckoo Sandbox のシグネチャでは，Python が提供する任意の機能を使って，マルウェア解析の結果得られる挙動ログから特徴的な挙動を抽出する検知条件を記述できる．ソースコード 1 の例では，検知と判断するには，match_file，match_batch_file の両方が真であり，mutex が indicators のどれか 1 つ以上と合致した場合である．

2.2 OpenIOC

OpenIOC は Mandiant/FireEye 社によって提案された IOC の一つの規格である [7]．内部は XML で記述され，マルウェアの痕跡情報を定義することができる．ファイルやレジストリやプロセスといった幅広い Entity に対して検知条件を記述することが可能である．Entity との評価方法として，完全一致とその否定である “`is`”および “`is not`” と，部分一致とその否定である “contain”および “does not

ソースコード 1 banker_cridex.py のシグネチャ

```
def on_complete(self):
 match_file = self.check_file(pattern=".*\\\\KB[0-9]{8}\.exe", regex=True)
 if match_file:
  self.mark_ioc("file", match_file) # 検知したEntity をログに記録する

 match_batch_file = self.check_file(pattern=".*\\\\Temp\\\\S{4,5}\.tmp\.bat", regex=True)
 if match_batch_file:
  self.mark_ioc("file", match_batch_file)

 if not match_file or not match_batch_file:
  return

 for indicator in self.indicators: # indicators = [".*Local.QM.*", ".*Local.XM.*", "634I", ...]
  match_mutex = self.check_mutex(pattern=indicator, regex=True)
  if match_mutex:
   self.mark_ioc("mutex", match_mutex)

  return self.has_marks(3) # 三つ以上のログが存在する場合, 真を返す
```

ソースコード 2 サンプルプログラム

```
i = get_input()
j = get_input()
if i > 10:
    if j > 100:
        return "A"
    else:
        return "B"
else:
    if i > 10:
        return "C"
...
```

contain”を利用することができる．

2.3 シンボリック実行

シンボリック実行は，King[9] によって提案されたプログラム解析手法の一つで，テストケース生成や静的解析に利用されている．ここでは，ソースコード 2 を例に用いてシンボリック実行を説明する．ソースコード 2 は，入力値，i と j に応じて戻り値を"A"，"B"，または"C"，に変化させるプログラムである．このプログラムから全ての挙動（すべての戻り値）を取得するには適切な入力値をこのプログラムに与える必要がある．

シンボリック実行では，プログラムに対する入力値を具体的な値に固定せず，あらゆる値を取り得るシンボルとして扱い，プログラムを逐次実行していく．例えば，ソースコード 2 では i と j をシンボル化する．そして，この逐次実行中に条件分岐に遭遇した場合，それぞれのパスを取るための制約を入力値に対して与える．例えば，"A"を得るためには，ソースコード 2 の 1，2，3，4，5 行目（プログラム実行する行のシーケンスをパスという）を通る必要がある．このパスのうち 3 行目の if 文で真になるには，入力値 i に対して，i > 10 といった制約を与える必要がある．また，4 行目の if 文で真になるには，j > 100 の制約を与える必要がある．一方で，"B"を得るためには，i > 10 と j <= 100 の制約を与える必要がある．

上記の例のように，条件分岐毎に入力値に対して制約を与えることで，特定のパスを通るために入力値が満たすべき制約条件（パス制約）を得ることができる．前述の例の場合，"A"を返すパス (1，2，3，4，5 行目）のパス制約は i > 10 かつ j > 100 であり，"B"を返すパス（1，2，3，4，6，7 行目）のパス制約は i > 10 かつ j <= 100 である．

このパス制約が充足可能な時，パス制約を満たす具体的な値（モデル）が存在し，テストケースを生成することができる．前述の例では，"A"を返すパスのパス制約は充足可能である．このモデルは，例えば i = 11，j = 101 がある．また，"B"を返すパスも充足可能であり，そのモデルは i = 11，j = 100 がある．一方で，パス制約が充足不可能な場合，そのパスを実行するモデルは存在しない．前述の例では，"C"を返すパスのパス制約は i <= 10 かつ i > 10 であり，これは充足不可能である．つまり，このプログラムは"C"を返すモデルは存在せず，このパスは実行されない．

3. 提案手法

ここでは，シンボリック実行を利用して，プログラム形式のシグネチャを静的なシグネチャへ変換する手法を提案する．その後，提案手法の実装について述べる．

3.1 基本アイデア

前述のように，シンボリック実行を利用することで，プログラムの特定の箇所へ到達するパスのパス制約を得ることができる．この性質を利用し，プログラム形式のシグネチャから，そのシグネチャが検知と判断する条件を抽出

する．この条件は，入力値に対する論理式，またはその組み合わせで表現される．これは，プログラム形式のシグネチャに比べ，より単純な項の組み合わせ（入力値に対する評価式の組み合わせ）で検知条件が表現されるため，他のフォーマットへの変換を容易に行うことができる．

3.2 動作概要

図 1 に動作概要を示す．まず Cuckoo Sandbox がマルウェアを解析した結果，生成する挙動ログをシンボル化する．具体的には，挙動ログ中の各カテゴリの挙動，例えばファイルアクセスの挙動やレジストリアクセスの挙動を，それぞれ異なるシンボル変数でシンボル化し，それらをシグネチャに入力値として渡す．この入力値を受け取ったシグネチャをシンボリック実行エンジン上で動作させ，検知と判断するすべてのパスのパス制約（検知パス制約）を抽出する．次に，この検知パス制約を OpenIOC 形式へ変換する．この際，パス制約のある値が正規表現を含んでいた場合，その値を正規表現を含まない形に変換する．また，シンボル化した挙動ログの各カテゴリと OpenIOC の各フィールドとの対応は変換テーブルを参照して行う．

3.3 実装

ここでは，提案手法の実装について述べる．まず Python 用シンボリック実行エンジン，pySym[2] とその拡張点について述べ，次に挙動ログのシンボル化について述べる．更に，OpenIOC 形式への変換を行う時の問題として，正規表現の扱いについて述べる．最後に，挙動ログの各カテゴリと OpenIOC の各フィールドの対応関係について述べる．

3.3.1 pySym の拡張

pySym は単純な Python プログラムを解析する目的で開発されたシンボリック実行エンジンである．angr[15] と同じ Online Executor 型のシンボリック実行エンジンであり，似たようなインタフェースで利用出来る．つまり，解析中に条件分岐に達すると，その時のステートを複製し，その条件分岐が真の場合と偽の場合の制約をそれぞれのパスに追加しつつ，複数パスの解析（マルチパス探索）を行う．pySym は，独自に Python インタープリタを持ち，それを通して Python プログラムを逐次実行し，ステートを更新していく．

現在の pySym の Python インタープリタは限られた変数タイプや文法にしか対応していない．そのため，以下のような拡張を行い，Cuckoo Sandbox のシグネチャを解釈するのに最低限なレベルにまで Python インタープリタの機能を拡張した．

(1) Dict，Tuple 等のデータタイプの追加
(2) 文字列の比較
(3) 外部関数ハンドラ（isinstance, set, re, 等）の追加

これらの実装のため，オリジナルの pySym と比較し，Python で 2,037 行のコードの追加，113 のテストケースの追加を行った．

3.3.2 挙動ログのシンボル化と検知パス制約

ソースコード 3 にシンボル化した Cuckoo Sandbox の挙動ログのデータ構造体を示す．挙動ログは Cuckoo Sandbox でマルウェアを解析した際，生成されるログであり，そのマルウェアが解析中に行った各種挙動がカテゴリ別に記載されている．Cuckoo Sandbox のシグネチャは，この挙動ログを入力として受け取り，自身が持っている検知条件に合致するログが対象とするカテゴリに含まれているかを調べている．

我々の提案手法では，入力値である挙動ログにかかる制約を抽出するため，各挙動カテゴリの要素など，シグネチャによる検知の対象となり得る箇所をシンボル化する．pySym では，pyState.String() でそれぞれ String 型のシンボル変数を生成できる．pyState.String() の第一引数には，文字列の長さを指定し，第二引数には，そのシンボル変数を識別するための名前を指定できる．

Cuckoo Sandbox のシグネチャには，ファイル，レジストリ，Mutex の各カテゴリで，ソースコード 3 で書かれていない処理（例えば，ファイルカテゴリの file_written など）も存在する．それらもシグネチャの検査対象になっている．しかし，ほとんどのシグネチャがそれらの処理の違いを考慮して書かれていなかったため，シンボリック実行の速度などを考慮し，シンボル化する対象も各カテゴリ 1 つの処理に絞った．

シンボリック実行によるパス探索が完了した後，充足可能なモデルを持つパスの中から真（True）を返すパスを取り出す．その際，シンボル化した各カテゴリのログに対してパス制約がかかっていた場合，その制約を取り出し，AND で接続する．それをそのパスにおけるパス制約とする．また，真を返すパスが複数存在していた場合，それらを全て OR で接続したものを，このシグネチャが真を返すための検知パス制約，つまり検知と判断するための制約とし，変換の対象とする．

3.3.3 正規表現の変換

ここでは変換先のフォーマットが正規表現をサポートしていない場合を考慮し，正規表現を含まない形への変換を説明する．実際，OpenIOC1.0 では正規表現はサポートされていない [7]．そのため，シンボリック実行の結果，検知と判断するためのパス制約に正規表現が含まれていた場合，正規表現の文字列を，以下の二つのルールに則り，正規表現を含まない具体的な文字列の組み合わせに変換する．

(1) 表現できるパターン数が閾値以下の場合は，全て具現化し OR で接続する
(2) 表現できるパターン数が閾値以上の場合は，その前後で文字列を分割し，AND で接続する

(1) の例として，「xyz[0-9]」のような正規表現を考える．

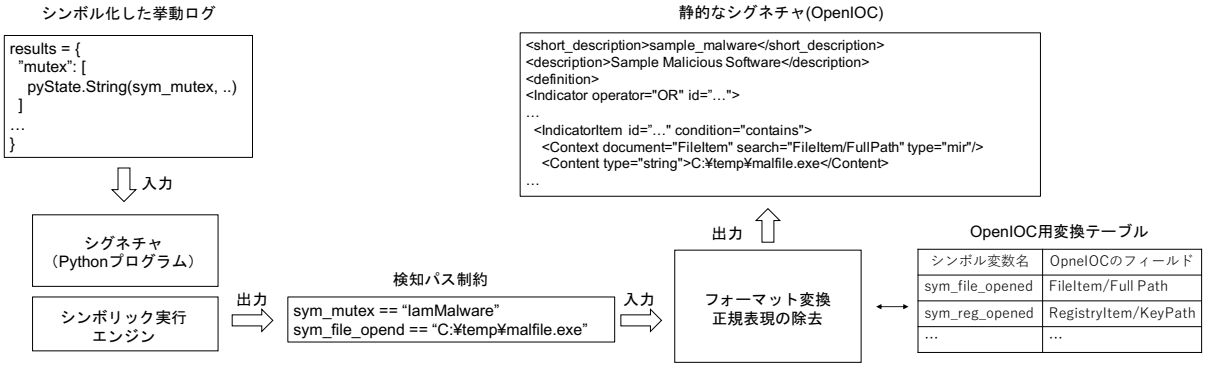

図 1 動作概要
Fig. 1 Overview

ソースコード 3 挙動ログのシンボル化

```
1  results = { ...
2    "behavior": {
3    ...
4    "summary": {
5      "mutex": [
6        pyState.String(256, "sym_mutex0"),
7        ... ],
8      "file_opened": [
9        pyState.String(256, "sym_file_opened0"),
10       ... ],
11     "regkey_opened": [
12       pyState.String(256, "sym_regkey_opened0"),
13       ... ],
14 }}}
```

この正規表現を具現化すると xyz0 から xyz9 までの 10 通りのパターンがある．パターンの閾値が 10 以上の値で設定していた場合，これらの 10 通りを全て OR で接続した「xyz0 OR xyz1 OR ... OR xyz9」といった論理式に変換する．

(2) の例としては，「ab.+de」のような正規表現がある．この場合，「.+」は任意の文字の 1 回以上の繰り返しのため，表現できる文字列のパターンが無限であり，特定の閾値以上となってしまう．そこで，この場合は「.+」の部分で文字列を分割し，「ab」と「de」の AND を接続した「ab AND de」といった変換を行う．このルールは，検知対象の空間を変化させてしまうため，False Positive を生む可能性がある．この問題に関しては，4.3 節で実験するとともに，6 章で考察する．

3.3.4 挙動ログと OpenIOC のフィールドの対応

シンボリック実行により，対象とするシグネチャの検知パス制約を算出した後，表 2 の対応に基づき OpenIOC 形式へ変換する．また Entity との評価方法に関しては，パス制約のうち，正規表現を含んでいる値を前述の方法で変換した場合は文字列の部分一致である “contain”または “does not contain”を利用する．また，それ以外の場合は，文字列の完全一致である “is”または “is not”を利用する．

表 2 シンボル変数名と OpenIOC フィールドの対応
Table 2 Table for Symbolic Variables and OpenIOC Conversion

シンボル名	OpenIOC フィールド
sym_file_opened	FileItem/Full Path
sym_reg_opened	RegistryItem/KeyPath
sym_mutex	ProcessItem/Process/Handles/Name

4. 実験

ここでは，提案手法の有効性を示すために行った 2 つの実験について述べる．1 つ目は，ケーススタディとして，Cuckoo Sandbox の一つのシグネチャ（banker_cridex.py）に対して提案手法を適用した際のプログラム形式から静的シグネチャへの変換動作を詳細に説明した．2 つ目は，Cuckoo Sandbox の中から検知対象のカテゴリ（ファイル，レジストリ，Mutex や正規表現の有無）を網羅できるように選んだ 5 つのシグネチャを OpenIOC に変換し，Cuckoo Sandbox の挙動ログに対して，プログラム形式のシグネチャと同程度に検知できるか実験した．

4.1 対象の範囲

今回の実験では，Cuckoo Sandbox のファイル，レジストリ，Mutex を対象とする制約に範囲を絞った．Cuckoo Sandbox のシグネチャには，これ以外にもネットワークやコマンドラインの挙動を対象とした制約も存在するが，今回はそれらは対象外とした．

また，Cuckoo Sandbox のシグネチャの検査プロセスの最後に実行される on_complete をシンボリック実行の解析対象とした．それ以外にも検査プロセスの途中に呼ばれる on_call も存在するが，これらは主にマルウェアが作成する痕跡ではなく，マルウェアの挙動そのものを検査対象とする場合に使われる．そのため，痕跡を検出する目的である OpenIOC の用途には適さないため対象外とした．

これらの対象範囲の絞り込みは実験のコストを意識して行ったもので，提案手法がこれらの対象のみに制限される

わけではない．上記で述べた対象以外にも拡張する際も上述の提案手法と実装の範囲内で適用可能だと考える．

4.2 ケーススタディ：banker_cridex.py

ここではケーススタディとして，Cuckoo Sandbox のシグネチャ（banker_cridex.py）を OpenIOC へ変換するまでの動作を追いかける．Mutex の Indicators を 1 つに絞り，banker_cridex.py をシンボリック実行エンジンで実行すると，11 のパスで Completed（充足可能），83 のパスで Deadended（充足不可能）となった．充足可能なモデルを表 3 に示す．充足可能な 11 のケースのうち，真の戻り値を返すケース，つまり検知したものは 2 つであった．順序を考慮しない場合は，これらの 2 つは同じものである．つまり，“.*\\Temp\\\S{4,5}\.tmp\.bat”と “.*\\KB[0-9]{8}\.exe”のファイルが存在し，“.*Local.QM.*”の Mutex が存在する場合，このシグネチャは検知と判断する．

次にこの検知条件の値が正規表現を含まない形に変換する．上記の文字列に対して，3.3.3 項で説明した正規表現の変換処理を行ったところ，以下の文字列の AND の組み合わせを得られた，「“\\Temp\\”と “\.tmp\.bat”」，「“\\KB” と “\.exe”」，「“Local”と “QM”」．

更に，3.3.4 項で説明したように，これらの文字列の評価対象を OpenIOC のフィールドに対応させる．その結果，ソースコード 4 の OpenIOC が生成された．「\KB と.exe」と「Local と QM」と「\Temp\ と.tmp.bat」の AND となっており，元になった banker_cridex.py のシグネチャと同等の検知基準となっていることが確認できる．

4.3 検知性能

ここでは，5 つの Cuckoo Sandbox のシグネチャ（banker_cridex.py, rat_trogbot.py, rat_zegost.py, rat_fynloski.py, trojan_dapato.py）を OpenIOC 形式へ変換し，変換前と変換後のシグネチャの検知結果を比較する．この実験の目的は，フォーマット変換によって生じるシグネチャの対象範囲の変化が検知結果に大きな影響を与えていないことを示すことである．

4.3.1 実験方法

5 つのシグネチャが対象としている挙動を持ったマルウェアを VirusTotal からダウンロードし，それらを Cuckoo Sandbox で解析し，挙動ログを作成する．その後，それぞれの挙動ログと OpenIOC 形式の 5 つのシグネチャを突合させ，Cuckoo Sandbox で解析した場合と同様の数だけシグネチャにヒットするか確認する．ここで利用する OpenIOC と挙動ログのマッチングツールはこの実験のために自作した．なお，これら 5 つのシグネチャは，ファイル，レジストリ，Mutex のそれぞれの挙動が網羅されるように選別した．

4.3.2 実験結果

表 4 にこの実験の結果を示す．変換後の OpenIOC 形式のシグネチャを使った場合でも，5 つのシグネチャすべてにおいて，Cuckoo Sandbox が出力したのと同じ数だけイベントを検知することが確認できた．また，この実験の範囲内では，変換後の OpenIOC による False Positive は発生しなかった．

5. 関連研究

Singh ら [16] はプログラム合成技術を利用し，Entity Matching 用ルールのフォーマットの変換を行っている．プログラム合成を利用した方法では，一定量の Positive Example と Negative Example を集める必要がある．そのため，Example の集めにくいマイナーなシグネチャの場合，正確なフォーマット変換が困難になってしまう．一方で，我々の提案手法では，Examples は必要としないため，マイナーなシグネチャの場合であっても変換を行うことが可能である．

Python 用のシンボリック実行エンジンとしては，pySym[2] の他にも PyExZ3[1] や Bruni ら [3] の研究のような Python が持っている機能を拡張することでシンボリック実行を実現する方法も提案されている．彼らのアプローチの場合，新しく Python のインタープリタを実装する必要がなく，大規模な Python プログラムに対しても適用可能だと考える．今後，より多様な Python プログラムを解析するため，これらのアーキテクチャの採用等も考慮したい．

OpenIOC を利用した研究，ツールは多数公開されている．ディスク上の痕跡の調査には Redline[6] や OpenIOC Finder[5] などがあり，メモリフォレンジックス用途には春山が提案する OpenIOC_scan[8] などがある．本提案手法を利用し Cuckoo Sandbox のシグネチャを OpenIOC 形式にすることで世界中のマルウェア解析者たちが開発した Cuckoo Sandbox のシグネチャをこれらのツールで利用することができる．

6. 考察

ここでは検知対象空間の変化と他のフォーマットへの応用について考察する．

検知対象空間の変化 提案手法では，検知パス制約の各項の順番は考慮されていない．つまり，プログラム形式のシグネチャ内で記述されている検知条件の順序に何らかの意味があった場合，その意味を静的なシグネチャ側に再現できない．順序を考慮した検知条件の変換を行うには，シンボリック実行エンジンの拡張等を検討する必要がある．

3.3.3 項の正規表現の変換では，検知ルールの対象空間が変化する可能性がある．例えば，「ab.*de」のような正規表現を含んだ文字列の場合，3.3.3 項のルールに従い「ab」と

ソースコード 4 banker_cridex.py の OpenIOC

```
<?xml version='1.0' encoding='UTF-8'?> ...
 <short_description>banker_cridex</short_description>...
    <Indicator operator="AND" id="...">
      <Indicator operator="AND" id="...">
        <IndicatorItem id="..." condition="contains">
          <Context document="FileItem" search="FileItem/FullPath" type="mir" />
          <Content type="string">\KB</Content>
        </IndicatorItem>
        <IndicatorItem id="..." condition="contains">
          <Context document="FileItem" search="FileItem/FullPath" type="mir" />
          <Content type="string">.exe</Content>
        </IndicatorItem>
      </Indicator>
      <Indicator operator="AND" id="...">
        <IndicatorItem id="..." condition="contains">
          <Context document="ProcessItem" search="ProcessItem/HandleList/Handle/Name" type="mir" />
          <Content type="string">Local</Content>
        </IndicatorItem>
        <IndicatorItem id="..." condition="contains">
          <Context document="ProcessItem" search="ProcessItem/HandleList/Handle/Name" type="mir" />
          <Content type="string">QM</Content>
        </IndicatorItem>
      </Indicator>
      <Indicator operator="AND" id="...">
        <IndicatorItem id="..." condition="contains">
          <Context document="FileItem" search="FileItem/FullPath" type="mir" />
          <Content type="string">\Temp\</Content>
        </IndicatorItem>
        <IndicatorItem id="..." condition="contains">
          <Context document="FileItem" search="FileItem/FullPath" type="mir" />
          <Content type="string">.tmp.bat</Content>
        </IndicatorItem>
      </Indicator>
    </Indicator>...
</ioc>
```

表 3 banker_cridex.py の解析結果（充足可能なもの）
Table 3 Analysis Result of banker_cridex.py (only satisfiable)

sym_file_opened0	sym_file_opened1	sym_mutex0	戻り値
".*\\Temp\\\S{4,5}\.tmp\.bat"	".*\\KB[0-9]{8}\.exe"	".*Local.QM.*"	真
".*\\KB[0-9]{8}\.exe"	".*\\Temp\\\S{4,5}\.tmp\.bat"	".*Local.QM.*"	真
".*\\Temp\\\S{4,5}\.tmp\.bat"	".*\\KB[0-9]{8}\.exe"	—	偽
".*\\KB[0-9]{8}\.exe"	".*\\Temp\\\S{4,5}\.tmp\.bat"	—	偽
".*\\KB[0-9]{8}\.exe"	—	—	偽
—	".*\\KB[0-9]{8}\.exe"	—	偽
".*\\KB[0-9]{8}\.exe"	".*\\KB[0-9]{8}\.exe"	—	偽
".*\\Temp\\\S{4,5}\.tmp\.bat"	—	—	偽
—	".*\\Temp\\\S{4,5}\.tmp\.bat"	—	偽
".*\\Temp\\\S{4,5}\.tmp\.bat"	".*\\Temp\\\S{4,5}\.tmp\.bat"	—	偽
—	—	—	偽

「de」の二つの文字列の AND に変換される．この際，変換後の二つの文字列の順序は考慮されていないため，「deab」といった文字列でも検知してしまい False Positive を生成する．4.3 節の実験では，この順序性の欠如は大きな問題にならなかったが，他のユースケースでは考慮が必要な可能性がある．また変換先のシグネチャのフォーマットに，変換前の項目と対象範囲が合致する項目がない場合，検知できる対象の範囲がずれてしまい，False Positive や Negative を発生させる可能性がある．

他のフォーマットへの変換 今回は Python で書かれた

表 4 検知性能の実験結果
Table 4 Result of Detection Experiment

シグネチャ名	マルウェア名	Cuckoo Sandbox ヒット数	OpenIOC ヒット数
banker_cridex.py	W32.Cridex	3 Events (file×2, mutex)	3 Events (file×2, mutex)
rat_trogbot.py	W32.Spybot.Worm	1 Event (mutex)	1 Event (mutex)
rat_zegost.py	W32.Ramnit!inf	1 Event (mutex)	1 Event (mutex)
rat_fynloski.py	Trojan.Zbot!gen43	4 Events (mutex, regkey×2, file)	4 Events (mutex, regkey×2, file)
trojan_dapto.py	Backdoor.Noppuca!gen	2 Events (mutex, file)	2 Events (mutex, file)

Cuckoo Sandbox のシグネチャを OpenIOC 形式へ変換したが，提案手法は OpenIOC 形式への変換に限定しているわけではない．3.3.3 項と 3.3.4 項で述べている点を考慮すれば，OpenIOC 以外の形式への変換も可能である．また，変換元も Python プログラムに限定していない．そのシグネチャが記述されたプログラムに対応するシンボリック実行エンジンを用意できれば，他の言語にも応用することができる．例えば，Bro のシグネチャを Snort で利用するといった応用もある．記述の自由度の高いスクリプト言語等でシグネチャを記述できた場合，シグネチャの開発コストやメンテナンスコスト等も下げることができる．

7. まとめ

本論文では，プログラム形式のシグネチャを静的なシグネチャに変換する手法を提案した．シンボリック実行により，プログラム形式のシグネチャを解析し，検知と判断するための条件を自動的に抽出し，静的なシグネチャに変換する．提案手法により，自由度の高いプログラム言語を使ってシグネチャを記述し，それを多くのソフトウェア・機器で利用可能な静的なシグネチャの形に自動変換して適用するといった運用を行うことが可能になる．具体的には Cuckoo Sandbox のシグネチャを OpenIOC 形式に変換可能なことを示した．また，マルウェア解析を行って得た挙動ログを元に評価を行い，変換前と変換後で同程度の検知が行えることを示した．これにより，特定のソフトウェア・機器固有になりがちなプログラム形式のシグネチャを他のソフトウェア・機器でも利用可能にし，知識の共有を加速させることができると考える．

参考文献

[1] Ball, T. and Daniel, J.: Deconstructing Dynamic Symbolic Execution., *Dependable Software Systems Engineering* (Irlbeck, M., Peled, D. A. and Pretschner, A., eds.), NATO Science for Peace and Security Series, D: Information and Communication Security, Vol. 40, IOS Press, pp. 26–41 (2015).

[2] bannsec: pySym, https://github.com/bannsec/pySym.

[3] Bruni, A., Disney, T. and Flangan, C.: A Peer Architecture for Lightweight Symbolic Execution, https://hoheinzollern.files.wordpress.com/2008/04/seer1.pdf.

[4] Cuckoo Sandbox: Community, https://github.com/cuckoosandbox/community.

[5] FireEye: IOC Finder — Free Security Software, https://www.fireeye.com/services/freeware/ioc-finder.html.

[6] FireEye: Redline — Free Security Software, https://www.fireeye.com/services/freeware/redline.html.

[7] Harrington, C.: Sharing Indicators of Compromise: An Overview of Standards and Formats, *RSA Conference 2013* (20013).

[8] Haruyama, T.: Fast and Generic Malware Triage Using openioc_scan Volatility Plugin, *The Digital Forensic Research Conference DFRWS 2015 EU* (2015).

[9] King, J. C.: Symbolic Execution and Program Testing, *Commun. ACM*, Vol. 19, No. 7, pp. 385–394 (1976).

[10] McCanne, S. and Jacobson, V.: The BSD Packet Filter: A New Architecture for User-level Packet Capture, *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, pp. 2–2 (1993).

[11] Oktavianto, D. and Muhardianto, I.: *Cuckoo Malware Analysis*, Packt Publishing (2013).

[12] Open Information Security Foundation: Suricata, https://suricata-ids.org/.

[13] Paxson, V.: Bro: A System for Detecting Network Intruders in Real-time, *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, pp. 3–3 (1998).

[14] Roesch, M.: Snort - Lightweight Intrusion Detection for Networks, *Proceedings of the 13th USENIX Conference on System Administration*, LISA '99, pp. 229–238 (1999).

[15] Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C. and Vigna, G.: SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis, *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 138–157 (2016).

[16] Singh, R., Meduri, V., Elmagarmid, A., Madden, S., Papotti, P., Quiané-Ruiz, J.-A., Solar-Lezama, A. and Tang, N.: Generating Concise Entity Matching Rules, *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pp. 1635–1638 (2017).

[17] STIXProject: openioc-to-stix, https://github.com/STIXProject/openioc-to-stix.

[18] Yara: The pattern matching swiss knife for malware researchers, https://virustotal.github.io/yara/.