スクリプト実行環境に対する マルチパス実行機能の自動付与手法

碓井 利宣[1,2,a)]　古川 和祈[3]　大月 勇人[1,†1]　幾世 知範[1]　川古谷 裕平[1]　岩村 誠[1]　三好 潤[1]
松浦 幹太[2]

概要：悪性スクリプトの挙動の解析には，難読化の影響を受けにくい動的解析が用いられてきた．しかし，悪性スクリプトには，特定の条件を満たさなければ実行されない実行経路が存在し，単一の経路のみを解析する方式では挙動を把握しきれない．こうした問題への対策として，条件分岐の発生時に両方の経路を実行するマルチパス実行が存在する．しかし，マルチパス実行はスクリプトエンジンごとに設計，実装が必要であり，悪性スクリプトの用いる多様なスクリプトエンジンに対して実現するのは現実的でない．
この問題を解決するため，本研究では，スクリプトエンジンに共通する構造に着目したマルチパス実行機能の自動付与手法を提案する．複数のテスト用のスクリプトを用いてスクリプトエンジンの持つ仮想機械の挙動の差分を抽出することでアーキテクチャの情報を取得し，それに基づいて仮想機械に解析用コードを付加することで，マルチパス実行を実現する．Lua と VBScript のスクリプトエンジンに対して本手法を適用し，必要な情報が得られることを確認した．さらに，それを用いてマルチパス実行基盤を構成し，解析妨害を具備する検体に対しても，従来の解析環境では実行されない経路を実行できることを確認した．

キーワード：悪性スクリプト，マルチパス実行，動的解析，機能拡張

Automatically Appending Multi-Path Execution Functionality to Vanilla Script Engines

Toshinori Usui[1,2,a)]　Kazuki Furukawa[3]　Yuto Otsuki[1,†1]　Tomonori Ikuse[1]　Yuhei Kawakoya[1]
Makoto Iwamura[1]　Jun Miyoshi[1]　Kanta Matsuura[2]

Abstract: Malicious scripts are generally analyzed by dynamic analysis which can evade the effect of obfuscation. However, since generic dynamic analysis analyzes only one executed path, its analysis is not effective for malicious scripts that have execution paths only triggered by specific conditions. To handle this problem, multi-path execution, which executes both paths of conditional branch is promising. However,building multi-path execution tools for various script engines widely used by malicious scripts is unrealistic because it requires design and implementation for respective script engines.
In this paper, we propose a method for automatically appending multi-path execution functionality to a vanilla script engine by focusing on the architecture commonly seen in generic script engines. Our method executes multiple test scripts to obtain execution traces and diffs them for extracting architecture information of the virtual machine of the target script engine. We implemented a prototype of our method and applied it to the real world script engines. The prototype showed that our method can extract the architecture information required for multi-path execution. Using the information, we built a multi-path execution tool and confirmed that it can effectively analyze real-world evasive malicious scripts.

Keywords: malicious script, multi-path execution, dynamic analysis, function enhancement

1. はじめに

マルスパムやファイルレスマルウェアなど，攻撃形態の多様化が進む中，悪意のある挙動を持ったスクリプト（悪性スクリプト）を用いた攻撃が拡大している．こうした悪性スクリプトに対策を講じるには，悪性スクリプトの持つ挙動を解析する技術が不可欠である．

悪性スクリプトを解析する際の障壁として，コードの難読化がある．悪性スクリプトの多くは難読化が施されており，コードの静的解析によって詳細な挙動を明らかにするのは困難である．一方，動的解析では，一般に実際に実行された単一の経路に解析範囲が限られるため，特定の条件を満たさなければ実行されない経路を持った悪性スクリプトは解析し切れない．この一例として，実行中の環境の情報を読み取り，解析環境に見られる特徴を持っている場合は動作しない解析妨害 [1] があるため，対策は急務である．

これに対し，条件分岐の発生時に実行状態をフォークし，両方の経路を実行するマルチパス実行による動的解析 [2][3][4] も提案されているが， スクリプトのマルチパス実行を実現するには 2 つの大きな問題が存在する．1 つ目は，スクリプトエンジンのアーキテクチャに関する知見を必要とする点である．2 つ目は，スクリプトエンジンごとに個別にマルチパス実行器の設計，実装を必要とする点である．攻撃者は多様なスクリプト言語を用いて悪性スクリプトを作成することができるため，それら全てのスクリプトエンジンのアーキテクチャの情報を得てマルチパス実行を実現するのは，現実的でない．

この問題を解決するため，本研究では，スクリプトエンジンを動的解析することでスクリプトエンジンがバイトコードを実行させる仮想機械 (VM) のアーキテクチャを解析し，自動的にマルチパス実行を実現する手法を提案する．テストスクリプトと呼ぶ解析用のスクリプトを用いて，VM の挙動の差分を抽出する．これにより，仮想プログラムカウンタや VM 命令のデコーダ・ディスパッチャ，条件分岐フラグなどのアーキテクチャ情報を取得する．また，VM が持つ命令セットアーキテクチャ (Instruction Set Architecture: ISA) も同様に，テストスクリプトを用いて解析し，VM の持つ分岐命令を明らかにする．これらのアーキテクチャ情報に基づいて，マルチパス実行を実現する．条件分岐の VM 命令のデコードを検出して，実行状態をフォークし，条件分岐フラグを書き換えることで，分岐を制御してマルチパス実行を実現する．これを，スクリプトエンジンに対する解析用コードを挿入で実現する．

提案手法に基づくプロトタイプを実装し，Lua と VBScript のスクリプトエンジンに対して実験を実施した．それらが持つ VM を解析した結果，マルチパス実行に必要なアーキテクチャ情報を抽出できることを確認した．また，各々の検出は，スクリプトエンジン一つあたり，数百秒程度で実現可能なことも確認できた．さらに，過去の我々の研究 [5] で作成した API トレーサに対して，提案手法に基づいてマルチパス実行機能を付与し，実際の攻撃に用いられた悪性スクリプトを解析した．その結果，単一経路のみを解析する状態のトレーサと比較して，より多くの悪性な挙動を抽出可能なことが確認できた．本研究により，今まで多くの解析ツールで解析が困難であった，特定の条件を満たさなければ実行されない実行経路を持つ悪性スクリプトに対しても，有効な解析を実現できることが期待される．

本研究の貢献をまとめると，以下の通りである．

- スクリプトエンジンの VM を解析し，得られたアーキテクチャ情報に基づいてマルチパス実行を実現する手法を初めて提案した．
- 実験を通して，提案手法によってアーキテクチャ情報を取得できることを確認した．
- 提案手法によってマルチパス実行機能を付与したスクリプト解析ツールを用いて，実際の悪性スクリプトを解析し，有用な情報を取得できることを示した．

ソースコード 1　解析対象の悪性スクリプトの一例

```
1 Set wmi = GetObject("winmgmts:\\.\root\cimv2")
2 Set cpu = wmi.ExecQuery("Select␣*␣from␣
      Win32_Processor")
3
4 If cpu.Item("Win32_Processor.DeviceID='CPU0'").
      NumberOfCores <= 2 Then
5     WScript.Quit
6 End If
7 do_malicious()
```

2. 悪性スクリプトとマルチパス実行

2.1 解析対象の悪性スクリプト

本研究の目的となっている悪性スクリプトの一例をソースコード 1 に示す．これは実際の悪性スクリプトの難読化を解除して得られたスクリプトを一部抜粋して整形したものである．この悪性スクリプトは，プロセッサの情報を取得 (2 行目) し，コア数が 2 個以下であれば，実行を終了する (4,5 行目)．そのため，解析環境のコア数が少ない場合には，悪性な挙動 (7 行目) を観測できない．本研究では，こうした解析環境では実行されない経路も実行し，挙動の観測による解析を可能にすることを目指す．

[1] NTT セキュアプラットフォーム研究所
NTT Secure Platform Laboratories
[2] 東京大学生産技術研究所
Institute of Industrial Science, The University of Tokyo
[3] 電気通信大学
The University of Electro-Communication
[†1] 現在，NTT セキュリティ・ジャパン株式会社
Presently with NTT Security (Japan) KK
[a)] toshinori.usui.rt@hco.ntt.co.jp

2.2 スクリプトエンジンのアーキテクチャ

スクリプトエンジンは一般に，入力されたスクリプトを解析する解析部と，解析結果を基に解釈実行する実行部を持つ．解析部では，字句解析および構文解析によって抽象構文木 (AST) が生成される．

実行部には，おもに 2 つのアーキテクチャが存在する．1 つ目は，AST を直接解釈実行するアーキテクチャであり，2 つ目は，AST をバイトコードに変換して VM で解釈実行するアーキテクチャである．また，この派生として，実行時のプロファイルに基づき，バイトコードをさらに機械語に動的に変換する仕組み（Just-In-Time (JIT) コンパイラ）を持つスクリプトエンジンも存在する．AST の解釈実行は一般に遅いため，多くのスクリプトエンジンはバイトコードに変換して VM で解釈実行している．そのため，本研究では，VM を用いるアーキテクチャに焦点を当てる．

バイトコードを解釈実行する VM には，大きく分けて 2 つの種類が存在する．1 つ目はデコード・ディスパッチ型であり，2 つ目はスレッデッドコード型である．本研究では，一般性の高い前者に焦点を当てている．

図 1 に VM の構成図を，ソースコード 2 に VM の擬似コードを示す．VM によるバイトコードの解釈実行は，**インタプリタループ**内で実施される．また，AST から変換されたバイトコードを保持するメモリ領域を，**コードキャッシュ**と呼ぶ．インタプリタループは，以下の処理を繰り返す．なお，括弧内はソースコード 2 の対応する行番号である．

- コードキャッシュ中で**仮想プログラムカウンタ (VPC)** が指す位置にある命令がフェッチされる．(2 行目)
- VM 命令は**デコーダ**によって解釈され．(5 行目)，命令の内容が実装された部分にディスパッチされる．(6 行目〜)
- 命令のオペランドは**仮想スタック**または**仮想レジスタ**を用いて受け渡され，実行される．

なお，オペランドを仮想スタックを用いて受け渡す VM をスタックマシン，仮想レジスタを用いるものをレジスタマシンと呼ぶ．

2.3 スクリプトのマルチパス実行のアプローチ

マルチパス実行を実現するためには，少なくとも，(1)VPC，(2) 条件分岐の VM 命令のオペコード，(3) 条件分岐フラグの 3 つの VM のアーキテクチャ情報を知る必要がある．これらが判明していれば，VPC の指す先を監視し，条件分岐の VM 命令が実行される際に，条件分岐のフラグを書き換えることで，分岐を制御できる．これにより，ある条件分岐に差し掛かった時に，実行状態をフォークして片方の条件分岐のフラグを変更することで，両方のパスを実行させるマルチパス実行が実現できる．

そのため本研究では，まず先に挙げた 3 つのアーキテクチャ情報を解析によって取得し，それに基づいて，前述のようなマルチパス実行を実現するための解析用コードを付与する，というアプローチをとる．

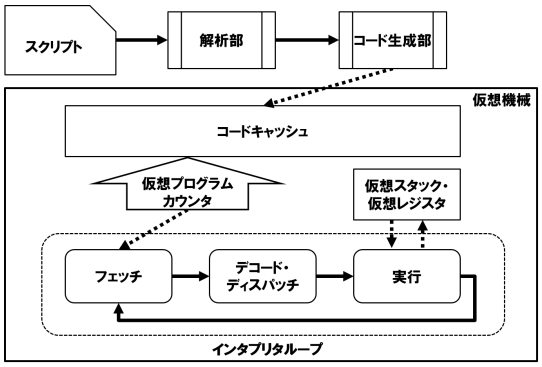

図 1　仮想機械の構成図
Fig. 1　Architecture of virtual machine

ソースコード **2**　仮想機械の擬似コード

```
1  vpc = 0
2  while (True) {
3      opcode = code_cache[vpc]
4      vpc += 1
5      switch (opcode) {
6          case VMOP_1:
7              ...
8      }
9  }
```

2.4 本研究での仮定

本研究では，以下のようなアーキテクチャのスクリプトエンジンを対象として仮定する．こうしたアーキテクチャは，スクリプトエンジンとしてごく一般的なものである．

- VM による解釈実行をする
- デコード・ディスパッチ型の VM である
- VM はレジスタマシンまたはスタックマシンである
- JIT コンパイラは持たないか，無効にできる
- 難読化はされていない

また，スクリプトエンジンのアーキテクチャに関する事前知識は仮定しない．一方，テストスクリプトの作成のため，スクリプト言語の言語仕様に関する事前知識は仮定する．

3. 提案手法

3.1 概要

提案手法では，2.3 節の 3 つのアーキテクチャ情報を動的解析によって取得する．それに基づいて，解析用コードを付与することで，マルチパス実行機能を付与する．解析手法には，我々の過去の研究 [5] で提案した，差分実行解析と呼ぶ手法を用いる．差分実行解析とは，複数の異なる条件で実行トレースを取得し，その差分を分析することで

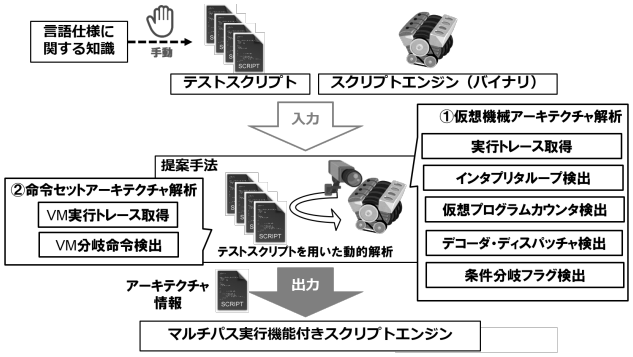

図 2 提案手法の概要図
Fig. 2 Overview of proposing method

動的解析する手法である．

図 2 に提案手法の概要を示す．まず，提案手法の準備として，テストスクリプトと呼ぶスクリプトの準備が必要となる．提案手法はスクリプトエンジン解析および ISA 解析，マルチパス実行機能の付与から構成される．スクリプトエンジン解析は，実行トレース取得，インタプリタループ検出，仮想プログラムカウンタ (VPC) 検出，VM 命令デコーダ・ディスパッチャ検出，条件分岐フラグ検出の 5 ステップからなる．また，ISA 解析は，VM 実行トレース取得，VM 分岐命令検出の 2 ステップからなる．最後に，マルチパス実行機能を付与する 1 ステップがある．以降で，それぞれのステップの詳細を述べる．

3.2 準備：テストスクリプト作成

テストスクリプトとは，スクリプトエンジンを動的解析する際に入力されるスクリプトである．本研究でのテストスクリプトは，命令の実行やメモリ読み書きの回数に着目し，異なる回数の間に生じる差分を捉えるために用いられる．検出手法にあわせたテストスクリプトの作成方法の詳細は，3.3.2 項および 3.3.5 項にて，それぞれ述べる．このテストスクリプトは解析の事前に準備するものであり，手動で作成するものである．なお，この作成には，対象のスクリプト言語の仕様に関する知識が必要となるが，2.4 節で述べた前提とは矛盾しない．

3.3 スクリプトエンジン解析

3.3.1 実行トレース取得

提案手法でのアーキテクチャ情報を得るためのスクリプトエンジンの動的解析は，実行トレースの取得に基づく．この実行トレースは，3.4.1 項の VM 実行トレースとは異なり，ネイティブプログラムの実行トレースを取得する．本手法での実行トレースは，ブランチトレースとメモリアクセストレースで構成される．ブランチトレースは，分岐のトレースである．ブランチトレースでは，実行の際の分岐命令の種類と，分岐元アドレスと分岐先アドレスを記録していく．命令フックによってログ出力用のコードを挿入し，分岐命令の呼び出しごとにそれを実行させて，記録していく．メモリアクセストレースは，メモリの読み書きのトレースである．メモリアクセストレースでは，実行の際のメモリ操作命令の種類と，その操作対象のメモリアドレスを記録していく．ブランチトレースと同じく，メモリ操作命令の命令フックによってログ出力用のコードを挿入する．

```
a = 0
For N=0 To 10
   a = a + N
   a = a * N
   a = a - N
   …
Next
```
M
N

図 3 インタプリタループ検出のためのテストスクリプト
Fig. 3 Test script for interpreter loop detection

3.3.2 インタプリタループ検出

このステップでは，ブランチトレースを解析し，インタプリタループを検出する．デコード・ディスパッチ型 VM のインタプリタループでは一般に，VM 命令の実行後に分岐命令でループの先頭に飛ぶ．そこで，ブランチトレース中の分岐命令の分岐先アドレスの中から，インタプリタループの先頭に該当するものを検出する．ここでは，分岐回数に着目した差分実行解析を用いる．この検出に用いるテストスクリプトの一例を図 3 に示す．このテストスクリプトでは，繰り返し処理を用いる．テストスクリプト内の繰り返し回数を増やすと，回数の増加に比例して VM のインタプリタループの先頭への分岐の回数も増加する．また，この分岐は，繰り返し処理の中の文の数にも比例する．

繰り返しの回数を N，繰り返される文の数を M としたとき，大まかには MN 程度のインタプリタループの先頭への分岐が発生する．これをそれぞれ $2N$ と $2M$，$3N$ と $3M$ などと増やした時に，$4MN$，$9MN$ という増え方をした分岐先をインタプリタループの先頭として検出する．

3.3.3 仮想プログラムカウンタ検出

このステップでは，インタプリタループ内でのメモリアクセスを解析し，VPC を検出する．VPC は一般にメモリ上に格納されており，VM 命令が実行されるたびに更新されるため，このメモリアドレスへの値の書き込みが発生する．この VPC を検出するため，メモリの書き込み回数に着目した差分実行解析を用いる．

繰り返しの回数を N，繰り返される文の数を M としたとき，大まかには MN 程度の VPC を保持するメモリへの書き込みが発生する．これをそれぞれ $2N$ と $2M$，$3N$ と $3M$ などと増やした時に，$4MN$，$9MN$ という増え方をしたメモリを VPC として検出する．

3.3.4 VM 命令デコーダ・ディスパッチャ検出

このステップでは，スクリプトエンジンのバイナリの静的解析し，インタプリタループ内のデコーダ・ディスパッ

チャを検出する．デコーダ・ディスパッチャの実装には一般に，2 つの種類が存在する．1 つ目はソースコード 2 のように Switch 文を用いた実装であり，2 つ目は関数テーブルやジャンプテーブルを用いた実装である．Switch 文および関数テーブル，ジャンプテーブルを認識する手法は，既存の技術で実現されている [6] ため，これを用いて検出する．検出された Switch 文およびテーブルジャンプのうち，インタプリタループ内に存在するものを，デコーダ・ディスパッチャとして検出する．この Switch 文およびテーブルジャンプは，ソースコード 2 の行目のように，VM 命令のオペコードを入力としてディスパッチ先を決定する．そのため，この検出によって VM 命令のオペコードを取得できる．

3.3.5 条件分岐フラグ検出

このステップでは，インタプリタループ内でのメモリアクセスを解析し，条件分岐フラグを検出する．前提として，レジスタマシンでは仮想レジスタが，スタックマシンでは仮想スタックが，条件分岐フラグを保持する．そのため，対象の VM がスタックマシンの場合は仮想スタックおよびスタックポインタを，レジスタマシンの場合は仮想レジスタを検出することになる．ソースコード 3 に，条件分岐フラグ検出のためのテストスクリプトの一例を示す．

ここでは，インタプリタループ内でのメモリアクセスから，2 段階の絞り込みをすることで，条件分岐フラグを検出する．このフラグには，分岐がなされる (Taken) か，なされない (Not taken) かの 2 つの状態がある．また，このフラグは，条件分岐の回数に比例した回数，読み込まれると考えられる．

このことから，1 段階目の絞り込みとして，条件分岐の回数に比例した回数のメモリ読み込みがあるメモリを抽出する．そして，2 段階目の絞り込みとして，各メモリ読み込み時の値が，テストスクリプトの条件分岐と対応付くように 2 つの値を行き来しているメモリを抽出する．例えば，Taken を X，Not taken を Y で保持している場合，ソースコード 3 では，条件分岐は Taken, Not taken, Taken, Taken, Not taken となるので，X, Y, X, X, Y と行き来しているメモリアドレスを抽出する．これを分岐の回数を変更しながら繰り返すことにより，条件分岐フラグを検出できる．また，メモリ読み込み時の値を検査し，上記の条件分岐のフラグを保持しているメモリアドレスを指すメモリがあれば，スタックポインタとして検出する．

ソースコード 3　条件分岐フラグ検出のテストスクリプト

```
1 bool_array = Array(True, False, True, True, False)
2 a = 0
3 For Each bool in bool_array
4     If bool Then
5         a = a + 1
6     End If
7 Next
```

3.4 命令セットアーキテクチャ解析

3.4.1 VM 実行トレース取得

このステップでは，VM 上で実行される命令のトレース（VM 実行トレースと呼ぶ）を取得する．提案手法による VM の ISA 解析は，この VM 実行トレースに基づく．この VM 実行トレースは，3.3.1 項の実行トレースとは異なり，VM の持つ実行情報の実行トレースを取得する．提案手法での VM 実行トレースは，VPC と VM 命令のオペコードで構成される．3.3.3 項および 3.3.4 項で検出した VPC と，VM 命令デコーダから得られるオペコードを記録していく．

3.4.2 VM 分岐命令検出

このステップでは，VM 実行トレースを分析し，VM における分岐命令（VM 分岐命令と呼ぶ）を検出する．ここでのテストスクリプトは，VM 分岐命令が含まれていればよいため，分岐の制御構文を含むスクリプトでありさえすればよい．例えば，インターネット上から収集したり，公式ドキュメントから取得したりして準備できる．

まず，テストスクリプトを実行し，多数の VM 命令の VM 実行トレースを取得する．これらの VM 実行トレースから，VM 命令のオペコードと，命令の実行前後での VPC のオフセットを，組として抽出する．このオフセット o は，命令の実行前の VPC の値を p_{prev}，実行後の値を p_{next} として，$o = p_{next} - p_{prev}$ で算出する．

ここで，ある VM 命令が分岐命令のとき，このオフセットは分岐先に依存して変化する一方，分岐命令以外のときは，オフセットは VM 命令のサイズに依存して変化する．そのため，VM 命令のオペコードとオフセットの組を収集し，オペコードごとにオフセットの値を見たとき，分岐命令であれば分岐先によって様々な値にばらつき，分岐命令以外であれば VM 命令のサイズという特定の値に集中する．

したがって，ばらつきを評価するため，分散 s^2 を用いる．あるオペコードに対するオフセットの集合 O を $O = o_0, o_1, ..., o_N$ とし，t を閾値としたとき，分岐命令か否かは以下のように判定される．テストスクリプトの数が，ある ISA の VM が持つ全ての VM 分岐命令を含むほど十分に大きければ，これによって，VM 分岐命令を網羅的に検出できる．

$$\overline{o} = \frac{1}{N}\sum_{k=0}^{N} o_k \tag{1}$$

$$s^2 = \frac{1}{N}\sum_{k=0}^{N} (o_k - \overline{o})^2 \tag{2}$$

$$isBranch(O) = \begin{cases} true & (s^2 > t) \\ false & (otherwise) \end{cases} \tag{3}$$

表 1　実験環境
Table 1　Experimental environment

CPU	Intel Core i7-6600U CPU @ 2.60GHz
メモリ	2GB
OS	Windows 7 32-bit
Lua	Lua 5.3.5
VBScript	vbscript.dll (ReactOS 0.4.11)
VBScript	vbscript.dll 7.01.1048

3.5　マルチパス実行機能の付与

ここまでで，3.3.3 項で VPC を，3.4.2 項で条件分岐の VM 命令のオペコードを，3.3.5 項で条件分岐フラグを得ている．これらに基づいて，下記のような解析用コードをスクリプトエンジンに付与する．まず，VPC の指す先を監視し，VM 命令が実行される度に，そのオペコードを確認できるようにする. 実行された VM 命令のオペコードが，条件分岐命令のものであるとき，実行状態をフォークする．フォークされた実行状態のうち，一方はそのまま実行し，もう一方は，条件分岐フラグを書き換えて実行する．これにより，マルチパス実行が実現される．

4.　評価

提案手法の評価のため，実行トレース取得とにマルチパス実行機能の付与に Intel Pin[7] を，デコーダ・ディスパッチャの検出に IDA Pro[6] を用いて，プロトタイプを実装した．このプロトタイプを，検出精度，実行時間, 実検体への解析性能の 3 点から評価した．

4.1　実験環境

実験環境を表 1 に示す．この環境を，仮想マシン上に構成した．CPU には 1 つの仮想 CPU を割り振ってある．本研究は本来は，プロプライエタリソフトウェアのスクリプトエンジンに対しての適用を想定しているが，実験後の検証を容易にするため，実験にはオープンソースとプロプライエタリの両方のスクリプトエンジンを用いた．ただし，オープンソースについては，ソースコードから得られる情報は結果の検証以外には一切用いず，プロプライエタリを対象とする場合と同等の状況としている．オープンソースには，Lua[8] と，ReactOS プロジェクト [9] で実装されている VBScript を用いた．前者は，簡易なオープンソースのスクリプトエンジンであり実験でよく用いられるため，後者は，攻撃者によく利用されるプロプライエタリなスクリプトエンジンのオープンソース実装であるため，実験に採用した．ReactOS 上では Intel Pin が正しく動作しないため，vbscript.dll のみを抽出して実験環境の Windows 上に移植して実験した．プロプライエタリには，Microsoft の正規の VBScript を用いた．

4.2　検出精度の評価

提案手法による解析および検出の精度を評価するため，アーキテクチャ情報を検出する実験を実施した．実験の結果を表 2 に示す．表頭に記載されたアーキテクチャ情報を検出できた場合は ✓，できなかった場合は ✗ としている．本研究の目的はマルチパス実行の実現であるため，条件分岐制御の列が ✓ となることが研究の一つの目標である．

なお，提案手法による検出の正しさは，以下のように検証している．オープンソースのスクリプトエンジンについては，検出された箇所に対応するソースコード上の記述を解析して確認する．その記述が検出した要素を実装したものであれば，正しく検出されたとしている．例えば，VPC の検出において，対応するソースコード上の変数が VPC の働きをするものであれば，正しく検出されたとする．また，プロプライエタリのスクリプトエンジンについては，上記を手動解析にてベストエフォートで実施した．

表 2 の通り，提案手法によって，いずれのアーキテクチャ情報も検出できていた．また，それに基づいてマルチパス実行用のコードを付加することで，条件分岐を制御できていた．Lua のディスパッチャは Switch 文を，VBScript はいずれも関数テーブルを用いており，どちらも提案手法によって検出されていた．また，Lua はレジスタマシンであり，あるメモリ領域が条件分岐フラグを保持する仮想レジスタとして検出されていた．このメモリ領域を調査したところ，ソースコード上では *cond* という変数であり，条件分岐フラグを保持する変数であることが確認された．一方，ReactOS の VBScript はスタックマシンであり，検出されたスタックポインタは，ソースコード上では *exec* という構造体変数の *top* というメンバ変数であった．プロプライエタリの VBScript も同様の設計だと推測される．

Lua の VM 分岐命令は，公式に記載のうち，13 種類の分岐命令を検出できていた．一方で, 1 種類の OP_TESTSET については，分岐命令としては検出できなかった．この分岐命令の分散は非常に小さかったため，コード生成器によって，特定の分岐先に大きく偏った使い方をされているのではないかと考えられる．その場合，マルチパス実行の際にも，この検出漏れはあまり問題にならない．ReactOS の VBScript の分岐命令は 8 種類と少なく，全てを検出できていた．プロプライエタリの VBScript では，12 種類の分岐命令が検出された．

以上の結果から，提案手法によるアーキテクチャ情報の検出や，条件分岐の制御が，解析者に有益な一定の精度を持つことを示した．

4.3　実行速度の評価

提案手法による解析および検出の速度を評価するため，4.2 節の実験のあいだ，提案手法の各ステップの実行時間を計測した．実行時間を図 4 に示す．なお，提案手法の手

表 2　実験結果
Table 2　Experimental result

言語	インタプリタループ	VPC	ディスパッチャ	条件分岐フラグ	VM 分岐命令	条件分岐制御
Lua	✓	✓	✓	✓	13/14	✓
VBScript (ReactOS)	✓	✓	✓	✓	8/8	✓
VBScript	✓	✓	✓	✓	12	✓

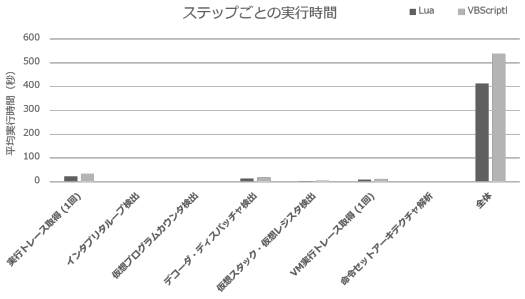

図 4　提案手法の実行時間
Fig. 4　Execution duration of proposing method

分岐条件フラグの変更なし

```
CreateObject Microsoft.XMLHTTP
CreateObject Adodb.streaM
CreateObject Wscript.shell
CreateObject Scripting.FileSystemObject
CreateObject WScript.Shell
GetObject winmgmts:¥¥.¥root¥cimv2
Invoke ExecQuery(Param1="Select * from Win32_Processor", Param2=NULL, Param3=48)
Invoke WScript.Quit
```

分岐条件フラグの変更あり

```
CreateObject Microsoft.XMLHTTP
CreateObject Adodb.streaM
CreateObject Wscript.shell
CreateObject Scripting.FileSystemObject
CreateObject WScript.Shell
GetObject winmgmts:¥¥.¥root¥cimv2
Invoke ExecQuery(Param1="Select * from Win32_Processor", Param2=NULL, Param3=48)
Invoke Scripting.FileSystemObject.GetSpecialFolder(Param1=0x0002)
Invoke Microsoft.XMLHTTP.Open(Param1="GeT", Param2="http://[anonymized by
author].top/search.php", Param3=False)
Invoke Microsoft.XMLHTTP.Send()
Invoke Microsoft.XMLHTTP.Status()
Invoke WScript.Shell.Type(Param1=Empty, Param2=0x0001)
Invoke WScript.Shell.Open()
Invoke MSXML.XMLHTTP.responseBody()
Invoke WScript.Shell.Write(Param1=<ARRAY>[snipped by author])
Invoke WScript.Shell.Savetofile(Param1="C:¥¥Users¥¥[anonymized by
author]¥¥AppData¥¥Local¥¥Temp¥¥tnuglovs.exe", Param2=0x0002)
Invoke WScript.Shell.Run(Param1="cmd.exe /c call "C:¥¥Users¥¥[anonymized by
author]¥¥AppData¥¥Local¥¥Temp¥¥tnuglovs.exe"")
```

図 5　解析ログの一例
Fig. 5　Example of analysis log

順のうち，テストスクリプトの作成については，あらかじめ用意するものとして，実行時間には含めていない．

図 4 より，実行トレースの取得および VM 実行トレースの取得に一定の時間を要していることが分かる．これは，実行トレースを得るために Intel Pin の VM 上で実行されているためであり，分岐やメモリアクセスの命令実行ごとにコールバックが発生することによる．また，デコーダ・ディスパッチャ検出の所要時間は，IDA Pro による静的解析に時間を要するためである．一方で，インタプリタループ検出や VPC 検出，条件分岐フラグ検出や VM 分岐命令検出は，いずれも実行トレースのログ行数に対して線形の計算量で処理できる程度のログ分析で済むため，多くが数秒程度までに収まっている．全体として，マルチパス実行に必要なアーキテクチャ情報は数百秒程度で得られており，現実的な時間内でマルチパス実行を実現できている．

4.4　実検体に対する解析性能の評価

研究用のマルウェア共有サービスの VirusTotal から，2017 年 1 月から 2019 年 7 月にかけてアップロードされた VBScript のスクリプトのうち，明示的に実行を終了するメソッドによって動的解析が途中で停止する 108 検体を抽出した．この動的解析には，我々の過去の研究 [5] で構成したスクリプト API トレーサを用いた．

これらの検体を，前述の API トレーサに提案手法に基づいてマルチパス実行機能を付与したスクリプト解析ツールを用いて解析した．なお，難読化解除の処理を無用に解析しないため，悪性スクリプトに見られる挙動が観測され始めてからマルチパス実行を開始するヒューリスティクスを導入した．解析ログの一例を図 5 に示す．本来は図の上側のログのように解析が途中で終了してしまうが，マルチパス実行の機能により，図の下側のログのように悪性な挙動が確認できている．破線枠内が差分である．

また，解析した 108 検体のうち，102 検体において，新たな挙動が観測されることを確認した．API トレースの引数として得られた URL およびファイルストリームのハッシュ値を VirusTotal で調査したところ，いずれも悪性であることが分かった．以上のことから，実際の悪性スクリプトに対しても，マルチパス実行によって有用な情報を抽出できることを確認した．

5.　議論

5.1　制約

提案手法は，VM に着目したマルチパス実行の構成手法のため，ネイティブコードに変換される JIT コンパイルは，提案手法では扱えず，対象外としている．しかし，JIT コンパイラを持たなかったり，無効化できるスクリプトエンジンであれば，提案手法は適用可能である．また，メモリ情報などに基づいて JIT コンパイルで生成されたコード領域を検出できれば，ネイティブコードに対するマルチパス実行の技術を適用できる，以上の理由から，JIT コンパイルは大きな問題とならないと考えられる．

提案手法では，スレッデッドコード型の VM もまた対象外としている．Python はビルドオプションでこの型を選択でき，Ruby はこの型のみを採用するなど，言語によっ

て採用実績がある．提案手法の汎用性を高めるため，この対応は今後の課題である．

5.2 マルチパス実行の有効性

本研究でのマルチパス実行は条件分岐のフラグを操作することで，経路を強制的に変更している．そのため，整合性の問題が生じる可能性がある．例えば，分岐条件に関わる変数の値が後段の悪性挙動にも影響を与える場合，条件分岐のフラグを操作しただけでは，後段の悪性挙動が正しいものとなるかの保証ができない．しかしながら，現状では有効な場合も少なくなく [4]，実行される可能性のある悪性な挙動が分かるため，価値はあると考えられる．

6. 関連研究

6.1 スクリプトのマルチパス実行の研究

スクリプトに対するマルチパス実行基盤の構成は数多く研究されてきた．例えば，JavaScript に対しては，Kudzu[2] は，シンボリック実行と呼ばれる，分岐条件の充足可能性を踏まえたマルチパス実行の一つを実現している．また，J-FORCE[4] は，経路強制実行と呼ばれるマルチパス実行の一つを実現するシステムである．そのほかの言語では，例えば Python には CutiePy[10] が Python に対するシンボリック実行エンジンである．これらはいずれも個別のスクリプトエンジンに対してマルチパス実行を実現したものであり，本研究とは目的が異なる．

Chef[3] は，手動で改造を施したスクリプトエンジンのバイナリ自体を，バイナリプログラム向けのシンボリック実行器の上で実行することで，スクリプトに対するシンボリック実行と同様の実行を実現しようとする研究である．多様なスクリプト言語に対するマルチパス実行の実現を解くべき課題としている点は，本研究と類似している．一方で，手動での改造を前提とし，スクリプトエンジンの解析はしないなど，前提やアプローチが本研究とは異なる．

6.2 仮想機械の解析の研究

例えば，Sharif[11] らの研究や Xu らの研究 [12] では，本研究と同じく，バイナリ解析によって VM のアーキテクチャを明らかにしている．しかし，これらはマルウェアが難読化に用いる VM を解析するもので，スクリプトエンジンの VM は対象としていない．スクリプトエンジンの VM の解析においては，任意のテストスクリプトを入力とした解析となるなど，前提やアプローチが異なる．

6.3 解析による機能拡張の研究

我々の過去の研究 [5] では，スクリプトエンジンを解析し，API トレース機能を付与している．これは単一の実行経路のみの解析機能を付与するものであり，マルチパス実行機能を付与する本研究とは，目的が異なる．

7. 結論

本研究では，悪性スクリプトの持つ，特定の条件を満たさなければ実行されない経路の解析の課題に着目し，その解決のために，スクリプトエンジンのバイナリにマルチパス実行機能を自動的に付与する手法を提案した．提案手法では，複数のテストスクリプトを実行した際の差分に着目し，スクリプトエンジンの持つ VM のアーキテクチャ情報を抽出する．実験を通して，提案手法がスクリプトエンジンにマルチパス実行機能を，現実的な時間で付与できることを確認した．また，実際の攻撃に用いられた悪性スクリプトに対して，単一の実行経路のみを解析する解析ツールと比較して，多くの有効な情報を抽出できることを示した．より汎用性の高い手法への改良が今後の課題である．

謝辞 本研究の一部は，JSPS 科研費 17KT0081 の助成を受けた．

参考文献

[1] Yokoyama, A., Ishii, K., Tanabe, R., Papa, Y., Yoshioka, K., Matsumoto, T., Kasama, T., Inoue, D., Brengel, M., Backes, M. et al.: SandPrint: fingerprinting malware sandboxes to provide intelligence for sandbox evasion, *RAID'16*, Springer, pp. 165–187 (2016).

[2] Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S. and Song, D.: A symbolic execution framework for javascript, *S&P'10*, IEEE, pp. 513–528 (2010).

[3] Bucur, S., Kinder, J. and Candea, G.: Prototyping symbolic execution engines for interpreted languages, *ACM SIGPLAN Notices*, Vol. 49, No. 4, ACM, pp. 239–254 (2014).

[4] Kim, K., Kim, I. L., Kim, C. H., Kwon, Y., Zheng, Y., Zhang, X. and Xu, D.: J-force: Forced execution on javascript, *WWW'17*, WWW Steering Committee, pp. 897–906 (2017).

[5] 碓井利宣，大月勇人，川古谷裕平，岩村 誠，三好 潤：スクリプト実行環境に対する解析機能の自動付与手法，コンピュータセキュリティシンポジウム 2018 論文集，情報処理学会，pp. 1016–1023 (2018).

[6] Hex-Rays: IDA, `https://www.hex-rays.com/products/ida/index.shtml`.

[7] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J. and Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation, *ACM SIGPLAN Notices*, Vol. 40, No. 6, ACM, pp. 190–200 (2005).

[8] Lua Community: Lua, `https://www.lua.org/`.

[9] ReactOS Project: ReactOS, `https://www.reactos.org/` (accessed: 2018-08-16).

[10] Sapra, S., Minea, M., Chaki, S., Gurfinkel, A. and Clarke, E. M.: Finding errors in python programs using dynamic symbolic execution, *ICTSS'13*, Springer, pp. 283–289 (2013).

[11] Sharif, M., Lanzi, A., Giffin, J. and Lee, W.: Automatic Reverse Engineering of Malware Emulators, *S&P'09*, IEEE, pp. 94–109 (2009).

[12] Xu, D., Ming, J., Fu, Y. and Wu, D.: VMHunt: A Verifiable Approach to Partially-Virtualized Binary Code Simplification, *CCS'18*, ACM, pp. 442–458 (2018).