スクリプト実行環境に対する実行遅延・実行停止を回避する機能の自動付与手法

碓井 利宣[1,2,a)]　幾世 知範[1]　川古谷 裕平[1]　岩村 誠[1]　松浦 幹太[2]

概要：マルウェアの動的解析を妨げる要因に，不要な命令の繰り返しによる実行の遅延や，例外による実行の停止がある．これらは，機械語形式のマルウェアでは，実行命令や実行状態を監視し，発生箇所を検出してスキップすることで回避できる．しかし，スクリプト形式のマルウェアでは，言語やエンジンごとに，未知のバイトコードの解析や仮想機械の監視の必要が生じるため，実現が困難である．
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，実行遅延および実行停止を回避する機能を自動付与する手法を提案する．提案手法では，仮想機械を解析して，実行状態の監視と制御を可能にするとともに，命令セットアーキテクチャを解析して，未知のバイトコードに対しても，制御フローグラフおよびコールグラフを構築可能にする．これらに基づいて，実行遅延・実行停止の発生箇所を検出してスキップする仕組みを付与する．これを Lua と VBScript のエンジンに適用し，本手法の解析結果が正しいことを確認した．さらに，実行遅延・実行停止を引き起こす 286 検体のうち 199 検体に対して，従来の解析環境では観測できない新たな挙動を観測できることを確認した．

キーワード：悪性スクリプト，実行遅延・実行停止，経路強制実行，動的解析，機能拡張

Automatically Appending Execution Stall/Stop Prevention to Vanilla Script Engines

Toshinori Usui[1,2,a)]　Tomonori Ikuse[1]　Yuhei Kawakoya[1]　Makoto Iwamura[1]
Kanta Matsuura[2]

Abstract: Malware widely uses an anti-analysis technique that stalls/stops execution with exceptions and unnecessary loops during analysis. This is generally prevented by detecting and skipping execution stall/stop based on the observation of executed instructions and execution context. However, for malicious scripts, developing such a mechanism is difficult because it requires observation of unknown virtual machines (VMs) with unknown instructions and construction of CFGs/CGs with them.
To address this problem, in this paper, we propose an approach that automatically appends execution stall/stop prevention to vanilla script engines. It first analyzes the target script engine VM for observing and controlling its execution context. It then analyzes the instruction set architecture of the VM for building a control-flow graph with unknown bytecode. With these steps, it appends the code that detects and skips the execution stall/stop. Through the experiments, we proved that our approach could indeed prevent the anti-analysis technique.

Keywords: malicious script, execution stall/stop, forced execution, dynamic analysis, function enhancement

1. はじめに

セキュリティ製品を回避する攻撃が増える中，正常利用に紛れ込ませやすい，悪性スクリプトによる攻撃が拡大し

[1] NTT 社会情報研究所
NTT Social Informatics Laboratories
[2] 東京大学生産技術研究所
Institute of Industrial Science, The University of Tokyo
[a)] toshinori.usui.rt@hco.ntt.co.jp

ている．こうした悪性スクリプトに対策を講じるには，その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，静的解析によって詳細な機能を明らかにするのは，困難である．そのため，悪性スクリプトの解析には，こうした難読化の影響を受けない動的解析が一般に用いられる．

一方，動的解析を妨げる障壁も存在する [1] [2] [3]．解析妨害と解析環境の不適合である．前者は，攻撃者が意図的に解析を妨げるものであり，後者は，意図されたものでなくとも，攻撃者が想定する環境と解析環境との不一致により，解析が妨げられるものである．我々は過去の研究で，スクリプトのマルチパス実行 [4] とテイント解析 [5] の 2 つの動的解析技術を実現した．これらは，特定の条件下でしか動作しない解析妨害を持つ悪性スクリプトを解析できる．

しかしながら，悪性スクリプトの解析においては，いまだ対策がなされていない，実用面における 2 つの大きな障壁が存在する．1 つ目は，実行の遅延による解析妨害である．これは，ストーリングコード (Stalling code) [1] と呼ばれ，不要な命令の繰り返しによって実行に遅延を引き起こすものである．一般的な動的マルウェア解析サンドボックスは，規定の時間内に観測された挙動のみを解析するため，こうした実行の遅延によって，本来解析するべき挙動に至らずに解析が終了してしまう．2 つ目は，解析環境の不適合によって発生する例外による実行の停止 [2] [3] である．これは，マルウェアが想定する環境と解析環境の間の差異によって例外が発生し，その例外が捕捉されなかった場合に，実行が停止して解析が終了してしまうものである．

これらは，機械語形式のマルウェアにおいては，経路強制実行によって解決できる [1] [2] [3]．すなわち，実行中にストーリングコードや未処理例外の箇所を検出し，適切な地点から強制的に実行を再開させることで，問題の箇所をスキップし，実行を継続させられる．この時，実行の再開地点は，制御フローグラフ (Control Flow Graph; CFG) やコールグラフ (Call Graph; CG) に基づいて決定される．

しかしながら，悪性スクリプトにおいては，この手法は適用が難しい．内部仕様のよく知られた実装を持つ JavaScript などを除けば，スクリプトエンジンの内部仕様は一般に公開されていない．特にプロプライエタリのスクリプトエンジンにおいては，リバースエンジニアリングによらずに内部仕様を知ることは困難である．この時，たとえば命令セットアーキテクチャ (Instruction Set Architecture; ISA) の内部仕様が未知の仮想機械 (Virtual Machine; VM) を持つスクリプトエンジンのバイトコードに対しては，分岐を認識できず，CFG および CG を構築できない．したがって，実行の再開地点を決定できなくなるため，前述の経路強制実行は適用できない．また，多様なスクリプトエンジンを，人手のリバースエンジニアリングで個別に内部仕様を把握し，CFG や CG の構築を可能にするのは，かかる労力の膨大さから現実的でない．

この問題を解決するため，本研究では，実行遅延および実行停止を回避する機能を，スクリプトエンジンに自動付与する手法を提案する．提案手法では，まず VM の解析により，バイトコードの解釈実行の仕組みを明らかにし，実行状態の監視と制御を可能にする．さらに，ISA の解析を通して，実行された未知のバイトコードに対しても，CFG および CG を構築可能にする．これらに基づいて，実行遅延や実行停止の発生箇所を検出してスキップする仕組みを，スクリプトエンジンに自動で付与する．

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．その結果，解析によって VM のアーキテクチャおよび ISA を明らかにして，未知のバイトコードに対して CFG および CG を構築し，実行遅延および実行停止を回避できることを確認した．また，この解析は，スクリプトエンジン 1 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 [6] で作成したスクリプト API トレーサに対して，提案手法に基づいて実行遅延・実行停止の回避機能を付与し，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，実行遅延・実行停止の回避機能を有さないトレーサと比較して，より多くの悪性な挙動を明らかにできることが確認できた．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，実行遅延・実行停止を引き起こす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や ISA の情報に基づいて実行遅延・実行停止を回避する手法を初めて提案した．
- 実験を通して，提案手法によって未知のバイトコードに対して CFG および CG を構築し，実行遅延・実行停止を回避できることを確認した．
- 提案手法によって実行遅延・実行停止の回避機能を付与した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．

2. スクリプトと実行遅延・実行停止

2.1 解析対象のスクリプト

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手動で解除した上で，一部抜粋して整形したものである．

この悪性スクリプトの前半部は，ストーリングコードによる解析妨害を具備しており，時間を要する処理を繰り返し実行する（2〜4 行目）．この繰り返しを完了する前に，規定の解析時間を超えてしまうと，悪性な挙動（5 行目）を観測できないまま，解析が終了してしまう．また，後半部

```
1  ' ストーリングコードによる解析妨害
2  For i = 0 To 10000000
3      do_something_time_consuming()
4  Next
5  do_malicious()
6
7  ' 解析環境の不適合による例外が発生するコード
8  Set obj = CreateObject("Object.WeDoNotHave")
9  do_malicious()
```

ソースコード 1　解析対象の悪性スクリプトの一例 (VBScript)

のように，オペレーティングシステムやスクリプトエンジンのバージョンの違いなどで存在しないオブジェクトを作成しようとした場合（8 行目），例外が発生して実行が停止してしまう．これにより，やはり悪性な挙動（9 行目）を観測できないまま，解析が終了してしまう．本研究では，こうした実行遅延や実行停止を回避し，その先にある悪性な挙動を解析可能にすることを目指す．

2.2　実行遅延・実行停止の回避

本研究の扱う問題を明らかにするために，前節の実行遅延・実行停止が，バイナリおよび特定の言語のスクリプトに対する既存研究で，どのように対処されているかの概要を述べる．HASTEN [1] は，バイナリでの実行遅延の回避機能を具備するシステムである．HASTEN ではまず，Sreedhar らによる既存手法 [7] を用いて CFG からループを検出する．次に，コールされてからまだリターンしていないコードブロックを生存中とみなす．それらを含む繰り返しのうち，さらに現在実行中のコードブロックを含むものを，ストーリングコードを構成する繰り返し（ストーリングループと呼ぶ）として検出する．HASTEN は，ストーリングループの終端での条件分岐フラグの書き換えにより，繰り返しを抜けて実行を再開する．

J-Force [2] は．JavaScript での実行停止の回避機能を具備するシステムである．解析対象のスクリプト全体をトップレベル例外ハンドラ内に配置し，全ての未処理例外を捕捉する．例外を捕捉した場合，例外の発生箇所から見て直近に呼ばれた関数を発見し，そこから実行を再開する．

これらを多様なスクリプトエンジンに対して自動的に実現するためには，次節で説明するスクリプトエンジンの VM の内部仕様を把握するとともに，バイトコードレベルでの CFG・CG を構築する必要がある．

2.3　スクリプトエンジンの仮想機械

スクリプトエンジンは一般に，まず入力されたスクリプトを解析し，抽象構文木を経て，バイトコードを生成する．そして，それを VM で解釈実行することで，スクリプトの実行を実現する．そのため，スクリプトの実行状態の把握や制御のためには，VM のアーキテクチャを認識すること

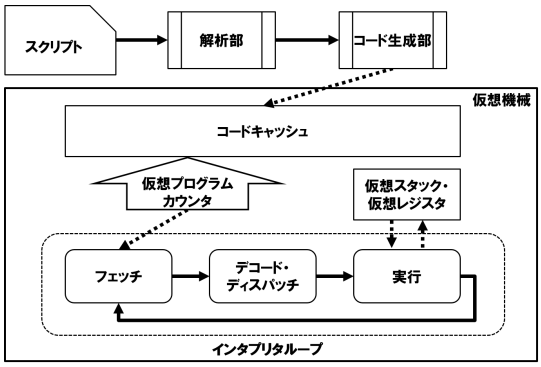

図 1　仮想機械の構成図
Fig. 1　Architecture of virtual machine

が重要である．そこで，本節では，今後の議論に向けて，VM のアーキテクチャを記述する．

バイトコードの解釈実行用の 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．ここでは，広く用いられている前者に焦点を当てる．なお，本研究で提案する手法は，どちらの型にも対応できる．

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**（2〜8 行目）内で実施される．また，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．

- 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．(3 行目)
- VM 命令は**デコーダ**によって解釈され．(4 行目)，**ディスパッチャ**によって命令の内容を実装した箇所にディスパッチされる．(5 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．そのうち，条件分岐は**条件分岐フラグ**に基づき実行される．

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

ソースコード 2　仮想機械の擬似コード

2.4　アプローチ

実行遅延・実行停止を回避するためには，2.2 節の通り，CFG・CG の構築を要する．そこで，ISA が未知のバイトコードに対し，この構築を実現するために必要な情報を考える．まず，バイトコードを正しく走査するため，(1) コー

ドキャッシュの把握を要する．また，実行される VM 命令の観測に，(2) デコーダ・ディスパッチャを要し，その命令が分岐か否か，さらにはどういった種類の分岐かを知るため，(3) 条件分岐・無条件分岐，(4) コール・リターンの VM 命令の判別を要する．そして，その分岐命令に基づいて，ベーシックブロックや分岐元，分岐先を把握し，ノードやエッジを追加するために，(5)VPC を要する．

これらが判明していれば，VPC の値とデコーダの解釈する VM 命令を監視し，各種の分岐命令が実行される度に，グラフに適切なノードやエッジを追加することで，CFG・CG を構築できる．また，CFG・CG に基づいて 2.2 節の手法で実行遅延・実行停止を検出した際に，当該箇所をスキップするには，(5)VPC や (6) 条件分岐フラグの操作を要する．そのため本研究では，これらの 6 つの情報を解析によって取得する．それに基づいて，2.2 節のように実行遅延・実行停止の回避を実現するための，解析用コードをスクリプトエンジンに付与する，というアプローチをとる．

2.5 本研究での仮定

本研究では，(1) 難読化されておらず，(2)VM による解釈実行をし，(3)Just-In-Time コンパイラは持たないか無効化できる，というスクリプトエンジンを対象として仮定する．こうしたスクリプトエンジンは，ごく一般的なものである．また，スクリプトエンジンの内部仕様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の知識は仮定する．

3. 提案手法

3.1 概要

提案手法では，2.4 節の 6 つのアーキテクチャ情報を動的解析によって取得する．この解析には，我々が過去の研究 [6] で提案した，スクリプトエンジンの差分実行解析と呼ぶ手法を用いる．それに基づいて，解析用コードを挿入することで，実行遅延・実行停止の回避機能を付与する．

図 2 に提案手法の概要を示す．まず，事前にテストスクリプトの準備を要する．提案手法は VM 解析，ISA 解析，機能付与からなる．VM 解析は，実行トレース取得，VPC 検出，デコーダ・ディスパッチャ検出，条件分岐フラグ検出，コードキャッシュ検出の 5 ステップからなる．また，ISA 解析は，VM 実行トレース取得，分岐 VM 命令判定の 2 ステップからなる．最後に，機能付与は，CFG・CG の構築機能の付与，実行遅延・実行停止の回避機能の付与の 2 ステップからなる．以降で，各ステップを詳述する．

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の動的解析の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では，メモリ読み書きの回数や値の遷移パターン

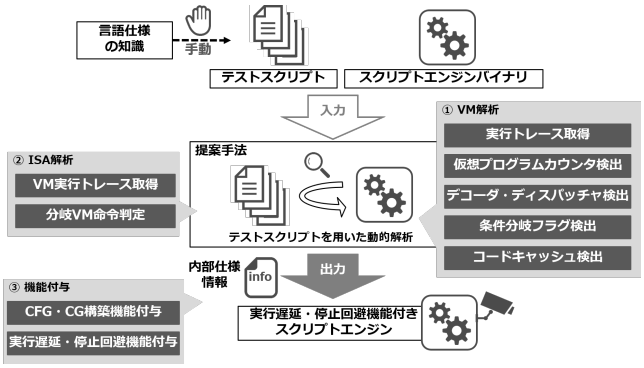

図 2 提案手法の概要図
Fig. 2 Overview of proposing method

```
1  a = 0
2  ' --VPC 検出のためのテストスクリプト
3  For i = 1 To 1000
4      a = a + 1
5  Next
6
7  ' --条件分岐フラグ検出のためのテストスクリプト
8  For Each bool in Array(True, False, True, True, False)
9      If bool Then
10         a = a + 1
11     End If
12 Next
```

ソースコード 3 テストスクリプトの一例 (VBScript)

を操作し，それらに生じる差分を捉えるために用いられる．

提案手法で用いるテストスクリプトの一例をソースコード 3 に示す．これらの詳細な説明は，それぞれを用いて検出する各ステップの 3.3.2 項および 3.3.3 項で参照しながら行う．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，本研究では手動での作成を想定している．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，2.5 項で述べた仮定とは矛盾しない．

3.3 仮想機械解析

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．ここでの実行トレースは，スクリプトエンジンのバイナリの実行状態を記録するものである．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り，実行の際の分岐命令の種類と，分岐元と分岐先のアドレスを記録していく．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り，実行の際のメモリ操作命令の種類（読み書き）と，その対象のメモリアドレスと値を記録していく．いずれのトレースも，命令フックによってログ出力用のコードを挿入し，分岐命令やメモリ操作命令の呼び出しごとにそれを実行させて，記録していく．

3.3.2 仮想プログラムカウンタ検出

このステップでは，メモリアクセストレースを解析し，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，このメモリ領域への値の書き込みが発生する．そこで，各メモリ領域への書き込み回数に着目した差分実行解析を用いる．

このステップでは，ソースコード 3 の 1〜5 行目の，規定回数の繰り返しをするテストスクリプトを用いる．繰り返しの回数を N，繰り返される VM 命令の数を M，初期化時などの固定の書き込み回数を c としたとき，VPC を保持するメモリ領域への書き込み回数 X は $X = MN + c$ となり，X と N は比例の関係がある．ここで，N はテストスクリプトで操作可能であり，X はメモリアクセストレースで観測可能であるため，異なる 3 つの N のテストスクリプトを用いて，前述の式を連立 3 元 1 次方程式とし，矛盾なく解けたメモリ領域を VPC として検出する．

3.3.3 条件分岐フラグ検出

このステップでは，メモリアクセストレースを解析し，条件分岐フラグを検出する．ここでは，ソースコード 3 の 8〜12 行目の，特定のパターンで条件分岐を繰り返すことで，条件分岐フラグの値を操作するテストスクリプトを用いる．条件分岐フラグには，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．このことから，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 つの値を遷移しているメモリ領域を抽出する．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，8,9 行目より，条件分岐は Taken, Not taken, Taken, Taken, Not taken となるので，X, Y, X, X, Y と値が遷移しているメモリ領域を抽出する．これを条件分岐の回数や遷移パターンを変更しながら繰り返すことで，条件分岐フラグを検出できる．

3.3.4 コードキャッシュ検出

このステップでは，VM 実行トレースとメモリアクセストレースを分析し，コードキャッシュを検出する．まず，スクリプトの実行時に，VPC が指すメモリ領域を VM 実行トレースから取得する．そして，そのメモリ領域を確保したメモリ割り当て関数を呼び出しているコード箇所を検出し，そこで確保されたメモリ領域をコードキャッシュとして検出する．さらに，コードキャッシュに書き込みをしているコード箇所をメモリアクセストレースから検出し，それをコードキャッシュへバイトコードを格納する処理とみる．これは，コードキャッシュの更新の検出に用いる．

3.3.5 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を静的解析し，デコーダ・ディスパッチャを検出する．デコーダ・ディスパッチャには一般に，ソースコード 2 のように Switch 文によるジャンプテーブルを用いた実装と，関数テーブルを用いた実装の 2 種類が存在する．これらのテーブルによる分岐の検出は，既存の静的解析技術で実現されている [8] ため，これを用いる．検出されたテーブルによる分岐のうち，VPC の更新が常に伴うものを，デコーダ・ディスパッチャとして検出する．デコーダは，ソースコード 2 の 4 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．

3.4 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 の持つ実行状態のトレースを取得する．ここでの 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．3.3.2 項で検出した VPC の値と，3.3.5 項で検出した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 分岐 VM 命令判定

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（分岐 VM 命令と呼ぶ）を検出する．ここでのテストスクリプトは，分岐 VM 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 を用いる．あるオペコードに対するオフセットの集合を $O = \{o_0, o_1, ..., o_N\}$ としたとき，O の平均を $\overline{o} = \frac{1}{N+1}\sum_{k=0}^{N} o_k$ として，$s^2 = \frac{1}{N+1}\sum_{k=0}^{N}(o_k - \overline{o})^2$ で算出される．ここで，t を閾値として，$s > t$ であれば分岐命令と判定し，そうでなければ，分岐命令でないと判定される．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判定できる．この判定ののち，その中からさらに，CFG・CG の構築に要する，条件分岐およびコール，

リターンの 3 つの VM 命令を判定する．

3.4.2.1 条件分岐 VM 命令の判定

条件分岐の際には，分岐先を決定するために，必ず条件分岐フラグへのアクセスが発生する．そのため，各分岐 VM 命令の実行の際に，条件分岐フラグにアクセスしているかを検証することで，条件分岐 VM 命令を判定できる．そこで，VM 実行トレースとメモリアクセストレースに基づいて，分岐命令 VM 命令のうち，条件分岐フラグへのアクセスを伴うものを，条件分岐 VM 命令と判定する．

3.4.2.2 コール・リターン VM 命令の判定

コール VM 命令による分岐では，呼び出し元の直後のアドレスが保存され，呼び出されたサブルーチンの実行後には，リターン VM 命令によって，その保存されたアドレスに戻ってくる特徴がある．そこで，(1) 分岐先がバイトコード上で隣接していない（サブルーチン呼び出しのため），(2) 以後の他の分岐 VM 命令によってバイトコード上の直後のアドレスに戻る，の条件を満たす分岐 VM 命令の組を，コールおよびリターンの VM 命令と判定する．

3.5 機能付与

3.5.1 制御フローグラフ・コールグラフ構築

このステップでは，これまでに獲得した内部仕様の情報に基づき，CFG・CG を構築する．バイトコードは実行時に動的に生成されるため，この構築も実行時に行われる．

まず, バイトコードがコードキャッシュに格納され, VPC に値が読み込まれた際には，その VPC の値をエントリーポイントとして，バイトコードを走査していく．そして，分岐 VM 命令の度に，新たなベーシックブロックをノードとして追加し，その分岐元から分岐先へのエッジを追加する．また，実行時にも，間接分岐の VM 命令があった場合には，上記と同様にしてノードとエッジを追加する．これは，動的 CFG と呼ばれる [1]．実行中の VPC と VM 命令の監視により，これを実現する．

3.5.2 実行遅延・実行停止回避

このステップでは，前節で構築された CFG・CG に基づき，実行遅延・実行停止を回避する機能を付与する．CFG と CG さえ構築できていれば，既存の手法に基づいて，ストーリングコードの検出や未処理例外の捕捉をし，スキップすることで，実行遅延・実行停止の回避を実現できる．この実現は，2.2 節で紹介した，HASTEN [1] の手法と J-Force [2] の手法にそれぞれ基づく．なお，実行中は VPC と VM 命令を監視し続け，CFG・CG 上のどの位置を実行中であるかを常に追跡することで，これらの手法を適用する．したがって，スクリプトエンジンに，上記を実現するための解析用コードを挿入し，出力する．

4. 評価

提案手法の評価のため，プロトタイプを実装した．実行トレース取得には，動的バイナリ計装のフレームワークの Intel Pin [9] を，デコーダ・ディスパッチャの検出には，逆アセンブラの IDA Pro [8] を，それぞれ用いた．実装したプロトタイプを，各ステップの精度，実行時間，実検体への解析性能の 3 点から評価した．

表 1 実験環境
Table 1 Experimental environment

CPU	Intel Core i7-6600U CPU @ 2.60GHz
メモリ	4GB
OS	Ubuntu 20.04 LTS, Windows 10 64-bit
Lua	Lua 5.4.2
VBScript	vbscript.dll (ReactOS 0.4.11)
VBScript	vbscript.dll 7.01.1048

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，プロプライエタリのスクリプトエンジンに対する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 と，ReactOS プロジェクト [10] 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．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

提案手法の各ステップでの精度を評価するため，アーキテクチャ情報の検出と分岐 VM 命令の判定，CFG・CG の構築をする実験を実施した．実験の結果を表 2 に示す．表頭に記載の各ステップにおいて，正しく検出できた場合は ✓，できなかった場合は ✗ を記載している．また，表頭に記載の各 VM 命令については，（正しく判定できた数）/（実際の数）を記載している．CFG・CG の構築精度については，手動で解析したスクリプトの CFG・CG を正例として，提案手法が構築したものとのグラフ編集距離を評価した．これは，2 つのグラフを一致させるために要する，ノードやエッジの追加や削除の数を評価するものである．なお，いずれのステップも，正しさはソースコードとバイナリの手動解析で確認した．

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．検出されたメモリ領域に対応

した変数をソースコード上で確認したところ，VPC は *pc* 変数や *exec_ctx_t* 構造体の *instr* メンバ変数，条件分岐フラグは *cond* 変数や *stack_pop_bool* 関数内の *b* 変数，コードキャッシュは *Proto* 構造体の *code* メンバ変数や *exec_ctx_t* 構造体の *code* メンバ構造体の *instrs* メンバ変数というように，検出した情報と変数名との間に対応が見られた．

分岐 VM 命令は，Lua で 21 種類，ReactOS の VBScript で 8 種類あり，それら全てを分岐 VM 命令と判定できていた．判定された VM 命令に対応したソースコード上での実装を確認したところ，条件分岐 VM 命令は，Lua ではいずれも *docondjump/donextjump* というマクロを用いる 11 種類の命令であり，VBScript では *OP_jmp_true* および *OP_jmp_false*，コール VM 命令は，Lua では *L_OP_CALL*，VBScript では *OP_icall*，*OP_icallv*，*OP_mcall*，*OP_mcallv*，リターン VM 命令は，Lua では *L_OP_RETURN*，*L_OP_RETURN0*，*L_OP_RETURN1*，VBScript では *OP_ret* という命令名でそれぞれ定義されていた．いずれも，判定した命令と，ソースコード上の命令名との間に対応が見られた．

また，CFG・CG の正例とのグラフ編集距離は，高々数ノード・エッジに収まり，概ね正確な CFG・CG を構築できていた．さらに，この CFG・CG に基づき，実行遅延・実行停止がどちらもをスキップできていることも確認できた．以上の結果から，提案手法による内部仕様の情報の検出や分岐 VM 命令の判定，CFG・CG の構築が，実行遅延・実行停止の回避に必要な一定の精度を持つことを示した．

4.3 実行速度の評価

提案手法による検出や判定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 3 に示す．

図より，まず，実行トレースの取得に，一定の時間を要している．これは，取得に際して Intel Pin の VM 上で実行していることと，分岐やメモリアクセスの命令の実行ごとにコールバックが発生することによる．一方，VPC および条件分岐フラグ，コードキャッシュの検出は，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分析で済むため，多くが数秒程度までに収まっている．また，デコーダ・ディスパッチャ検出の所要時間は，おもに IDA Pro による静的解析による．VM 実行トレースの取得は，実行トレースと比べて，コールバックの発生が VPC とデコーダ・ディスパッチャの箇所に限られるため，数秒程度のみで済んでいる．条件分岐とコール・リターンの VM 命令の判定が最も時間を要しており，前者は全ての VM 命令に対する VPC オフセットの計算に，後者はコールとリターンの関係の探索に，要因があると考えられる．全体として，実行遅延・実行停止の回避に要する情報は数百秒程度で得られており，現実的な時間内での解析を実現できている．

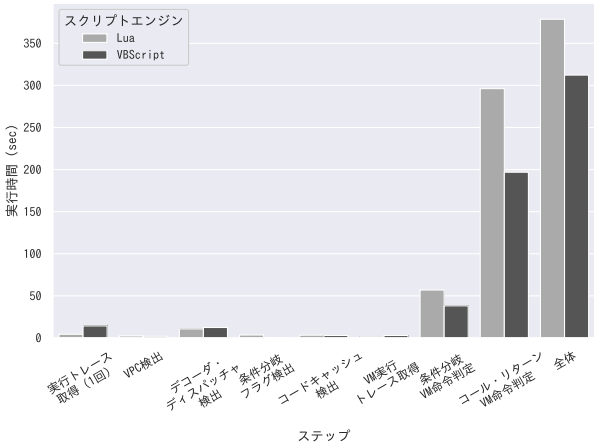

図 3 提案手法の実行時間
Fig. 3 Execution duration of proposing method

4.4 実検体に対する解析性能の評価

提案手法の実検体への有効性を評価するため，我々の過去の研究 [6] で構成したスクリプト API トレーサに対して，実行遅延・実行停止の回避機能を追加し，解析を実施した．まず，ソースコード 1 の悪性スクリプトを解析した．その結果，本来は実行遅延・実行停止が発生する箇所をスキップし，その先の悪性な挙動を解析できた．例外の先では，*obj* の不在により，関連した挙動は失敗するものの，*obj* と独立の挙動は観測された．また，トレーサの仕様から，*obj* に関する挙動を試みた痕跡も観測された．

さらに，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，実行中の未処理例外によって実行が停止する 218 検体と，繰り返しによって 5 分以上の実行時間を要する 68 検体を抽出して解析した．その結果，それぞれ 161 検体と 38 検体において，新たな挙動が観測されることを確認した．スクリプト 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，いずれも悪性なものが見られた．以上より，実際の悪性スクリプトに対しても，提案手法による実行遅延・実行停止の回避で，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，経路強制実行の一種であるため，変数の一貫性に問題が生じうる．すなわち，実行経路の変更により，本来あるべき値と異なる値を変数が保持しうる．これに対し，影響を受ける変数をテイント解析で特定し，その変数に関連したコードのみを切り出して高速に実行する手法 [1] が提案されている．我々の過去の研究 [5] では，スクリプト向けのテイント解析を実現しており，これを適用可能だ

表 2　検出・判定・構築の精度の実験結果
Table 2　Experimental result on detection/identification/construction accuracy

エンジン	VPC	デコーダ・ディスパッチャ	条件分岐フラグ	コードキャッシュ	VM 命令の判定			CFG・CG 構築精度	実行遅延・停止の回避
					条件分岐	コール	リターン		
Lua	✓	✓	✓	✓	11/11	1/1	3/3	3.0	✓
VBScript (ReactOS)	✓	✓	✓	✓	2/2	4/4	1/1	5.0	✓
VBScript	✓	✓	✓	✓	4	1	1	3.0	✓

と考えられる．また，経路強制実行では，たとえ変数の一貫性に一定の問題を抱えた状態でも，可能な範囲で対応しつつ実行を継続することが重要である，という考え方が見られる [2]．たとえば，実行が継続しさえすれば，定数を用いた API 呼び出しなどは正しく観測しうる．これは，マルウェア解析においては重要な情報となる．以上から，提案手法によって変数の一貫性に問題が生じたとしても，解析者に対して一定の価値を提供できると考えられる．

5.2　他のスクリプト解析への応用

CFG・CG は，実行遅延・実行停止の回避のみならず，様々なプログラム解析に用いられる．そのため，未知のバイトコードに対して，スクリプトエンジンの実行状態と対応した CFG・CG が構築できることは，実行遅延・実行停止の回避のみならず，悪性スクリプトの動的解析に貢献することが期待できる．たとえば，実行時の動的な CFG や CG の比較により，悪性スクリプトの挙動の類似性を抽出する研究などに応用可能だと考えられる．

6.　関連研究

6.1　実行遅延・実行停止の回避の研究

2.2 節で紹介した HASTEN [1] および J-Force [2] のほかに，たとえば，IgnorEx [3] は，例外を無視する機構を持つ実行環境を実装することにより，実行停止を回避している．HASTEN および IgnorEx はバイナリを，J-Force は JavaScript のみを対象にしており，多様なスクリプトに対しての汎用的な実現を目指す我々とは，目的が異なる．

6.2　スクリプトエンジンの機能拡張の研究

Chef [11] は，バイナリ向けのシンボリック実行エンジンを用いて，スクリプトエンジンにシンボリック実行機能を付与する研究である．また，我々の過去の研究 [4] [6] [5] では，スクリプトエンジンを解析し，API トレース機能およびマルチパス実行機能，テイント解析機能をそれぞれ付与している．これらはいずれも実行遅延や実行停止の問題に着目したものではなく，本研究とは目的が異なる．

7.　結論

本研究では，悪性スクリプトの解析中に発生する実行遅延・実行停止の解決のために，当該箇所を検出してスキップする機能を，スクリプトエンジンに自動的に付与する手法を提案した．提案手法は，検出とスキップに要する CFG・CG を，未知のバイトコードに対しても構築するために，VM のアーキテクチャと ISA を，テストスクリプトを用いた動的解析で明らかにする．実験を通して，提案手法がこれらを正しく解析でき，実行遅延・実行停止を回避する機能を現実的な時間内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，実行遅延・実行停止の回避機能を有さない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] Kolbitsch, C., Kirda, E. and Kruegel, C.: The Power of Procrastination: Detection and Mitigation of Execution-Stalling Malicious Code, *CCS '11*, pp. 285–296.
[2] 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*, pp. 897–906.
[3] 大山恵弘，小久保博崇：例外を発生させるマルウェアのための動的解析手法，*CSS2019*, pp. 953–960.
[4] 碓井利宣，古川和祈，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マルチパス実行機能の自動付与手法，*CSS2019*, pp. 961–968.
[5] 碓井利宣，幾世知範，川古谷裕平，岩村　誠，三好　潤：スクリプト実行環境に対するテイント解析機能の自動付与手法，*CSS2020*, pp. 932–939.
[6] Usui, T., Otsuki, Y., Kawakoya, Y., Iwamura, M., Miyoshi, J. and Matsuura, K.: My Script Engines Know What You Did in the Dark: Converting Engines into Script API Tracers, *ACSAC '19*, p. 466–477.
[7] Sreedhar, V. C., Gao, G. R. and Lee, Y.-F.: Identifying Loops Using DJ Graphs, *ACM Trans. Program. Lang. Syst.*, Vol. 18, No. 6, p. 649–658 (1996).
[8] Hex-Rays: IDA, `https://hex-rays.com/ida-pro/`.
[9] 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, *PLDI '05*, pp. 190–200.
[10] Project, R.: ReactOS, `https://reactos.org/`.
[11] Bucur, S., Kinder, J. and Candea, G.: Prototyping symbolic execution engines for interpreted languages, *ASPLOS '14*, pp. 239–254.