Computer Security Symposium 2018
22 - 25 October 2018

# スクリプト実行環境に対する解析機能の自動付与手法

碓井 利宣[1]　大月 勇人[1]　川古谷 裕平[1]　岩村 誠[1]　三好 潤[1]

**概要**：マルスパムやファイルレスマルウェアなど，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は一般に難読化が施されており，コードの静的解析で挙動の詳細を知るのは困難であるため，動的解析が主に用いられる．しかしながら，動的解析においても，独自の実行基盤を用いた解析では，実スクリプトエンジンとの実装の差異が課題となり，システム Application Programming Interface (API) の監視による解析では，スクリプトとの間に生じるセマンティックギャップの課題が残る．そこで，本研究では，スクリプトエンジンのバイナリに対して，実行中のスクリプトを解析する機構を自動的に付与する手法を提案する．スクリプト固有の言語要素を考慮した監視により，スクリプトとの間に生じるギャップを解消する．あらかじめ用意したテストスクリプトを，スクリプトエンジンを監視しながら実行させることで，その内部挙動を解析する．それに基づき，解析用のコードを挿入するポイントと，ログ出力するメモリを参照するポイントを特定する．これにより，任意のスクリプトエンジンに対する解析機能の付与を可能とし，悪性スクリプト解析ツールに応用する．

**キーワード**：悪性スクリプト，動的解析，機能拡張

# Automatic Enhancement of Script Engines by Appending Behavior Analysis Capabilities

Toshinori Usui[1]　Yuto Otsuki[1]　Yuhei Kawakoya[1]　Makoto Iwamura[1]　Jun Miyoshi[1]

***Abstract:***
Malicious scripts have been a crucial attack vector in the recent attacks such as malware spams (malspams) and fileless malware. Since malicious scripts are generally obfuscated, statically analyzing them is difficult due to reflections. Therefore, dynamic analysis, which is not affected by the obfuscation, is used for malicious script analysis. However, despite its wide adoption, some problems remain unsolved in dynamic analysis. For emulator-based approach, implementation differences between real script engines and the emulators cause an incorrect execution problem. For system level hook approach, semantic gaps occur between executed scripts and observed system Application Programming Interfaces (APIs). In this paper, for narrowing down the semantic gaps, we propose a method for automatically enhancing script engines by appending behavior analysis capabilities to their binaries. We reduces the semantic gaps by considering language elements that are specific to observing scripts. Our method conducts dynamic analysis on script engines using predefined test scripts. Through the analysis, we mine the knowledge of script engine internals that are required to append behavior analysis capabilities. This enables to add analysis functionalities to arbitrary script engines. Experimental results showed that we can apply this method for building malicious script analysis tools.

***Keywords:*** malicious script, 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，多様な攻撃の形態が生じるに伴い，悪性スクリプトを用いた攻撃が拡大している．近年の悪性スクリプトには，単純なダウンローダやドロッパのみならず，端末情報の収集や C&C サーバとの通信，他プロセスへの感染などの複雑な挙動を

[1] NTT セキュアプラットフォーム研究所
NTT Secure Platform Laboratories

持ったものも存在する．また，悪性 Web サイトによる攻撃でも，脆弱性の悪用のために，悪性スクリプトは継続的に利用されている．こうした攻撃に対策を講じるためには，挙動を正確に把握する必要があり，悪性スクリプトを解析する技術が希求される．

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難しい．特に，コードの一部を外部から動的に取得する場合は，実行しなければ解析できない．したがって，悪性スクリプトを実行し，その振る舞いを監視する動的解析が主に用いられる．

悪性スクリプトのコードを監視するには，その実行をフックしてログを出力する方式が一般的である．悪性スクリプトをフックするために，大きく 3 つの方式が存在する．スクリプトレベルフック，システムレベルフック，スクリプトエンジンレベルフックである．これらはいずれも実際に用いられている手法ではあるが，それぞれが課題を抱えている．スクリプトレベルフックは，スクリプト言語の提供する機能を用いてスクリプトにフックを挿入することで挙動を監視する方式である．特定の言語仕様を持ったスクリプト言語に対してのみ実現が可能であるため，汎用性に乏しい．

システムレベルフックは，システムの Application Programming Interface (API) の監視により挙動を解析する方式である．スクリプト言語に依存せずに実現できるため汎用性はあるものの，監視対象のスクリプトに対して監視箇所がシステム API と隔たりがあるため，セマンティックギャップが生じる．これらに対して，スクリプトエンジン内で挙動を監視するスクリプトエンジンレベルフックは，特定の言語仕様に依存せず，監視対象のスクリプトとも隔たりがないためセマンティックギャップも生じない．一方で，スクリプトエンジン内のどの部分にどのようなフックを挿入すれば望む情報が得られるかが前提知識を必要とし，これをいかに獲得するかが課題となっている．

この課題を解決するため，本研究では，スクリプトエンジンのバイナリを解析し，得られた情報に基づいてスクリプトの解析機能を自動的に付与する手法を提案する．解析により，フックを施して解析用のコードを挿入する箇所（フックポイントと呼ぶ）と，解析用のコードによってログ出力するためのメモリ監視箇所（タップポイントと呼ぶ）を自動抽出する．提案手法では，あらかじめ用意した，特定の言語要素のみを呼び出すテストスクリプトを，システム API トレースとブランチトレースを取得しながら解析する．そして，それらの実行トレースをバックトレースによる分析手法と，差分実行解析による分析手法で分析することにより，言語要素に対応したフックポイントおよびタップポイントを取得する．これらのフックポイントおよびタップポイントを監視してログ出力する機能をスクリプトエンジンに付与することで，悪性スクリプトを効率的に解析するツールを生成する．なお，言語要素とは，スクリプトエンジンが提供する機能の呼び出しの単位であり，たとえば，ビルトイン関数や VBScript のステートメント，PowerShell のコマンドレットなどを指す．

提案手法を実装し，実験を行った．その結果，VBScript と PowerShell の言語要素について，スクリプトのセマンティクスを保ったログを出力できるフックポイントおよびタップポイントを検出できることを確認した．また，その検出は，一つの言語要素あたり数十秒程度の短時間で実現可能なことも確認できた．さらに，実際の攻撃に用いられた悪性スクリプトの解析に適用し，その有効性を確認できた．本研究により，今まで解析ツールの構成が困難であったスクリプト言語に対しても，解析ツールを生成し，防御を実現可能とな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を解析することによりスクリプトへの解析機能を付与する手法を初めて提案した．
- 実験を通して，提案手法によってフックポイントおよびタップポイントを検出できることを確認した．
- 本手法によって作成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．

## 2. スクリプト解析基盤の構成

### 2.1 スクリプト解析基盤の要件

まず，スクリプト解析基盤の構成方式が満たすべき 3 つの要件を明らかにする．

**(1) 言語仕様からの独立性**: 攻撃者は，多種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．したがって，スクリプト言語の仕様によらずに汎用的に適用できる構成方式が求められる．

**(2) スクリプトのセマンティクスの保存性**: スクリプトの解析に際しては，出力されるログがスクリプトのセマンティクスを失うほど，解析者の得られる情報は減少していく．そのため，スクリプトのセマンティクスを保ち，解析結果に反映できる構成方式が必要となる．

**(3) バイナリへの適用可能性**: スクリプト解析基盤の構成に際し，スクリプトエンジンがプロプライエタリソフトウェアの場合は，ソースコードが得られない．しかし，攻撃者はそうしたスクリプト言語も用いるため，スクリプトエンジンバイナリのみを用いて構成可能な方式が望ましい．

ここで，解析基盤が，どのような解析ログを出力すればよいかについても議論する．前述の要件 (2) の観点から，スクリプトのセマンティクスを持ったログが希求される．すなわち，スクリプトが用いた言語要素とその引数を復元できるログを出力することが望ましい．たとえ

ば，スクリプトの難読化を解除して実行すると CreateObject(“WScript.Shell”) が実行される場合に，解析機能によるログには，CreateObject が実行され，WScript.Shell が引数に渡されたことが記録される状態が理想的である．本研究では，こうしたログの取得を目指す．

### 2.2 構成方式と課題

スクリプトの解析基盤は一般に，スクリプトの実行をフックし，解析用のコードを挿入することで実現される．ここでは，フックをどこに施すかによって構成法を分類する．図 1 は，スクリプト解析基盤の構成法のフックポイントでの分類を示す図である．以降では，図中のスクリプトレベルフック，システムレベルフック，スクリプトエンジンレベルフックを説明し，それぞれの抱える課題を挙げる．

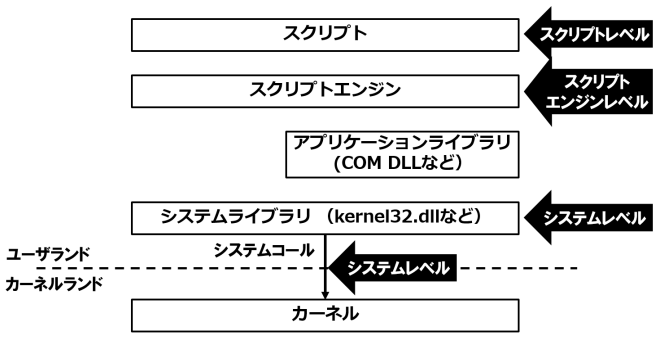


**図 1** スクリプト解析基盤構成法のフックポイントでの分類
**Fig. 1** Hookpoint-based categorization of script analysis system construction.

#### 2.2.1 スクリプトレベルフック

**構成方式**: スクリプトレベルフックは，スクリプトに対して直接フックを施す方法である．悪性スクリプトは一般に難読化されているため，解析者に有用な情報を得られるフックポイントをスクリプト内から発見するのは容易ではない．したがって，フックは特定の言語要素をオーバーライドすることで実現される．ソースコード 1 は，JavaScript においてスクリプトレベルフックを実現するコード片の例である．ここでは，eval 関数のオーバーライドでフックを施し（2 行目），引数をログ出力する解析用コードを挿入している（3 行目）．

**課題**: スクリプトレベルフックの課題として，実装の汎用性が低いことが挙げられる．この方式は言語要素のオーバーライドを要するため，それを許容する言語仕様を持った特定のスクリプト言語に対してしか実現できない．したがって，2.1 節の言語仕様からの独立性の要件を満たさない．

#### 2.2.2 システムレベルフック

**構成方式**: システムレベルフックは，システム API やシステムコールにフックを施す方法である．その上で，スクリプトエンジンのプロセスを監視しながら，スクリプトを実行させることで解析する．

**ソースコード 1** スクリプトレベルフックの例

```
1 original_eval = eval;
2 eval = function(input_code) {
3     console.log('[eval] code: ' + input_code);
4     original_eval(input_code); }
```

**課題**: システムレベルフックの課題として，解析対象のスクリプトとフックポイントとの間に隔たりがあるため，セマンティックギャップが生じることが挙げられる．

セマンティックギャップによって引き起こされる問題として，雪崩効果 [1] とセマンティクスの喪失がある．雪崩効果は，監視対象と監視箇所の間に隔たりとして層が存在するとき，監視の際に大量のノイズが観測されてしまう現象である．Ralf らの研究 [1] では，雪崩効果の層として Component Object Model (COM) オブジェクトを挙げているが，我々の研究では，スクリプトエンジンの層がある場合にも，同様に雪崩効果がみられた．また，セマンティクスの喪失では，解析者の得られる情報が減少する．たとえば，スクリプトのセマンティクスでは Document.Cookie.Set という言語要素であるものが，システム API では WriteFile として観測されるとすると，Cookie を操作しているというセマンティクスが失われる．これらの課題により，システムレベルフックは 2.1 節のスクリプトのセマンティクスの保存性の要件を満たさない．

#### 2.2.3 スクリプトエンジンレベルフック

**構成方式**: スクリプトエンジンレベルフックは，スクリプトエンジン内の特定の機能にフックを施す方法である．

**課題**: スクリプトエンジンレベルフックの課題として，実現が容易でないことが挙げられる．Antimalware Scan Interface (AMSI)[2] のように，スクリプトエンジンが解析用の機能を提供している場合は容易に実現できるが，限られた例に過ぎず，一般には，プログラム中の適切なフックの挿入位置を探す必要がある．オープンソースのスクリプトエンジンについては，ソースコードの解析によってフックの追加が可能であるものの，ソースが得られるスクリプト言語に限られ，一定の工数も要する．さらに，プロプライエタリのスクリプトエンジンについては，リバースエンジニアリングの必要があり，その自動化は確立されていない．また，人手で実施するには熟練したリバースエンジニアと多大な工数が必要となり，現実的でない．したがって，2.1 節のバイナリへの適用可能性の要件を満たさない．

### 2.3 アプローチ

前節で述べた通り，スクリプトレベルフックおよびシステムレベルフックは，2.1 節の要件を満たせず，それを満たす改良も原理的に困難である．一方で，スクリプトエンジンレベルフックの課題であるバイナリへの適用可能性は，

スクリプトエンジンの自動的なリバースエンジニアリングが可能となれば，解決できる．そこで，本研究では，スクリプトエンジンバイナリを解析し，フックに必要な情報を自動的に取得することで，スクリプトエンジンレベルフックをバイナリに適用可能とする．

スクリプトエンジンの解析に際して，対象のスクリプト言語の仕様に関する知識は保有していることを前提とする．これは，解析時に入力するスクリプトを作成する必要があるためである．一方で，スクリプトエンジンの内部実装に関する知識は前提としない．したがって，スクリプトエンジンに対して事前に何らかのリバースエンジニアリングをする必要はない．

## 3. 提案手法

### 3.1 概要

図 2 は，手法の全体像を示した図である．提案手法は，スクリプトエンジンの解析に基づいて，フックポイントとタップポイントを自動的に検出するものである．提案手法では，テストスクリプトと呼ばれる，スクリプトを動的解析する際に入力されるスクリプトを用いる．これは，あらかじめ手動で作成されるものである．まず，このテストスクリプトを実行させながらスクリプトエンジンを監視し，実行トレースを取得する．この実行トレースに対して，2 つのフックポイント検出手法を順に適用し，フックポイント候補を抽出する．そして，タップポイント検出手法を用いて，フックポイントを確定するとともに，タップポイントを抽出する．得られたフックポイントおよびタップポイントにしたがって，スクリプトエンジンにフックを施し，解析機能を付与されたスクリプトエンジンを出力する．

なお，本研究では，フックポイントはスクリプトエンジンの内部関数を単位とし，フックは内部関数の先頭に挿入されるものとする．また，タップポイントは内部関数の引数であるとする．

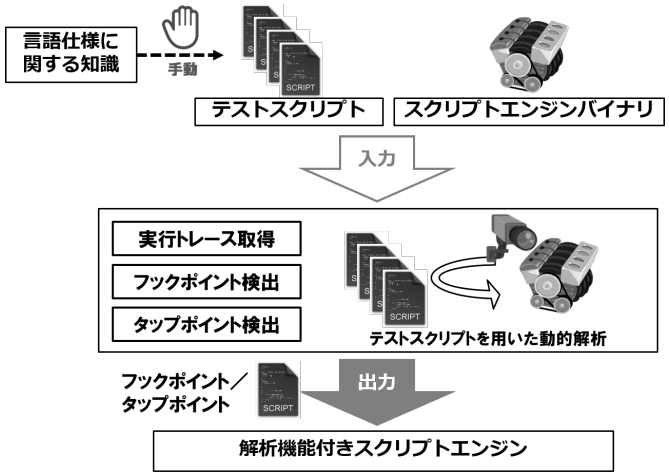


**図 2** 提案手法の全体像
**Fig. 2** Overview of the proposing method.

**ソースコード 2** テストスクリプトの例

```
1 Dim objShell
2 Set objShell = CreateObject("WScript.Shell")
```

### 3.2 テストスクリプトの作成

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．このテストスクリプトは，スクリプトエンジンにおいて解析したい言語要素を指定する働きを果たす．したがって，解析対象の言語要素のみを含んだスクリプトを用いる．たとえば，スクリプトエンジン内の CreateObject という言語要素に関連した処理を解析してフックポイントを獲得したい場合は，ソースコード 2 のように，CreateObject のみを呼び出すテストスクリプトを作成する．フックポイント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，3.4.1 項および 3.4.2 項にて述べる．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．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，2.3 節で述べた前提とは矛盾しない．

### 3.3 実行トレースの取得

提案手法による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．本手法での実行トレースは，API トレースとブランチトレースで構成される．API トレースでは，実行の際に，呼び出されたシステム API とその引数を記録していく．API フックによってログ出力用のコードを挿入し，システム API の呼び出しごとにそれを実行させて，記録していく．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．これは，命令フックによるログ出力用コードの挿入で実現する．

### 3.4 フックポイント検出

#### 3.4.1 バックトレースによる検出

ここでは，提案手法のフックポイント検出手法の一つである，バックトレースに基づく検出を説明する．提案手法におけるフックポイントの検出はどちらも，3.3 節で取得したトレースログを分析することで実現される．このバックトレースに基づく検出手法は，システムとのインタラクションを要する言語要素のフックポイントの検出に用いる．すなわち，システム API の呼び出しを伴う言語要素に対して有効である．

本手法の概念図を図 3 に示す．この手法は，システムとのインタラクションが必要な言語要素は，スクリプトエンジン内のその言語要素に関わるコード領域から，必要なシステム API を呼び出すであろう，という仮定に基づいている．したがって，この検出手法は，システム API の呼び出

しからスクリプトエンジンのコード領域まで遡って辿っていくことで，言語要素に関わるコード部分を特定し，そこに存在する，言語要素に関わるフックポイントを検出する．

この検出手法のためのテストスクリプトでは，言語要素の引数のうちで任意に設定可能なものに，特徴的な値を設定する．これにより，スクリプトで渡された引数が，スクリプトエンジンによって最終的にどのシステム API に渡されたのかを判別する．

このテストスクリプトを実行させて得られた実行トレースを分析する．まず，スクリプト内で実行している言語要素の引数が，システム API の引数として現れている呼び出しを発見する．そして，そこからブランチトレースに基づいて呼び出し元を辿り，遡っていく．スクリプトエンジンまで遡ったら，そのスクリプトエンジン内の呼び出し箇所から $N$ 回分の分岐を，フックポイント候補として検出する．ここで，遡りがスクリプトエンジンに辿り着いた点をフックポイントとするのではなく，そこから $N$ 回遡っているのは，スタブ等を経由して呼び出される場合を考慮しているためである．

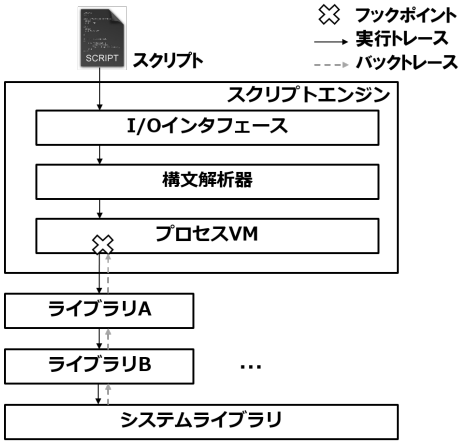


**図 3** バックトレースによるフックポイント検出の概念図
**Fig. 3** Concept of hookpoint detection by backtrace.

### 3.4.2 差分実行解析による検出

ここでは，もう一つのフックポイント検出手法である，差分実行解析に基づく検出を説明する．差分実行解析とは，条件を変更しつつ複数の実行トレースを取得し，その差分を分析することで動的解析する方法である．

この検出手法は，システム API の呼び出しを伴わない言語要素に対して有効である．たとえば，Eval 関数はスクリプトエンジンの中で閉じた言語要素であり，システム API を呼び出す必要がない．しかし，手動解析の際には，解析者に有用な情報を持つため，興味の対象となる．こうした言語要素に対しては，システム API に依存するバックトレースによる検出手法は効果をなさないが，この差分実行解析による手法では，検出が可能となる．

本手法の概念図を図 4 に示す．この手法で，特定の言語要素のみを 1 回呼び出した場合の実行トレースと，複数回実行した場合の実行トレースでは，その言語要素に関わるコードの実行トレースのみが差分として現れるであろう，という仮定に基づいている．

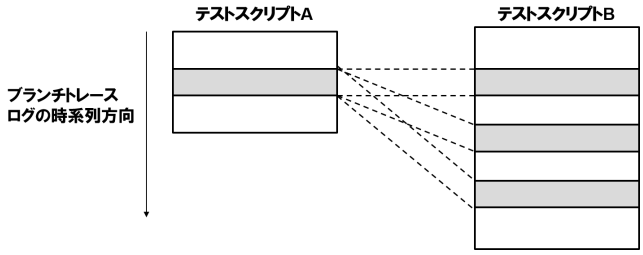


**図 4** 差分実行解析によるフックポイント検出の概念図
**Fig. 4** Concept of hookpoint detection by differential execution analysis.

この検出手法では，複数のテストスクリプトを用いる．それぞれに対する実行トレースを比較することで，検出する．テストスクリプトには，解析対象の言語要素を 1 回のみ呼び出すスクリプトと，複数回呼び出すスクリプトとを用意する．これにより，実行トレースを比較した際に，差分に現れる言語要素に関わるトレース部分を捉える．

この各々のテストスクリプトを実行し，得られた実行トレースを比較して分析する．そのために，ブランチトレースから共通部分を取得する．ただし，この共通部分は，言語要素を 1 回のみ呼び出しているスクリプトのトレースには 1 回のみ，$N$ 回呼び出しているスクリプトのトレースには $N$ 回のみ現れるブランチトレースの部分集合であるとする．これを抽出するため，2 つ以上の系列から共通した部分を抽出するローカルアラインメントの検出手法である，Smith-Waterman アルゴリズムを用いた．ここで，Smith-Waterman アルゴリズムには先ほど述べたような回数の制約の考慮が存在しないため，改良を施した．

Smith-Waterman アルゴリズムは, 動的計画法 (Dynamic Programming: DP) に基づく系列アライメントアルゴリズムであり，2 つ以上の系列から，相同性の高い部分系列を抽出できる．このアルゴリズムでは，DP 表と呼ばれる表を用いる．DP 表では，1 つの系列を表頭に，もう 1 つの系列を表側に配置し，各セルにマッチスコアを記入する．$x$ 軸方向の添字を $i$，$y$ 軸方向の添字を $j$ として，下記の式 (1) に基づいて，セル $(i, j)$ のスコア $F(i, j)$ を算出していく．

$$F(i,j) = max \begin{cases} 0 \\ F(i-1, j-1) + s(i,j) \\ F(i-1, j) + d \\ F(i, j-1) + d \end{cases} \quad (1)$$

ただし本研究では，

$$s(i,j) = \begin{cases} 2 & (match) \\ -1 & (unmatch) \end{cases} \quad (2)$$

$$d = -2 \quad (3)$$

であるとする．この DP 表の作成までは，通常の Smith-Waterman アルゴリズムと同一である．ここで，説明のために DP 表を図 5 に示す．図 5 中の A, B, C は合わさって図 4 のグレー部分の 1 つ分を構成し，S は実行トレースの最初に現れる白部分，E は最後に現れる白部分である．M はグレー部分の間に現れる白部分である．本来，これらの各要素は複数の分岐のログ行で構成されるが，ここでは簡略化のため圧縮している．ここから，スコアが最大のセル（図 5 の 8 のセル）からバックトラックしていくことで，最も相同性の高い部分文字列（図 5 内の 8 のセルを含む破線部分，SABC）を発見して終了するが，提案手法では，ここからさらに探索を行う．アルゴリズムで抽出された部分文字列を除いた部分のうち，同じ行（図 5 の点線部分）に対して，あらためて相同性の高い部分文字列を抽出する．この処理をテストスクリプトでの呼び出し回数分繰り返し，抽出された部分文字列（図 5 の 3 つの破線部分）同士のうち，各々の部分文字列の類似度がいずれも閾値以上であれば，その部分を構成する分岐のログ行がフックポイント候補であるとして検出する．さもなければ，次にスコアの高いセルについて調べていく．

| | | S | A | B | C | M | A | B | C | M | A | B | C | M | E |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 0 | 4 | 2 | 0 | 0 | 2 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 |
| B | 0 | 0 | 2 | 6 | 4 | 2 | 0 | 4 | 2 | 0 | 0 | 4 | 2 | 0 | 0 |
| C | 0 | 0 | 0 | 4 | 8 | 6 | 4 | 2 | 6 | 4 | 2 | 2 | 6 | 4 | 2 |
| E | 0 | 0 | 0 | 0 | 6 | 7 | 5 | 3 | 4 | 5 | 0 | 1 | 4 | 5 | 3 |

**図 5** Smith-Waterman アルゴリズムの改変
**Fig. 5** Modified Smith-Waterman algorithm.

### 3.5 タップポイント検出

タップポイントの検出は，以下の 2 点の役割を担う．1 点目は，フックポイント検出で得られたフックポイント候補の中から，最終的なフックポイントを確定することである．2 点目は，フック時にログ出力するメモリ位置を特定することである．

タップポイントは，フックポイントの引数を探索していくことで実施する．そのために，ここまでで得られたフックポイント候補にフックを施して，実行トレースを再度取得する．引数は，呼出規約に沿って参照していくことで取得できる．このとき，各引数の型情報は得られないため，さらなる探索にはヒューリスティクスが必要となる．本研究では，次のようなヒューリスティクスを用いる．まず，引数がポインタとして参照できない場合は，それが値であるとみる．参照できる場合は，ポインタとして参照する．値としてみる場合には，様々な型としてみていく．この結果，テストスクリプトで用いている引数がフックポイント候補で観測されれば，そのフックポイントを確定し，引数が得られた点をタップポイントとする．

この探索方法は，型情報が得られれば改善できるため，TIE[3] をはじめとする型情報をリバースエンジニアリングする研究を適用すれば，より正確な探索が期待できる．

### 3.6 解析機能の付与

解析機能の付与には，ここまでで得られたフックポイントおよびタップポイントを用いる．フックポイントでスクリプトエンジンをフックし，タップポイントのメモリをログ出力する解析用コードを挿入することで実現される．

## 4. 評価

### 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のスクリプトエンジンを用いた．ただし，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，プロプライエタリソフトウェアを対象とする場合と同等の状況としている．具体的には，ReactOS プロジェクト [4] で実装されている VBScript と，オープンソース版の PowerShell[5] を用いた．これらは，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．また，バックトレースでスクリプトエンジンに到達した際の遡り回数 $N$ は 10 としている．

**表 1** 実験環境
**Table 1** Experimental environment

| | |
|---|---|
| OS | Windows 7 |
| CPU | Intel Core i7-6600U CPU @ 2.60GHz |
| メモリ | 2GB |
| VBScript | vbscript.dll (ReactOS 0.4.9) |
| PowerShell | PowerShell 6.0.3 |

### 4.2 検出精度の評価

フックポイント検出およびタップポイント検出の精度を評価するため，提案手法でこれらのポイントを検出する実

験を実施した．実験では，悪性スクリプトが高頻度で用いる言語要素のフックポイントおよびタップポイントの検出を試みた．VBScript は，言語の設計思想として，OS との直接的なインタラクションはできず，必要に応じて COM オブジェクトを用いるようになっている．したがって，実際の攻撃では COM オブジェクトの操作に関わる言語要素と Eval や Execute などの，文字列を動的に評価する言語要素が用いられる．PowerShell は，コマンドレットと呼ばれる言語要素を用いて，OS とのインタラクションも含めた様々な動作を実現できる．それらの中から，オブジェクトの生成，ファイル操作，プロセス実行，インターネットアクセス，リフレクションなど，攻撃に用いられるコマンドレットを選択した．

実験の結果を表 2 に示す．表中のオリジナルの列は，ブランチトレースによって得られた，絞り込みを行っていない全てのユニークな分岐の数である．フック検出の列は，フックポイント検出によって絞り込んだ後のフックポイント候補の数である．ログ可否の列は，タップポイント検出によって最終的に一点に特定されたフックポイントおよびタップポイントから，2.1 節の要件を満たすログが正しく得られたか否かを表す．

VBScript では，CreateObject の実行によって，スクリプトエンジン内で CLSIDFromProgID や CoGetClassObject などの COM に関わるシステム API が呼び出されていた．CreateObject の引数は CLSIDFromProgID に渡されており，そこからバックトレースによって，フックポイントが検出できていた．結果の検証のため，ソースコード内でのフックポイントの位置を確認したところ，`create_object` という，正にオブジェクトを作成する内部関数にフックがなされていた．また，COM メソッドの Invoke については，システム API の呼び出しはないものの，差分実行解析によって検出できていた．同じくソースコードを確認したところ，`disp_call` という内部関数にフックがなされており，これも正に，IDispatch::Invoke の COM インタフェースの呼び出しに至る部分であった．また，ログ可否の列からも分かるように，CreateObject についても Invoke についても，正しくログ出力できるタップポイントを検出できていた．一方，Eval および Execute については，ReactOS の VBScript では，関数はあるものの中身が実装されていなかったため，検出できなかった．

PowerShell についても表 2 の通り，VBScript と同様に，正しいログ出力が可能なフックポイントおよびタップポイントを検出できていた．VBScript と PowerShell の違いとして，スクリプトエンジンとカーネルの間に，Microsoft .NET Framework の共通言語基盤が存在する点が挙げられる．したがって，この結果から，共通言語基盤などの追加の層が存在する場合であっても，解析機能の付与には問題がないことが分かる．

**表 2** フックポイントおよびタップポイントの検出結果
**Table 2** Result of hookpoint detection and tappoint detection

| スクリプト | 言語要素 | オリジナル | フック検出 | ログ可否 |
|---|---|---|---|---|
| VBScript | CreateObject | 89213 | 10 | ○ |
| | Invoke (COM) | 128511 | 43 | ○ |
| | Eval | - | - | × |
| | Execute | - | - | × |
| PowerShell | New-Object | 210852 | 10 | ○ |
| | Import-Module (DLL) | 185192 | 10 | ○ |
| | New-Item (File) | 198327 | 10 | ○ |
| | Set-Content (File) | 200822 | 10 | ○ |
| | Start-Process | 152841 | 10 | ○ |
| | Invoke-WebRequest | 315380 | 10 | ○ |
| | Invoke-Expression | 271054 | 82 | ○ |

### 4.3 実行速度の評価

フックポイント検出およびタップポイント検出の速度を評価するため，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 6 に示す．なお，提案手法の手順のうち，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．

実行トレースの取得およびタップ検出は，Intel Pin による実行とログ出力を伴うため，テストスクリプトに応じて 10 秒前後の時間がかかっている．バックトレースは，ごく少しの実行トレースログの探索で済むため，ほとんど時間を要さない．一方，差分実行解析は，一定の時間を要している．Smith-Waterman アルゴリズムを解析すると，一方の系列の長さを $M$，もう一方の系列の長さを $N$ とすると，計算量は $O(MN)$ であるため，実行トレースログが長くなればなるほど，実行時間は増大する．全体として，一つの言語要素に対するフックポイントおよびタップポイントの検出の実行時間は，高々数十秒となっている．VBScript の全言語要素は 100 件を下回る程度であり，その中でも悪性スクリプト解析の際に興味のあるものは限られる．このことから，提案手法は実用に十分な速度でスクリプトエンジンに解析機能を付与できると考えられる．

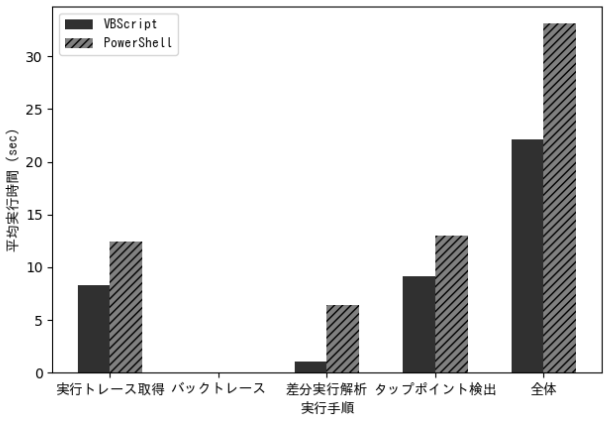


**図 6** 提案手法の実行時間
**Fig. 6** Execution duration of the proposing method.

### 4.4 実検体の解析ログの評価

マルウェア共有サービスの VirusTotal から，2017 年 1 月から 7 月にかけてアップロードされた PowerShell のスクリプト 205 検体を収集し，実験で構成したスクリプト解析ツールを用いて解析した．その結果，スクリプトの実行している言語要素と引数を抽出することができた．引数として得られた 105 件の URL を VirusTotal で調査したところ，いずれも悪性 (positive > 1) と判定された．また，複数のスクリプトによって共通して出力されたファイルストリームのハッシュを調査したところ，Dridex ランサムウェアであった．以上のことから，実際の悪性スクリプトに対しても，有用な情報を抽出できることを確認した．

## 5. 議論

提案手法の制限として，フックポイントおよびタップポイントの検出が困難なケースを 2 点挙げる．1 点めは，解析対象の言語要素に，任意に設定できる引数が存在しない場合である．フックポイントの確定およびタップポイントの検出には引数の照合を用いているため，これが利用できない場合には検出がうまくいかない．2 点目は，解析対象の言語要素が，システム API を呼び出さず，中身の実装もごく少量の場合である．このとき，バックトレースは適用できず，差分実行解析も差分が十分に出ないために働かない可能性がある．しかしながら，これらを満たしつつ複雑な処理やシステムに影響を与える処理を実装することはできないため，悪性スクリプト解析で解析者の興味のある言語要素にはなりづらいと考えられる．

## 6. 関連研究

### 6.1 スクリプト解析基盤の研究

スクリプト解析基盤の構成は数多く研究されてきた．たとえば，スクリプトレベルフックであれば，JS-Walker[6] などが存在する．また，スクリプトエンジンレベルフックであれば，Flash のオープンソース実装の改変によりフックする FlashDetect[7] などがある．システムレベルフックであれば，スクリプトのみを主眼としないが，API Chaser[8] などを応用することになる．しかしながら，これらはいずれも 2.2 節の課題を解決できていない．

### 6.2 セマンティックギャップの削減の研究

セマンティックギャップの解消を目指した研究は，仮想マシンモニタのゲスト OS とホスト OS の間でのギャップに焦点をあてたものが見られる．

Virtuoso[9] は，仮想マシン (VM) 内でのツールの実行トレースを取得し，その実行トレースから必要な部分のみをスライシングで抽出することで，VM 外からそのツールと同等の出力を得るイントロスペクションツールを構成する．事前に実行トレースを取得し，その解析によって必要な情報を取得するアプローチは，本研究と類似している．

Tappan Zee (North) Bridge[10] は，メモリアクセスを監視しながらソフトウェアに様々な値を入力し，その値がメモリ上のどの位置に現れるかを特定することで，イントロスペクションに有効なタップポイントを発見する．

## 7. 結論

本研究では，悪性スクリプトの動的解析におけるセマンティックギャップの課題に着目し，その解決のために，スクリプトエンジンのバイナリに解析機能を自動的に付与する手法を提案した．提案手法では，テストスクリプトを用いた 2 つの動的解析の手法に基づいて，スクリプトエンジン中の適切なフックポイントとタップポイントを検出する．実験を通して，提案手法がスクリプトエンジンに解析機能を付与できることを確認し，実際の攻撃に用いられた悪性スクリプトから情報を抽出できることを示した．より高機能な解析機能の付与が今後の課題である．

### 参考文献


[1] Hund, R.: The beast within Evading dynamic malware analysis using Microsoft COM, Blackhat USA briefings 2016.

[2] Microsoft: Antimalware Scan Interface, `https://docs.microsoft.com/en-us/windows/desktop/amsi/antimalware-scan-interface-portal`. (accessed: 2018-08-16).

[3] Lee, J., Avgerinos, T. and Brumley, D.: TIE: Principled Reverse Engineering of Types in Binary Programs, pp. 1–18 (2011).

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

[5] Team, P.: PowerShell, `https://github.com/powershell`. (accessed: 2018-08-16).

[6] 柴田龍平，羽田大樹，横山恵一：Js-Walker: JavaScript API hooking を用いた解析妨害 JavaScript コードのアナリスト向け解析フレームワーク，コンピュータセキュリティシンポジウム 2016 論文集, 情報処理学会, pp. 951–957 (2016).

[7] Van Overveldt, T., Kruegel, C. and Vigna, G.: FlashDetect: ActionScript 3 Malware Detection, *International workshop on recent advances in intrusion detection*, Springer, pp. 274–293 (2012).

[8] Kawakoya, Y., Iwamura, M., Shioji, E. and Hariu, T.: Api chaser: Anti-analysis resistant malware analyzer, *International Workshop on Recent Advances in Intrusion Detection*, Springer, pp. 123–143 (2013).

[9] Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J. and Lee, W.: Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection, *Proceedings of the IEEE Symposium on Security and Privacy (SP) 2011*, IEEE, pp. 297–312 (2011).

[10] Dolan-Gavitt, B., Leek, T., Hodosh, J. and Lee, W.: Tappan Zee (North) Bridge: Mining Memory Accesses for Introspection, *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, ACM, pp. 839–850 (2013).