Computer Security Symposium 2020
26 - 29 October 2020

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

碓井 利宣[1,2,a)]　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を詳細に解析するには，制御フローの解析のみならず，データフローの解析も求められる．このデータフローの解析には，テイント解析がよく利用されるが，既存のスクリプト向けのテイント解析手法はスクリプトエンジンごとに設計，実装する必要がある．
この問題を解決するため，本研究では，バイナリ（機械語のプログラム）向けのテイント解析機能を，スクリプトにも適用可能にすることで，言語やエンジンに非依存でテイント解析機能を自動付与する手法を提案する．まず，バイナリのデータ型とスクリプトのデータ型の間に生じるセマンティックギャップがこの実現を妨げる問題だと実験的に示す．そして，こうした型のセマンティックギャップによって起こる伝播漏れを検出し，テイントの強制伝播によってそれを解消することで，テイント解析機能を実現する．Python と VBScript のスクリプトエンジンに対して提案手法を適用し，テイント解析を実現できることを確認した．さらに，それを用いて悪性スクリプトを解析し，データフローを追跡可能になったことも確認した．

キーワード：悪性スクリプト，テイント解析，動的解析，機能拡張

Automatically Armoring Script Engine with Taint Analysis Capability

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

Abstract: Data flow analysis is an essential technique for understanding the detailed behavior of malicious scripts. For such analysis, taint analysis is widely used in existing studies. However, taint analysis techniques of those studies are dependent on the respective design and implementation of each script engine.
In this paper, we propose a method that automatically appends taint analysis capability to vanilla script engines by leveraging taint analysis capability designed for native binaries for scripts. We first conduct experiments for clarifying that the problem preventing this method is caused by the semantic gaps between data types of binaries and ones of scripts. Our method detects such gaps and bridges them by generating forced propagation rules. It enables script engines to obtain taint analysis capabilities. We built taint analysis tools for Python and VBScript with our method and confirmed that they could effectively analyze the data flow of real-world malicious scripts.

Keywords: malicious script, taint analysis, 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，静的解析によってコードの詳細な挙動を明らかにする

[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
[a)] toshinori.usui.rt@hco.ntt.co.jp

のは困難である．そのため，悪性スクリプトの解析には動的解析が多く用いられる．我々は過去の研究で，スクリプト API のトレース [1] とスクリプトのマルチパス実行 [2] の 2 つの動的解析技術を実現した．これらは，スクリプトの制御フローに着目してその挙動を解析する技術である．一方，さらなる詳細な挙動の解析のためには，制御フローの解析のみならず，データフローの解析も求められる．悪性スクリプトが扱うデータの流れを精緻に追跡できれば，解析者はそのデータの属性（たとえば，復号鍵であるか，C&C のコマンドであるかなど）を把握できる．これにより，悪性スクリプトの挙動をより詳細に明らかにできる．

こうしたデータの追跡を実現する手法として，テイント解析が存在する．テイント解析とは，データにテイントタグ（以降，タグと呼ぶ）という属性情報を付与し，それをデータの移動にあわせて伝播させていくことで，データフローを解析する技術である．既存のスクリプト向けのテイント解析 [3][4][5] では，スクリプトの言語やエンジンごとにテイント解析機能を設計・実装している．悪性スクリプトが多様な言語で作成され得ることを考慮すると，あらゆるスクリプト言語のエンジンに対して，個別にテイント解析の機能を実装するのは，現実的でない．

そこで，本研究では，バイナリ（機械語のプログラム）向けに実装されたテイント解析機能を用いて，スクリプトエンジンのバイナリ上で動作するスクリプトのデータフローを解析する手法を提案する．これにより，スクリプトエンジンがバイナリであれば，悪性スクリプトの言語に非依存での解析を可能とする．この提案におけるチャレンジは，テイントの伝播漏れへの対応である．スクリプトはスクリプトエンジンの仮想機械上で動作するため，スクリプトとスクリプトエンジンのバイナリの間には，セマンティックギャップがある．そのため，バイナリでのテイント伝播がスクリプトに対しても十分な伝播であるとは限らず，伝播漏れが起こり得る．そこで，まず，スクリプトにおける伝播漏れの原因を調査した．その結果，スクリプトとバイナリの用いるデータ型（以降，型と呼ぶ）にセマンティックギャップがあり，スクリプトエンジン内での型変換が，スクリプトに固有の伝播漏れを起こすことを明らかにした．

これに対し，提案手法では，スクリプトエンジンを動的解析することで，こうした伝播漏れを解決し，スクリプト向けのテイント解析機能を自動的に実現する．テストスクリプトと呼ぶ解析用のスクリプトを用いて，型の変換を担う関数（型変換関数と呼ぶ）を検出する．この関数の入出力に着目したテイント解析を実施し，型変換の過程で伝播漏れを起こすものを明らかにする．そして，タグを入力から出力に強制的に伝播させる特殊な伝播ルールを生成する．この強制伝播ルールをバイナリ向けのテイント解析ツールに適用することで，伝播漏れを解消し，スクリプト向けのテイント解析機能を実現する．

ソースコード 1　解析対象の悪性スクリプトの一例 (Python)

```
1 current_time = time.time()
2 if current_time <= 1597910400:
3       sys.exit()
4 do_malicious()
```

提案手法に基づくプロトタイプを実装し，Python 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，伝播漏れを起こす型変換関数を検出し，強制伝播ルールを生成して，それらを解消できることを確認した．また，強制伝播ルールは，スクリプトエンジンによって，数十秒から数百秒程度で実現可能なことも確認できた．さらに，提案手法に基づいてテイント解析ツールを生成し，解析妨害に用いられる情報を検出するアプリケーションを実装して，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，解析妨害に関わる情報を正しく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，多様な悪性スクリプトに対して有効なテイント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．

- 調査を通して，型のセマンティックギャップにより，スクリプトに固有の伝播漏れが引き起こされることを明らかにした．
- スクリプトエンジンを解析し，型変換関数に強制伝播ルールを適用することで，スクリプト向けのテイント解析を自動で実現する手法を初めて提案した．
- 実験を通して，提案手法によって型変換関数に対して正しい強制伝播ルールを生成できることを確認した．また，それによってテイント解析機能を付与したスクリプトエンジンを用いて，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. スクリプトとテイント解析

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は，条件分岐による解析妨害を具備しており，システムの時刻情報を取得 (1 行目) して，特定の時刻以前であれば，実行を終了する (2,3 行目)．そのため，解析環境が特定の時刻以前の場合には，悪性な挙動 (4 行目) を観測できない．したがって，解析妨害を無効化して動的解析を進めるためには，何の情報に基づいて条件分岐しているかを知る必要がある．本研究では，こうした解析妨害のための分岐条件に至るデータの流れを追跡し，何の情報に基づいているかを把握可能にすることを目指す．

2.2 スクリプトエンジンのデータ型

本研究の扱う問題を明らかにする準備として，前節で挙げたソースコード 1 を参照しながら，スクリプトエンジンの用いる型の特徴を示す．スクリプトエンジンの多くは，バイナリとして提供される．スクリプトで扱うデータは，スクリプトエンジンの内部では一般に，オペレーティングシステムが提供するプリミティブ型ではなく，独自に拡張された参照型として保持される．これは，多倍長整数などのより複雑な型の提供や，ガベージコレクションに用いる参照カウントなどの情報の付加のためである．

たとえば，ソースコード 1 の time.time メソッドが返す浮動少数点数の値は，CPython のスクリプトエンジン内では，PyFloat 型として保持される．このように，スクリプト内で用いられる任意のデータは，浮動小数点数に関わらず，整数であれば PyLong 型，文字列であれば PyString 型というように，独自の参照型を用いて保持されている．こうした型の扱いは Python に限らず，多くのスクリプト言語で共通している．こうしたバイナリでの型とスクリプトでの型の間で発生する意味論上の隔たりを，本研究では型のセマンティックギャップと呼ぶ．

2.3 スクリプトエンジンでのテイントタグの伝播漏れ

ソースコード 1 を参照して，スクリプトエンジン内で伝播漏れが発生する原因を明らかにしていく．まず，スクリプトエンジンをテイント解析ツール上で実行し，条件分岐による解析妨害を検出できるか，実験した．

条件分岐による解析妨害は，おもにシステム情報に基づいて実施される [6]．したがって，その検出には，システム情報を取得するシステムコールの出力値をタグの付与点（以降，ソースと呼ぶ），分岐条件をタグの確認点（以降，シンクと呼ぶ）としたテイント解析を一般に用いる [7]．分岐条件に関わる変数にタグが付いていれば，何のシステム情報に基づいた解析妨害かが分かる．ソースコード 1 では，時刻情報を用いているため，それを取得するシステムコールである clock_gettime の返り値にタグを付与し，分岐条件まで正しく伝播するかを確認した．理論上は，これによって解析妨害の分岐条件が時刻情報に基づくことを検出できる．実際，ソースコード 1 と同等の処理を，スクリプトでなくバイナリで実装したものでは，正しく検出できた．しかしながら，スクリプトであるソースコード 1 では，タグが分岐条件まで伝播しておらず，正しく検出できなかった．

そこで，スクリプトエンジンを解析し，この伝播漏れの原因を調査した．その結果，clock_gettime システムコールの返り値が current_time 変数に格納されるまでに，複数回の型変換があり，その過程で伝播漏れが発生することが分かった．具体的にはまず，clock_gettime システムコールの返り値の timespec 構造体が pytime_fromtimespec 関数によって _PyTime_t に変換される．そして，それ

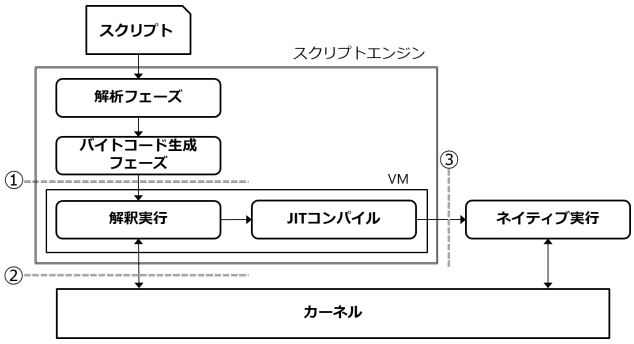

図 1 スクリプトエンジンの構造と伝播漏れの発生箇所

Fig. 1 Script engine mechanism and points that cause under-tainting

が _PyTime_AsSecondsDouble 関数によって double に変換された後，さらに PyFloat_FromDouble 関数によって，time.time メソッドの返り値である PyFloatObject 構造体に変換される．このうち，_PyTime_AsSecondsDouble 関数において，伝播が完全に途切れてしまう．

さらなる調査の結果，複数の型変換において上記の伝播漏れが発生することが分かった．図 1 に，スクリプトエンジンの構造と，伝播漏れの発生箇所を示す．この図は，スクリプトの入力を起点とした，スクリプトエンジンの処理の流れを示している．図中の破線は，伝播漏れが発生した箇所であり，スクリプトとネイティブのコンテキストの境界である．破線①では，入力されたスクリプトがバイトコードに変換される過程で，スクリプトを構成する文字列からスクリプトの型への変換により，伝播漏れが発生していた．このため，悪性スクリプトの初期化変数をソースとしたテイント解析ができない．また，破線②では，スクリプトエンジンによるシステムインタラクションの過程で，スクリプトの型からシステムの API に合わせた型への変換により，伝播漏れが発生していた．このため，システムコールの入出力をソース・シンクとしたテイント解析ができない．破線③では，バイトコードがバイナリに Just-In-Time (JIT) コンパイルされる過程で，スクリプトの型からネイティブの型への変換により，伝播漏れが発生していた．このため，JIT コンパイルで生成されたコードへタグを追跡できない．また，これらの境界に加えて，VM 命令の実行においても，型変換による伝播漏れが発生していた．

以上より，型のセマンティックギャップが，バイナリにはない多様な型変換を誘発し，伝播漏れを起こすおもな原因となっている．そこで，本研究では，型のセマンティックギャップによる伝播漏れを解消し，バイナリ向けのテイント解析をスクリプトにも適用可能にすることを目指す．

2.4 スクリプトのテイント解析のアプローチ

前節で挙げた型変換関数による伝播漏れを解消し，テイント解析を実現するためには，(1) 型変換関数の検出，(2)

型変換関数が伝播漏れを起こすか否かの判定，(3) 伝播漏れを解消する強制伝播ルールの生成，の 3 つが必要となる．これらが実現できていれば，伝播漏れを起こす型変換関数が呼び出された場合に，その変換元のタグを変換先に強制的に伝播させることで，伝播漏れを解消できる．これにより，バイナリ向けのテイント解析の機能を，スクリプトに対しても提供できるようになる．そのため本研究では，先に挙げた 3 つの要素を実現し，得られた強制伝播ルールに基づいて，スクリプトエンジンに強制伝播を実現し，伝播漏れを解消する，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，以下のような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．

- 難読化されていないバイナリである
- 独自の参照型による変数管理をする

また，スクリプトエンジンの内部実装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

提案手法ではまず，スクリプトエンジンの型変換関数を，動的解析によって検出する．この動的解析には，我々が過去の研究 [2] 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．次に，その関数の入出力がどの変数かを明らかにする．それに基づいて，入力をソース，出力をシンクとしてテイント解析をし，シンクにタグがなければ，伝播漏れとして検出する．そして，入力のタグを出力へ強制的に伝播させる解析用コードを追加することで，その伝播漏れを解消し，テイント解析機能を付与する．

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が必要となる．提案手法は，実行トレース取得，型変換関数検出，入出力検出，テイント伝播漏れ検出，強制伝播ルール生成，テイント解析機能付与の 6 つのステップからなる．以降で，各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は，分岐命令の実行やメモリ読み書きの回数に着目し，異なる回数で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 2 に示す．これは，型変換関数を内部的に呼ぶ関数を，規定の回数呼び出している．ここでは，ソースコード 1 で挙げた time.time メソッドを例に用いた．差分実行解析では，この呼び出し回数を変更した複数のスクリプトを用意

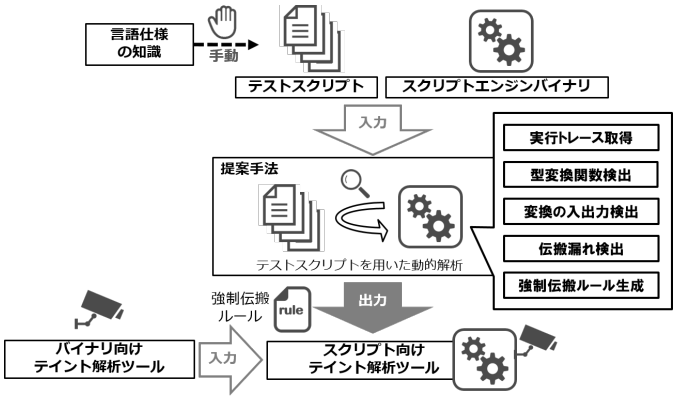

図 2 提案手法の概要図
Fig. 2 Overview of proposing method

ソースコード 2 テストスクリプトの一例 (Python)

```
1 import time
2 t1 = time.time()
3 t2 = time.time()
4 t3 = time.time()
```

し，それらの実行トレースの差分をみることで，求める情報を得る．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，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 節で述べた仮定とは矛盾しない．

3.3 実行トレース取得

提案手法での型変換関数を検出す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．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．

3.4 型変換関数検出

このステップでは，ブランチトレースを解析し，型変換関数を検出する．この検出には，スクリプトの差分実行解析 [1] を用いる．図 3 に，差分実行解析による型変換関数の検出の概要を示す．図のように，1 回のみ型変換をするスクリプトと，N 回型変換するスクリプトを用意し，ブランチトレースの差分をみると，型変換に対応した分岐の系列はそれぞれ 1 回と N 回出現する．したがって，この差分

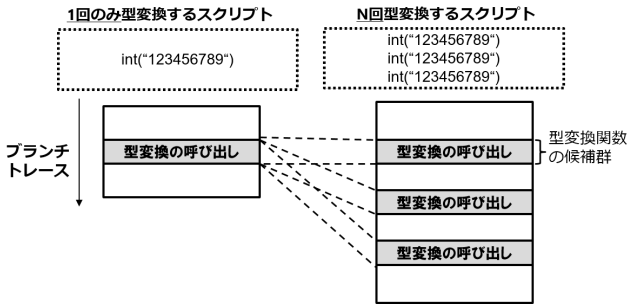

図 3　差分実行解析による型変換関数の検出

Fig. 3　Type conversion function detection with differential execution analysis

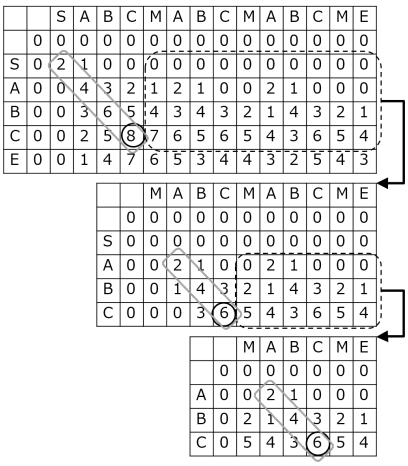

図 4　改変版の Smith-Waterman アルゴリズム

Fig. 4　Our modified Smith-Waterman algorithm

を捉えれば，型変換関数を検出できる．提案手法では，独自の改変を加えた Smith-Waterman アルゴリズムによって捉える．原版は，2 つの系列から類似性の高い部分系列を検出するアルゴリズムである．これに，一方に 1 回，一方に N 回というように，出現回数を加味する改変を施した．

改変版を図 4 に示す．まず，表側に 1 回の、表頭に N 回のテストスクリプトのブランチトレースの系列を配置する．図中の ABC が，図 3 の灰色部分にあたる求める部分系列であり，それ以外は，白色部分にあたる．動的計画法の表を作成してセルにスコアを埋め，最大のセルからバックトラックをするまでは，原版と同じである．改良版ではさらに，残余部分（図中の破線部分）に対して，同一の処理を $N-1$ 回，再帰的に繰り返し，全部で N 個の系列を抽出する．抽出された系列の類似度が閾値以上であれば，それを求める系列として出力する．こうした操作により，図 3 の型変換関数に関わる分岐の系列が検出できる．

3.5　入出力検出

このステップでは，スクリプトエンジンのバイナリを解析し，型変換関数の入出力の関係を検出する．ここで，型変換関数の入力とは，変換元の値であり，出力とは変換先の値である．型変換関数において，引数は複数ある場合があり，そのどれが入力かは自明でない．出力も同じく，返り値で渡す場合と，引数に渡されたポインタで渡す場合とがあり，やはり自明でない．したがって，今後の解析のステップを進めるためには，まず入出力の渡され方を検出する必要がある．そのため，まず引数および返り値の型や構造体の定義を推論する．2.5 節の通り，スクリプトエンジンは難読化されていないことを仮定しているため，データの依存関係の静的解析 [8] によって実現できる．

次に，入力と出力の関係にある変数を検出する．これは，データの間の関係性を明らかにする作業であるため，本来はテイントを用いたい．しかし，そもそも伝播漏れをする関数を探しているという前提から，テイントは使えない．したがって，ここではテイントの代替として，値の照合によってデータ間の関係性を明らかにする．型変換関数の入力と出力の間には，型は異なるが，何らかの依存関係を保持しているという特徴が見られる．たとえば，整数を 10 進文字列に変換する型変換関数では，入力が 1234567890 である時，出力は “1234567890” となる．これは，型が異なるため 16 進数での表現は異なるが，既知の型への変換をすれば一致する．したがって，こうした依存関係を明らかにするために，型の変換をかけて値の類似性をみれば，入力と出力の関係にある変数を検出できる．これにより，入力となる引数と型，出力となる引数または返り値と型を得る．

3.6　テイント伝播漏れ検出

このステップでは，前節で得られた入出力の関係に基づいてテイント解析をし，型変換関数によって引き起こされる伝播漏れを検出する．前節で検出した入力の変数をソースとし，タグを付与する．また，出力の変数をシンクとし，タグを確認する．タグの付与および確認は，検出された入出力の型に基づいて値のサイズを決定して実施する．テイント解析の結果，シンクにタグが付いていれば，その型変換関数は伝播漏れを起こさない．一方，タグが付いていなければ，伝播漏れを引き起こすとして検出する．

3.7　強制伝播ルール生成

強制伝播とは，通常命令レベルで定められる伝播ルールでは伝播漏れが発生してしまう箇所に，命令レベルによらない特殊な伝播ルール（強制伝播ルールと呼ぶ）を適用することで，正しく伝播させる手法である．そのためには，強制伝播のソースとなる入力とシンクとなる出力に対して，各々のメモリ上の位置と，何の型であるかを与える必要がある．提案手法では，入力および出力は型変換関数の引数または返り値であるため，そのメモリ上の位置は，型変換関数のオフセットと，入力および出力が何番目の引数または返り値であるかで示せる．したがって，強制伝播ルールは，これらの情報によって構成される．こうした情報は，

型変換関数検出および入出力検出で得られているため，それらを強制伝播ルールとしてまとめ，出力する．

3.8 テイント解析機能付与

前節で得られた強制伝播ルールを用いて，スクリプト向けのテイント解析を実現する．まず，スクリプトエンジンを，バイナリ向けのテイント解析ツール上で実行する．そして，伝播漏れを起こす型変換関数をフックし，呼び出しの前後のコールバックを用いて，強制伝播ルールを適用する．これにより，スクリプトに固有の伝播漏れを解消でき，スクリプト向けのテイント解析が実現される．

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 Intel Pin [9] を，入出力の検出に IDA Pro [10] を，テイント解析には AngoraFuzzer 版の libdft64 [11] をそれぞれ用いた．AngoraFuzzer 版では，テイント伝播に対応した命令の拡充など，独自の実装が追加されているため，本研究ではこちらを用いる．

実装したプロトタイプを，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，CPython と，ReactOS プロジェクト [12] で実装されている VBScript を用いた．前者は，オープンソースのスクリプトエンジンであり一定の成熟した型の実装を持つため，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．

表 1 実験環境
Table 1 Experimental environment

CPU	Intel Core i7-6600U CPU @ 2.60GHz
メモリ	2GB
OS	Ubuntu 20.04 LTS, Windows 7 32-bit
Python	CPython 3.8.3
VBScript	vbscript.dll (ReactOS 0.4.11)
VBScript	vbscript.dll 7.01.1048

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，型変換関数およびテイント伝播漏れの検出と，伝播ルールの生成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ついて，（正しく検出できた数）/（検出を試みた数）を記述している．なお，検出の正しさは，ソースコードとバイナリの手動解析で確認した．本研究のゴールはスクリプト向けのテイント解析機能の付与であるため，テイント解析機能付与の列が全て満たされることが一つの目標である．この列は，テストスクリプトの対象の関数における伝播漏れが解消され，想定されるタグの伝播が見られた数を記述した．なお，テストスクリプトは，事前の調査で，想定される伝播が見られなかった関数を対象に作成した．

表 2 の通り，提案手法によって，いずれのステップでも情報を検出できていた．また，それに基づいて強制伝播ルールを適用することで，伝播漏れを解消できていた．提案手法が型変換関数として検出した関数は，ソースコードにおいて確かに型変換関数であることが確認できた．Python ではたとえば，PyLong_FromString をはじめとする {（Python の）型名 }_From{ 型名 } という名前の API 関数や，long_to_decimal_string や mystrtoul といった内部関数が検出されていた．また，2.3 節で挙げた_PyTime_AsSecondsDouble 関数も型変換関数として検出されていた．ReactOS の VBScript では，Global_{ 型名 } という内部関数がおもに検出されていた．

Python は PyLong 型や PyFloat 型など，型ごとに独自の構造体を定義しており，いずれも入出力検出の過程で構造体を解析し，入出力の関係を明らかに出来ていた．一方，ReactOS では Windows OS での標準である VARIANT 型を用いており，既知の構造体の型として検出されていた．こうした情報により，伝播漏れの箇所を検出でき，強制伝播ルールを生成できていた．また，それを用いて，図 1 の全ての破線で，伝播漏れを解消できていた．以上の結果から，提案手法による型変換関数に関する情報の検出が，テイント解析の付与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 5 に示す．なお，提案手法の各ステップのうち，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．

図より，実行トレースの取得に最も時間を要している．これは，取得に際して Intel Pin の VM 上で実行している

表 2　実験結果
Table 2　Experimental result

言語	テスト件数	型変換関数検出	入出力検出	伝播漏れ検出	強制伝播ルール生成	テイント解析機能付与
Python	14	14/14	14/14	14/14	14/14	14/14
VBScript (ReactOS)	9	9/9	9/9	9/9	9/9	9/9

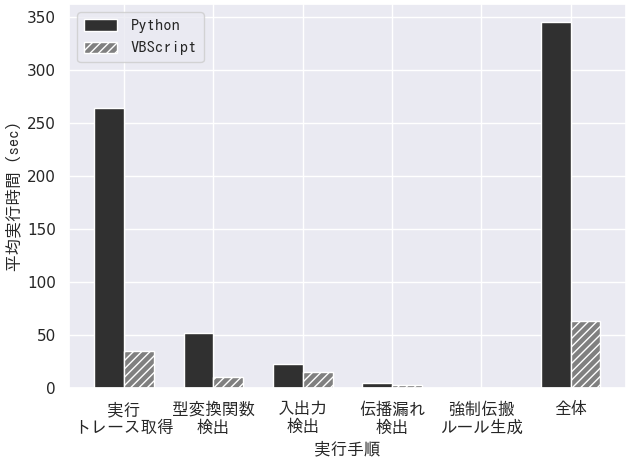

図 5　提案手法の実行時間
Fig. 5　Execution duration of proposing method

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．特に，複雑な実装とメモリモデルを持つ CPython で時間を要した．型変換関数の検出にも，一定の時間を要している．これは，2 つの実行トレースのサイズを M, N としたとき，アルゴリズムの計算量は $O(MN)$ と，比較的大きいためである．また，入出力検出の所要時間は，おもに IDA Pro による型の静的解析による．伝播漏れ検出は，テイント解析の一回の実行に要する時間のみである．さらに，強制伝播ルールの生成は，ここまでで得られた情報をまとめるのみのため，ほぼ時間を要していない．全体として，強制伝播ルールは，数十秒から数分程度で得られている．型変換関数の数は一般に，型の数に依存して限りがあるため，現実的な時間内でスクリプト向けのテイント解析を実現できることが期待される．

4.4　実検体に対する解析性能の評価

悪性スクリプトの解析妨害の条件を検出するため，提案手法で付与されたテイント解析機能を用いて，それを実現するツールを実装した．ソースをシステムコールの出力に，シンクを分岐条件に関わる命令のオペランドに設定した．まず，ソースコード 1 の元となった悪性スクリプトを解析した．その結果，clock_gettime システムコールの出力値のタグが分岐条件に伝播し，確かに時刻情報に基づく解析妨害であることが検出できた．さらに，過去の我々の研究 [1] の実験で，解析妨害すると判明した 102 検体を同様に解析した．その結果，76 検体が何の情報に基づいて解析妨害をするか検出できた．検出できなかったものは，テストスクリプトの拡充で対応可能とみられた．以上により，型のセマンティックギャップによる伝播漏れが解消され，有効なテイント解析が実現できていることが確認できた．

5.　議論

5.1　制約

提案手法は，入出力検出に値の照合を用いているため，型変換の過程で値が大きく変わる場合に，検出に失敗する可能性がある．しかし，型変換関数は型の変換のみを担うのが一般的な設計であり，値が大きく変わるのは考えにくい．実際，実験の範囲ではこうした問題は発生しなかった．

また，本研究では，スクリプトのレベルでのソースとシンクの設定は実現していない．たとえば，ソースやシンクにスクリプト API や VM 命令を設定したい場合は，さらにスクリプトエンジンを解析し，該当箇所を特定する必要がある．これには，我々の過去の研究 [1][2] が活用できる．

5.2　テストスクリプトの作成

本研究では，型変換をするテストスクリプトを可能な限り網羅的に作成する必要がある．これは，公式ドキュメントの関数やメソッドを網羅することで，実現できると考えられる．実用的には，悪性スクリプトが高頻度に用いるものを優先的に作成することで，限られた数のテストスクリプトで，有効なテイント解析ツールを効率的に構成できる．

5.3　テイント伝播の粒度

提案手法で生成される強制伝播ルールは，伝播元と伝播先のみを定めたものである．それらの間でタグが，どのような粒度でどのように伝播されるべきかは自明でなく，アプリケーションに応じて定められるべきである．たとえば，char 型の 10 進数の文字列を整数型に変換する関数では，変換元の”1234567890”は 10 バイトであるが，変換先の 1234567890 は 4 バイトで格納できる．ここで，変換元の各バイトに異なるタグが付与されていた場合，どのように伝播させるべきかは自明でない．こうしたタグの伝播のさせ方として，すべてのタグを一つに結合する方法や，新たにタグを定義する方法が考え得る．ただし，こうした伝播は，テイント解析の粒度を下げる可能性がある．なお，これはスクリプトに固有の問題ではなく，バイナリ向けのテイント解析においても，発生し得る．

5.4 テイント解析の応用

本研究では，提案手法によって実現したテイント解析機能を，悪性スクリプトで解析妨害のための条件の検出に応用したが，実際にはそれのみによらない．たとえば，情報漏洩の検出や，C&C 通信のコマンドの検出，復号鍵の検出など，一般的なマルウェア対策と同じく多様な応用先が期待できる．また，悪性スクリプトの解析以外に，Web アプリケーション脆弱性の検出などにも応用可能である．

6. 関連研究

6.1 スクリプトのテイント解析の研究

スクリプトに対するテイント解析の構成は数多く研究されてきた．例えば，Phan [4] では PHP の Zend の VM に，Vogt らの研究 [3] では JavaScript の VM に，それぞれテイント解析機能を実装している．また，Ichnaea [5] では，抽象機械を用いることで，JavaScript の多様なエンジンでテイント解析を可能にしている．これらはいずれも個別のスクリプト言語やエンジンに対してテイント解析を実現したものであり，本研究とは目的が異なる．

6.2 テイント伝播ルール生成の研究

例えば，TaintInduce [13] は，命令の入出力の組み合わせを収集し，未知のプロセッサに対して，テイント伝播ルールを生成する．また，DTA++ [14] は，暗黙的なフローによる伝播漏れを検出し，強制伝播ルールを生成する．しかし，スクリプトエンジンに着目した研究はなく，アプローチも我々の研究とは異なる．

6.3 スクリプトエンジンの機能拡張の研究

Chef [15] は，手動で改造を施したスクリプトエンジンを，バイナリ向けのシンボリック実行器の上で実行することで，スクリプト向けのシンボリック実行を実現する研究である．スクリプトエンジンに対してバイナリ向けの解析技術を適用することで，スクリプト向けの解析技術を構成する点は，本研究と類似している．また，我々の過去の研究 [1][2] 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をそれぞれ付与している．

これらはいずれも制御フローのみに着目しており，データフローの解析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，悪性スクリプトに対するテイント解析を実現するため，スクリプトエンジンにテイント解析機能を自動的に付与する手法を提案した．そのために，バイナリ向けのテイント解析ツールをスクリプトエンジンに適用し，スクリプト向けのテイント解析ツールの構成を試みた．この際に伝播漏れが発生したため，その原因を調査した．その結果，型のセマンティックギャップが，スクリプトに固有の伝播漏れを引き起こすことを特定した．そこで，伝播漏れを起こす型変換関数を検出し，強制伝播ルールを生成して自動で伝播漏れを解消する手法を提案した．

実験を通して，提案手法がスクリプトエンジンにテイント解析機能を，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，提案手法により，有用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] 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 論文集，pp. 961–968.

[2] 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.

[3] Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C. and Vigna, G.: Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis., *NDSS '07*, p. 12.

[4] Monga, M., Paleari, R. and Passerini, E.: A hybrid analysis framework for detecting web application vulnerabilities, *IWSESS '09*, pp. 25–32.

[5] Karim, R., Tip, F., Sochurkova, A. and Sen, K.: Platform-Independent Dynamic Taint Analysis for JavaScript, *IEEE TSE* (2018).

[6] 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*, pp. 165–187.

[7] Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Song, D. and Yin, H.: Automatically Identifying Trigger-based Behavior in Malware, *Botnet Detection*, Springer, pp. 65–88 (2008).

[8] Maier, A., Gascon, H., Wressnegger, C. and Rieck, K.: TypeMiner: Recovering Types in Binary Programs using Machine Learning, *DIMVA '19*, pp. 288–308.

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

[11] Chen, P. and Chen, H.: Angora: Efficient Fuzzing by Principled Search, *S&P '18*, pp. 711–725.

[12] ReactOS Project: ReactOS, `https://www.reactos.org/` (accessed: 2020-08-19).

[13] Chua, Z. L., Wang, Y., Baluta, T., Saxena, P., Liang, Z. and Su, P.: One Engine To Serve'em All: Inferring Taint Rules Without Architectural Semantics., *NDSS '19*.

[14] Kang, M. G., McCamant, S., Poosankam, P. and Song, D.: DTA++: dynamic taint analysis with targeted control-flow propagation., *NDSS '11*.

[15] Bucur, S., Kinder, J. and Candea, G.: Prototyping Symbolic Execution Engines for Interpreted Languages, *ASPLOS '14*, pp. 239–254.