Vol.2019-DPS-178 No.18
Vol.2019-CSEC-84 No.18
2019/3/4

難読化JavaScriptコード解析支援システムの自動化の実現

上原 渓一郎[1]　山内 利宏[2,a)]

概要：難読化された JavaScript コードの解析を支援するため，上川らは難読化 JavaScript コード解析支援システムを提案した [1] [2]．しかし，この解析支援システムを利用した動的解析を行う際の操作は，解析者の手動作業を必要とし，迅速な解析作業の妨げとなり得る．そこで，我々は，解析支援システムの利用に必要な手動作業にかかる時間を削減するため，解析支援システムの自動化を実現した．解析支援システムを利用する際に必要となる手動作業を Web ブラウザ操作自動化ツールやシェルスクリプトを用いて自動化することで，解析時間の短縮が期待できる．また，この自動化の実現により，解析結果ログの可読性が低下する問題に対処するため，解析ログ変換手法を実現した．本稿では，解析支援システムの自動化および解析ログ変換手法について，実現するための考え方と実現方式を述べる．また，解析支援システムの自動化により短縮される解析時間の評価について述べる．

1. はじめに

JavaScript は Web ブラウザ上で動作し，動的な Web サイト構築の開発に用いられる．また，JavaScript は，Ajax 技術や HTML5 の登場により，単に動的な Web サイトを提供するだけでなく，高いユーザビリティを実現するための重要な手段として，多くの Web サイトにおいて広く利用されている．

一方で，JavaScript は，Web を介するサイバー攻撃においても利用される．Web ページに埋め込んだ JavaScript コードにより，ユーザを攻撃 Web サイトへ誘導したり，Web ブラウザや Web アプリケーションの脆弱性を突いたりする攻撃がある．

このようなサイバー攻撃に対処するためには，攻撃コードを解析して攻撃の概要や手法を明らかにするとともに，攻撃を防ぐための対策を講じる必要がある．しかし，サイバー攻撃に利用される JavaScript コードには，セキュリティソフトの検知回避やコードの解析妨害を目的として，コードの難読化が施されている場合がある．難読化は，攻撃コード中の文字列のエンコード処理やデータ構造の変換により，コードの可読性を低下させ，解析を妨害する．難読化されたコードは人間による静的解析が困難であり，解析にかかる時間が長くなってしまう．

難読化 JavaScript コードの解析を支援するため，上川らは文献 [1,2] において難読化 JavaScript コード解析支援システム（以降，解析支援システム）を提案した．この解析支援システムでは，ブラウザ API 操作の捕捉やコード変形手法による静的フックを行い，コード実行時情報を取得可能にする．

この解析支援システムでは, 解析支援システムへの HTML ファイルの読み込みや，Web ブラウザのコンソール上に表示される解析結果の表示といった処理に，手動での作業を必要とする．解析対象のファイルが 1 つの場合，これらの手動作業にかかる解析時間は数分程度である．しかし，解析対象のファイル数が多くなるにつれて，この手動作業にかかる時間が長くなり，解析時間が増大してしまう．

本研究では，この問題に対処するため，Web ブラウザ上の処理を自動化するツールである Selenium [3] を利用した解析支援システムの自動化手法を実現した．一方で，Selenium を利用し，解析ログを取得した場合，解析ログの可読性が低下する．これは，自動化した解析支援システムでは，オブジェクト形式の解析ログを JSON 形式で取得するためである．そこで，解析ログの可読性を低下させないために，解析ログ変換手法についても述べる．解析ログ変換手法は，自動化により可読性が低下した解析ログに対して変換処理を行い，オブジェクト形式に戻すことで可読性の低下を抑制し，難読化 JavaScript コードの解析を支援する．また，本稿では，自動化した解析支援システムの評価内容と結果についても述べる．

[1] 岡山大学工学部
[2] 岡山大学大学院自然科学研究科
[a)] yamauchi@cs.okayama-u.ac.jp

Vol.2019-DPS-178 No.18
Vol.2019-CSEC-84 No.18
2019/3/4

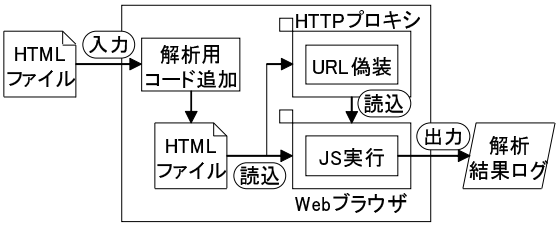

図 1　解析支援システムの全体図

2. 難読化 JavaScript コード解析支援システムの概要

2.1 実現方式

上川らが文献 [1,2] において提案した解析支援システムの概要を述べる．解析支援システムの全体図を図 1 に示す．解析支援システムは，Web ブラウザでの動的解析をベースとしたシステムである．解析者は，解析対象の JavaScript コードを含む HTML ファイルを解析支援システムに入力し，出力として解析結果のログを得る．解析支援システムに HTML ファイルが入力されると，その HTML ファイルに解析のための処理を行う JavaScript コードが自動で追加される．解析用コードを追加した HTML ファイルを Web ブラウザで読み込み，JavaScript コードを実行することで，Web ブラウザのコンソールに解析結果のログが出力される．解析支援システムは，以下に示す 3 つの手法を利用して解析を行う．

（手法 1）　Proxy 手法

解析支援システムでは，Proxy オブジェクトを利用することにより，ブラウザ API 操作の捕捉を実現する．具体的には，ブラウザ API の操作をブラウザ API（オブジェクト）に対する内部関数呼び出しと定義し，これらの内部関数を捕捉する．Proxy オブジェクトは，JavaScript のオブジェクトに対する基本的な操作を内部関数呼び出しレベルで捕捉することができる．

（手法 2）　変数の参照置き換え手法

（手法 1）の Proxy 手法では，置き換えが禁止されているオブジェクト（window や location）に対する操作を捕捉できない．変数の参照置き換え手法では，これらの手法の適用可能範囲を拡張し，置き換えが禁止されているオブジェクトに対する操作を捕捉可能とする．

（手法 3）　コード変形手法

JavaScript は，代入文による変数への代入や制御構文による実行制御を動的に捕捉する仕組みが存在しない．このため，例えば，変数に代入される値や条件分岐文による条件式の評価についての情報を実行時に JavaScript で得られない．そこで，解析対象 JavaScript コード中の実行時情報を得たい特定操作を行う式や文に対し，関数呼び出しを行うようにコードを静的に書き換える（静的フック）．このとき，呼び出す関数の引数として，実行時情報およびコード情報を渡す．呼び出すようにした関数が実行された際，それぞれの操作に関する実行時情報とコード情報をログとして出力する．これにより，JavaScript コード中の式や文について，実行時の情報を取得し，コード中の記述と紐付けることができる．

表 1　解析支援システム実行時の引数と適用手法の対応

引数	適用手法
a	Proxy 手法
b	参照置き換え手法
c	コード変形手法
@	全手法

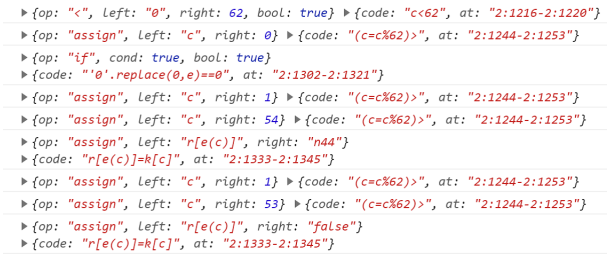

図 2　コンソールに出力された解析ログ

2.2 利用方法

解析支援システムは，Linux 上で実行できる．解析支援システムを利用するには，解析対象の JavaScript コードが埋め込まれた HTML ファイルを用意し，解析を行うスクリプト apply.pl を実行すればよい．以下に，具体的なターミナル上でのコマンド例を示す．

```
\$ ./analysis/apply.pl -abc < sample.html
  > sample-abc.html
```

この例では，解析対象ファイルである sample.html を解析支援システムに入力として与え，解析コード付き HTML ファイルが sample-abc.html として出力される．なお，コマンド中の引数 a，b，c の指定によって，2.1 節で述べた 3 つの手法のうち，どの手法を適用するかが決まる．また，全ての手法を適用する場合は，-@のように引数を指定することで，全手法を適用できる．引数と適用手法の対応を表 1 に示す．

この解析コード付き HTML ファイルを Web ブラウザに読み込むことで，Web ブラウザのコンソールに解析結果が出力される．図 2 に，コンソールに出力される解析結果の例を示す．

2.3 課題

上川らが提案した解析支援システムについて，現状の課題を述べる．

（課題 1）　手動作業にかかる解析時間

解析支援システムを利用して，JavaScript コードが埋

め込まれた HTML ファイルを解析する場合，解析者が手動ですべき作業がいくらかある．具体的に必要な作業は，HTML ファイルの解析支援システムへの読み込み，解析コード付き HTML ファイルの出力，および解析コード付き HTML ファイルの Web ブラウザへの読み込みである．さらに，Web ブラウザのコンソールへ出力された解析結果をログファイルとして保存する場合，その保存作業も手動で行う必要がある．
これらのコマンドを毎回手入力し，ファイルを Web ブラウザに読み込ませるといった作業は，解析対象のファイルが増えるにつれて，解析時間に対する割合として多くなる．これらにかかる時間を自動化により短縮し，得られた動的解析結果を用いて静的解析を行うことが重要である．

（課題 2）　ページ遷移によりコンソールが消失する問題
攻撃に利用される JavaScript コードは，ユーザを攻撃サイトへ誘導するため，リダイレクトを発生させる場合がある．解析支援システムにおいて，リダイレクトによってページ遷移が起こると，コンソールに出力された解析ログが消失するという問題がある．この問題に対処するには，手動でページ遷移を抑制するように攻撃コードを修正する必要がある．

（課題 3）　参照置き換え手法によりグローバル変数がグローバルでなくなる問題
2.1 節で述べた通り，参照置き換え手法の適用により，window 変数や location 変数が指している window オブジェクトや location オブジェクトに対する操作が捕捉できるようになる．しかし，この手法の適用により，コード中で定義されるグローバル関数がグローバルでない関数になり，プログラムがエラーで停止してしまう例があった．この問題については，今後対処方法を検討する必要がある．

以降では，上記の課題のうち，（課題 1）への対処について述べる．

3. 難読化 JavaScript コード解析支援システムの自動化

3.1 目的と要件

2.3 節で述べた解析支援システムの（課題 1）へ対処するため，以下の要件が提案システムに求められる．

（要件 1）　手動作業への対処
解析者が 1 つのスクリプトを実行するだけで，解析支援システムを自動実行可能とする．従来の解析支援システムにおいて手動で行っていた作業を自動化することにより，解析にかかる手動時間を短縮する．

（要件 2）　解析ログの同一性
提案システム実現以前の解析支援システムと同じ内容のログを出力可能とする．自動化により，解析ログに含まれる情報が欠落したり変更されたりしないようにする．

3.2 考え方

解析支援システムでは，解析時に，ファイルの読み込みやログの取得といった作業を手動で行う必要がある．その際，コマンド入力のようなターミナル上で実行できる作業は，シェルスクリプトを利用することにより，自動化を実現できる．しかし，Web ブラウザのコンソールに出力される情報を取得するといった作業は，ターミナル上で実行しないため，シェルスクリプトを利用した自動化は不可能である．これらの Web ブラウザ上での作業は，Web ブラウザ自動操作ツールを利用することで自動化を実現できる．

以上のことを踏まえて，3.1 節で述べた要件を満たし，解析支援システムによるログ出力までの作業を自動で実行するシステムを以下の方針に基づき実現する．

(1) 提案システムを利用する際，引数として解析対象ファイル名と適用手法を与えるだけで，解析ログを出力・保存する．これにより，従来手動で行っていた複数の作業が 1 つのスクリプトで実行可能になる．
(2) 提案システム実現以前の解析支援システムと同じ内容のログを出力可能とする．自動化により，解析ログに含まれる情報が欠落したり変更されたりしないようにする．
(3) Web ブラウザ上の処理の自動化には，Selenium を利用する．Selenium とは，Web ブラウザの自動操作機能を提供するツールのことである．Selenium は，Node.js 上で動作する JavaScript から利用できる．解析支援システムでは Node.js を利用しているため，Node.js を新たにインストールする必要が無く，Node.js で Selenium モジュールをインストールするだけでよい．
(4) 提案システムにより得た解析ログの変換手法は，（3）と同じく，Node.js 上で動作する JavaScript により実現する．さらに，解析ログの変換手法は，（3）の Selenium を用いた Web ブラウザ自動操作を行う JavaScript コード内に，関数として実現する．このような方針にした理由は，解析ログを保存してから別のプログラムを呼び出して変換処理を行うよりも，内部で関数を定義して呼び出し，変換処理を行った方が高速に変換できると考えたためである．

以上の対処により，従来の解析支援システムの手動作業を自動化でき，解析にかかる時間を短縮可能である．さらに，提案システムにより得られる解析ログの可読性を低下させないことが可能である．

3.3 提案システム

3.2 節で述べた考え方を踏まえて設計した提案システムの全体図を**図 3** に示す．提案システムは，シェルスクリ

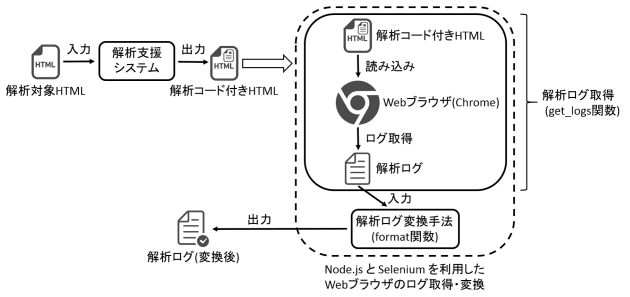

図 3 提案システムの処理の流れ

プト，Node.js，および Selenium を組み合わせることで，解析支援システムを自動化する．なお，提案システムは，Linux 仮想マシン上で動作する．解析者は，解析対象の JavaScript コードが埋め込まれた HTML ファイルのファイル名と適用手法を引数として提案システムを実行し，出力として解析ログを得る．

解析支援システムは，解析対象の JavaScript コードが埋め込まれた HTML ファイルに対して，Web ブラウザの API 操作の捕捉結果やコード中の変数代入の情報を Web ブラウザのコンソールに出力するように，解析用コードの挿入を行う．解析者は，この解析用コード付き HTML ファイルを Web ブラウザに読み込み，動的解析を行い，解析結果が Web ブラウザのコンソールに出力される．この部分の自動化については，Selenium を利用して Web ブラウザの処理の自動操作を行うことにより，自動化を行う．

しかし，Selenium を利用した自動化により得られる解析ログ中にオブジェクト形式のログが含まれる場合，3.4 節の（3）で後述する理由により，それらのログが JSON 文字列という文字列形式で出力される．JSON 文字列は文字列に対してエスケープ処理が施されているため，解析ログの可読性が低下してしまう．このため，自動化したシステムにより出力された解析ログは，3.5 節で述べる変換手法により，可読性が元に戻るように形式を変換し，出力する．

提案システムは，1 つのスクリプトを実行するだけで解析ログを得ることができ，これは（要件 1）を満たす．また，Selenium を用いてコンソールへの出力を自動化しても，取得できるログの情報は同じであることから，（要件 2）を満たす．これにより，3.1 節で述べた要件を全て満たすことができる．

3.4 解析支援システムの自動化手法

3.3 節で述べた提案システムの実現方式について述べる．提案システムは，シェルスクリプト，Node.js および Selenium を組み合わせることにより解析支援システムの自動化を実現する．

解析支援システムの利用に必要な作業のうち，自動化が必要な手動作業を以下に示す．

(1) 解析対象 HTML ファイルの解析支援システムへの読み込み
(2) 解析コード付き HTML ファイルの Web ブラウザへの読み込み
(3) Web ブラウザのコンソールに出力される解析ログの取得
(4) 解析ログの保存

以降では，この 4 つの作業の自動化の実現方式についてそれぞれ述べる．

(1) 解析対象 HTML ファイルの解析支援システムへの読み込み

2.2 節で述べた利用方法の通り，コマンドを実行するだけでよい．よって，単にシェルスクリプトを利用するだけで自動化が可能である．シェルスクリプト実行の際に必要な情報は，以下の 2 つである．

- 解析対象 HTML ファイルのファイル名
- 解析支援システムで利用する適用手法

(2) 解析コード付き HTML ファイルの Web ブラウザへの読み込み

（1）によって出力された解析コード付き HTML ファイルを自動で Web ブラウザに読み込ませる必要がある．Selenium を利用して Web ブラウザ上で特定のページやファイルを読み込むためには，`get` メソッドを利用すればよい．ファイル名が `file_path` の場合，以下のようなコードで読み込みを行う．

```
driver.get("file:///" + file_path);
```

このコードにより，Web ブラウザが起動し，`file_path` で指定された HTML ファイルが開く．

(3) Web ブラウザのコンソールに出力される解析ログの取得

（2）で実行する `driver.get()` によって読み込まれた HTML ファイルに含まれる JavaScript コードは，Web ブラウザ上で動的実行され，Web ブラウザのコンソールに解析結果が出力される．ここでは，このコンソールへの出力を Selenium により取得する必要がある．

Selenium によってコンソールへの出力を取得するには，出力されるログを全て取得するように Selenium Webdriver のオプションをセットする．また，利用する Web ブラウザもオプションとして指定する．本稿では，Web ブラウザとして Google Chrome を利用した．セットしたオプションを引数として下のコードを実行すればよい．

```
driver.manage().logs().get();
```

これにより，コンソールに出力される文字列が取得できるようになる．しかし，**図 4** のように，解析結果がオブジェクトとして返される場合，Selenium でこのログを取得しようとしても，単に “Object” という文字列のみが返され，オブジェクトの中身が取得できないという問題点がある．

この問題に対処するため，解析ログにオブジェクトが含

Vol.2019-DPS-178 No.18
Vol.2019-CSEC-84 No.18
2019/3/4

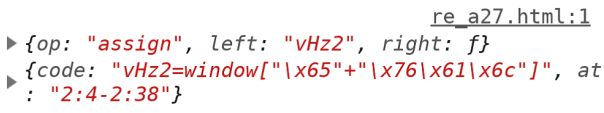

図 4　オブジェクト形式の解析ログ

```
op_eq(ci, a, b) {
  var r = a == b;
  return sh_isobj(a) && sh_isobj(b) && console.log(JSON.stringify({
    op: "==",
    left: a,
    right: b,
    bool: r
  }) + "\n" + JSON.stringify({
    code: ci.code,
    at: sh_range2str(ci)
  })), r
},
```

図 5　解析支援システムの改変部分

まれる場合，それらを文字列に変換して取得する．具体的な方法としては，出力すべきオブジェクトを JSON 文字列に変換してコンソールに出力し，Selenium でその文字列をログとして取得した後，その文字列をオブジェクトに変換し直すという手法である．ここで JSON 文字列とは，JavaScript において，オブジェクトデータを文字列形式のデータで表したものである．この JSON 文字列を利用した方法により，オブジェクトの中身を取得できると考えた．

この手法を適用するには，解析支援システムのログ出力部分のコードを改変する必要がある．解析支援システムの出力部分のうち，改変したコードの一部を**図 5** に示す（太枠で囲った部分が改変した部分）．解析支援システムが解析ログを出力する際に，これまでオブジェクト `obj` を単に `console.log(obj)` と出力していた部分を以下のように改変した．

```
console.log(JSON.stringify(obj))
```

ここで，`JSON.stringify()` 関数は JavaScript において，オブジェクトを JSON 文字列に変換し，その文字列を返す関数である．解析支援システムの出力部分を改変した後にログを取得したところ，変換した JSON 文字列を取得できていることが確認できた．なお，元の解析支援システムでは，オブジェクトをカンマ区切りで出力する仕様である．

ここで，Selenium を利用した解析ログ取得の自動化において，1 つ目のオブジェクトしか取り出せないという問題がある．このため，複数の JSON 文字列を `console.log()` で表示する場合，図 5 のように文字列同士を改行コード（“\n”）でつなぐことにより，全ての JSON 文字列を取り出せるようにしている．取得した JSON 文字列は，`JSON.parse()` 関数を利用することで元のオブジェクトに復元できる．

(4) 解析ログの保存

（3）での自動化の実現により，Web ブラウザ上のコンソールへの出力を取得可能となった．これらの出力は，Node.js 上で `console.log()` 関数により Linux のターミナルへ出力される．よって，この出力を解析ログとして保存するには，リダイレクトを行うようにシェルスクリプトを作成し，ログファイルとして保存すればよい．

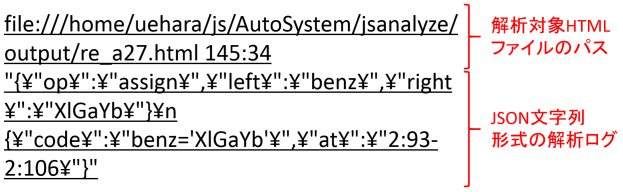

図 6　提案システムで得られる解析ログの例

{"op":"assign","left":"benz","right":"XlGaYb"}
{"code":"benz='XlGaYb'","at":"2:93-2:106"}

図 7　変換手法により変換された解析ログの例

3.5　解析ログ変換手法

提案システムの実現により，解析ログの自動取得が可能となる．一方で，Selenium を利用して解析ログを取得した場合，解析ログの可読性が低下する．提案システムにより得られる解析ログの例を**図 6** に示す．

解析ログの可読性が低下した原因として，以下の 2 点が挙げられる．

(1) 解析結果に追加されたファイルパス
2.2 節の（3）で述べた実現方式により解析結果を自動取得した場合，各解析結果の行頭に，解析対象 HTML ファイルのパスが追加されてしまう．このパスは解析の際に不要な情報である上，パスが長くなると解析ログの可読性が低下する．

(2) JSON 文字列形式
提案システムでは，解析結果がオブジェクトとして返される場合，オブジェクトを JSON 文字列形式に変換したものを解析ログとして取得する．しかし，図 6 に示す通り，JSON 文字列のままでは解析ログの可読性が低い．

(1)へ対処するため，解析ログの中から，HTML ファイルのパスを削除する．また，(2) に対処するため，`JSON.parse()` 関数を利用し，JSON 文字列形式のログをオブジェクト形式のログに変換する．ここで，`JSON.parse()` 関数は JavaScript で利用できる関数であり，JSON 文字列形式のデータを JavaScript のオブジェクトデータに変換できる．

解析ログ変換手法は JavaScript で記述し，Node.js を用いて実行する．ファイル入出力のためのディスクアクセス時間を減らすため，提案システムの JavaScript ファイル内に，解析ログ変換手法（`format` 関数）を実現した（図 3 中の破線で囲まれた部分）．提案システムに解析ログ変換手法を統合して 1 つの JavaScript コードにしたため，提案システムの実行用シェルスクリプトを変更する必要がない．また，解析ログ変換手法を追加したことにより追加で必要

となる Node.js モジュールは無い．

変換手法により変換を施した解析ログを図 **7** に示す．パスの削除および JSON 文字列の変換により，解析ログの可読性が自動化前の状態に戻っている．

3.6 期待される効果

提案システムの実現により，以下の効果が期待できる．

（効果 **1**） 解析時間の短縮

解析対象ファイルの Web ブラウザへの読み込みや解析結果の取得作業を自動化することで，従来より解析時間を短縮できる．

（効果 **2**） 一度に複数の難読化 JavaScript ファイルを処理可能

提案システムは，一度に複数の難読化 JavaScript ファイルを処理できる．これにより，提案システムを利用して複数のファイルを処理している間に別の解析作業を行うことができ，解析作業を効率化できる．

4. 評価

4.1 評価内容

提案システムおよび解析ログ変換手法の有用性を示すため，以下の評価を実施した．評価環境を**表 2** に示す．

（評価 **1**） 提案システムにより短縮できる解析時間の評価

難読化 JavaScript コードに対して，まず，解析支援システムを利用して手動で解析する．その後，提案システムを利用して解析する．それぞれの解析に要した時間を計測し，解析時間を比較する．この比較により，提案システムの利用によって短縮された解析時間を確認する．

（評価 **2**） 複数 **HTML** ファイル連続処理により短縮できる解析時間の評価

提案システムは，複数の HTML ファイルを連続で処理可能である．複数の HTML ファイルを連続で処理するためには，サンプル HTML ファイル用のディレクトリ内に解析対象 HTML ファイルを用意し，連続実行用のスクリプトを実行すればよい．解析対象 HTML ファイル数が 5，10，15，20 の場合に，解析支援システムを利用して HTML ファイルを 1 つずつ手動で処理した場合と，提案システムの連続実行スクリプトにより処理した場合の解析時間を比較する．この比較により，複数 HTML ファイルの連続処理により短縮された解析時間を確認する．

（評価 **3**） 出力された解析ログの正当性の評価

提案システムおよび解析ログ変換手法を利用して出力される解析ログの正しさを検証する．解析支援システムを手動で利用して取得した解析ログと提案システムで取得した解析ログを比較し，同じ内容の解析ログが得られていることを確認する．

表 2 評価環境

ホスト	OS	Windows 8.1 Pro (64bit)
	CPU	Intel Core i5-4460
	メモリ	12.0 GB
	仮想化ソフトウェア	Oracle VM VirtualBox 5.2.12
ゲスト	OS	CentOS Linux release 7.5.1840
	CPU	2 コア
	メモリ	4.0 GB
	Node.js	v9.2.0
	Web ブラウザ	Google Chrome 70.0.3538.102

表 3 提案システムにより短縮できる解析時間 （単位: s）

解析対象 HTML	手動での解析時間	提案システムでの解析時間	短縮された解析時間
a27.html	71.61	9.83	61.78
a55.html	56.79	6.50	50.29
a133.html	53.15	7.80	45.35
b9.html	56.83	7.20	49.63
b39.html	48.16	6.78	41.38

評価には，Malware-Traffic-Analysis.net [6] から収集した難読化 JavaScript コードを使用した．

4.2 提案システムにより短縮できる解析時間の評価

提案システムの利用により短縮できる解析時間の評価について述べる．5 つの難読化 JavaScript コードに対して，解析支援システムを手動で利用した場合の解析時間と提案システムを利用した場合の解析時間を比較した．ここで解析時間とは，解析支援システムと提案システムの両方において，システムを利用するためのコマンドをターミナルに打ち込み始めてから，最終的に解析ログが出力されるまでの時間を指す．この比較結果から，提案システムの利用によって短縮される解析時間について検証する．なお，手動での解析時間については，著者の一人が手動で解析支援システムを利用して解析作業を行い，かかった時間を計測した．**表 3** に評価結果を示す．

表 3 から，全ての HTML サンプルファイルについて，提案システムによって解析作業時間が短縮されていることが分かる．手動での作業時間は解析者によって個人差が出るものの，解析時間が 40 秒から 1 分程度短縮されていることが分かる．この結果から，解析対象 HTML ファイル数が非常に多い場合，提案システムが解析時間の短縮に大きく貢献できると推察できる．

4.3 複数 HTML ファイル連続処理により短縮できる解析時間の評価

複数 HTML ファイルの連続処理により短縮される解析時間の評価について述べる．解析対象 HTML ファイルの数が 5，10，15，20 の場合に対して，解析支援システムを利

表 4　連続処理により短縮できる解析時間　（単位: s）

解析対象コード数	手動での解析時間	連続処理での解析時間	短縮された解析時間
5	282.77	9.67	273.10
10	489.25	19.12	470.13
15	700.16	30.66	669.50
20	888.91	39.11	849.80

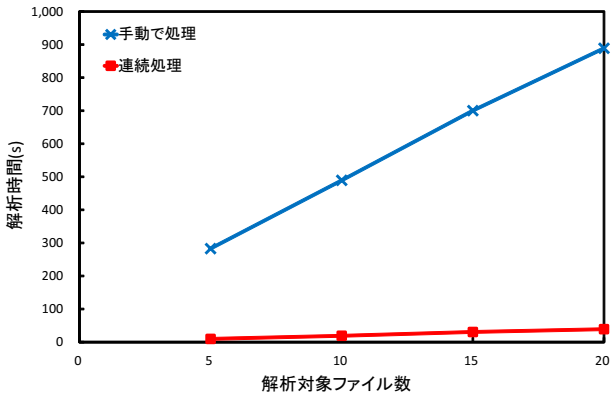

図 8　提案システムを利用した場合の解析時間（単位: s）

用して手動で処理した場合の解析時間と，連続処理を行った場合の解析時間を比較した．評価結果を**表 4** および**図 8** に示す．

表 4 から，解析対象コードの数が 5，10，15，20 のどの場合でも，連続処理により解析時間が $\frac{1}{22}$ から $\frac{1}{30}$ 程度に短縮されていることが分かる．3.6 節の（効果 2）で述べた通り，この連続処理中に，解析者は別の解析作業を行うことができ，解析作業の効率化が期待できる．

4.2 節と比較して解析時間が大きく短縮されたのは，手動操作部分の自動化に加え，ファイルを読み込むたびにファイル名を指定することなく，連続で処理したことが主な要因だと推察する．

4.4　出力された解析ログの正当性の評価

解析支援システムを手動で利用して取得した解析ログと提案システムで取得した解析ログを比較し，同じ内容の解析ログが得られているか否かを確認した．**図 9** に，提案システムによって得られた解析ログの一部を示す．

解析支援システムを手動で利用した場合の解析ログと比較した結果，同じ内容の解析ログが得られていた．解析ログ変換手法によって，解析ログが文字列とオブジェクトのどちらの場合においても行頭のファイルパスの削除ができている．また，解析結果がオブジェクトの場合は，JSON 文字列形式からオブジェクト形式へのログ変換が正しく行われている．

しかし，解析結果オブジェクトの文字列内にエスケープシーケンスが含まれる場合，変換手法が上手く適用できない．**図 10** に例を示す．図 10 のように，解析結果オブジェクトの文字列内にエスケープシーケンスが含まれる場合，そのエスケープシーケンスがエスケープ処理されて出力される．この例では，ダブルクオーテーションマークがエスケープ処理されており，実行されたコードを直感的に把握するのが難しい．このような意図しないエスケープ処理により，他の JavaScript コードの解析の際にも解析ログの可読性が低下する可能性がある．解析ログ変換手法適用前の解析ログを調査したところ，解析ログが既にエスケープ処理されていた．一方，提案システムを導入していない従来の解析支援システムではこのエスケープ処理が起こっていない．以上のことから，3.4 節で述べた Selenium を用いた実現方式が原因でエスケープ処理がなされていると推察している．この問題への対処は今後の課題とする．

```
"[[Get]] HTMLDocument.URL val: file:///home/uehara/
js/AutoSystem/jsanalyze/output/re_a27.html“

{"op":"assign","left":"benz","right":"XlGaYb"}
{"code":"benz='XlGaYb'","at":"2:93-2:106"}

{"op":"<","left":"0","right":62,"bool":true}
{"code":"c<62","at":"2:1216-2:1220"}
```

図 9　解析ログ変換手法の出力例　（a27.html）

```
{"op":"assign","left":"D5074","right":"/"}
{"code":"D5074=¥"/¥"","at":"3:0-3:9"}
```

図 10　一部がエスケープ処理された解析ログの出力例　（b39.html）

5.　関連研究

5.1　コンテナ上で実現される難読化 JavaScript コード解析システム

JS-Walker [7] は，難読化 JavaScript コード動的解析ツールであり，文献 [8] で提案されている．JS-Walker は，Docker コンテナ上で稼働するブラウザで解析対象のコードを実際に動作させることで，難読化 JavaScript の挙動を明らかにする [8]．これに対し，提案システムでは，Docker コンテナを利用せず，Linux 仮想マシン上の Web ブラウザで解析対象のコードを実際に動作させ，難読化 JavaScript コードの挙動を明らかにする．JS-Walker と提案システムは両者とも，隔離した環境上で Web ブラウザを用いて動的解析を行う．しかし，その隔離環境がコンテナか仮想マシンであるかという点で異なっている．コンテナは仮想マシンに比べて消費リソースが少なく，起動も高速であるため，提案システムにも応用できると考える．

5.2　Web ブラウザの自動操作に関する研究

Web ブラウザの自動操作に Selenium を利用している研究として，文献 [9]，文献 [10]，および文献 [11] がある．文

献 [9] では，Selenium を利用して，Web ブラウザの操作とクローリングを自動化している．文献 [10] では，Selenium を利用して，Web アプリケーションの動作時に出力されるページソースから，JavaScript コードを自動的に抽出し保存している．文献 [11] では，Web Cache Deception 脆弱性 [11] の有無を判定するために，Selenium を利用して，特定の URL への Web アクセスを行っている．いずれの研究も，Web ブラウザの操作やページソースの取得のために Selenium を利用しており，提案システムのように，Selenium を利用して Web ブラウザのコンソールの出力を取得する研究ではない．

一方，文献 [12] では，Web ブラウザの自動操作に Puppeteer [13] を利用している．Puppeteer は，Web ブラウザの情報の取得や，Web ブラウザの操作を行うためのライブラリである．Selenium と Puppeteer は両者とも，Web ブラウザの自動操作や情報の取得を行うことができる．一方 Puppeteer は，Selenium では取得できない Web ページのステータスコードを取得するインタフェースを持ち，JavaScript のみがサポートされているなど，Selenium とのわずかな違いがある．Puppeteer は Node.js 上で動作し，Google Chrome をサポートするため，本研究で行った Web ブラウザの自動操作に利用することができると考える．

6. おわりに

文献 [1,2] で提案された難読化 JavaScript コード解析支援システムの概要，実現方式，および課題について述べ，解析支援システムの自動化を提案し，解析支援システムにおける課題へ対処した．提案システムでは，シェルスクリプト，Node.js，および Selenium を利用することで解析支援システムの手動作業部分を自動化する．また，提案システムにより得られる解析ログの可読性の低さに対処するため，解析ログ変換手法を提案した．

評価では，提案システムにより，解析時間が最大で $\frac{1}{30}$ 程度に短縮されることを示し，難読化 JavaScript コードの解析支援に有効であることを示した．

残された課題として，4.4 節で述べた解析ログに対して変換手法が上手く適用できない問題への対処がある．

参考文献

[1] 上川先之，山内利宏：API 操作ログ取得による難読化 JavaScript コード解析支援システム，コンピュータセキュリティシンポジウム 2017（CSS2017）論文集，Vol.2017，No.2，pp.370–377 （2017）．
[2] 上川先之，山内利宏：コード実行時情報対応付けによる難読化 JavaScript コード解析支援手法，暗号と情報セキュリティシンポジウム 2018 （SCIS2018） 論文集，電子媒体，pp.1–8 （2018）．
[3] SeleniumHQ （online），available from ⟨https://www.seleniumhq.org/⟩ （accessed 2018–11–12）．
[4] Node.js （online），available from ⟨https://nodejs.org/⟩ （accessed 2018–11–13）．
[5] Mihai Bazon：UglifyJS（online），available from ⟨http://lisperator.net/uglifyjs/⟩ （accessed 2018–11–08）．
[6] Malware-Traffic-Analysis.net （online），available from ⟨http://malware-traffic-analysis.net/⟩（accessed 2018–11–16）．
[7] NTT セキュリティ株式会社，難読化 JavaScript 動的解析ツール（JS-Walker） （online），available from ⟨https://www.nttsecurity.com/docs/librariesprovider3/resources/jswalker⟩（accessed 2019–01–08）．
[8] 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，Vol.2016，No.2，pp.951–957（2016）．
[9] 西尾祐哉，廣友雅徳，神薗雅紀，福田洋治，毛利公美，白石善明: マルチ環境解析と JavaScript 解析を組み合せた悪性 Web サイトのクローキング分析手法，情報処理学会論文誌，Vol.59，No.9，pp.1624–1638（2018）．
[10] 井上佳祐，本多俊貴，向山浩平，大木哲史，西垣正勝: ホワイトリスト型 XSS 攻撃検知の効果的な実現方法に関する検討，暗号と情報セキュリティシンポジウム 2018（SCIS2018） 論文集，電子媒体，pp.1–7 （2018）．
[11] 小川怜和，奥田祐也，齊藤泰一: Web Cache Deception 脆弱性検知，暗号と情報セキュリティシンポジウム 2018（SCIS2018） 論文集，電子媒体，pp.1–7 （2018）．
[12] 高田雄太，渡邉卓弥，中野弘樹，波戸邦夫，秋山満昭: Web プッシュ通知の悪用に関する実態調査，情報処理学会研究報告，Vol.2018–CSEC–83，No.17，pp.1–8 （2018）．
[13] Puppeteer （Online），available from ⟨https://github.com/GoogleChrome/puppeteer⟩ （accessed 2019–1–8）．