プログラムスライシングを用いた環境依存コードの実行網羅性向上による潜在的 URL の抽出

高田 雄太 †　　秋山 満昭 †　　八木 毅 †　　針生 剛男 †

†NTT セキュアプラットフォーム研究所
180-8585 東京都武蔵野市緑町 3-9-11
{takata.yuta，akiyama.mitsuaki，yagi.takeshi，hariu.takeo}@lab.ntt.co.jp

あらまし ドライブバイダウンロード攻撃は，JavaScript を用いて踏台 URL を経由した後，ユーザを攻撃 URL へリダイレクトする．ユーザが攻撃 URL にアクセスすると，ブラウザやプラグインの脆弱性を悪用する攻撃コードが実行され，ユーザはマルウェア配布 URL からマルウェアをダウンロード，インストールしてしまう．さらに攻撃者は，ユーザのクライアント環境に応じてリダイレクト先 URL を変更し，踏台 URL や攻撃 URL を隠蔽する．したがって，おとりの環境で悪性 Web サイトを検査する従来技術は，攻撃 URL へリダイレクトされない場合は有効に機能しない．そこで本研究では，抽象構文木およびプログラムスライシングを用いた JavaScript 解析を行うことで，リダイレクト先となる潜在的な踏台 URL や攻撃 URL を抽出する技術を提案する．

Improving Coverage of Environment-dependent Code Using Program Slicing to Extract Potential URLs

Yuta Takata†　　Mitsuaki Akiyama†　　Takeshi Yagi†　　Takeo Hariu†

†NTT Secure Platform Laboratories
3-9-11 Midori-cho, Musashino-shi, Tokyo, 180-8585 JAPAN
{takata.yuta，akiyama.mitsuaki，yagi.takeshi，hariu.takeo}@lab.ntt.co.jp

Abstract Drive-by Download attacks redirect the user to the exploit URL via redirection URLs using JavaScript. The user installs and downloads malware from the malware distribution URL by accessing the exploit URL that exploits vulnerabilities of browser and/or plugin. In addition, an attacker changes redirect destinations depending on the client-environment in order to conceal redirection or exploit URLs. Therefore, conventional URL inspection methods based on a decoy system do not adequately work when they cannot access exploit URLs. In this paper, we propose a technique to extract potential redirection URLs and exploit URLs by JavaScript analysis using abstract syntax tree and program slicing.

1　はじめに

Web サイトを通じてクライアントをマルウェアに感染させるドライブバイダウンロード攻撃が深刻化している [2]．攻撃者は，入口 URL にアクセスしたユーザを，JavaScript を用いて踏台 URL へアクセスさせた後，攻撃コードを実行する攻撃 URL へリダイレクトする．ユーザが攻撃 URL にアクセスすると，ブラウザやプラグインの脆弱性を悪用する攻撃コードが実行され，ユーザはマルウェア配布 URL からマルウェアをダウンロード，インストールしてしまう．

この攻撃に対して，おとりのクライアント環境で攻撃 URL にアクセス，攻撃コードを収集し，収集した情報に基づいて対策を講じる技術が提案されている [3, 4]．しかし攻撃者は，アクセスしてきたユーザのブラウザやプラグインの種類やバージョンといったクライアント環境を，JavaScript を用いて識別し，

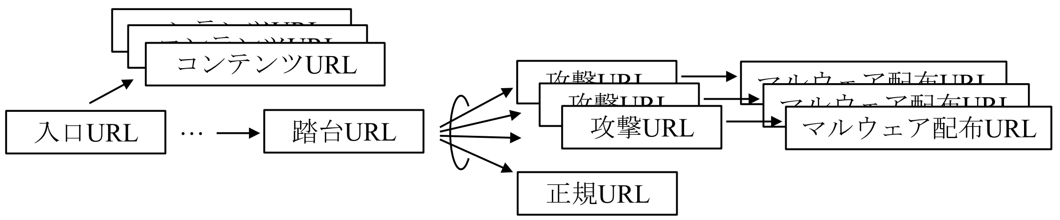

図 1: ドライブバイダウンロード攻撃における多段リダイレクト

環境に応じて実行するコードを制御することで，踏台 URL や攻撃 URL を隠蔽する．攻撃 URL へリダイレクトされないと，攻撃コードを検知する従来技術は有効に機能しない．

そこで本稿では，JavaScript コードの抽象構文木を用いた静的解析によりコードカバレッジを向上させることで，実行されない可能性のあるコードを特定する手法を提案する．さらに本手法では，特定したコードに対して，プログラムスライシングを用いた関連コードの抽出を行い，特定したコードとともに動的実行することで，潜在的な踏台 URL や攻撃 URL を抽出する．D3M [1] に含まれているパケットキャプチャデータを基に HTTP 通信を再現し，本手法を実装したブラウザエミュレータで 入口 URL にアクセスした結果，825 の巡回中，463（56%）の巡回において D3M の HTTP 通信に含まれていない新たな URL を抽出することに成功した．本手法により得た新たな URL やドメイン情報は，ドライブバイダウンロード攻撃の対策技術に活用できる．

2　研究背景

2.1　多段リダイレクト

ドライブバイダウンロード攻撃の起点となる入口 URL にアクセスしたユーザは，主に JavaScript を用いて踏台 URL を経由した後，攻撃 URL へリダイレクトされる．ユーザが攻撃 URL にアクセスすると，ブラウザやプラグインの脆弱性を悪用する攻撃コードが実行され，ユーザはマルウェア配布 URL からマルウェアをダウンロード，インストールしてしまう（図 1）．

JavaScript の実行によって，ユーザを異なる URL へリダイレクト（アクセス）させる方法は大きく分けて二種類存在する．異なる URL へ転送するリダイレクトコードによる方法と新たな外部コンテンツを取得するコードによる方法である．リダイレクト

表 1: リダイレクトに使用されるコードと外部コンテンツを取得する HTML タグ

リダイレクトコード	window.location = ‘URL’; location.href = ‘URL’; location.assign(‘URL’); location.replace(‘URL’);
コンテンツ取得コード	element.innerHTML = ‘HTML タグ’; document.write(‘HTML タグ’); document.writeln(‘HTML タグ’); element.setAttribute(‘src’, ‘URL’); XMLHttpRequest.send(‘URL’);
コンテンツ取得 HTML タグと属性名	<iframe src>, <frame src>, <script src> <embed src>, <applet archive> <object data>, <meta conent>

コードやコンテンツ取得コードは，表 1 のような JavaScript 関数やプロパティを使用する．また，コンテンツ取得コードは，表 1 に示したコンテンツを取得する HTML タグを使用し，属性名に値として URL を指定することで外部コンテンツを取得する．

上記二種類のコードは，実行されると自動的に指定された URL へアクセスするように動作する．ドライブバイダウンロード攻撃では，非表示設定で HTML タグを挿入したり，アクセス先 URL を操作したりすることで，ユーザに気づかれないようにリダイレクトさせている．

2.2　解析妨害，検知回避

攻撃者は，多段リダイレクトのほか，攻撃手法を複雑化・巧妙化することで解析妨害や検知回避を行っている [5]．事前に行った悪性 Web サイトの実態調査では，すべての調査対象 Web サイトにおいて，クライアント上で JavaScript によるブラウザフィンガープリンティングが使用されていることがわかった [6]．ブラウザフィンガープリンティングとは，Web サイトにアクセスしてきたクライアント環境を識別する手法のことで，一般的にはユーザトラッキングや環境に応じたコンテンツの提供を実現するために利用される．攻撃者は，JavaScript によるブラウザフィンガープリンティングを用いて，ブラウザやプラグイ

ンなどといったクライアント環境を識別し，取得した環境情報に応じてリダイレクト先 URL や実行する攻撃コードを変更する制御を加えることで，攻撃の成功率を向上させている．図 1 における正規 URL へのアクセスは，例えばアクセスしてきたユーザが攻撃対象のクライアント環境ではなかった場合に，正規 URL へリダイレクトし，正常な Web サイトとして装う場合を示している．すなわち，攻撃対象のクライアント環境を用いた解析を行わないと，攻撃 URL まで到達できず，攻撃コードやマルウェア配布 URL，およびマルウェアを収集することはできない．攻撃 URL へ到達するための踏台 URL もしくは攻撃 URL を特定できれば，アクセスすることで攻撃コードやマルウェアを収集し，シグネチャによる攻撃検知等に活用することができる．

3　提案手法

本稿では，JavaScript の制御命令に依存するリダイレクトコードやコンテンツ取得コードに着目し，これらのコードから URL を抽出する手法を提案する．

本手法は，まず URL アクセス時に取得したコンテンツから JavaScript コードを抽出し，コードを抽象構文木に変換する．次に，抽象構文木を用いた静的解析を通じてリダイレクトコードおよびコンテンツ取得コードを特定する．さらに，抽象構文木で特定したコードに加え，プログラムスライシングを用いて特定したコードに関連するコードを抽出する．特定したコードと関連コードを動的実行することで，リダイレクトコードおよびコンテンツ取得コードに使用されている URL を抽出する．

3.1　JavaScript コードの抽出

本手法では，コンテンツの取得や取得したコンテンツのパースを行うために，既存研究 [4] の JavaScript 解析に使用されているブラウザエミュレータ HtmlUnit [7] を使用する．HtmlUnit では，取得したコンテンツをパースする過程で，以下の規則に一致したコンテンツを JavaScript のコードとして評価している．

- script タグに囲まれたコンテンツ：<script>code</script>
- script タグの src 属性に指定された URL のコンテンツ：<script src="URL"></script>
- HTML エレメントの属性値 "javascript:" で記述されたコンテンツ："javascript:code"

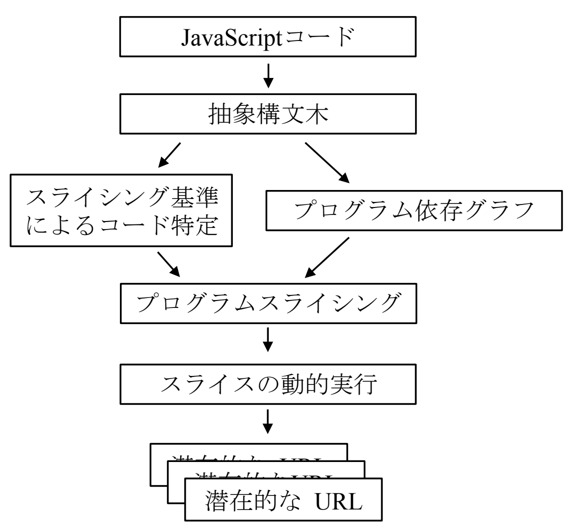

図 2: 提案手法の処理手順

```
 1 var jre_version = pd.getVersion('Java');
 2 var path = "next_redirection";
 3 var src = "http://exploit.com/malicious/?jre=";
 4 var jre = jre_version.split(",");
 5 if (jre[1] == "6") {
 6   var ifr = document.createElement("iframe");
 7   ifr.setAttribute("src", src + jre_version);
 8   document.body.appendChild(ifr);
 9 }
10 else {
11   location.href = "http://redirect.com/" + path;
12 }
```

図 3: クライアント環境依存コード

そのほか，HtmlUnit では JavaScript 関数 `eval()`，`setInterval()`，`setTimeout()` の引数として指定された文字列を JavaScript コードとして解釈する．そのため，本手法では上記の JavaScript コードに加え，これらの関数の引数文字列に関しても JavaScript コードとして抽出する．抽出された JavaScript コードは，HtmlUnit の JavaScript エンジン Rhino [8] によって抽象構文木に変換され，コンパイラによるコードの最適化が行われる．

3.2　抽象構文木を用いた JavaScript の静的解析

抽象構文木（*Abstract Syntax Tree*）とは，プログラム構造を抽象的な木構造で表すデータ構造である．抽象構文木を探索することで，プログラムを網羅的に解析できる．すなわち，プログラム構造に依存せずコードを解析できるため，JavaScript の制御命令によって実行されないようなコードも静的解析するこ

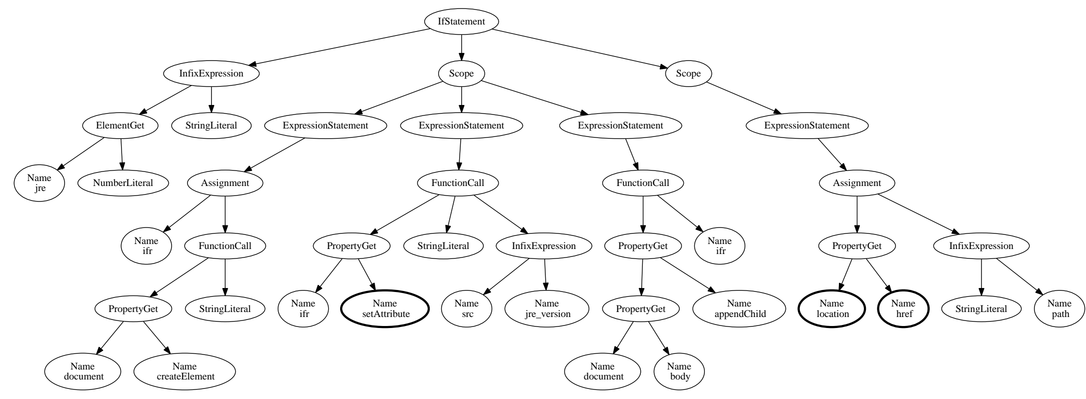

図 4: 条件分岐コードの抽象構文木

とができる．本手法では，あらかじめ JavaScript エンジンによって変換された抽象構文木を探索し，リダイレクトコードとコンテンツ取得コードに使用される関数の有無を調査する．例えば，クライアント環境の情報を取得し，その情報に基づいてリダイレクト先 URL を変更するコード例を図 3 に示す．さらに，図 3 のコードにおける if 文による条件分岐コード（5 行目 – 12 行目）を抽象構文木に変換した結果を図 4 に示す．

図 4 の抽象構文木を探索すると，表 1 に示した `location.href` や `setAttribute()` が使用されていることがわかる．このように抽象構文木を解析することで，制御命令に依存することなくリダイレクトコードやコンテンツ取得コードを特定できる．しかし，特定したコードの引数には変数が使用されているため，変数の値を解決しなければ正確な URL を抽出できない．そこで，次節に示すように，プログラムスライシングを用いて関連コードを抽出して動的実行することで変数の値を解決し，正確な URL を抽出する．

3.3 プログラムスライシングを用いた JavaScript の動的実行

プログラムスライシング（*Program Slicing*）とは，スライシング基準（*Slicing Criterion*）$<s, v>$と呼ばれる，プログラム内の任意の文 s において着目する変数 v に関連する一部の文集合をプログラムから抽出する手法である [9]．スライシング基準にしたがって抽出された文集合はスライス（*Slice*）と呼ばれ，ある変数に影響を与えるスライスを抽出することで，プログラムデバッグやテスト等に用いられる．本手法では，3.2 節より特定したリダイレクトコードやコンテンツ取得コードに関連するコードを抽出するために，2.1 節に列挙した関数やプロパティをスライシング基準とし，スライスを抽出する．

また本手法では，3.2 節における抽象構文木探索の過程で，プログラム依存グラフを構築することができるため，プログラム依存グラフによるプログラムスライシングを用いる．プログラム依存グラフとは，制御依存関係およびデータ依存関係の二つの依存関係を用いて導出される有向グラフである．

制御依存関係 プログラム中の任意の文 p における条件式の実行結果が，任意の文 q が実行されるかどうかに影響を与える場合，文 p から文 q に制御依存関係（*Control Dependence*）があるという．

データ依存関係 プログラム中の任意の文 p における変数 v の定義が，変数 v を参照する任意の文 q に到達可能である場合，文 p から文 q にデータ依存関係（*Data Dependence*）があるという．ただし，文 p から文 q までの間に変数 v の再定義がないものとする．

上述した依存関係を用いて導出したプログラム依存グラフは，ノード（節点）として，代入文，関数定義文，関数呼び出し文，条件分岐文，繰り返し文を持ち，エッジ（有向辺）として二つのノード間の制御依存関係およびデータ依存関係を持つ．図 3 のコードにおけるプログラム依存グラフを図 5 に示す．エッジには，制御依存関係における制御命令名やデータ依存関係にある変数名が表示されている．

図 5 のようなプログラム依存グラフを用いて，制御依存関係やデータ依存関係のエッジをたどることにより，スライスを抽出する．エッジをたどる方法には，たどる方向によって順方向スライス（*forward*

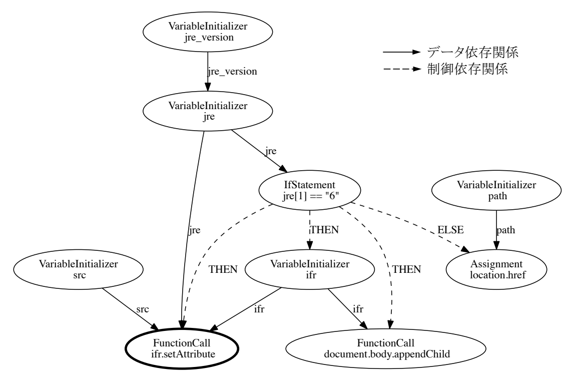

図 5: プログラム依存グラフ

```
1  var jre_version = pd.getVersion('Java');
2  var path = "next_redirection";
3  var src = "http://exploit.com/malicious/?jre=";
4  var jre = jre_version.split(",");
5  if (jre[1] == "6") {
6    var ifr = document.createElement("iframe");
7    ifr.setAttribute("src", src + jre_version);
8    document.body.appendChild(ifr);
9  }
10 else {
11   location.href = "http://redirect.com/" + path;
12 }
```

図 6: スライシング基準 <7, `setAttribute()` >で抽出したスライス

slice）と逆方向スライス（*backward slice*）の二種類が存在する．順方向スライスは，エッジを順にたどっていくことで，スライシング基準のノードが影響を与えるノードをスライスとして抽出することができる．一方で，逆方向スライスは，エッジを逆にたどっていくことで，スライシング基準のノードが影響を受けるノードをスライスとして抽出することができる．本手法では，制御命令に依存しない潜在的な URL の抽出を目的としているため，データ依存関係のエッジのみを使用した逆方向スライスによるスライス抽出を行う．また，制御依存関係にないノードは，JavaScript を評価する際に自動的に実行されるため，何らかの制御依存関係にあるノードをスライシング基準として選出する．

例えば，スライシング基準 <7, `setAttribute()` >で，逆方向スライスによるスライス抽出を行った結果を図 6 に示す．図 6 からは，スライシング基準 <7, `setAttribute()` >が影響を受けるノードをスライスとして抽出しており，スライスは解釈できる状態にあることがわかる．そこで，JavaScript エンジンを用いて図 6 のスライスを動的実行すると，`http://exploit.com/malicious/?jre=1,6,0,22` という URL を得ることができる（JRE バージョン 1.6.0.22 使用の場合）．この時，本来の JavaScript 処理にスライスの動的実行による影響を与えないよう JavaScript のコンテキスト情報（変数定義や関数定義等）は，スライス実行用に複製し，実行後は破棄する．

以上のように，本手法では，抽象構文木を解析することでリダイレクトコードおよびコンテンツ取得コードを特定し，特定したコードを基にプログラムスライシングによって抽出したスライスを動的実行する．その結果，HtmlUnit が JavaScript を解釈する過程でアクセスする URL に加え，リダイレクトコードおよびコンテンツ取得コードに使用される URL を抽出することができる．

4　実験

4.1　実験方法

本手法では，従来手法では抽出できない URL を抽出できる一方で，本手法による JavaScript の解析時間が必要となる．そこで本稿では，D3M [1] に含まれているパケットキャプチャデータを基に HTTP 通信を再現し，本手法を実装した HtmlUnit で入口 URL にアクセスすることで，D3M から抽出した URL 数と解析時間の評価を行う．

本手法の有効性を評価するために，従来の HtmlUnit（Normal）および Normal に本手法（Slice）を実装した HtmlUnit（Proposal）を用いて，D3M2010 – D3M2014 における合計 825 の入口 URL にアクセスした．この時，HtmlUnit は D3M に含まれるデータを収集した環境と同様の OS，ブラウザ，プラグイン（Oracle JRE, Adobe Acrobat, Adobe Flash Player のみ）を模擬する設定で使用した．

さらに，本手法の効率性を評価するために，Normal のクライアント環境として，プラグインの脆弱性を網羅する全 52 パターンを模擬（Multi-Pattern）[10] し，各パターンで入口 URL にアクセスした．

4.2　実験結果

D3M を用いた提案手法の評価結果を表 2 に示す．表 2 から，Proposal の方が Normal より多くの新たな URL を抽出できていることがわかる．表中の Slice は，Proposal におけるスライス実行で得たユニーク

表 2: D3M を用いた提案手法の評価

		D3M2010	D3M2011	D3M2012	D3M2013	D3M2014	合計
入口 URL（巡回）数		386	217	133	39	50	825
巡回に含まれるユニーク URL 数		925	466	286	128	260	2,017
新たに抽出した が含まれる巡回数 URL	Normal	130	77	25	14	34	280
	Multi-Pattern	130	87	80	20	39	356
	Proposal	186	141	80	15	41	463
巡回中に抽出した ユニーク URL 数	Normal	549	281	145	81	107	1,123
	Multi-Pattern	1,467	490	220	372	1,008	3,517
	Proposal	651	439	300	97	195	1,642
新たに抽出した ユニーク URL 数	Normal	99	83	8	8	23	212
	Multi-Pattern	1,017	292	47	295	924	2,566
	Proposal (Slice)	141 (52)	217 (136)	122 (116)	24 (21)	81 (59)	576 (382)
抽出できなかった ユニーク URL 数	Normal	475	268	149	55	176	1,106
	Multi-Pattern	475	268	113	51	176	1,066
	Proposal	415	244	108	55	146	951
平均巡回時間 [sec]	Normal	3.20	2.98	7.96	4.11	3.67	3.99
	Multi-Pattern	166.43	224.84	289.64	213.89	186.97	205.15
	Proposal	3.49	3.05	8.26	5.00	3.84	4.24
Slice の平均解析時間 [sec]		0.42	0.24	0.49	0.32	0.63	0.39

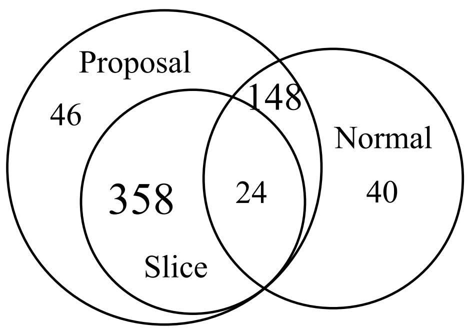

図 7: 新たに抽出したユニーク URL の抽出方法

表 3: 抽出したユニーク URL の差異の例

Normal	http://DOM1.net/forum/go.php?sid=1
	http://DOM1.net/forum/got.php?sid=1
	http://DOM2.com/main.php?page=bbd8c7cc65c2cfb5
	http://DOM2.com/content/Qai.jar
Proposal	http://DOM1.net/forum/go.php?sid=1
	http://DOM1.net/forum/got.php?sid=1
	http://DOM2.com/main.php?page=bbd8c7cc65c2cfb5
	http://DOM2.com/content/Qai.jar
	http://DOM2.com/content/ap2.php?f=7a2b1
	http://DOM2.com/content/field.swf
	http://DOM2.com/content/hhcp.php?c=7a2b1
	http://DOM2.com/content/ap1.php?f=7a2b1

URL 数を表しており，Proposal の 66%の URL は，Slice による抽出であることがわかる．

また，Slice で必要となる解析時間（抽象構文木の探索，プログラムスライシング，スライスの動的実行）は，総巡回数で平均して 0.39 sec，総巡回時間に対して 9.29 %であった．解析によるオーバーヘッドは少なからずあるものの，巡回中に抽出したユニーク URL 数 1,642 の内，新たに抽出したユニーク URL 数は 576（54%増）であるため，本手法により効率よく URL を抽出できていると言える．

ここで，どの方法によって新たに URL が抽出されたか把握するため，抽出された URL とその抽出方法の関係を図 7 に示す．図 7 から，本手法によるスライス実行でのみ抽出できた URL（$Slice \setminus (Slice \cap Normal) = 358$）は，図 7 の中で最も多くの URL を抽出できていることがわかる．同一の入口 URL へアクセスした際に抽出したユニーク URL の差異の例を表 3 に示す．Proposal は，Normal よりも太字で示す URL を新たに抽出できていることがわかる．また，これらの URL の約半数には拡張子として.pdf, .swf, .jar が使用されていることがわかった．これは，図 3 のようなクライアント環境情報に依存して .pdf, .swf, .jar といったコンテンツを外部から取得するコードに対して，それぞれの分岐パスから URL を抽出できていることを意味する．この結果は，本稿が対象としていた潜在的な URL に対して，本手法が有効に機能していることを示しており，抽出した URL のコンテンツは，攻撃コードを含む悪性コンテンツである可能性が高いと考えられる．

さらに，Proposal のみ（$Proposal \setminus Normal \setminus Slice = 46$）および Normal のみ（$Normal \setminus Proposal = 40$）で抽出している URL は，手動で URL を調査したところ，アクセス解析に使用されるワンタイム URL であることがわかった．ワンタイム URL は，アクセスごとに URL を変化させるため，このように新たな URL として一定数抽出されてしまう．

そのほか，Proposal と Normal に共通して出現する URL（$Proposal \cap Normal = 172$）は，HtmlUnit によるアクセスで抽出できる URL であるが，その

中でも Slice にも共通して出現する URL（$Slice \cap Proposal \cap Normal = 24$）は，URL のパラメータクエリに環境情報（e.g. `&br=MSIE&vers=6.0`）が使用されていた．これは，今回使用した HtmlUnit のクライアント環境設定では抽出に成功しているが，異なるクライアント環境設定の HtmlUnit を使用すると抽出できない，もしくは URL が変化する可能性があることを意味している．

Multi-Pattern は，Proposal と比べて 4 倍以上の新たな URL を抽出できているが，その差分 URL（$Multi\text{-}Pattern \setminus Proposal$）の内容は，ほぼすべてアクセス解析等のワンタイム URL であった．また，平均巡回時間は Proposal の約 48 倍を要している．したがって本手法は，Multi-Pattern と比べて，対策に必要のない URL を過剰に抽出することなく，効率的に URL を抽出できていると言える．

5 関連研究

本手法と同様に，ブラウザエミュレータを用いて JavaScript を解析し，悪性 Web サイトを検知する方法が提案されている．JSAND [4] は，ブラウザエミュレータを用いて，JavaScript 関数の実行回数や引数，ActiveXObject の使用といった情報を取得し，機械学習による悪性 Web サイト検知手法を提案している．しかし，JSAND は JavaScript 解析に単一の実行パスしか考慮していない．すなわち，クライアント環境に依存してリダイレクト先 URL や実行する攻撃コードを変更する攻撃に対しては，収集できない情報が存在する．そのため，抽象構文木やマルチパス実行を用いた解析手法により，JavaScript 解析のカバレッジを向上させる方法が研究されてきた．

Wang らは，抽象構文木とコールグラフを用いた JavaScript 解析を行い，JavaScript に埋め込まれている URL を抽出することで，検索エンジンのカバレッジ向上手法を提案している [11]．コールグラフとは，プログラムにおけるサブルーチン同士の呼び出し関係を表現した有向グラフである．Wang らは，抽象構文木を用いて URL が埋め込まれるような箇所を選別した後，コールグラフを用いて実行されていない関数を網羅的に実行することで，URL を抽出している．しかし，Wang らの手法は，独自の JavaScript エンジンによる実装を行っており，クライアントのプラグイン情報を参照するコードには対応していない．また，検索エンジンのカバレッジ向上を目的としているため，a タグや form タグなど静的リンクに使用される URL も抽出している．攻撃に使用される可能性の低い URL を多く抽出しており，余分な解析を必要とする恐れがある．

最後に，マルチパス実行による環境依存攻撃の解析手法を提案している ROZZLE という手法がある [12]．ROZZLE は，多くの悪性 Web サイトに使用されるクライアント環境情報に基づく条件分岐コードに着目し，クライアント環境に基づく分岐をすべて実行（マルチパス実行）することで，新たな悪性 URL を抽出する方法を提案している．しかし，ROZZLE は，実ブラウザへの組み込み実装であるため，クライアント環境に制限が存在する．

6 考察

6.1 提案手法による再帰的評価

本稿で実施した実験は，D3M に含まれる HTTP 通信データのみによるコンテンツ評価である．すなわち，D3M に含まれていないコンテンツは評価していない．そのため，本手法で新たに抽出した URL のコンテンツにおいても評価を行っていない．本手法は，一連のリダイレクトで取得したコンテンツに対して再帰的に適用できる．したがって，本手法で新たに抽出した URL を基にコンテンツを取得し，本手法による解析を行うことで更なる新たな URL を抽出することができると期待できる．

6.2 サーバサイドブラウザフィンガープリンティング

悪性 Web サイトの中には，サーバサイドにて，OS やブラウザ，IP アドレスや国，言語情報を取得し，応答するコンテンツを変更する Web サイトが存在する [13]．本手法は，クライアントで実行される JavaScript に対する解析手法であるため，クライアントサイドにおけるリダイレクト先 URL の変更には有効だが，サーバサイドにおける URL 変更には有効ではない．しかし，本手法ではブラウザエミュレータを使用しており，任意のクライアント環境をエミュレートできる．したがって，サーバサイドブラウザフィンガープリンティングに対しては，クライアント環境設定の変更に応じて，応答コンテンツが変化するかどうかを監視することで検知できる．

6.3 マルウェア配布 URL やマルウェアの収集

本手法は，ブラウザによる JavaScript の解釈に割り込み，JavaScript コードの解析を行うことから，ブラウザの実装に変更を加えることのできるブラウザエミュレータを用いている．しかし，ブラウザエミュレータは，ブラウザの動作を完全に模擬するわけではないため，ブラウザやプラグインの脆弱性を悪用した攻撃コードを解釈することはできない．すなわち，本手法では攻撃コードがアクセスするマルウェア配布 URL を抽出することはできない．表 2 における抽出できなかった URL は，これらマルウェア配布 URL が含まれていると考えられる．また，本手法では新たに抽出した URL の悪性判定は行っていない．マルウェア配布 URL およびマルウェアの収集やその悪性判定を行うには，本手法で抽出した URL に対して，エミュレータではなく実環境を用いた高対話型ハニークライアントによる巡回を行う必要がある．

6.4 制御命令によるリダイレクト先 URL の部分変更

本稿におけるプログラムスライシングでは，スライス抽出に制御依存関係のエッジを考慮していない．そのため，リダイレクト先 URL が変数によって決定し，かつ制御命令によってその変数の値が変化する場合に，制御命令において最後に代入された変数の値しか評価することができない．6.3 節同様，表 2 における抽出できなかった URL には，ほかに代入される変数の値を利用した URL が含まれると考えられる．スライシング基準の関数やプロパティが，制御命令に依存して値を変える変数を参照する場合は，すべての場合を評価する必要がある．

6.5 提案手法の検知および回避

本手法は，プログラムスライシングによる JavaScript コードの動的実行を行うため，論理的に到達できないコードまで実行してしまう恐れがある．例えば，あらかじめ攻撃者が到達不能コードを用意し，到達不能コードに含まれる URL に対してアクセスを試みるユーザがいた場合，アクセスログから本手法を検知される恐れがある．これに対して，コールグラフを用いて到達不能コードを判別し，解析対象から除外する等の対応策がある．

7 まとめ

本稿では，JavaScript の制御命令によってリダイレクト先 URL や実行する攻撃コードを変更するドライブバイダウンロード攻撃に対して，抽象構文木およびプログラムスライシングを用いた解析手法を適用することで，制御命令に依存せず URL を抽出する手法を提案した．本手法を実装したブラウザエミュレータを用いて評価実験を行い，少ないオーバーヘッドでより多くの新たな URL を抽出できていることを確認した．本手法で抽出した新たな URL は，攻撃コード収集や高対話型ハニークライアントによるマルウェア収集等に活用できる．

参考文献

[1] 秋山満昭, 神薗雅紀, 松木隆宏, 畑田充弘, “マルウェア対策のための研究用データセット 〜MWS Datasets 2014〜,” 情報処理学会 研究報告コンピュータセキュリティ Vol. 2014-CSEC-66, No. 19, pp. 1 – 7, 2014.

[2] N. Provos, et al., “All Your iFRAMEs Point to Us,” In Proceedings of the USENIX Security Symposium, 2008.

[3] M. Akiyama, et al., “Design and Implementation of High Interaction Client Honeypot for Drive-by-Download Attacks,” IEICE TRANS. COMMUN., 2010.

[4] M. Cova, et al., “Detection and Analysis of Drive-by-Download Attack and JavaScript Code,” In Proceedings of the International World Wide Web Conference (WWW), 2010.

[5] M. A. Rajab, et al., “Trends in Circumventing Web-Malware Detection,” Google Technical Report 2011, Jusly 2011.

[6] 高田ほか, “ドライブバイダウンロード攻撃に使用される悪性な JavaScript の実態調査,” 信学技報, ICSS2013-72, 2014.

[7] Gargoyle Software Inc., “HtmlUnit,’ http://htmlunit.sourceforge.net/

[8] Mozilla Developer Network, “Rhino,” http://www.mozilla.org/rhino

[9] M. Weiser, “Program slicing,” In Proceedings of the 5th International Conference on Software Engineering. IEEE Press, 1981.

[10] Y. Takata, et al., “Extracting Redirect-Chain Variations in Drive-by Download Attacks Using Emulation of Various Client Environments,” In the USENIX Security Symposium: Poster Session, 2014.

[11] Q. Wang, et al., “Extracting URLs from JavaScript via program analysis,” In Proceedings of the European Software Engineering Conference/the Foundations of Software Engineering (ESEC/FSE), 2013.

[12] C. Kolbitsch, et al., “ROZZLE: De-Cloaking Internet Malware,” In Proceeings of the IEEE Symposium on Security and Privacy (SP), 2012.

[13] G. De Maio, et al., “PExy: The other side of Exploit Kits,” In Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), 2014.