Revolver: An Automated Approach to the Detection of Evasive Web-based Malware

Alexandros Kapravelos and Yan Shoshitaishvili, *University of California, Santa Barbara;*
Marco Cova, *University of Birmingham;*
Christopher Kruegel and Giovanni Vigna, *University of California, Santa Barbara*

This paper is included in the Proceedings of the
22nd USENIX Security Symposium.

August 14–16, 2013 • Washington, D.C., USA

ISBN 978-1-931971-03-4

Open access to the Proceedings of the
22nd USENIX Security Symposium
is sponsored by USENIX

Revolver: An Automated Approach to the Detection of Evasive Web-based Malware

Alexandros Kapravelos
UC Santa Barbara
kapravel@cs.ucsb.edu

Yan Shoshitaishvili
UC Santa Barbara
yans@cs.ucsb.edu

Marco Cova
University of Birmingham
m.cova@cs.bham.ac.uk

Christopher Kruegel
UC Santa Barbara
chris@cs.ucsb.edu

Giovanni Vigna
UC Santa Barbara
vigna@cs.ucsb.edu

Abstract

In recent years, attacks targeting web browsers and their plugins have become a prevalent threat. Attackers deploy web pages that contain exploit code, typically written in HTML and JavaScript, and use them to compromise unsuspecting victims. Initially, static techniques, such as signature-based detection, were adequate to identify such attacks. The response from the attackers was to heavily obfuscate the attack code, rendering static techniques insufficient. This led to dynamic analysis systems that execute the JavaScript code included in web pages in order to expose malicious behavior. However, today we are facing a new reaction from the attackers: *evasions*. The latest attacks found in the wild incorporate code that detects the presence of dynamic analysis systems and try to avoid analysis and/or detection.

In this paper, we present *Revolver*, a novel approach to automatically detect evasive behavior in malicious JavaScript. *Revolver* uses efficient techniques to identify similarities between a large number of JavaScript programs (despite their use of obfuscation techniques, such as packing, polymorphism, and dynamic code generation), and to automatically interpret their differences to detect evasions. More precisely, *Revolver* leverages the observation that two scripts that are similar should be classified in the same way by web malware detectors (either both scripts are malicious or both scripts are benign); differences in the classification may indicate that one of the two scripts contains code designed to evade a detector tool.

Using large-scale experiments, we show that *Revolver* is effective at automatically detecting evasion attempts in JavaScript, and its integration with existing web malware analysis systems can support the continuous improvement of detection techniques.

1 Introduction

In the last several years, we have seen web-based malware—malware distributed over the web, exploiting vulnerabilities in web browsers and their plugins—becoming a prevalent threat. Microsoft reports that it detected web-based exploits against over 3.5 million distinct computers in the first quarter of 2012 alone [22]. In particular, drive-by-download attacks are the method of choice for attackers to compromise and take control of victim machines [12, 29]. At the core of these attacks are pieces of malicious HTML and JavaScript code that launch browser exploits.

Recently, a number of techniques have been proposed to detect the code used in drive-by-download attacks. A common approach is the use of honeyclients (specially instrumented browsers) that visit a suspect page and extract a number of features that help in determining if a page is benign or malicious. Such features can be based on static characteristics of the examined code [5, 7], on specifics of its dynamic behavior [6, 20, 25, 28, 32, 40], or on a combination of static and dynamic features [34].

Drive-by downloads initially contained only the code that exploits the browser. This approach was defeated by static detection of the malicious code using signatures. The attackers started to obfuscate the code in order to make the attacks impossible to be matched by signatures. Obfuscated code needs to be executed by a JavaScript engine to truly reveal the final code that performs the attack. This is why researchers moved to dynamic analysis systems which execute the JavaScript code, deobfuscating this way the attacks regardless of the targeted vulnerable browser or plugin. As a result, the attackers have introduced evasions: JavaScript code that detects the presence of the monitoring system and behaves differently at runtime. Any diversion from the original targeted vulnerable browser (e.g., missing functionality, additional objects, etc.) can be used as an evasion.

As a result, malicious code is not a static artifact that, after being created, is reused without changes. To the contrary, attackers have strong motivations to modify the code they use so that it is more likely to evade the defense mechanisms employed by end-users and security researchers, while continuing to be successful at exploiting vulnerable browsers. For example, attackers may obfuscate their code so that it

does not match the string signatures used by antivirus tools (a situation similar to the polymorphic techniques used in binary malware). Attackers may also mutate their code with the intent of evading a specific detection tool, such as one of the honeyclients mentioned above.

This paper proposes *Revolver*, a novel approach to automatically identify evasions in drive-by-download attacks. In particular, given a piece of JavaScript code, *Revolver* efficiently identifies scripts that are *similar* to that code, and automatically classifies the differences between two scripts that have been determined to be similar. *Revolver* first identifies syntactic-level differences in similar scripts (e.g., insertion, removal, or substitution of snippets of code). Then *Revolver* attempts to explain the semantics of such differences (i.e., their effect on page execution). We show that these changes often correspond to the introduction of evasive behavior (i.e., functionality designed to evade popular honeyclient tools).

There are several challenges that *Revolver* needs to address to make this approach feasible in practice. First, typical drive-by-download web pages serve malicious code that is heavily obfuscated. The code may be mutated from one visit to the page to the next by using simple polymorphic techniques, e.g., by randomly renaming variables and functions names. Polymorphism creates a multitude of differences in two pieces of code. From a superficial analysis, two functionally identical pieces of code will appear as very different. In addition, malicious code may be produced on-the-fly, by dynamically generating and executing new code (through JavaScript and browser DOM constructs such as the `eval()` and `setTimeout()` functions). Dynamic code generation poses a problem of coverage; that is, not all JavaScript code may be readily available to the analyzer. Therefore, a naive approach that attempts to directly compare two malicious scripts would be easily thwarted by these obfuscation techniques and would fail to detect their similarities. Instead, *Revolver* dynamically monitors the execution of JavaScript code in a web page so that it can analyze both the scripts that are statically present in the page and those that are generated at runtime. In addition, to overcome polymorphic mutations of code, *Revolver* performs its similarity matching by analyzing the Abstract Syntax Tree (AST) of code, thereby ignoring superficial changes to its source code.

Another challenge that *Revolver* must address is scalability. For a typical analysis of a web page, *Revolver* needs to compare several JavaScript scripts (more precisely, their ASTs) with a repository of millions of ASTs (potential matches) to identify similar ones. To make this similarity matching computationally efficient, we use a number of machine learning techniques, such as dimensionality reduction and clustering algorithms.

Finally, not all code changes are security-relevant. For example, a change in a portion of the code that is never executed is less interesting than one that causes a difference in the runtime behavior of the script. In particular, we are interested in identifying code changes that cause detection tools to misclassify a malicious script as benign. To identify such evasive code changes, *Revolver* focuses on modifications that introduce control flow changes in the program. These changes may indicate that the modified program checks whether it is being analyzed by a detector tool (rather than an unsuspecting visitor) and exhibits a different behavior depending on the result of this check.

By automatically identifying code changes designed to evade drive-by-download detectors, one can improve detection tools and increase their detection rate. We also leverage *Revolver* to identify benign scripts (e.g., well-known libraries) that have been injected with malicious code, and, thus, display malicious behavior.

This paper makes the following contributions:

1. **Code similarity detection**: We introduce techniques to efficiently identify JavaScript code snippets that are similar to each other. Our tool is resilient to obfuscation techniques, such as polymorphism and dynamic code generation, and also pinpoints the precise differences (changes in their ASTs) between two different versions of similar scripts.
2. **Detection of evasive code**: We present several techniques to automatically classify differences between two similar scripts to highlight their purpose and effect on the executed code. In particular, *Revolver* has identified several techniques that attackers use to evade existing detection tools by continuously running in parallel with a honeyclient.

2 Background and Overview

To give the reader a better understanding of the motivation for our system and the problems that it addresses, we start with a discussion of malicious JavaScript code used in drive-by-download attacks. Moreover, we present an example of the kind of code similarities that we found in the wild.

Malicious JavaScript code. The web pages involved in drive-by-download attacks typically include malicious JavaScript code. This code is usually obfuscated, and it fingerprints the visitor's browser, identifies vulnerabilities in the browser itself or the plugins that the browser uses, and finally launches one or more exploits. These attacks target memory corruption vulnerabilities or insecure APIs that, if successfully exploited, enable the attackers to execute arbitrary code of their choice.

Figure 1 shows a portion of the code used in a recent drive-by-download attack against users of the Internet Explorer browser. The code (slightly edited for the sake of clarity) instantiates a shellcode (Line 8) by concatenating the variables defined at Lines 1–7; a later portion of the code (not shown in the figure) triggers a memory corruption

```
1 var nop="%uyt9yt2yt9yt2";
2 var nop=(nop.replace(/yt/g,""));
3 var sc0="%ud5db%uc9c9%u87cd...";
4 var sc1="%"+"yutianu"+"ByutianD"+ ...;
5 var sc1=(sc1.replace(/yutian/g,""));
6 var sc2="%"+"u"+"54"+"FF"+...+"8"+"E"+"E";
7 var sc2=(sc2.replace(/yutian/g,""));
8 var sc=unescape(nop+sc0+sc1+sc2);
```

Figure 1: Malicious code that sets up a shellcode.

vulnerability, which, if successful, causes the shellcode to be executed.

A common approach to detect such attacks is to use honeyclients, which are tools that pose as regular browsers, but are able to analyze the code included in the page and the side-effects of its execution. More precisely, low-interaction honeyclients emulate regular browsers and use various heuristics to identify malicious behavior during the visit of a web page [6, 13, 25]. High-interaction honeyclients consist of full-featured web browsers running in a monitoring environment that tracks all modifications to the underlying system, such as files created and processes launched [28, 38, 40]. If any unexpected modification occurs, it is considered to be a manifestation of a successful exploit. Notice that this sample is difficult to detect with a signature, as strings are randomized on each visit to the compromised site.

Evasive code. Attackers have a vested interest in crafting their code to evade the detection of analysis tools, while remaining effective at exploiting regular users. This allows their pages to stay "under the radar" (and actively malicious) for a longer period of time, by avoiding being included in blacklists such as Google's Safe Browsing [11] or being targeted by take-down requests.

Attackers can use a number of techniques to avoid detection [31]: for example, code obfuscation is effective against tools that rely on signatures, such as antivirus scanners; requiring arbitrary user interaction can undermine high-interaction honeyclients; probing for arcane characteristics of browser features (likely not correctly emulated in browser emulators) can thwart low-interaction honeyclients.

An effective way to implement this kind of circumventing techniques consists of adding some specialized "evasive code" whose only purpose is to cause detector tools to fail on an existing malicious script. Of course, the evasive code is designed in such a way that regular browsers (used by victims) effectively ignore it. Such evasive code could, for example, pack an exploit code in an obfuscation routine, check for human interaction, or implement a test for detecting browser emulators (such evasive code is conceptually similar to "red pills" employed in binary malware to detect and evade commonly-used analysis tools [10]).

Figure 2 shows an evasive modification to the original exploit of Figure 1, which we also found used in the wild. More precisely, the code tries to load a non-existent ActiveX

```
1  try {
2      new ActiveXObject("yutian");
3  } catch (e) {
4      var nop="%uyt9yt2yt9yt2";
5      var nop=(nop.replace(/yt/g,""));
6      var sc0="%ud5db%uc9c9%u87cd...";
7      var sc1="%"+"yutianu"+"ByutianD"+ ...;
8      var sc1=(sc1.replace(/yutian/g,""));
9      var sc2="%"+"u"+"54"+"FF"+...+"8"+"E"+"E";
10     var sc2=(sc2.replace(/yutian/g,""));
11     var sc=unescape(nop+sc0+sc1+sc2);
12 }
```

Figure 2: An evasion using non-existent ActiveX controls.

control, named `yutian` (Line 2). On a regular browser, this operation fails, triggering the execution of the catch branch (Lines 4–11), which contains an identical copy of the malicious code of Figure 1. However, low-interaction honeyclients usually emulate the ActiveX API by simulating the presence of *any* ActiveX control. In these systems, the loading of the ActiveX control does not raise any exception; as a consequence, the shellcode is not instantiated correctly, which stops the execution of the exploits and causes the honeyclient to fail to detect the malicious activity.

Detecting evasive code using code similarity. Code similarity approaches have been proposed in the past, but none of them has focused specifically on malicious JavaScript. There are several challenges involved when processing malicious JavaScript for similarities. Attackers actively try to trigger parsing issues in analyzers. The code is usually heavily obfuscated, which means that statically examining the code is not enough. The malicious code itself is designed to evade signature detection from antivirus products. This renders string-based and token-based code similarity approaches ineffective against malicious JavaScript. We will show later how regular code similarity tools, such as Moss [37], fail when analyzing obfuscated scripts. In *Revolver*, we extend tree-based code similarity approaches and focus on making our system robust against malicious JavaScript. We elaborate on our novel code similarity techniques in §3.4.

At a high-level overview, we use *Revolver* to detect and understand the similarity between two code scripts. Intuitively, *Revolver* is provided with the code of both scripts and their classification by one or more honeyclient tools. In our running example, we assume that the code in Figure 1 is flagged as malicious and the one in Figure 2 as benign. *Revolver* starts by extracting the Abstract Syntax Tree (AST) corresponding to each script. *Revolver* inspects the ASTs rather than the original code samples to abstract away possible superficial differences in the scripts (e.g., the renaming of variables). When analyzing the AST of Figure 2, it detects that it is similar to the AST of the code in Figure 1. The change is deemed to be interesting, since it introduces a difference (the try-catch statement) that may cause a change in the control flow of the original program. Our system also determines that the added code (the statement that tries to

load the ActiveX control) is indeed executed by tools visiting the page, thus increasing the relevance of the detected change (*execution bits* are described in more detail in §3.1). Finally, *Revolver* classifies the modification as a possible evasion attempt, since it causes the honeyclient to change its detection result (from malicious to benign).

Assumptions and limitations. Our approach is based on a few assumptions. *Revolver* relies on external detection tools to collect (and make available) a repository of JavaScript code, and to provide a classification of such code as either malicious or benign (i.e., *Revolver* is not a detection tool by itself). To obtain code samples and classification scores, we can rely on several publicly-available detectors [6, 13, 25].

Attackers might write a brand new attack with all components (evasion, obfuscation, exploit code) written from scratch. In such cases, *Revolver* will not be able to find any similarities the first time it analyzes these attacks. The lack of similarities though can be used to our advantage, since we can isolate brand-new attacks (provided that they can be identified by other means) based on the fact that we have never observed such code before.

In the same spirit, to detect evasions, *Revolver* needs to inspect two versions of a malicious script: the "regular" version, which does not contain evasive code, and the "evasive" version, which attempts to circumvent detection tools. Furthermore, if an evasion is occurring, we assume that a detection tool would classify these two versions differently. In particular, if only the evasive version of a JavaScript program is available, *Revolver* will not be able to detect this evasion. We consider this condition to be unlikely. In fact, trend results from a recent Google study on circumvention [31] suggest that malicious code evolves over time to incorporate more sophisticated techniques (including evasion). Thus, having a sufficiently large code repository should allow us to have access to both regular and evasive versions of a script. Furthermore, we have anecdotal evidence of malware authors creating different versions of their malicious scripts and submitting them to public analyzers, until they determine that their programs are no longer detected (this situation is reminiscent of the use of anti-antivirus services in the binary malware world [18]).

Revolver is not effective when server-side evasion (for example, IP cloaking) is used: in such cases, the malicious web site does not serve at all the malicious content to a detector coming from a blocked IP address, and, therefore, no analysis of its content is possible. This is a general limitation of all analysis tools and can be solved by means of a better analysis infrastructure (for example, by visiting malicious sites from IP addresses and networks that are not known to be associated with analysts and security researchers and cannot be easily fingerprinted by attackers).

3 Approach

In this section, we describe *Revolver* in detail, focusing on the techniques that it uses to find similarities between JavaScript files.

A high-level overview of *Revolver* is presented in Figure 3. First, we leverage an existing drive-by-download detection tool (an "Oracle") to collect datasets of both benign and malicious web pages (§3.1). Second, *Revolver* extracts the ASTs (§3.2) of the JavaScript code contained in these pages and, leveraging the Oracle's classification for the code that contains them, marks them as either benign or malicious. Third, *Revolver* computes a similarity score for each pair of ASTs, where one AST is malicious and the other one can be either benign or malicious (§3.3–§3.4). Finally, pairs that are found to have a similarity score higher than a given threshold are further analyzed to identify and classify their similarities (§3.5).

If *Revolver* finds similarities between two malicious scripts, then we classify this case as an instance of *evolution* (typically, an improvement of the original malicious code). On the other hand, if *Revolver* detects similarities between a malicious and a benign script, it performs an additional classification step. In particular, similarities can be classified by *Revolver* into one of four possible categories: *evasions*, *injections*, *data dependencies*, and *general evolutions*. We are especially interested in identifying evasions, which indicate changes that cause a script that had been found to be malicious before to be flagged as benign now.

It is important to note that, due to JavaScript's ability to produce additional JavaScript code on the fly (which enables extremely complex JavaScript packers and obfuscators), performing this analysis statically would not be possible. *Revolver* works dynamically, by analyzing all JavaScript code that is compiled in the course of a web page's execution. By including all these scripts, and the relationships between them (such as what code created what other code), *Revolver* is able to calculate JavaScript similarities among malicious web pages to an extent that is not, to our knowledge, possible with existing state-of-the-art code comparison tools.

3.1 Oracle

Revolver relies on existing drive-by-download detection tools for a single task: the classification of scripts in web pages as either malicious or benign. Notice that our approach is not tied to a specific detection technique or tool; therefore, we use the term "Oracle" to generically refer to any such detection system. In particular, several popular low- and high-interaction honeyclients (e.g., [6, 13, 25, 38]) or any antivirus scanner can readily be used for *Revolver*.

Revolver analyzes the Abstract Syntax Trees (ASTs) of individual scripts rather than examining web pages as a whole. Therefore, *Revolver* performs a refinement step, in which

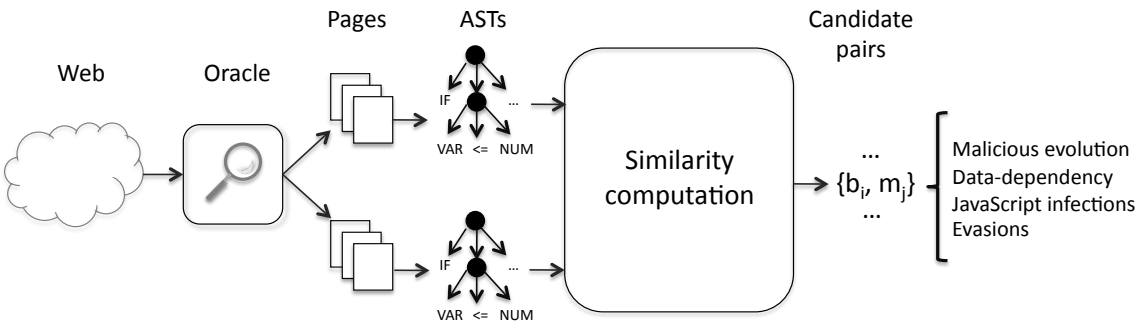

Figure 3: Architecture of *Revolver*.

i) individual ASTs are extracted from the web pages obtained from the Oracle, ii) their detection status is determined (that is, each AST is classified as either benign or malicious), based on the page classification provided by the Oracle, and iii) for each node in an AST, it is recorded whether the corresponding statement was executed. Of course, if an Oracle natively provides this fine-grained information, this step can be skipped.

More precisely, *Revolver* executes each web page using a browser emulator based on HtmlUnit [1]. The emulator parses the page and extracts all of its JavaScript content (e.g., the content of `script` tags and the body of event handlers). In particular, the ASTs of the JavaScript code are saved for later analysis. In addition, to obtain the AST of dynamically-generated code, *Revolver* executes the JavaScript code. At the end of the execution, for each node in the AST, *Revolver* keeps an **execution bit** to record whether the code corresponding to that node was executed. Whenever it encounters a function that generates new code (e.g., a call to the `eval()` or `setTimeout()` functions), *Revolver* analyzes the code that is generated by these functions. It also saves the parent-child relationship between scripts, i.e., which script is responsible for the execution of a dynamically-generated script. For example, the script containing the `eval()` call is considered the parent of the script that is evaluated. Similarly, *Revolver* keeps track of which script causes network resources to be fetched, for example, by creating an `iframe` tag.

Second, for each AST, *Revolver* determines if it is malicious or benign, based on the Oracle's input. More precisely, an AST is considered malicious if it is the parent of a malicious AST, or if it issued a web request that led to the execution of malicious code. This makes *Revolver* flexible enough to work with any Oracle.

3.2 Abstract Syntax Trees

Revolver's core analysis is based on the examination of ASTs rather than the source code of a script. The rationale for using ASTs is that they abstract away details that are irrelevant for our analysis (and, in fact, often undesirable), while retaining enough precision to achieve good results.

For example, consider a script obtained from the code in Figure 1 via simple obfuscation techniques: renaming of variables and function names, introduction of comments, and randomization of whitespace. Clearly, we want *Revolver* to consider these scripts as similar. Making this decision can be non-trivial when inspecting the source code of the scripts. In fact, as a simple validation, we ran Moss, a system for determining the similarity of programs, which is often used as a plagiarism detection tool [37], on the original script and the one obtained via obfuscation. Moss failed to flag the two scripts as similar, as shown in the tool's output here [23]. However, the two scripts are identical when their AST representations are considered, since, in the trees, variables are represented by generic `VAR` nodes, independently of their names, and comments and whitespaces are ignored. This makes tree-based code similarity approaches more suitable for malicious JavaScript comparisons (and this is the reason why our analysis leverages ASTs as well). However, as shown in §3.4, we need to treat malicious code in a way that is different from previous techniques targeting benign codebases. Below, we describe our approach and necessary extensions in more detail.

Revolver transforms the AST produced by the JavaScript compiler into a *normalized node sequence*, which is the sequence of node types obtained by performing a pre-order visit of the tree. In total, there are 88 distinct node types, corresponding to different constructs of the JavaScript language. Examples of the node types include `IF`, `WHILE`, and `ASSIGN` nodes.

Figure 4 summarizes the data structures used by *Revolver* during its processing. We discuss *sequence summaries* in the next Section.

3.3 Similarity Detection

After extracting an AST and transforming it in its normalized node sequence, *Revolver* finds similar normalized node sequences. The result is a list of normalized node sequence

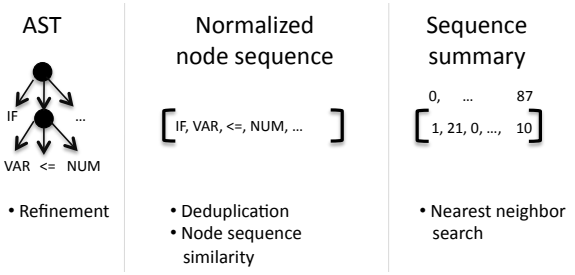

Figure 4: Data structures used by *Revolver*.

pairs. In particular, pairs of malicious sequences are compared to identify cases of evolution; pairs where one of the sequences is benign and the other malicious are analyzed to identify possible evasion attempts.

The similarity computation is based on computing the directed edit distance between two node sequences, which, intuitively, corresponds to the number of operations that are required to transform one benign sequence into the malicious one. Before discussing the actual similarity measurement, we discuss a number of minimization techniques that we use to make the computation of the similarity score feasible in datasets comprising millions of node sequences.

Deduplication. As a first step to reduce the number of similarity computations, we discard duplicates in our dataset of normalized node sequences. Since we use a canonical representation for the ASTs, we can easily and efficiently compute hashes of each sequence, which enables us to quickly identify groups of identical node sequences. In the remaining processing phases, we only need to consider one member of a group of identical node sequences (rather than all of its elements). Notice that identical normalized node sequences may correspond to different scripts, and may also have a different detection classification (we describe such cases in §3.5). Therefore, throughout this processing, we always maintain the association between node sequences and the scripts they correspond to, and whether they have been classified as malicious or benign.

Approximate nearest neighbors. Given a repository of n benign ASTs and m malicious ones, *Revolver* needs to compute $n \times m$ comparisons over (potentially long) node sequences. Even after the deduplication step, this may require a significantly large number of operations.

To address this problem, we introduce the idea of *sequence summaries*. A sequence summary is a compact summarization of a regular normalized node sequence, which stores the number of times each node type appears in the corresponding AST. Since there are 88 distinct node types, each node sequence is mapped into a point in an 88-dimensional Euclidean space. An advantage of sequence summaries is that they bound the length of the objects that will be compared (from potentially very large node sequences, corresponding to large ASTs, down to more manageable vectors of fixed length).

Then, for each sequence summary s, we identify its *malicious neighbors*, that is, up to k malicious sequence summaries t, such that the distance between s and t is less than a chosen threshold τ_n. Intuitively, the malicious neighbors correspond to the set of ASTs that we expect to be most similar to a given AST. Determining the malicious neighbors of a sequence summary is an instance of the k-nearest neighbor search problem, for which various efficient algorithms have been proposed. In particular, we solve it by using the FLANN library [24].

In the remaining step, we compare sequence summaries only with their malicious neighbors, thus dramatically reducing the number of comparison to be performed.

Normalized node sequence similarity. Finally, we can compute the similarity between two normalized node sequences. More precisely, *Revolver* compares the normalized node sequence corresponding to a sequence summary s with each normalized node sequence that corresponds to a sequence summary of the malicious neighbors of s.

The similarity measurement is based on the pattern matching algorithm by Ratcliff et al. [33]. More precisely, given two node sequences, a and b, we first find their longest contiguous common subsequence (LCCS). Then, we recursively search for LCCS between the pieces of a and b to the left and right of the matching subsequence. The similarity of a and b is then returned as the number of nodes in common divided by the total number of nodes in the malicious node sequence. Therefore, identical ASTs will have similarity 1, and the similarity values decrease toward zero as two ASTs contain higher amounts of different nodes. This technique is robust against injections, where one benign script includes a malicious one, since all malicious nodes will be matched.

In addition to a numeric similarity score, the algorithm also provides a list of insertions for the two node sequences, that is, a list of AST nodes that would need to be added to one sequence to transform it into the other one. This information is very useful for our analysis, since it identifies the actual code that was added to an original malicious code.

After the similarity score is computed, we discard any pairs that have a similarity below a predetermined threshold τ_s.

Expansion. Once pairs of ASTs with high similarity have been identified, we need to determine the Oracle's classification of the scripts they originate from. We, therefore, expand out any pairs that we deduplicated in the initial *Deduplication* step so that we associate the AST similarities to the scripts that they correspond to.

3.4 Optimizations

There are several techniques that we utilize to improve the results produced by the similarity detection steps. In particular, our objective is to restrict the pairs identified

as similar to "interesting" ones, i.e., those that are more likely to correspond to evasion attempts or significant new functionality. The techniques introduced here build upon tree-based code similarity approaches and are specific to malicious JavaScript.

Size matters. We observed that JavaScript code contains a lot of very small scripts. In the extreme case, it includes scripts comprising a single statement. We determined that the majority of such scripts are generated dynamically through calls to `eval()`, which, for example, dynamically invoke a second function. Such tiny scripts are problematic for our analysis: they have not enough functionality to perform malicious actions and they end up matching other short scripts, but their similarity is not particularly relevant. As a consequence, we **combine** ASTs that contain less than a set number of nodes (τ_z). We do this by taking into account how a script was generated: if another script generated code under our threshold, we inline the generated script back to its parent. If the script was not dynamically generated, then we treat it as if one script contained all static code under our threshold. This way the attacker cannot split the malicious code into multiple parts under our threshold in order to evade *Revolver*.

Repeated pattern detection. We also observed that, in certain cases, an AST may contain a set of nodes repeated a large number of times. This commonly occurs when the script uses some JavaScript data structure that yields many repeated AST nodes. For example, malicious scripts that unpack or deobfuscate their exploit payload frequently utilize a JavaScript *Array* of elements to store the payload. Their ASTs contain a node for every single element in the *Array*, which, in many cases, may have thousands of instances. An unwanted consequence, then, is that any script with a large *Array* will be considered similar to the malicious script (due to the high similarity of the array nodes), regardless of the presence of a decoding/unpacking routine (which, instead, is critical to determine the similarity of the scripts from a functional point of view). These obfuscation artifacts affect tree-based similarity algorithms, which will result in the detection of similar code pairs where the common parts are of no interest in the context of malicious JavaScript. To avoid this problem, we identify sequences of nodes that are repeated in a script more than a threshold (τ_p) and truncate them.

Similarity fragmentation. Although we have identified blocks of code that are shared across two scripts, it can be the case that these blocks are not continuous. One script can be broken down into small fragments that are matched to the other script in different positions. This is why we take into account the fragmentation of the matching blocks. To prune these cases, we recognize a similarity only if the fragmentation of the similarities is below a set threshold τ_f.

AST	Executed nodes	Classification
$=$	$*$	Data-dependency
$*$	$=$	Data-dependency
$B \subseteq M$	$\neq$	JavaScript injection
$M \subseteq B$	$\neq$	Evasion
$\neq$	$\neq$	General evolution

Table 1: Candidate pairs classification (B is a benign sequence, M is a malicious sequence, $*$ indicates a wildcard value).

3.5 Classification

The outcome of the previous similarity detection step is a list of pairs of scripts that are similar. As we show in §5.1 we can have hundreds of thousands of similar pairs. Therefore, *Revolver* performs a classification step of similar pairs. That is, *Revolver* interprets the changes that were made between two scripts and classifies them. There are two cases, depending on the Oracle's classification of the scripts in a pair. If the pair consists solely of malicious scripts, then we classify the similarity as a malicious evolution. The other case is a pair in which one script is malicious and one script is benign. We call such pairs *candidate pairs* (they need to be further tested before we can classify their differences). While the similarity detection has operated on a syntactic level (essentially, by comparing AST nodes), *Revolver* now attempts to determine the semantics of the differences.

In practice, *Revolver* classifies the scripts and their similarities into one of several categories, corresponding to different cases where an Oracle may flag differently scripts that are similar. Table 1 summarizes the classification algorithm used by *Revolver*.

Data-dependency category. *Revolver* checks if a pair of scripts belongs to the data-dependency category. A typical example of scripts that fall into this category is packers. Packers are tools that allow JavaScript authors to deliver their code in a packed format, significantly reducing the size of the code. In packed scripts, the original code of the script is stored as a compacted string or array, and its clear-text version is retrieved at run-time by executing an unpacking routine. Packers have legitimate uses (mostly, size compression): in fact, several open-source popular packers exist [9], and they are frequently used to distribute packed version of legitimate JavaScript libraries, such as jQuery. However, malware authors also rely on these very same packers to obfuscate their code and make it harder to be fingerprinted.

Notice that the ASTs of packed scripts (generated by the same packer) are identical, independently of their (unpacked) payload: in fact, they consist of the nodes of the unpacking routine (which is fixed) and of the nodes holding the packed data (typically, the assignment of a string literal

to a variable). However, the actual packed contents, which eventually determine whether the script is malicious or benign, are not retained at the AST level of the packer, but the packed content will eventually determine the nature of the overall script as benign or malicious.

Revolver categorizes as data-dependent pairs of scripts that are identical and have different detection classification.

As a slight variation to this scenario, *Revolver* also classifies as data-dependent pairs of scripts for which the ASTs are not identical, but the set of nodes that were actually executed are indeed the same. For example, this corresponds to cases where a function is added to the packer but is never actually executed during the unpacking.

Control-flow differences. The remaining categories are based on the analysis of AST nodes that are different in the two scripts, and, specifically, of nodes representing control-flow statement. We focus on such nodes because they give an attacker a natural way to implement a check designed to evade detection. In fact, such checks generally test a condition and modify the control flow depending on the result of the test.

More precisely, we consider the following control-flow related nodes: *TRY, CATCH, CALL, WHILE, FOR, IF, ELSE, HOOK, BREAK, THROW, SWITCH, CASE, CONTINUE, RETURN, LT (<), LE (<=), GT (>), GE (>=), EQ (==) , NE (! =), SHEQ (===), SNE (! ==), AND, and OR*. Depending on where these control-flow nodes were added, whether in the benign or in the malicious script, a candidate pair can be classified as a JavaScript injection or an evasion. Notice that we leverage here the *execution bits* to detect control flow changes that were actually executed and affected the execution of code that was found as malicious before.

JavaScript injection category. In some cases, malware authors insert malicious JavaScript code into existing benign scripts on a compromised host. This is done because, when investigating a compromise, webmasters may neglect to inspect files that are familiar to them, and thus such injections can go undetected. In particular, it is common for malware authors to add their malicious scripts to the code of popular JavaScript libraries hosted on a compromised site, such as jQuery and SWFObject.

In these cases, *Revolver* identifies similarities between a benign script (the original, legitimate jQuery code) and a malicious script (the library with the added malicious code). In addition, *Revolver* detects that the difference between the two scripts is due to the presence of control-flow nodes in the malicious script (the additional code added to the library), which are missing in the benign script. *Revolver* classifies such similarities as JavaScript injections, since the classification of the analyzed script changes from benign to malicious due to the introduction of additional code in the malicious version of the script.

Evasions category. Pairs of scripts that only differ because of the presence of additional control-flow nodes in the benign script are categorized as evasions. In fact, these correspond to cases where a script, which was originally flagged as malicious by an Oracle, is modified to include some additional functionality that modifies its control flow (i.e., an evasive check) and, as a consequence, appears to be benign to the Oracle.

General evolution cases. Finally, if none of the previous categories applies to the current pair of scripts, it means that their differences are caused by the insertion of control-flow nodes in both the benign and malicious scripts. Unlike similarities in the evasion category, these similarities may signify generic evolution between the two scripts. *Revolver* flags these cases for manual review, at a lower priority than evasive cases.

4 Implementation

In this section, we discuss specific implementation choices for our approach.

We used the Wepawet honeyclient [6] as the Oracle of *Revolver*. In particular, the input to *Revolver* was the web pages processed by the Wepawet tool at real-time together with their detection classification. We used *Revolver* to extract ASTs from the pages analyzed by Wepawet, and to perform the similarity processing described in the previous sections.

As our processing infrastructure, we used a cluster of four worker machines to process submissions in parallel with the Oracle. Notice that all the steps in *Revolver*'s processing can be easily parallelized. In terms of performance, we managed to process up to 591,543 scripts on a single day, which was the maximum number of scripts that we got on a single day from the Oracle during our experiments.

We will now discuss the parameters that can be tuned in our algorithms (discussed in §3), explaining the concrete values we have chosen for our experiments.

Minimum tree size (τ_z). We chose 25 nodes as the minimum size of the AST trees that we will process before combining them to their parent. Smaller ASTs can result from calls to *eval* with tiny arguments, and from calls to short event handlers, such as *onLoad* and *onMouseOver*. We expect that such small ASTs correspond to short scripts that do not implement any interesting functionality alone, but complement the functionality of their parent script.

Minimum pattern size (τ_p). Another threshold that we set is the minimum pattern size. Any node sequence that is repeated more than this threshold is truncated to the threshold value. The primary application of pattern detection is to handle similar packers that decode payloads of different size. We chose 16 for this value, as current packers either work on relatively long arrays (longer than 16, and thus detected) or on single strings (one node, and thus irrelevant to this issue). This amount also excludes the possibility of compressing interesting code sequences, since we rarely see such long

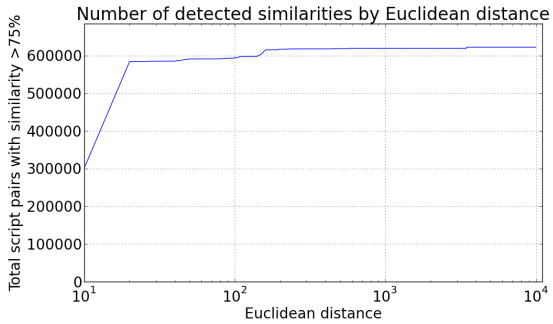

Figure 5: Number of detected similarities as a function of the distance threshold.

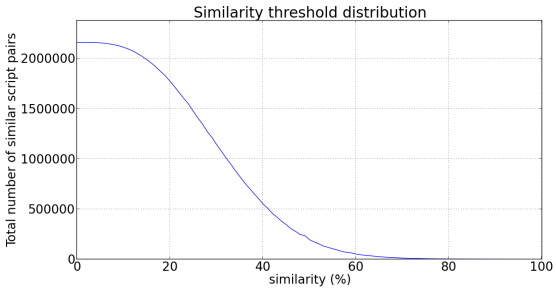

Figure 6: The resulting amount of similarities for different similarity thresholds.

patterns outside of packed payloads. Reducing this value would have the effect of making the tree similarity algorithm much more lax.

Nearest neighbor threshold (τ_n). In the nearest neighbors computation, we discard node sequences that are farther than a given distance d from the node sequence currently being inspected. We empirically determined a value for this parameter, by evaluating various values for d and inspecting the resulting similarities. From Figure 5, it is apparent that the amount of similarities that are detected levels off fairly sharply past $d = 1,000$. We determined that $10,000$ is a safe threshold that includes a majority of trees while allowing the similarity calculation to be computationally feasible.

Normalized node sequence similarity threshold (τ_s). Care has to be taken when choosing the threshold used to identify similar normalized node sequences. Intuitively, if this value is too low, we risk introducing significant noise into our analysis, which will make *Revolver* consider as similar scripts that in reality are not related to each other. On the contrary, if the value is too high, it will discard interesting similarities. Experimentally (see Figure 6), we determined that this occurs for similarity values in the 70%–80% interval. Therefore, we chose 75% as our similarity threshold (in other words, only node sequences that are 75% or more similar are further considered by *Revolver*).

Category	Similar Scripts	# Groups by malicious AST
JavaScript Injections	6,996	701
Data-dependencies	101,039	475
Evasions	4,147	155
General evolutions	2,490	273
Total	114,672	1,604

Table 2: Benign scripts from Wepawet that have similarities with malicious scripts and their classification from *Revolver*.

5 Evaluation

We evaluated the ability of *Revolver* to aid in detecting evasive changes to malicious scripts in real-world scenarios. While *Revolver* can be leveraged to solve other problems, we feel that automatically identifying evasions is the most important contribution to improving the detection of web-based malware.

5.1 Evasions in the wild

Revolver identifies possible evasion attempts by identifying similarities between malicious and benign code. Therefore, *Revolver*'s input is the output of any Oracle that classifies JavaScript code as either malicious or benign. To evaluate *Revolver*, we continuously submitted to *Revolver* all web pages that Wepawet examined. Since September 2012, we collected $6,468,623$ web pages out of which $265,692$ were malicious. We analyzed $20,732,766$ total benign scripts and $186,032$ total malicious scripts. Out of these scripts, we obtained $705,472$ unique benign ASTs and $5,701$ unique malicious ASTs.

Revolver applied the AST similarity analysis described in Section 3, and extracted the pairs of similar ASTs. Table 2 summarizes the results of classifying these similarities in the categories considered by *Revolver*. In particular, *Revolver* identified $6,996$ scripts where malicious JavaScript was injected, $101,039$ scripts with data-dependencies, $4,147$ evasive scripts, and $2,490$ scripts as general evolutions. We observe that many of these scripts can be easily grouped by their similarities with the same malicious script. Therefore, for ease of analysis, we group the pairs by their malicious AST component, and identify 701 JavaScript injections, 475 data-dependencies, 155 evasions, and 273 general evolutions. Our results indicate a high number of malicious scripts that share similarities with benign ones. This is due to the fact that injections and data-dependent malicious scripts naturally share similarities with benign scripts and we are observing many of these attacks in the wild.

To verify the results produced by *Revolver*, we manually analyzed all groups categorized as "evasions". For the

rest of the categories we grouped the malicious ASTs into families based on their similarities with each other and examined a few similar pairs from each family. We found the results for the JavaScript injection and data-dependencies categories to be correct. The reason why *Revolver* classified a large number of scripts as data-dependencies is due to the extensive use of a few popular packers, such as the Edwards' packer [9]. For example, the jQuery library was previously officially distributed in a packed state to reduce its size.

Of the 155 evasions groups, we found that only five were not intended evasion attempts. We cannot describe all evasions in detail here, but we provide a brief summary for the most interesting ones in the next section.

The pairs in the "general evolutions" category consisted of cases where *Revolver* identified control flow changes in both the benign and malicious scripts. We manually looked into them and did not find any behavior that could be classified as evasive.

5.2 Evasions case studies

The evasions presented here exploit differences in the implementation of Wepawet's JavaScript interpreter and the one used by regular browsers. Notice that these evasions can affect Oracles other than Wepawet; in particular, low-interaction honeyclients, such as the popular jsunpack [13] and PhoneyC [25].

We describe in more detail a subset of the evasions that we found from our experiment on real-world data. In the 22 evasion groups described here, we identified seven distinct evasion techniques, and one programming mistake in a malicious PDF.

We found three cases which leveraged subtle details in the handling of regular expressions and Unicode to cause a failure in a deobfuscation routine when executing in the Oracle (on the contrary, regular browsers would not be affected). In another case, the attackers replaced the JavaScript code used to launch an ActiveX exploit code with equivalent VBScript code. This is done because Internet Explorer can interpret VBScript, while most emulators do not support it. In a different case, the evasive code creates a `div` tag and checks for specific CSS properties, which are set correctly in Internet Explorer but not when executing in our Oracle. We will examine in more detail the next four evasion techniques.

Variable scope inside eval. We found that a successful evasion attack can be launched with minor changes to a malicious script. In one such case, shown in Figure 7, the authors of the malicious script changed a *replace* call with a call to `eval`, which, in turn, executed the same *replace*. While this change did not affect the functionality of the script in Internet Explorer, it did change it for our Oracle. In fact, in Wepawet's JavaScript engine, the code inside the `eval` runs in a different scope, and thus, the locally-defined variable on which *replace* is called is not accessible. While

```
1  // Malicious
2  function foo() {
3  ...
4  W6Kh6V5E4 = W6Kh6V5E4.replace(/\W/g,Bm2v5BSJE);
5  ...
6  }
7  // Evasion
8  function foo(){
9  ...
10 var enryA = mxNEN+F7B07;
11 F7B07 = eval;
12 {}
13 enryA = F7B07('enryA.rep' + 'lace(/\\W/g,CxFHg)↩
       ');
14 ...
15 }
```

Figure 7: Evasion based on differences in the scope handling inside `eval` in different JavaScript engines.

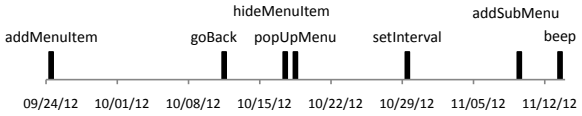

Figure 8: Timeline of PDF evasions automatically detected by *Revolver*.

the code successfully exploits Internet Explorer, it fails in our Oracle and is marked as benign.

Adobe-specific JavaScript execution. Figure 9 shows an evasion that leverages a specific characteristic of the JavaScript engine implementation included in Adobe Reader. In particular, in Adobe's engine (unlike other engines), the `eval` function is accessible as a property of native objects, e.g., a string (line 8) [2]. Wepawet's analyzer is not aware of this difference and fails on scripts that make use of this feature (marking them as benign). The functionally-identical script that does not use this trick, but directly invokes the `eval()` function, is correctly marked as malicious. We also found several instances of evasions related to PDF specific objects, like *app* and *target*, where missing functionality was used to render the malicious script harmless. We show a snippet of this evasion type found in the wild in Figure 10.

In Figure 8 we see the evasions related to the *app* object that were automatically detected by *Revolver* as found in the wild. Every time *Revolver* detected an evasion there is a spike in the figure, and we made the appropriate patches to Wepawet as soon as possible. What is of particular interest is the fact that attackers react to Wepawet's patching by introducing a new evasion within a few days, making a tool like *Revolver* necessary to automatically keep track of this behavior and keep false negative detections as low as possible.

Evasion through exceptions. Another interesting evasion that *Revolver* found also leverages subtle differences in the JavaScript implementation used in Wepawet and in real browsers. In this case, the malicious script consists of a decryption function and a call to that function. The function

```
1 // Malicious
2 OlhG='evil_code'
3 wTGB4=eval
4 wTGB4(OlhG)
5
6 // Evasion
7 OlhG='evil_code'
8 wTGB4="this"["eval"]
9 wTGB4(OlhG)
```

Figure 9: Evasion based on the ability to access the `eval` function as a property of native objects in Adobe's JavaScript engine.

```
1 if((app.setInterval+/**/"")["indexOf"](aa)!=-1){
2     a=/**/target.creationDate.split('|')[0];}
```

Figure 10: Evasion based on PDF specific objects *app* and *target*.

first initializes a variable with a XOR key, which will be used to decrypt a string value (encoding a malicious payload). The decoded payload is then evaluated via `eval`.

The evasion that we found follows the same pattern (see Figure 11), but with a few critical changes. In the modified code, the variable containing the XOR key is only initialized the first time that the decryption function runs; in sequential runs, the value of the key is modified in a way that depends on its prior value (Lines 16–17). After the key computation, a global variable is accessed. This variable is not defined the first time the decryption function is called, so that the function exits with an exception (Line 19). On Internet Explorer, this exception is caught, the variable is defined, and the decryption function is called again. The function then runs through the key calculation and then decrypts and executes the encrypted code by calling `eval`.

On our Oracle, a subtle bug (again, in the handling of `eval` calls) in the JavaScript engine caused the function to throw an exception the first *two* times that it was called. When the function is called the third time, it finally succeeds, modifies the XOR key, and attempts to decrypt the string. However, since the key calculation is run *three* times instead of two, the key is incorrect, and the decrypted string results in garbage data. We found three variations of this technique in our experiments.

A very interesting exception-based evasion that we found with *Revolver* was based on the immutability of *window.document.body*. The attacker checks if she can replace the *body* object with a *string*, something that should not be possible and should result in an exception, but it does not raise an exception in our Oracle because the *body* object is mutable. The interesting part is that we found three completely different malicious scripts evolving to incorporate this evasion, one of them being part of the popular exploit kit *Blackhole 2.0*. This is the first indication that evasion techniques are propagating to different attacking components and indicates that attackers care to keep their attacks as stealthy as possible.

```
1 // Malicious
2 function deobfuscate(){
3     ... // Define var xorkey and compute its ↩
          value
4     for(...) { ... // XOR decryption with xorkey↩
           }
5     eval(deobfuscated_string);
6 }
7 try {
8   eval('deobfuscate();')
9 }
10 catch (e){
11   alert('err');
12 }
13
14 // Evasion
15 function deobfuscate(){
16     try { ... // is variable xorkey defined? }
17     catch(e){ xorkey=0; }
18     ... // Compute value of xorkey
19     VhplKO8 += 0; // throws exception the first ↩
        time
20     for(...) { ... // XOR decryption with xorkey↩
           }
21     eval(deobfuscated_string);
22 }
23 try { eval('deobfuscate();') } // 1st call
24 catch (e){
25   // Variable VhplKO8 is not defined
26   try {
27     VhplKO8 = 0; // define variable
28     eval('deobfuscate();'); // 2nd call
29   }
30   catch (e){
31     alert('ere');
32   }
33 }
```

Figure 11: An evasion taking advantage of a subtle bug in Wepawet's JavaScript engine in order to protect the XOR key.

Unicode deobfuscation evasion. This evasion leveraged the fact that Unicode strings in string initializations and regular expressions are treated differently by different JavaScript engines. For example, *Revolver* found two scripts with a similarity of 82.6%. The script flagged as benign contained an additional piece of code that modified the way a function reference to `eval` was computed. More precisely, the benign script computed the reference by performing a regular expression replacement. While this operation executes correctly in Internet Explorer, it causes an error in the JavaScript engine used by Wepawet due to a bug in the implementation of regular expressions.

Incorrect PDF version check. Another similarity that *Revolver* identified involved two scripts contained inside two PDF files, one flagged as benign by Wepawet and the other as malicious. These scripts had a similarity of 99.7%. We determined that the PDF contained an exploit targeting Adobe Reader with versions between 7.1 and 9. The difference found by *Revolver* was caused by an incorrect version check in the exploit code. The benign code mistakenly checked for version greater or equal to 9 instead of less or equal to 9,

which combined with the previous checks for the version results in an impossible browser configuration and as a consequence the exploit was never fired. This case, instead of being an actual evasion, is the result of a mistake performed by the attacker. However, the authors quickly fixed their code and re-submitted it to Wepawet just 13 minutes after the initial, flawed submission.

False positives. The evasion groups contained five false positives. In this context, a false positive means that the similarity identified by *Revolver* is not responsible for the Oracle's misdetection. More precisely, of these false positives, four corresponded to cases where the script execution terminated due to runtime JavaScript errors before the actual exploit was launched. While such behavior could be evasive in nature, we determined that the errors were not caused by any changes in the code, but by other dependencies. These can be due to missing external resources required by the exploit or because of a runtime error. In the remaining case, the control-flow change identified by *Revolver* was not responsible for the misdetection of the script.

Revolver's impact on honeyclients. By continuously running *Revolver* in parallel with a honeyclient, we can improve the honeyclient's accuracy by observing the evolution of malicious JavaScript. The results from such an integration with Wepawet indicate a shift in the attackers' efforts from hardening their obfuscation techniques to finding discrepancies between analysis systems and targeted browsers. Popular exploit kits like *Blackhole* are adopting evasions to avoid detection, which shows that such evasions have emerged as a new problem in the detection of malicious web pages. *Revolver*'s ability to pinpoint, with high accuracy, these new techniques out of millions of analyzed scripts not only gives a unique view into the attackers' latest steps, but indicates the necessity of such system as part of any honeyclient that analyzes web malware.

6 Discussion

As with any detection method, malware authors could find ways to attempt to evade *Revolver*. One possibility consists in splitting the malicious code into small segments, each of which would be interpreted separately through `eval`. *Revolver* is resilient against code fragmentation like this because it combines such scripts back to the parent script that generated them, reconstructing this way the original non-fragmented script.

It is also possible for malware authors to purposefully increase the Euclidean distance between their scripts so that otherwise similar scripts are no longer considered neighbors by the nearest neighbor algorithm. For example, malware authors could swap statements in their code, or inject junk code that has no effect other than decreasing the similarity score. Attackers could also create fully metamorphic scripts, similar to what some binary malware does [19]. We can counteract these attacks by improving the algorithms we use to compute the similarity of scripts. For example, we could use a preprocessing step to normalize a script's code (e.g., removing dead code). A completely different approach would be to leverage *Revolver* to correlate differences in the code of the same web pages when visited by multiple oracles: if *Revolver* detects significant differences in the code returned during these visits, then we can identify metamorphic web pages. In addition, metamorphic code raises the bar, since an attack needs to be programmatically different every time, and the code must be automatically generated without clearly-detectable patterns. Therefore, this would force attackers to give up their current obfuscation techniques and ability to reuse code.

An attacker could include an evasion and dynamically generate the attack code only if the evasion is successful. The attacker has two options: He can include the evasion code as the first step of the attack, or after initial obfuscation and environment setup. Evasions are hard to find and require significant manual effort by the attackers. Therefore, attackers will not reveal their evasion techniques since they are almost as valuable as the exploits they deliver. Moreover, introducing unobfuscated code compromises the stealthiness of the attack and can yield into detection through signature matching. The second option works in *Revolver*'s favor, since it allows our system to detect similarities in obfuscation and in environmental setup code.

Finally, an operational drawback of *Revolver* is the fact that manual inspection of the similarities that it identifies is currently needed to confirm the results it produces. The number of similarities that were found during our experiments made it possible to perform such manual analysis. In the future, we plan to build tools to support the inspection of similarities and to automatically confirm similarities based on previous analyses.

7 Related Work

Detection of evasive code. The detection of code that behaves differently when run in an analysis environment than when executed on a regular machine is a well-known problem in the binary malware community. A number of techniques have been developed to check if a binary is running inside an emulator or a virtual machine [10, 30, 36]. In this context, evasive code consists of instructions that produce different results or side-effects on an emulator and on a real host [21, 26]. The original malware code is modified to run these checks: if the check identifies an analysis system, the code behaves in a benign way, thus evading the detection.

Researchers have dealt with such evasive checks in two ways. First, they have designed systems that remain transparent to a wide range of malware checks [8, 39]. Second, they have developed techniques to detect the presence of such checks, for example by comparing the behavior of a sample

on a reference machine with that obtained by running it on an analysis host [3, 15].

Similar to the case of evasions against binary analysis environments, the results produced by honeyclients (i.e., the classification of a web page as either malicious or benign) can be confused by sufficiently-sophisticated evasion techniques. Honeyclients are not perfect and attackers have found ways to evade them [16, 31, 40]. For example, malicious web pages may be designed to launch an exploit only after they have verified that the current visitor is a regular user, rather than an automated detection tool. A web page may check that the visitor performs some activity, such as moving the mouse or clicking on links, or that the browser possesses the idiosyncratic properties of commonly-used modern browsers, rather than being a simple emulator. If any of these checks are not satisfied, the malicious web page will refrain from launching the attack, and, as a consequence, will be incorrectly classified as benign, thus evading detection.

The problem of evasive code in web attacks has only recently been investigated. Kolbitsch et al. [17] have studied the "fragility" of malicious code, i.e., its dependence for correct execution on the presence of a particular execution environment (e.g., specific browser and plugins versions). They report several techniques used by malicious code for environment matching: some of these techniques may well be used to distinguish analysis tools from regular browsers and evade detection. They propose ROZZLE, a system that explores multiple execution paths in a program, thus bypassing environment checks. Rozzle only detects fingerprinting that leverages control flow branches and depends upon the environment. It can be evaded by techniques that do not need control-flow branches, e.g., those based on browser or JavaScript quirks. For example, the property `window.innerWidth` contains the width of the browser window viewport in Firefox and Chrome, and is undefined in Internet Explorer. Therefore, a malicious code that initialized a decoding key as *xorkey=window.innerWidth*0+3* would compute a different result for xorkey in Firefox/Chrome (3) and IE (Not a Number error), and could be used to decode malicious code in specific browsers. Rozzle will not trigger its multi-path techniques in such cases and can be evaded.

Revolver takes a different approach to identifying evasive code in JavaScript programs. Instead of forcing an evasive program to display its full behavior (by executing it in parallel on a reference host and in an analysis environment [3], or by forcing the execution through multiple, interesting paths [17]), it leverages the existence of two distinct but similar pieces of code and the fact that, despite their similarity, they are classified differently by detection tools. In addition, *Revolver* can precisely and automatically identify the code responsible for an evasion.

JavaScript code analysis. In the last few years, there have been a number of approaches to analyzing JavaScript code. For example, Prophiler [5] and ZOZZLE [7] have used characteristics of JavaScript code to predict if a script is malicious or benign. ZOZZLE, in particular, leverages features associated with AST context information (such as, the presence of a variable named `scode` in the context of a loop), for its classification.

Cujo [34] uses static and dynamic code features to identify malicious JavaScript programs. More precisely, it processes the static program and traces of its execution into q-grams that are classified using machine learning techniques.

Revolver performs the core of its analysis statically, by computing the similarity between pairs of ASTs. However, *Revolver* also relies on dynamic analysis, in particular to obtain access to the code generated dynamically by a script (e.g., via the `eval()` function), which is a common technique used by obfuscated and malicious code.

Code similarity. The task of automatically detecting "clones," i.e., segments of code that are similar (according to some notion of similarity), is an established line of work in the software engineering community [27, 35]. Unfortunately, many of the techniques developed here assume that the code under analysis is well-behaved or at least not adversarial, that is, not actively trying to elude the classification. Of course, this assumption does not hold when examining malicious code.

Similarity between malicious binaries has been used to quickly identify different variants of the same malware family. The main challenge in this context is dealing with extremely large numbers of samples without source code and large feature spaces from runtime data. Different techniques have been proposed to overcome these issues: for example, Bayer et al. [4] rely on locality sensitive hashing to reduce the number of items to compare, while Jong et al. [14] use feature hashing to reduce the number of features.

As a comparison, *Revolver* aims not only to identify pieces of JavaScript code that are similar, but also to understand why they differ and especially if these differences are responsible for changing the classification of the sample.

8 Conclusions

In this paper, we have introduced and demonstrated *Revolver*, a novel approach and tool for detecting malicious JavaScript code similarities on a large scale. *Revolver*'s approach is based on identifying scripts that are similar and taking into account an Oracle's classification of every script. By doing this, *Revolver* can pinpoint scripts that have high similarity but are classified differently (detecting likely evasion attempts) and improve the accuracy of the Oracle.

We performed a large-scale evaluation of *Revolver* by running it in parallel with the popular Wepawet drive-by-detection tool. We identified several cases of evasions that are used in the wild to evade this tool (and, likely, other tools

based on similar techniques) and fixed them, improving this way the accuracy of the honeyclient.

Acknowledgements: This work was supported by the Office of Naval Research (ONR) under grant N00014-12-1-0165 and under grant N00014-09-1-1042, and the National Science Foundation (NSF) under grants CNS-0845559 and CNS-0905537, and by Secure Business Austria.

References

[1] HtmlUnit. `http://htmlunit.sourceforge.net/`.

[2] JavaScript for Acrobat API Reference. `http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/js_api_reference.pdf`.

[3] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna. Efficient Detection of Split Personalities in Malware. In *Proc. of the Symposium on Network and Distributed System Security (NDSS)*, 2010.

[4] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, Behavior-Based Malware Clustering. In *Proc. of the Symposium on Network and Distributed System Security (NDSS)*, 2009.

[5] D. Canali, M. Cova, G. Vigna, and C. Kruegel. Prophiler: A Fast Filter for the Large-scale Detection of Malicious Web Pages. In *Proc. of the International World Wide Web Conference (WWW)*, 2011.

[6] M. Cova, C. Kruegel, and G. Vigna. Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code. In *Proc. of the International World Wide Web Conference (WWW)*, 2010.

[7] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. Zozzle: Low-overhead Mostly Static JavaScript Malware Detection. In *Proc. of the USENIX Security Symposium*, 2011.

[8] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2008.

[9] D. Edwards. Dean Edwards Packer. `http://bit.ly/TWQ46b`.

[10] P. Ferrie. Attacks on Virtual Machines. In *Proc. of the Association of Anti-Virus Asia Researchers Conference*, 2003.

[11] Google. Safe Browsing API. `http://code.google.com/apis/safebrowsing/`.

[12] 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. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2012.

[13] B. Hartstein. jsunpack – a generic JavaScript unpacker. `http://jsunpack.jeek.org/dec/go`.

[14] J. Jang, D. Brumley, and S. Venkataraman. BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2011.

[15] M. G. Kang, H. Yin, S. Hanna, S. McCamant, and D. Song. Emulating Emulation-Resistant Malware. In *Proc. of the Workshop on Virtual Machine Security (VMSec)*, 2009.

[16] A. Kapravelos, M. Cova, C. Kruegel, and G. Vigna. Escape from Monkey Island: Evading High-Interaction Honeyclients. In *Proc. of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2011.

[17] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-Cloaking Internet Malware. In *Proc. of the IEEE Symposium on Security and Privacy*, 2012.

[18] B. Krebs. Virus Scanners for Virus Authors, Part II. `http://krebsonsecurity.com/2010/04/virus-scanners-for-virus-authors-part-ii/`, 2010.

[19] F. Leder, B. Steinbock, and P. Martini. Classification and detection of metamorphic malware using value set analysis. In *Proc. of the Conference on Malicious and Unwanted Software (MALWARE)*, 2009.

[20] L. Lu, V. Yegneswaran, P. Porras, and W. Lee. BLADE: An Attack-Agnostic Approach for Preventing Drive-By Malware Infections. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2010.

[21] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi. Testing CPU Emulators. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*, 2009.

[22] Microsoft. Microsoft Security Intelligence Report, Volume 13. Technical report, Microsoft Corporation, 2012.

[23] Moss. Moss with obfuscated scripts. http://goo.gl/XzJ7M.

[24] M. Muja and D. G. Lowe. Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration. In *Proc. of the Conference on Computer Vision Theory and Applications (VISAPP)*, 2009.

[25] J. Nazario. PhoneyC: A Virtual Client Honeypot. In *Proc. of the USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2009.

[26] R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi. A Fistful of Red-Pills: How to Automatically Generate Procedures to Detect CPU Emulators. In *Proc. of the USENIX Workshop on Offensive Technologies (WOOT)*, 2009.

[27] J. Pate, R. Tairas, and N. Kraft. Clone Evolution: a Systematic Review. *Journal of Software Maintenance and Evolution: Research and Practice*, 2011.

[28] N. Provos, P. Mavrommatis, M. Rajab, and F. Monrose. All Your iFRAMEs Point to Us. In *Proc. of the USENIX Security Symposium*, 2008.

[29] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The Ghost in the Browser: Analysis of Web-based Malware. In *Proc. of the USENIX Workshop on Hot Topics in Understanding Botnet*, 2007.

[30] T. Raffetseder, C. Kruegel, and E. Kirda. Detecting System Emulators. In *Proc. of the Information Security Conference*, 2007.

[31] M. A. Rajab, L. Ballard, N. Jagpal, P. Mavrommatis, D. Nojiri, N. Provos, and L. Schmidt. Trends in Circumventing Web-Malware Detection. Technical report, Google, 2011.

[32] P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: A Defense Against Heap-spraying Code Injection Attacks. In *Proc. of the USENIX Security Symposium*, 2009.

[33] J. W. Ratclif. Pattern Matching: the Gestalt Approach. *Dr. Dobb's*, 1988.

[34] K. Rieck, T. Krueger, and A. Dewald. Cujo: Efficient Detection and Prevention of Drive-by-Download Attacks. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, 2010.

[35] C. K. Roy and J. R. Cordy. A Survey on Software Clone Detection Research. Technical report, School of Computing, Queen's University, 2007.

[36] J. Rutkowska. Red Pill... or how to detect VMM using (almost) one CPU instruction. http://www.invisiblethings.org/papers/redpill.html, 2004.

[37] S. Schleimer, D. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *Proc. of the 2003 ACM SIGMOD international conference on Management of data*, 2003.

[38] The Honeynet Project. Capture-HPC. https://projects.honeynet.org/capture-hpc.

[39] A. Vasudevan and R. Yerraballi. Cobra: Fine-grained Malware Analysis using Stealth Localized Executions. In *Proc. of the IEEE Symposium on Security and Privacy*, 2006.

[40] 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. In *Proc. of the Symposium on Network and Distributed System Security (NDSS)*, 2006.