Kizzle: A Signature Compiler for Exploit Kits

MSR-TR-2015-12

Ben Stock
FAU Erlangen-Nuremberg

Benjamin Livshits and Benjamin Zorn
Microsoft Research

***Abstract*—In recent years, the drive-by malware space has undergone significant consolidation. Today, the most common source of drive-by downloads are the so-called *exploit kits*. Exploit kits signify a drastic consolidation of the process of malware creation and delivery. Exploit kit authors are often significantly more skillful and better financially motivated than an average "standalone" JavaScript malware author. This paper presents Kizzle, the first prevention technique *specifically* designed for finding exploit kits.**

Our analysis of some of the code found in exploit kits shows that while the actual JavaScript delivered by kits varies greatly, the code observed after it is sufficiently unpacked and deobfuscated varies much less. The approach taken by Kizzle is based in our observation that, while exploit kits change the malware they deliver frequently, kit authors generally *reuse* much of their code from version to version. Ironically, this well-regarded software engineering practice allows us to build a scalable and precise detector that is able to quickly respond to superficial but frequent changes in exploit kits.

Kizzle is a signature compiler that is able to generate signatures for detecting exploit kits delivering JavaScript to browsers. These signatures compare favorably to those created by hand by security analysts. Yet Kizzle is highly responsive in that able to automatically generate new malware *signatures* within hours. Our approach will reduce the imbalance between the attacker who often only needs to make cosmetic changes to their malware to thwart detection, and the defender, whose role requires much manual effort.

Our experiments show that Kizzle produces high-accuracy signatures using a scalable cloud-based analysis. When evaluated over a month-long period, false positive rates for Kizzle are under 0.03%, while the false negative rates are under 5%. Both of these numbers compare favorably with the performance of commercial AV engines using hand-written signatures.

I. Introduction

The landscape of drive-by download malware has changed significantly in recent years. There has been a great deal of consolidation in malware production and a shift from attackers writing custom malware to almost exclusively basing drive-by download attacks on exploit kits (EKs) [20]. This approach gives attackers an advantage by allowing them to share and quickly reuse malware components in line with the best of software engineering guidelines. It is the natural evolution of the malware ecosystem [12] to specialize in individual tasks such as CVE discovery, packer creation, and malware delivery. We observe that this consolidation, while benefiting attackers, also allows defenders significant opportunities.

From ad-hoc to structural diversity: Indeed, the reliance on EKs implies a great deal of structural diversity in deployed malware. While five years ago we may have had deployed variants of the same JavaScript-driven heap spraying exploit written by two independent hackers, today much of the time malware variants will come by as a result of malware individualization performed by an EK as well as natural changes in the EK over time.

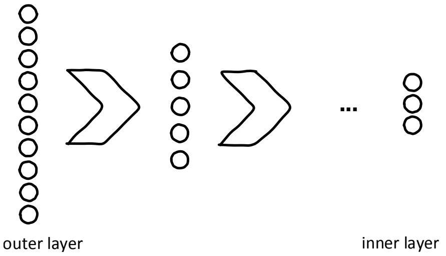


Fig. 1: Malware variants at different layers of the "malware onion". As we unpack the malware further, we observe fewer variants and more code reuse.

There is virtually no deployed JavaScript malware that is not obfuscated in some way; at runtime, the obfuscated code is *unpacked*, often multiple times, to get to the ultimate payload. We observe that in practice, much EK-generated malware operates like an onion: the outer layers change fast, often via randomization created by code packers, while the inner layers change more slowly, for example because they contain rarely-changing CVEs.

Arresting the malware cycle: Malware development is chiefly reactive in nature: malware producers create and test their wares against the current generation of readily available detection tools such as AV engines, EmergingThreats signatures, or sites such as VirusTotal. In a constant arms race with the AV industry, these kits change part of their code to avoid detection by signature-based approaches. While these changes may be distinct between two versions of the same kit, the underlying structure of the fingerprinting and exploitation code rarely changes.

This attack-defense pattern is the fundamental nature of the adversarial cycle (Figure 7), as has been noted as

EK	Flash	Silverlight	Java	Adobe Reader	Internet Explorer	AV check
Sweet Orange	2014-0515		Unknown[1]		2013-2551, 2014-0322	No
Angler	2014-0507, 2014-0515	2013-0074	2013-0422		2013-2551	Yes
RIG	2014-0497	2013-0074	Unknown		2013-2551	Yes
Nuclear exploit kit	(2013-5331), 2014-0497		2013-2423, 2013-2460	2010-0188	2013-2551	Yes

Fig. 2: CVEs used for each malware kit (as of September 2014). The CVEs are broken down into categories.

early as 20 years ago [25]. Unfortunately, presently, in the drive-by malware space the attacker has a considerable advantage in terms of the amount of work involved. Unfortunately, malware variants are relatively easy to create, most typically by changing the unpacker the attacker uses and testing their variant against VirusTotal. Indeed, the attacker can easily automate the process of variant creation and testing.

A. Kizzle

This paper proposes Kizzle, a *signature compiler* that automates the process of synthesizing signatures from captured malware variants. Kizzle gives defenders greater automation in the creation of AV signatures. Our scalable cloud-based implementation allows defenders to generate new signatures for malware variants observed the same day within a matter of hours. Moreover, because Kizzle creates signatures in an existing AV format, it takes advantage of a well-established *deployment channel*, i.e. AV signature update, and fits seamlessly into the existing malware defense ecosystem. Of course, the focus of Kizzle is on exploit kits *only*, whereas AV engines handle a much wider range of threats.

At the heart of Kizzle is a malware clustering approach that matches new malware clusters with previously-recognized malicious clusters by understanding the process of malware *unpacking*. As shown in Figure 1, more clusters exist at outer malware layers than exist at inner, more slowly changing layers. While we used a simple variant of k-means that we deployed on a cluster of computers, note, however, that a different clustering technique may be substituted in place of the one we used. We do not claim clustering as a contribution of this paper, as it is a standard technique in malware detection research and more complex techniques are available in the literature. In our implementation, however, we aimed to avoid complexity when it came to machine learning. This deliberate simplicity of our technical approach is a boon to our technique, making it more accessible to engineers who are the end-users of Kizzle. On the basis of created clusters, Kizzle generates AV signatures which we used for evaluation.

B. Contributions

This paper makes the following contributions:

Insight: Through detailed examination of existing exploit kits we document the evolution of these kits over time, noting how they grow and evolve. In particular, EKs often evolve by appending new exploits, the outer packer changes without major changes to the inner layer, and different EK families "borrow" exploits from each other. These observations suggest that there is a great deal of commonality and code reuse, both across EK versions, and between the different EKs, which enables EK-focused detection strategies.

Clustering in the cloud: Based on these observations we built a high-performance processing pipeline for preprocessing the JavaScript code into a structured token stream and parallelizing the process of clustering to be run in the cloud (50 machines in our deployment). Lastly, the clusters are matched with known exploit kits for both marking them as either benign or malicious and kit identification.

Signature generation: Out of the detected code clusters, we propose an algorithm for quickly automatically generating *structural signatures* which may be deployed within an anti-virus engine. Kizzle signatures are comparable in quality and readability to those a human analyst may write. With these structural signatures whose accuracy rivals those written by analysts, we can track EK changes in *minutes* rather than days. Kizzle signatures can be also deployed within the browser, enabling fast detection at JavaScript execution runtime.

Evaluation: Our longitudinal experimental evaluation shows that automatically-produced structural signatures are comparable to those produced manually. We compare Kizzle against a widely used commercial AV engine and find that it produces comparable false positive and false negative rates for the exploit kits we targeted. When evaluated over a month-long period, false positive rates for Kizzle are under 0.03%, while the false negative rates are under 5%.

C. Paper Organization

The rest of the paper is organized as follows. Section II gives some background on exploit kits and how they are typically constructed. Section III provides some of the technical details. Section IV contains a detailed experimental evaluation. Section V talks about the limitations of our approach. Finally, Sections VI and VII summarize related work and conclude.

II. Background

The last several years have witnessed a shift from unique drive-by downloads to a consolidation into so-

[1] Note that for some of the kits, while a Java exploit was present, no version checking was conducted by the kit, thus determining the specific CVE is difficult if not impossible.

called Exploit Kits, which incorporate a variety of exploits for known vulnerabilities. This has several advantages for all malicious parties involved. As mentioned in the Microsoft Security Intelligence Report V16, "Commercial exploit kits have existed since at least 2006 in various forms, but early versions required a considerable amount of technical expertise to use, which limited their appeal among prospective attackers. This requirement changed in 2010 with the initial release of the Blackhole exploit kit, which was designed to be usable by novice attackers with limited technical skills."

Exploit kits bring the benefits of specialization to malware production. Indeed, a botnet herder can now focus on development of his software rather than having to build exploits that target vulnerabilities in browser and plugins. On the other hand, the maintainer of a single exploit kit may use it to distribute different pieces of malicious software, optimizing his revenue stream. This trend is also shown by glimpses security researchers sometimes got into the backend functionality and operation of such kits, such as detailed information on the rate of successful infection from the kits [15]. Interested readers are referred to [20] for a more comprehensive summary. Note, however, that most up-to-date information can be found via blogs like SpiderLabs[2] and "Malware don't need Coffee[3]" which are updated regularly as exploit kit updates emerge.

In this paper we primarily focus on detecting four popular exploit kits (EKs). Figure 2 shows a brief summary of information about these EKs. As can be seen from the table, the EKs under investigation are targeting vulnerabilities in five browser and plugin components. An interesting observation in this instance is the fact that the Nuclear contains an exploit targeting a CVE from 2010 in Adobe Reader, highlighting the fact that exploitable vulnerabilities are hard to come by.

Note that in September 2014, three of the exploit kits used the exact same code to check for certain system files belonging to AV solutions. If such solutions are discovered, the exploit kits effectively stop the attack to avoid detection.

A. *Exploit Kit Structure*

Exploit kits (EKs) are typically comprised of several components organized into layers (see Figure 3). The components typically include an *unpacker*, a plugin and AV detection component, an `eval`/execution trigger, and, finally, at least one malicious payload.

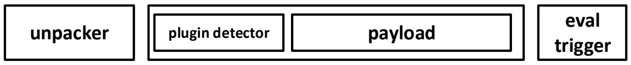


Fig. 3: Structure and components of a typical EK.

```
var buffer="";
var delim="y6";

function collect(text) {
 buffer += text;
}

collect("47y642y6100y6");
collect("102y6103y6104..");

pieces = buffer.split(delim);

screlem = document.createElement("script");

for (var i=0; i<pieces.length; i++) {
 screlem.text += String.fromCharCode(pieces[i]);
}

document.body.appendChild(screlem);
```

(a) RIG

```
var payload =
    "69172243452601227643 7
    188215239887038218819 7
    642634057014376927622 1
    2757616434526211272.."
var cryptkey =
    "Io^Rg_U8$\\ep6kAu.rVvn!'Ti15SQqd-
    #2@\"{(l4xcbt?>[3E/sP:O<D7*yz|m+Z;JBf)
    hX9Gw L0CF%KN},&YaMHj=]W";
...

getter = function(a){
  return a;
};

thiscopy = this;
doc = thiscopy[thiscopy["getter"]("document")]
bgc = doc[thiscopy["getter"]("bgColor")];

evl = thiscopy["getter"]("ev#333366al")
win = thiscopy["getter"]("win#333366dow")

thiscopy
    [win["replace"](bgc,"")]
    [evl["replace"](bgc, "")](payload);
```

(b) Nuclear exploit kit

Fig. 4: Two typical code unpackers from exploit kits.

To stay competitive, a real-life exploit kit usually comes equipped with a range of CVEs it attempts to exploit, which target a wide range of operating systems and browsers. In the following, we shed light on these components, showing how they evolve over time.

Unpackers: The outer-most layer of this onion is typically used to ensure that a security analyst or a web site administrator cannot effortlessly determine the inner workings of the exploit kit. This can either be achieved by packing the underlying pieces or by at least applying obfuscation techniques such as string encoding.

Figure 4 shows samples of packers from RIG and Nuclear Exploit Kit, highlighting the differences between them across families. While Nuclear relies on an encryption key that is used when unpacking the malicious payload, RIG uses a buffer which is dynamically filled during runtime with the ASCII codes for the payload, intermixed with a delimiter. We found that this delimiter is randomized between different versions of the kit. In contrast, the encryption key — and therefore the encrypted payload — for the Nuclear exploit kit differs in every response, highlighting the fact that the obfuscated code is difficult to pattern-match on.

Plugin and AV detection code: All major EKs use

[2] `http://blog.spiderlabs.com/`

[3] `http://malware.dontneedcoffee.com/`

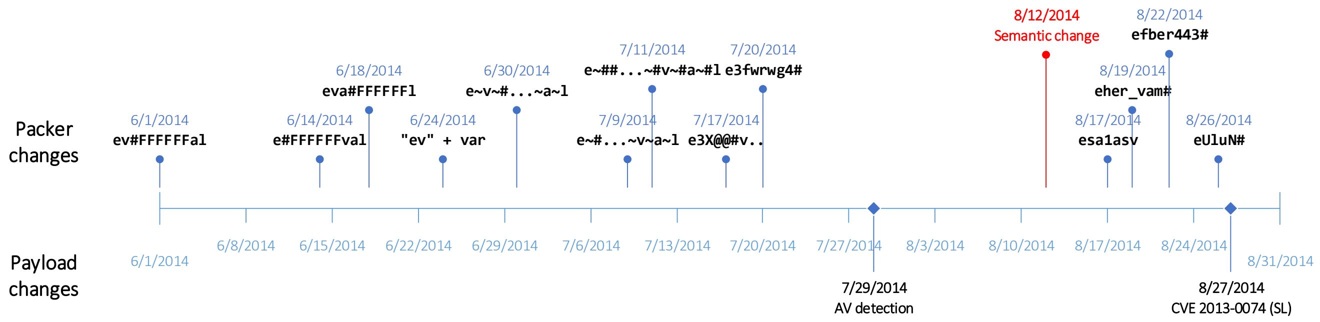


Fig. 5: Evolution of the Nuclear exploit kit over a three-months period in 2014. In this timeline, packer changes are shown above the axis and payload changes below the axis. The lion's share of changes are superficial changes to the packer.

```
function chavs(a) {
    var xmlDoc = new ActiveXObject("Microsoft.XMLDOM");
    xmlDoc.async = true;
    xmlDoc.loadXML('<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML
         1.0 Translation//EN" "res://c:\\Windows\\System32\\
         drivers\\' + a + '">');
    if (xmlDoc.parseError.errorCode != 0) {
        var err = "Error Code: " + xmlDoc.parseError.errorCode
            + "\n";
        err += "Error Reason: " + xmlDoc.parseError.reason;
        err += "Error line: " + xmlDoc.parseError.line;
        if (err.indexOf("-2147023083") > 0) {
            return 1;
        } else {
            return 0;
        }
    }
    return 0;
}

if (chavs("kl1.sys") || chavs("tmnciesc.sys") || chavs("tmtdi.
      sys") || chavs("tmactmon.sys") || chavs("TMEBC32.sys")
      || chavs("tmeext.sys") || chavs("tmcomm.sys") || chavs("
      tmevtmgr.sys")) {
    exit();
}
```

Fig. 6: Detecting and avoiding Anti-Virus solutions.

some form of plugin or browser version detection code. Apart from the fact that, e.g., downloading a malicious PDF file to a machine which is not running a vulnerable version of Adobe Reader causes unnecessary traffic, attackers have different reasons to ensure specific versions of client-side software are running for better targeting [19].

As an example, let us consider a heap-spraying vulnerability in a browser. While the attack might eventually lead to execution of attacker-controlled payload, exhausting memory in a browser that is not susceptible to this attack might crash the browser, which raises the attention of the user and possibly sends reports to the browser vendor, prompting attention the attacker would like to avoid. Also, downloading a malicious file might trigger AV software on the machine, which again draws attention to the site that was just visited. We also found that several exploit kit use measures to determine if a certain AV software is present, in which case the EK would abort the execution entirely, as shown in Figure 6. The code leverages the fact that the XMLDOM implementation in Internet Explorer returns a different error code depending on whether a file was not on disk or access was denied to that file. This can be used to enumerate files on the system. Here, the Trend Micro (tm*.sys) as well the Kaspersky (kl1.sys) AV engines are being detected based on drivers installed on the machine.

Version checking code is closely tied with the exploits that are used — ensuring that a given version of Java is running on a machine is unnecessary when only targeting Silverlight vulnerabilities. As we will see, exploits or CVEs are not frequently added (due to the lack of sufficient exploitable vulnerabilities) to EKs and so, changes to plugin detection code also occur quite infrequently. While in the past client-side detection was developed by each individual malware author, today, EK authors tend to use the PluginDetect JavaScript library[4] which allows the extraction of browser and plugin version information with a simple function call.

Malicious payload: The actual payload typically targets a set of known vulnerabilities in the browser or plugins. As can be seen from the Exploit Pack Table[5], 5–7 CVEs per kit is fairly typical. The payload is often interleaved with plugin detection code, e.g. an HTML element is added to the DOM pointing to a malicious flash file if a vulnerable version was detected previously.

Eval trigger: After the malware is fully unfolded, there is usually a short trigger that starts the EK execution process. Examples of these triggers are shown on line 19 of Figure 4(a) and lines 23–25 of Figure 4(b).

B. *Evolution of an Exploit Kit*

To understand how EK components evolve in the wild, we captured samples of the Nuclear exploit kit over the course of 3 months and tracked changes to the kit (see Figure 5).

Changing the packer: The figure illustrates specific changes that were made to the packer and the payload in the kit to avoid AV detection on a time line. Many of the changes were very local, changing the way that the kit obscured calls to eval. For example, between 6/1 and 6/14, the attacker changed ev#FFFFFFal to e#FFFFFFval.

[4] http://www.pinlady.net/PluginDetect/

[5] http://contagiodump.blogspot.com/2010/06/overview-of-exploit-packs-update.html

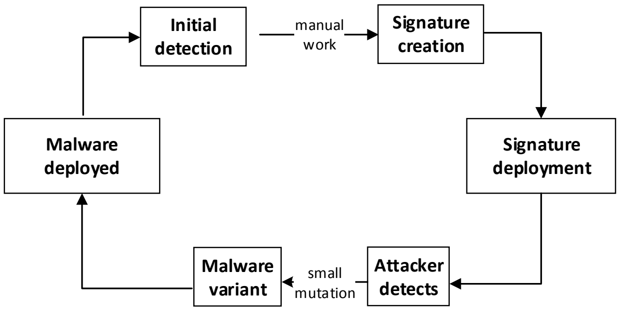


Fig. 7: Adversarial cycle illustrated. There is a built-in asymmetry that Kizzle aims to remedy: the response time for the attacker is only a fraction of that of the defender.

Over the course of the three months, we see a total of 13 small syntactic changes in this category. Only one of these packer changes (on 8/12) changed the semantics of the packer.

Appending new exploits: We observed changes to the other components of the kit, namely plug-in detection, and payload, much less frequently. On 7/29, AV detection was added to the plug-in detector component, and on 8/27 a new CVE was added. Nothing was removed from either the plug-in detector or the payload over this period. This data supports our claim that exploit kits change their outer layer frequently to avoid AV detection, but typically modify their inner components by appending to them and even then only infrequently.

Code borrowing: A noteworthy fact in this instance is that in June, Nuclear exploit kit did not utilize any code aiming at detecting AV software running on the victim's machine. We initially observed this behavior in the RIG exploit kit starting in May. The exact code we had observed for RIG was then used in Nuclear starting in August, obviously having been copied from the rivaling kit.

Summary: While we use Nuclear exploit kit as a specific example here, we observed similar changes in all the exploit kits we studied. In summary, we observe that kits typically change in three ways, namely **changing** the unpacker (frequent), **appending** new exploits (infrequent), and **borrowing** code from other kits (infrequent).

C. Adversarial Cycle

Exploit kit authors are in a constant arms race with anti-virus companies as well as rivaling kits' authors. While on the one hand, the kits try to avoid detection by anti-virus engines, their revenue stream is dependent on the amount of machines they can infect. Therefore, kit authors always try to include multiple exploits, and if one kit includes a new exploit, we observed that these exploits are quickly incorporated into other kits as well.

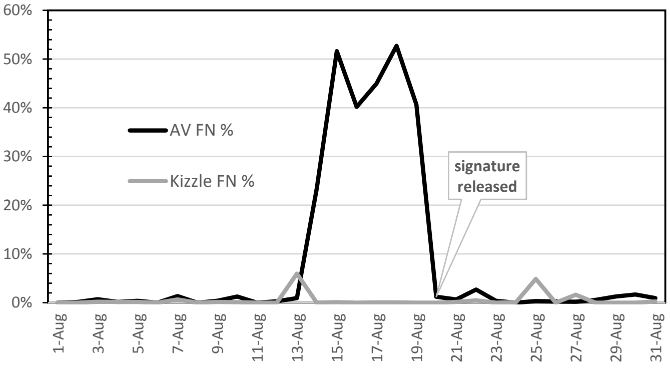


Fig. 8: Window of vulnerability for Angler in August, 2014 for a commercial AV engine. The window starts around August 13th and continues to roughly August 19th.

As we have seen in the example of the Nuclear exploit kit above, kit authors attempt to avoid detection by anti-virus engines by modifying the code, whereas in turn AV analysts try to create new signatures matching the modified variants. This process can be abstracted to the adversarial cycle shown in Figure 7.

Initially, an exploit kit is not detected by AV, which presents a very challenging problem for an analyst. First they have to find examples of the undetected variant and then they need to write a new signature which matches this undetected variant, trading off precision and recall – i.e. the signature has to be able to catch all occurrences of the exploit kit while not blocking benign components. Naturally this takes time and effort and despite this cost, AV engines update their signatures frequently to keep pace with the malware writers. After an analyst is satisfied they have a sufficiently precise and unique signature, they deploy it.

At this point, the attacker responds, determining that his kit is now detected by deployed AV signatures. Once this step has occurred (left-hand side of the figure), he takes measures to counter detection, such as slight modification to the part of the code that would be suspicious (such as calls to `eval`), or, in more drastic cases we observed in the wild, exchanging entire pieces, such as the unpacker.

Depending on the type of change, this task can be easily accomplished within minutes — and more importantly, an attacker can typically scan his code with an AV solution to determine if now passes detection. This shows the imbalance between the involved parties, i.e. the effort and reaction time of the malicious player can be much lower than that of the AV vendors.

Example 1 (Angler in August) Figure 8 shows false negatives for the Angler exploit kit in the month of

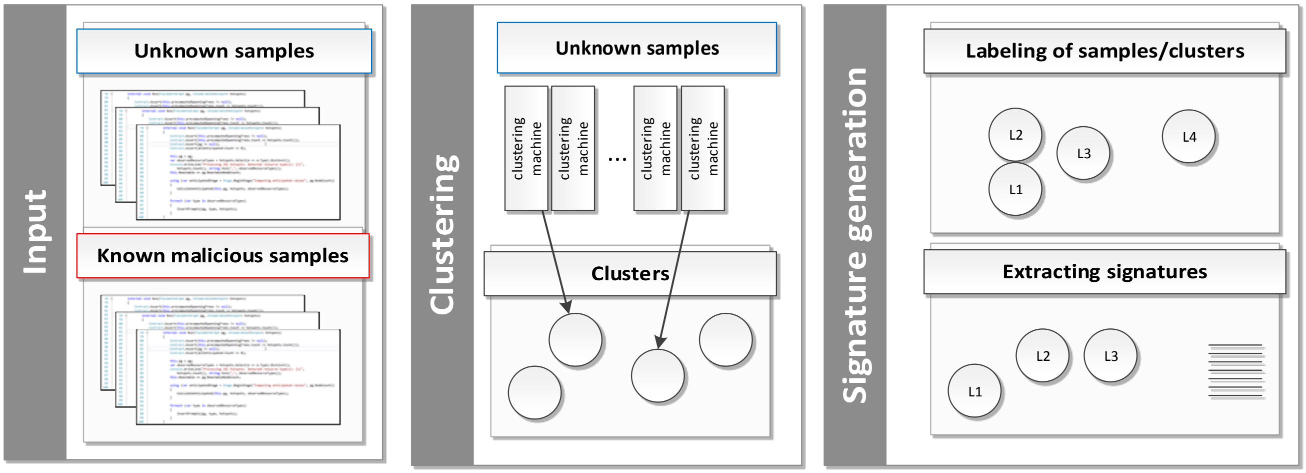


Fig. 9: Architecture of Kizzle.

August, 2014 for a widely used commercial AV engine[6]. By observing the changes to the kit over this time, we understand what happened. Before August 13th, the exploit kit contained an unencrypted HTML snippet that included an exploit targeting Java with a specific unique string. On August 13th, this string was incorporated into the encrypted body of kit and only written to the document if a vulnerable version of Java was installed on the system. This change negated the AVs ability to identify the kit, resulting in a window of time where variants of the kits were not detected. □

III. Techniques

In this section, we describe the implementation of Kizzle, which is illustrated at a high level in Figure 9. The input to Kizzle is a set of new samples and a set of existing unpacked malware samples which correspond to exploit kits Kizzle is aiming to detect. The algorithmic elements of Kizzle include abstracting the samples into token sequences, clustering the samples, labeling the clusters, and generating signatures from the elements of clusters labeled as malicious. We describe each of these steps in turn with examples, referring to the pseudocode in Figures 10–15 and Figure 17.

The processing starts with a new collection of samples (Figure 10). The `Main` routine shows how Kizzle breaks the new, "unknown" samples into a set of clusters, labels each cluster either as benign or corresponding to a known kit, and if the cluster is malicious, generates a new signature for that cluster based on the samples in it. We consider each of the parts of the task in turn in the subsections below.

```
Main
Inputs: unlabeled samples
Returns: generated signatures for all malicious samples
clusters = Cluster(samples)
for all cluster in clusters do
    unpackedSample = Unpack(cluster.prototype)
    label = LabelSample(unpackedSample)
    if label != ”unknown” then
        yield CreateSignature(cluster)
    end if
end for
```

Fig. 10: Kizzle `Main(samples)`. Main function of Kizzle, invokes the components, namely clustering and unfolding, followed by the process of labeling of resulting clusters and signature creation for each malicious cluster.

```
Cluster
Inputs: unlabeled samples
Returns: clustered, unlabeled samples
1: partitions = Partition(samples)
2: for all partition in partitions do
3:     partitiontokens = map(partition, lambda x: tokenize(x))
4:     clusters[i] = DBSCAN(partitiontokens)
5:     i++
6: end for
7: return DBSCAN(clusters)
```

Fig. 11: Function `Cluster(samples)`. After partitioning the input, the content is abstracted to the tokenized form and subsequently, DBSCAN is used to cluster the samples.

A. *Clustering Samples*

The process of clustering the input samples is computationally expensive, and as a result, benefits from parallelization across a set of machines. Figure 11 explains the process.

The first step is to randomly *partition* the samples across a cluster of machines (line 1). For each partition, the samples are first tokenized from the concrete JavaScript source code represented as Unicode to a sequence of abstract tokens that include `Keyword`, `Identifier`, `Punctuation`, and `String`. Figure 12 gives

[6] We anonymize the exact engine for two reasons: first, we believe that all engines exhibit the same behavior despite constant efforts by the analysts to keep them current, and second because EULA agreements typically prevent disclosing such comparisons.

Token	Class
`var`	Keyword
`Euur1V`	Identifier
`=`	Punctuation
`this`	Identifier
`[`	Punctuation
”`l9D`”	String
`]`	Punctuation
`(`	Punctuation
”`ev#333399al`”	String
`)`	Punctuation

Fig. 12: Example of tokenization in action.

```
LabelSample
Inputs: single unlabeled sample
Returns: label for that sample
 1: global knownMalwareSamples
 2: sampleWinnows = WinnowHistogram(sample)
 3: for all kMS in knownMalwareSamples do
 4:     kMSWinnows = WinnowHistogram(kMS.sample)
 5:     overlap = HistogramOverlap(kMSWinnows, sampleWinnows)
 6:     if overlap > kMS.threshold then
 7:         return kMS.Family
 8:     end if
 9: end for
10: return "unknown"
```

Fig. 13: Function `LabelSample(sample)`. Given a set of known malicious samples and their winnow histograms, the histogram is calculated for the sample in question to quickly check the code overlap.

an example of tokenization in action.

We cluster the samples based on these token strings in order to eliminate artificial noise created by an attacker in the form of randomized variable names, etc. We apply a hierarchical clustering algorithm, using the edit distance between token strings as a means of determining the distance between any two samples. We have experimentally determined that a threshold of 0.10 is sufficient to generate a reasonably small number of clusters, while not generating clusters that are too generic, i.e. contain samples that do not belong to the same family of malware (or snippets of benign code). As an algorithm, we use the DBSCAN clustering algorithm [11] (line 4). In the reduction phase, the clusters determined by each partition are combined in a final step (line 7).

Jumping back to the `Main` function in Figure 10, we then consider each cluster in turn (line 2), selecting a single prototype sample (piece of JavaScript code) from the cluster, unpacking it (if it is packed) and then attempting to label it (line 4)[7]. With the unpacked cluster prototype, we then attempt to label the cluster.

B. Labeling Clusters

Figure 13 shows how we label clusters using winnowing [34], a technique originally proposed for detecting plagiarism in code. Using a collection of known unpacked malware samples (with exploit family labels), we generate a winnow histogram for the cluster prototype and compare it against the winnow histograms for all the known malware samples. If there is sufficient overlap (based on a threshold that we determined empirically is malware family specific) (line 6), then we consider the cluster represented by the prototype to be malicious and from the corresponding family.

```
CreateSignature
Inputs: malicious cluster
Returns: signature to match all files in the cluster
 1: DistinctValues = []
 2: CommonSubseq = BinarySearch(extractNgrams, cluster, max=200)
 3: if CommonSubseq.length < 20 then
 4:     return <invalid>
 5: end if
 6: for all member in cluster do
 7:     RelevantTokens = ExtractTokens(member, CommonSubseq)
 8:     for all token in RelevantTokens do
 9:         if token.value not in DistinctValues[token.offset] then
10:             DistinctValues[token.offset].add(token.value)
11:         end if
12:     end for
13: end for
14: signature = ""
15: for all values in DistinctValues do
16:     signature += GenerateMatchingRegexp(values)
17: end for
18: if signature.length < 100 then
19:     return <invalid>
20: end if
21: return signature
```

Fig. 14: Function `CreateSignature(cluster)`. After finding the longest common token sequence, the concrete values for those tokens are collected and a matching regular expression is generated.

```
extractNgrams
Inputs: cluster, length of n-gram
Returns: distinct n-grams that can be found once if every file of the cluster
 1: for all member in cluster do
 2:     tokens = tokenize(member)
 3:     tokenNgrams = getNgrams(tokens, n)
 4:     NgramHistogram = histogram(tokenNgrams)
 5:     UniqueNgrams = NgramHistogram.extract(count=1)
 6:     for all ngram in UniqueNgrams do
 7:         GlobalHistogram[ngram].count += 1
 8:     end for
 9: end for
10: for all ngram in GlobalHistogram do
11:     if ngram.count == size(cluster) then
12:         yield ngram
13:     end if
14: end for
```

Fig. 15: Function `extractNgrams(cluster,n)`. Finds all n-grams for a given n that occur exactly once in every file of the analyzed cluster.

C. Signature Creation

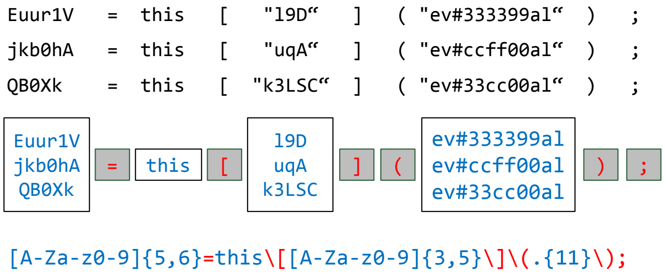
For each cluster that is labeled as malicious, we generate a signature from the packed samples in that cluster with the following method (Figure 14). The first step in signature creation is to find a maximum value of N such that every sample in a cluster has a common token string subsequence of length up to N tokens. We cap this maximum length at 200 tokens. We find this subsequence with binary search, varying N and calling the function

[7] In our experiments, because we focus on a small number of exploit kits, we chose to implement an unpacker for each exploit kit manually. More general techniques such as emulation or collecting the unpacked sample over the course of executing it in JavaScript engine are possible as well.

in Figure 15, `extractNgrams`, to determine if a common subsequence of length N exists. A additional constraint, imposed during the search for a common subsequence, is that the subsequence is unique in every sample (Figure 15, line 5).

Once the length of the common subsequence is known and sufficiently long (sequences that are too short are discarded), the exact sequences of tokens and characters in each of the samples from the malicious cluster are extracted (Figure 14, line 4). In addition, for each offset in the token sequence, the algorithm determines the distinct set of concrete strings found in the different samples at that token offset.

Figure 16 illustrates this process, showing a cluster with three samples and the process of determining the distinct values at each offset. Note, that although the original string contains quotation marks, these are automatically removed by AV scanners in a normalization step. Therefore, we omit these in the final signature. Finally, with the array `DistinctValues` containing all variants appearing in the samples at each offset, the algorithm generates a regular expression-based signature, one token at at time (Figure 14, line 13).



```
Euur1V   =  this   [   "l9D“   ]   ( "ev#333399al“  )   ;
jkb0hA   =  this   [   "uqA“   ]   ( "ev#ccff00al“  )   ;
QB0Xk    =  this   [  "k3LSC“  ]   ( "ev#33cc00al“  )   ;
```

```
[A-Za-z0-9]{5,6}=this\[[A-Za-z0-9]{3,5}\]\(.{11}\);
```

Fig. 16: An example of signature generation in action.

Figure 17 describes how a regular expression is generated from a set of distinct values. If the value is the same across *all* samples (line 1), add the concrete value to the signature. Otherwise, the algorithm must generate a regular expression that will match all elements of this set. While this is a well-studied problem in general (The $L\star$ algorithm [2] can infer a minimally accepting DFA), we implement a domain-specific approach focusing on our expectations of the kinds of diversity that malware writers are likely to inject into their code.

Our approach turns out to be highly scalable. We compute an expression that will accept strings of the observed lengths, and containing the characters observed by drawing on a predefined set of common patterns such as $[\mathtt{a-z}]+$, $[\mathtt{a-zA-Z0-9}]+$, etc. The current approach uses brute force to determine a working pattern (line 9) but a more selective approach could build a more efficient decision procedure from the predefined templates.

```
GenerateMatchingRegexp
Inputs: distinct values
Returns: regular expression that matches all distinct values
 1: if values.length == 1 then
 2:     return values[0]
 3: end if
 4: minlength = min(values, key=length)
 5: maxlength = max(values, key=length)
 6: lengthdef = ”” + minlength + ”,” + maxlength + ””
 7: AllUsedChars = set(values.join().split())
 8: PredefinedRegexp = ([”^[A-Z+]$”, ”^[A-Z0-9+]$”, ...])
 9: for all regexp in PredefinedRegexp do
10:     if regexp.matches(AllUsedChars) then
11:         return regexp + lengthdef
12:     end if
13: end for
14: return ”.” + lengthdef
```

Fig. 17: Function `GenerateMatchingRegexp(values)`. Generates a regular expression that expresses all distinct values collected in the previous step.

```
(?<var0>[0-9a-zA-Z]{3,6})=\[\[(?<var1>
[0-9a-zA-Z]{3,6})\[(?<var2>[0-9a-zA-Z"’]
{5,8})]\("cUluNoUluNnUluNcUluNaUluNtUluN"\),
\k<var1>\[\k<var2>]\("sUluNuUluNbUluNsUluNtUluNrUluN"\),
\k<var1>\[\k<var2>]\("dUluNoUluNcUluNuUluNmUluNeUluNnUluNtUluN"\),
\k<var1>\[\k<var2>]\("CUluNoUluNlUluNoUluNrUluN"\),
\k<var1>\[\k<var2>]\("lUluNeUluNnUluNgUluNtUluNhUluN"\)],
\[\k<var1>\[\k<var2>]\((?<var3>.{57})\),\k<var1>
\[\k<var2>]\((?<var4>.{67})\),\k<var1>
\[\k<var2>]\("rUluNeUluNpUluNlUluNaUluNcUluNeUluN"\)]]
var(?<var5>[0-9a-zA-Z]{3,7})
```

(a) Nuclear exploit kit

```
\)\)\)\{varaa=xx\.join\(""\)ar\[\(Math\.exp\(1\)-Math\.E\)]
\[\(1\)\*2]="l"ar\[\(1\)]\[3]="WWWWWWWbEWsjdhfW"varq=
\(Math\.exp\(1\)-Math\.E\)for
\(qq<ar\[\(Math\.exp\(1\)-Math\.E\)]\.length\+
\+q\)\{aa=aa\.cnvbsdfYTQUWETQWUEASA\(newRegExp\(ar\[\(1\)]
\[q\+\(1\)],"g"\),ar\[\(Math\.exp\(1\)-Math\.E\)]\[q]\)\}
returnaa\}return""\}function(?<var0>[a-zA-Z]{6})\(\)
\{varok=\[(?<var1>[0-9a-zA-Z"’]{17})\.charAt\(Math\.sqrt
\(196\)\),(?<var2>[0-9a-zA-Z"’]{17})\.charAt\(Math\.sqrt
\(196\)\),(?<var3>[0-9a-zA-Z"’]{17})\.charAt\(Math\.sqrt
\(196\)\),(?<var4>[0-9a-zA-Z"’]{21})\.charAt\(Math\.sqrt
\(324\)\),(?<var5>[0-9a-zA-Z"’]{21})\.
```

(b) Sweet orange

Fig. 18: Examples of Kizzle-generated signatures.

Example 2 (Kizzle **signatures)** Figure 18 shows signature generated by Kizzle that match the Nuclear and Sweet orange. For the first signatures, Kizzle picked up on the strings delimited by `Ulun`. While such long strings that do not naturally occur in benign applications make the creation of a signature easy, the kit author can easily change these to avoid detection by a matching signature. Since, however, Kizzle generates these automatically, this advantage vanishes.

Also, Kizzle picked up on the usage of templatized variable names, as can be observed by the combination of `var1` and `var2` in lines 4 to 9. While the Sweet orange EK does not use such delimiters, it uses a simple obfuscation technique, namely exchanging static integer values with calls to the `Math.sqrt` function, allowing it to simply change this obfuscation by using other mathematical operations. Again, Kizzle picked up on this property of the packed payload, generating a precise signature. □

IV. Evaluation

In this section we evaluate the effectiveness of Kizzle. To evaluate Kizzle, we gathered potentially malicious samples

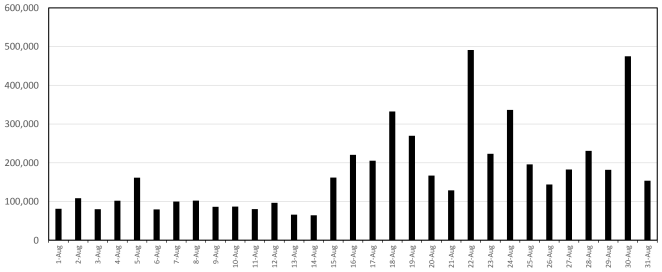


Fig. 19: Grayware volume over time for a month-long time window.

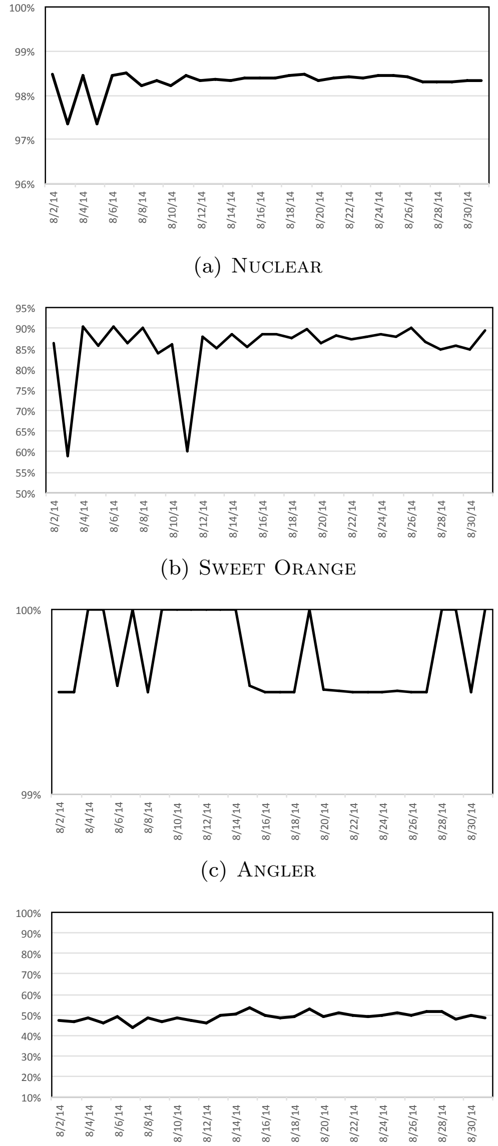
using a crawler with a browser instrumented to gather telemetry collected from IE 11 installations on pages that have ActiveX content. We worked with an anti-malware vendor to hook into the `IExtensionValidation` interface[8] for data extraction.

The `Validate` method of this interface allows the capture of `htmlDocumentTop` and `htmlDocumentSubframe` which is how we get access to the underlying HTML and JavaScript. Our focus on IE 11 biases our collection towards newer, more actively supported kits that have payloads that work in IE 11. Because we sample data during a potentially suspicious operation, the fraction of malware we see is likely to be substantially higher than a typical browser will see, hence we consider our data stream "grayware". We ran the crawler daily for a month (August 2014) and the overall number of samples in the stream of grayware data is shown in Figure 19.

Throughout the rest of this section, we focus on the four exploit kits that are most prevalent in our data: NUCLEAR, SWEET ORANGE, ANGLER, and RIG. All these kits follow the pattern we describe in Section II: they are packed on the outside and are relatively similar when unpacked.

We compare KIZZLE with a state-of-the-art commercial AV implementation, which we anonymize to avoid drawing generalizations based on our limited observations (our position is that all commercial AV vendors have similar challenges). Because we focus solely on only four exploit kits, drawing more general conclusions about the quality of the product based on these results would be incorrect.

A. Experimental Setup



(a) NUCLEAR

(b) SWEET ORANGE

(c) ANGLER

(d) RIG

Fig. 20: Similarity over time for a month-long time window. Note that the y axis is *different* across the graphs: sometimes, the range of similarities is very narrow.

Figure 20 shows our measurements of how these kits change over the course of a month in our data. We measure the overlap between the centroids of malicious clusters on each day with centroids of the clusters of all previous days based on winnowing (Section III) and report the maximum overlap. Figure 20 shows that, for three of the four kits we measured, the amount of change from over the course of the entire month, is quite small, often only a few percent. This contrasts greatly from the *external* changes at the level of the packed kits as shown in Figure 5, which happen every few days. The example in Figure 4(b) shows that EKs use payload encryption minimizing the amount of syntactic similarity between the variants that employ different encryption keys.

These observations confirm our hypothesis that most of the change is *external* and happens on the packer that surrounds the logic of the kit. In the case of NUCLEAR EXPLOIT KIT, there is very little change at all. We do note that RIG (Figure 20(d)) is an outlier, showing changes of 50% from day over day. This behavior is explained by noticing that the changes reflect changes in the embedded URLs of the kit, and, given the body of the kit is relatively short, these URLs alone represent a significant enough part

[8] `http://msdn.microsoft.com/en-us/library/dn301826(v=vs.85).aspx`.

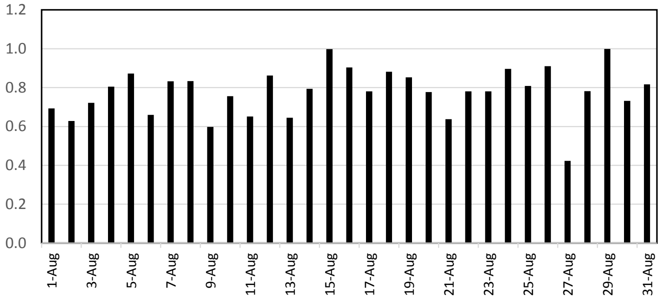


Fig. 21: Running time in megabytes per second, for a month-long time window.

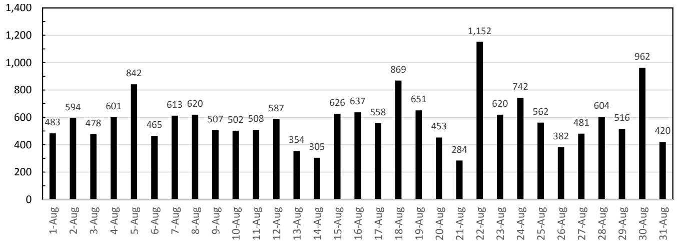


Fig. 23: Number of clusters generated daily over time for a month-long time window.

of the code to indicate a 50% churn.

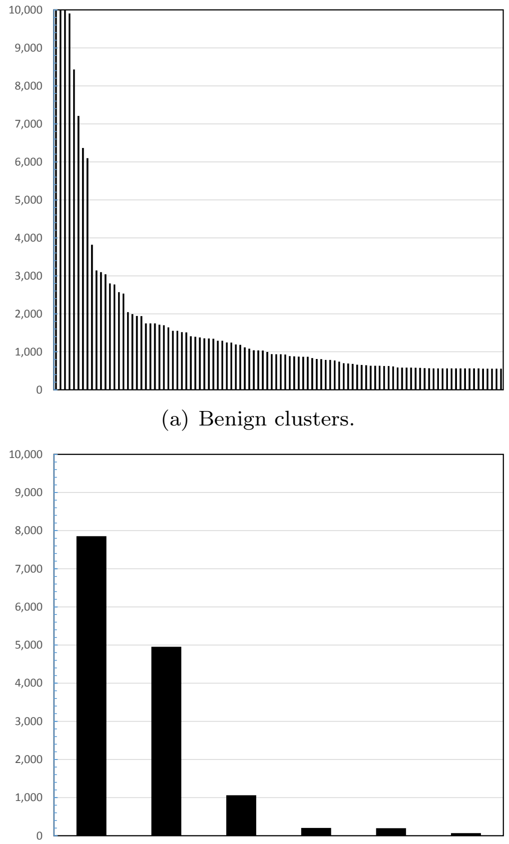
B. Cluster-Based Processing Performance

Our implementation is based on using a cluster of machines and exploiting the inherent parallelism of our approach. For the performance numbers found in this section, we used 50 machines for sample clustering and signature generation. Figure 21 which shows the normalized running time over the same month-long period of data consistently shows running times of about 1 MB per second.

Our experience shows that clustering takes the majority of this time, as opposed to signature generation. The reduce step described in Section III-A is often the bottleneck when we needed to reconcile the clusters computed across the distributed machines. With more effort, we believe that the reduction step can be parallelized as well in future work to improve our scalability. In practice, our runs consistently completed in about 90 minutes when processing the data collected by our crawl. We believe that this processing time is adequate and can be further improved with more machines.

C. Clustering Process

Figure 23 shows the number of clusters over time detected by the Kizzle clustering process. The majority of clusters every day correspond to benign code, while a handful are detected as malicious and labeled as one of the exploit kits above. The figure shows that despite the fact that we are analyzing grayware, much of what we observe is benign code that falls into a relatively small number of frequently observed clusters.



(a) Benign clusters.

(b) Malicious clusters.

Fig. 24: A bar plot showing the number of samples per cluster on the y axis, for both benign and malicious clusters. Each bar represent a single cluster, sorted by the number of contained samples.

When we compared the number of samples per cluster by focusing on a single day within the month of August, we do not see significant differences between the number of samples in benign and malicious clusters, as shown in the two charts in Figure 24. Each bar represents a single detected cluster. We see a wide variety of benign clusters — over a hundred of them — some with as many as 10,000 samples in size. There only several malicious clusters corresponding to the exploit kits we are looking for — a small fraction of the total detected, although, two of them contain at least 1,000 elements. At the same time, we cannot, for example, conclude that clusters with many samples are necessarily malicious; this is simply not the case.

D. Characterizing Kizzle-Generated Signatures

Figure 18 shows some examples of the signatures produced by Kizzle. Overall, two things immediately stand out for the automatically-generated signatures: they are long and they are very specific. Both of these characteristics contribute to Kizzle signatures being less prone to false positives, as we show in Figure 25.

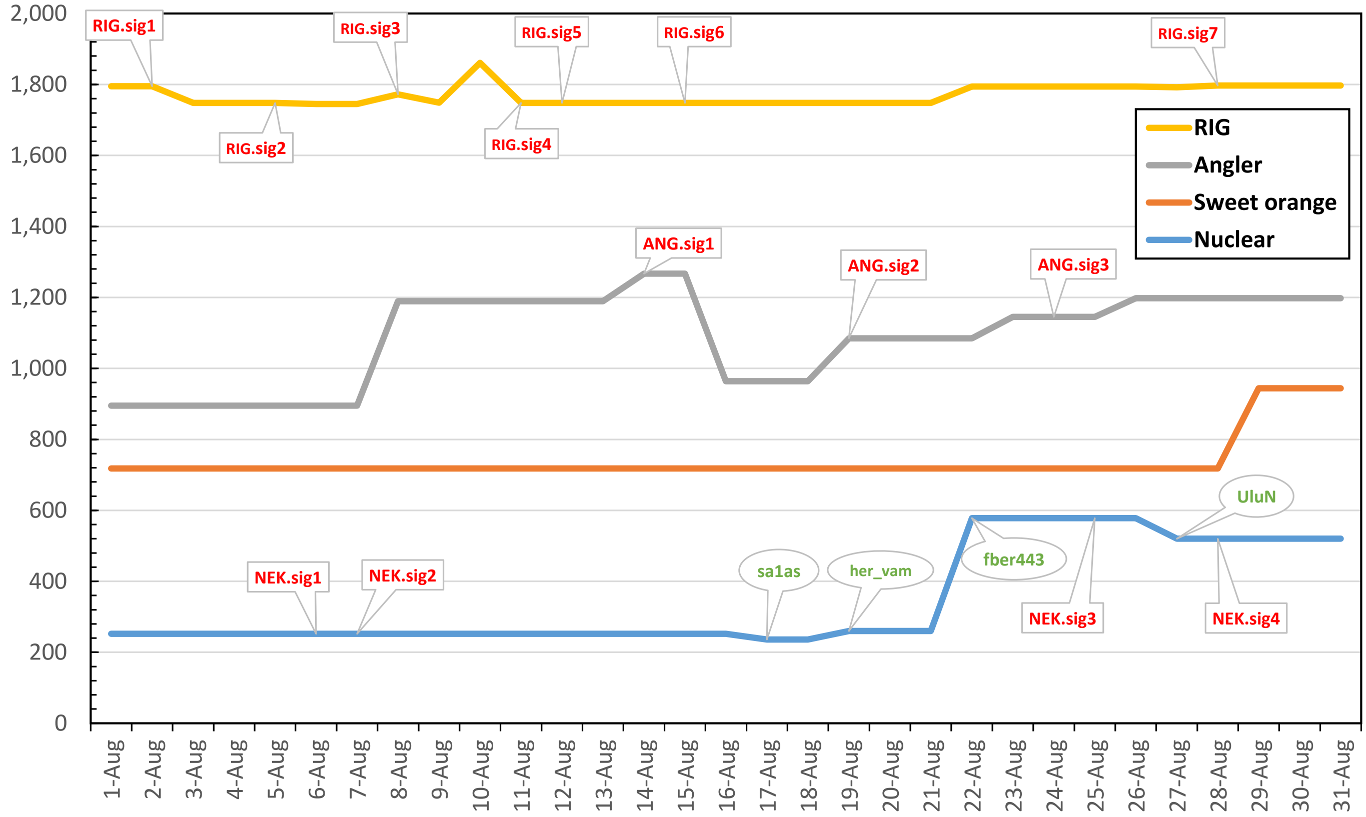


Fig. 22: Signature lengths over time for a month-long time window. Red call-outs are new signatures issued by AV. Green oval call-outs show delimiter changes in NUCLEAR EXPLOIT KIT. In this figure, signature names have been anonymized to avoid violating the terms of use.

Figure 22 shows the lengths of KIZZLE-generated signatures over the month-long period of time. We show the length of the signatures, in characters, on the y axis of the graph. Every time there is a "bump" in the line for one of the EKs, this means that KIZZLE decides to create a new signature. To help the reader correlate KIZZLE signatures with manually-generated signatures, we show the labels for hand-crafted signatures created to address these EKs over the same period of time.

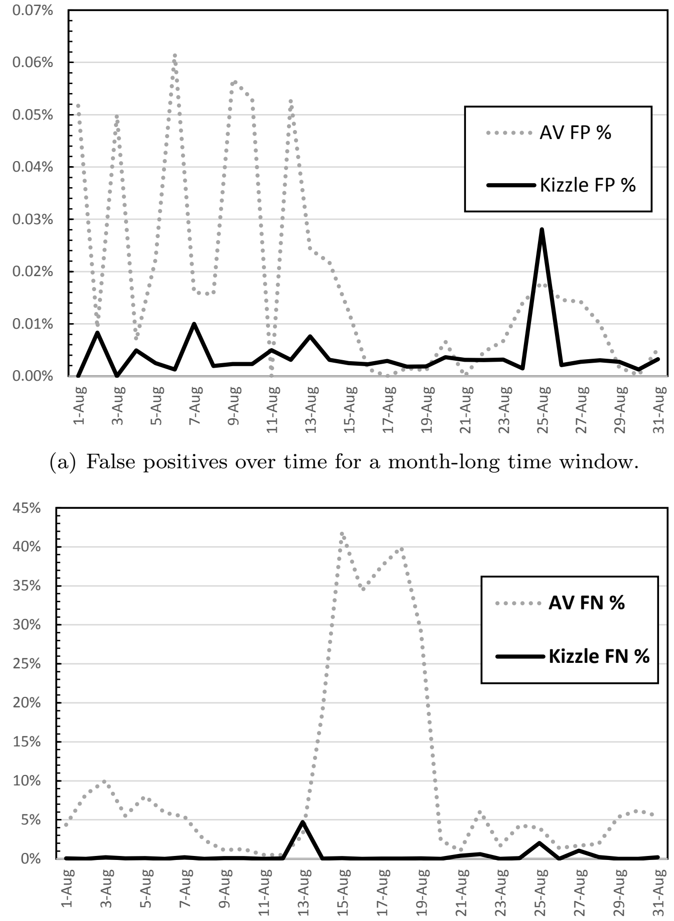
To highlight some of the insights, consider NUCLEAR EXPLOIT KIT (blue line) starting on August 17th. As a packing strategy, NUCLEAR EXPLOIT KIT uses a delimiter, which separates characters `ev` and `al` and `win` and `dow` in Figure 4(b). You can also spot this delimeter-based approach in Figure 18 where a string like `dUluNoUluNcUluNuUluNmUluNeUluNnUluNtUluN` unpacks into `document`. The strategy that NUCLEAR EXPLOIT KIT uses is to change this delimiter frequently because the malware writer suspects that AV signatures will try to match this code.

The green oval call-outs in the figure show signature-avoiding changes to the delimiter which is part of kit evolution. The KIZZLE algorithm can immediately react to these minor changes in the body of the kit, as indicated by daily changes in KIZZLE signatures. To contrast with manually-produced signatures, the first AV signature that we see responding to these changes emerges on August 25th. Note that it may be the case the the AV signature did not need to be updated to be effective, but the figure illustrates that KIZZLE will automatically respond to kit changes on a day-by-day basis.

E. Precision of KIZZLE*-Generated Signatures*

The quality of any anti-virus solution is based on its ability to find most viruses with a very low false positive rate. Our goal for KIZZLE is to provide rates comparable to human-written AV signatures when tested over the month of data we collected. Figure 25 shows false positive and false negative rates for KIZZLE compared to those for AV. Overall, false positive rates are lower for KIZZLE (except for a period between August 21 and August 29th). Figure 27 shows a representative false positive. False positive rates for KIZZLE overall are very small: under 0.03%.

Figure 26 shows some of the details of our evaluation. The kit that gave KIZZLE the most challenge was RIG, which occurred with low frequency in our sample set. RIG also changed more on a daily basis, as illustrated in Figure 20. To be able to confirm false positives and false negatives, we manually evaluated X samples. To obtain the ground truth, we took the *union* of samples matched



(a) False positives over time for a month-long time window.

(b) False negatives over time for a month-long time window.

Fig. 25: False positives and false negatives over time for a month-long time window: Kizzle vs. AV.

EK	Ground truth	AV FP	AV FN	Kizzle FP	Kizzle FN
Nuclear	6,106	1	1,671	25	8
Sweet Orange	11,315	0	2	0	1
Angler	40,026	635	4,213	0	196
RIG	1,409	11	30	241	144
Sum	58,856	647	7,587	266	349

Fig. 26: False positives and false negatives: absolute counts comparing Kizzle vs. AV.

by both AV signatures and the Kizzle approach and examined the overlap. We were able to manually review all samples over the month of August within the union of the two approaches. The manual validation time is a tedious and sometimes frustrating manual process. We estimate the overall manual examination time to be 10–15 hours.

Notably, false negative rates for Kizzle are smaller than those for AV as well. This shows that Kizzle successfully *balances* false positive and false negative requirements. In particular, there is a spike in false negatives between August 13 and August 21st for AV which Kizzle does not suffer from. The majority of these false negatives are due to AV's failure to detect a variant of Angler, as we showed in Figure 8.

```
\},toString:\(\{\}\)\.constructor\.prototype\
.toString,isPlainObject:function\(c\)\{vara=
this,bif\(!c\|\|a\.rgx\.any\.test\(a\.toString\
.call\(c\)\)\|\|c\.window==c\|\|a\.rgx\.num\
.test\(a\.toString\.call\(c\.nodeType\)\)\)\
{return0\}try\{if\(!a\.hasOwn\(c,"constructor"\)
&&!a\.hasOwn\(c\.constructor\.prototype,
"isPrototypeOf"\)\)\{return0\}\}catch\(b\)\
{return0\}return1\},isDefined:function\(b\)\
{returntypeofb!="undefined"\},isArray:
function\(b\)\{returnthis\.rgx\.arr\.test\
(this\.toString\.call\(b\)\)\},isString:
function\(b\)\{returnthis\.rgx\.str\.test\
(this\.toString\.call\(b\)\)\},isNum:function\(b\)
```

Fig. 27: A false positive for Kizzle: this sample extracted from `PluginDetect` shares a very high (79%) overlap with Nuclear exploit kit.

Overall, the false negative rate for Kizzle is under 5% for the month of August. While a fundamental weakness of the Kizzle approach is that if the kit changes *drastically* overnight theKizzle is no longer able to connect previous samples to subsequent ones, for the period of time we examined, we did not see this in practice. This is because kit authors reuse either the unpacked body of the kit (Figure 20), or because they reuse the packer. When we experience false negatives, it is generally because of changes in the kit that are not numerous enough in our grayware stream to warrant a separate cluster. An example of this is a small bump in false negatives for Angler on August 13th in Figure 8, which produced some, but not enough new variants of Angler for Kizzle to produce a new signature.

V. Discussion

Our approach combines automatically building signatures on the packed versions of exploit kits by reasoning about maliciousness based on comparing the unpacked versions to previous known attacks.

While our initial results are promising, there are still many questions that remain and possible threats to validity of our approach. First, we depend on reliably unpacking malware samples that are collected either by crawling or via detection in a deployed browser. We have had previous success collecting malware samples via crawling but detection in a browser would also be valuable at the cost of additional overhead. While there are situations in which this overhead may be unacceptable, we believe there are approaches, including sampling, that can mitigate the overhead.

Second, our results are based on studying the behavior of four exploit kits over a period of one month. We have looked at the same kits over a longer period (and observed consistent behavior) and these kits do represent a significant fraction of all malware we observe in our data stream, but our experiments are still limited. Longer studies with more kits would provide stronger support for this result.

Also, as with any solution based on clustering and abstracting streams of data, a number of tuning knobs control the effectiveness of the approach. For example, how far apart can clusters be, how many samples do we need to define a cluster, how long should the generated signatures be, etc. Finding the right values for these parameters and adjusting them due to the dynamic nature of a malicious opponent make keeping such a system well-tuned challenging. Likely, observing detection accuracy over time as part of operations with the attacker adjusting to Kizzle is needed.

Our approach is based on the fact that while the effort to change the outer layer of an exploit kit is relatively small, changing the syntax, yet not the semantics, of the inner layer is non-trivial. However, research by Payer has highlighted that in the space of binary files, automated rewriting to achieve just such syntactic changes is feasible [28]. While this work aims specifically at binaries, adoption of such automated rewriting schemes would impair Kizzle's ability to track the kits in their unfolded form, if exploit kits move closer to the "fully polymorphic" model.

Finally, our algorithm generates signatures that match all samples in a given cluster while trying to maximize the length of a signature to avoid false positives, resulting in long signatures. While this helps our approach to ensure low false positive rates, longer signatures potentially make the adversary's job of avoiding them easier, since minor changes to any part of what is being matched by the signature is sufficient to bypass detection. They could also reduce the performance of scanning, however, they would only be applied to JavaScript samples.

VI. Related Work

Below, we cover the three most closely related areas of research in turn.

A. Exploit Kits

Previous work on exploit kits has focused mainly on examining their server-side components. In their work, Kotov *et al.* analyze the server-side code for 24 (partially inactive) different families and found that, similar to what our work has shown, 82% of the analyzed kits use some form of obfuscation of the client payload [20]. They also detail the average age of the exploits involved and show that only about one half of the used exploits are less than two years old, with the oldest exploits reaching the age of six years. Finally, they investigate the code reuse of the server-side code to ascertain whether a common code base was used by the malware authors. While they did find some overlap and sharing between the kits, they could not discover evidence of a common codebase. This resonates with some of our observations, i.e., same AV checking code was incorporated into different kits over a period of several weeks.

Additional research into the server-side components has been conducted by De Maio *et al.* [8]. Contrary to the results from [20], the authors conclude from the large overlap of exploit code that several of the analyzed kits are derived from one another. Using static taint-tracking and a constraint solver, they are able to produce combinations of both `User-Agents` and GET parameters such that an infection is more likely to occur. The authors propose to use such signatures to either "milk" a known exploit kit to extract as many different exploits as possible or to deploy honey-clients which are more likely to get infected.

Esthe and Venkatakrishnan recently proposed an application of machine learning to determine if a given URL is part of an exploit kit [10]. They focus on two types of exploit kit-specific characteristics, namely the fact that EKs are *attack-centric* — e.g. they utilize client targeting and obfuscation — and use *self-defense* approaches, e.g., IP blocking or cloaking [19]. They also discover that several exploit kits actively probe URL blacklists and AV signatures databases, enabling the EK author to quickly react to such countermeasures. Using their proposed scheme, they were able to determine with a high accuracy if a given URL belonged to an exploit kit, and outperform existing approaches such as Wepawet [6].

Apart from research that has focused on the technical aspects of exploit kits, Grier *et al.* [12] have conducted an analysis into the effects exploit kits have on the malware ecosystem, finding that the analyzed kits are used to deliver several different families of malware. They show that exploit kits are an integral part of that ecosystem, putting additional emphasis of effective countermeasures. Recently, Allodi *et al.* conducted experiments with the server-side code of exploit kits to determine how resilient kits are to changes to the targeted systems. In doing so, they found that while some exploit kits aim for a lower, yet steadier infection rate over time, other kits are designed to deliver a small number of the latest exploits, achieving a higher infection rate [1].

Additional analyses have been conducted to analyze exploits used in attacks. Bilge *et al.* show that exploits, which were later on also used in exploit kits, could be found in the wild as zero-days before the disclosure of the targeted vulnerability by the vendor [3]. They show that even with AV that can react to known threats, the window of exposure to zero-days is often longer than expected. In 2011, Dan Guido present a case study highlighting the fact that exploit kits encountered by his customers typically incorporated exploits from whitehats or APTs, rather than actually using a zero-day in their attacks [14].

B. Drive-by Attacks

Drive-by downloads or drive-by attack have received much attention [5, 12, 30]. Many studies [24, 30, 31, 35, 38, 39] rely on a combination of high- and low-interaction client honeypots to visit a large number of sites, detecting suspicious behavior in environment state after being compromised

Below we mention some of the more closely related projects. Apart from the work that specifically investigates issues related to exploit kits, researchers have also focused on drive-by downloads. In Zozzle, Curtsinger *et al.* develop a solution that can detect JavaScript malware in the browser using static analysis techniques to classify a given piece of JavaScript as malicious [7]. This system uses a naïve Bayes classifier to finding instances of known, malicious JavaScript. A similar approach was followed by Rieck *et al.* for CUJO, in which the detection and prevention components were deployed in a proxy rather than the browser [33].

Cova *et al.* describe JSAND [6] for analyzing and classifying web content based on static and dynamic features. Their system provides a framework to emulate JavaScript code and determine characteristics that are typically found in malicious code.

Ratanaworabhan *et al.* describe Nozzle, a dynamic system that uses a global heap health metric to detect heap-spraying, a common technique used in modern browser exploits [32].

In addition to detection, researchers have also analyzed ways to mitigate the effects of drive-by download attacks. To this end, Egele *et al.* check strings that are allocated during runtime for patterns that resemble shellcode and ensure that this code is never executed [9]. Lu *et al.* propose a system called BLADE, which effectively ensures that all executable files that are downloaded via the browser are automatically sandboxed such that they can only be executed with explicit user consent [23]. Additionally, Zhang *et al.* provide a means of identifying malware distribution networks (*MDNs*), which are used to host malware that is being download to the victim's machine. After these have been identified, their system ARROW can generate URL signatures which can be used to block these malicious downloads.

Kapravelos *et al.* present Revolver, a system that leverages the fact that in order to avoid detection by emulators or honey clients, authors of exploits use small syntactic changes to throw of such detection tools. In order to find such evasive malware, they compare the structure of two pieces of JavaScript, allowing them to determine these minor changes [16].

C. Signature Generation

Automated signature generation based has been researched to counter both network-based attacks and generate AV-like signatures for malicious files. In 2003, Singh *et al.* proposed an automated method to detect previously unknown worms based on traffic characteristics und subsequently create content-based signatures [36]. In the following year, two research groups presented similar works, generating signatures from honeypot [21] and DMZ traffic [18]. In subsequent years, additional research has focused on improving false positive rates of such systems [27], enabling privacy-preserving exchange of signatures to quickly combat detected attacks [37] and shifting the detection towards commonalities between all different attacks against a single vulnerable service [22, 26]

Work by Brumley *et al.* proposes a deeper analysis of the vulnerabilities rather than exploits to detect malicious packets and subsequently create matching signatures [4]. The concept of clustering HTTP traffic was then used in 2010 by Perdisci *et al.* to find similar patterns in different packets to improve the quality of generated signatures [29].

While much focus has been on the detection of network-based attacks, research into automatic generation of virus signatures dates back to 1994, when Kephart and Arnold propose a system that leverages a large base of benign software to infer which byte sequences in malicious binaries are unlikely to cause false positives if used a signature [17].

In recent years, this idea was picked up when Griffin *et al.* presented Hancock, which determines the probability that an arbitrary byte sequence occurs in a random file and improves the selection of signature candidates by automatically identifying library code in malicious files. This allows them to ensure that signatures are not generated on such shared code, helping them achieve a false positive rate of 0.1% [13]. One way to think of Kizzle is that it is a specialized, narrowly-focused tool which achieves a rate that is lower still.

VII. Conclusions

This paper proposes Kizzle, a malware signature compiler that targets exploit kits. Kizzle automatically identifies malware clusters as they evolve over time and produces signatures that can be applied to detect malware with a lower false negative and similar false positive rates, when compared to hand-written anti-virus signatures. While we have seen a great deal of consolidation in the space of web malware, which leads to sophisticated attacks being accessible to a broad range of attackers, we believe that Kizzle can tip the balance in favor of the defender.

Kizzle is designed to run in the cloud and scale to large volumes of streaming data. Our longitudinal evaluation shows that Kizzle produces high-accuracy signatures. When evaluated over a month-long period in August 2014, false positive rates for Kizzle are under 0.03%, while the false negative rates are under 5%.

Acknowledgments

We greatly appreciate the cooperation and help we received from Dennis Batchelder, Edgardo Diaz, Jonathon Green, and Scott Molenkamp in the course of working on this project.

References

[1] L. Allodi, V. Kotov, and F. Massacci. Malwarelab: Experimentation with cybercrime attack tools. In *Workshop on Cyber Security Experimentation and Test*, 2013.

[2] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2), 1987.

[3] L. Bilge and T. Dumitras. Before we knew it: An empirical study of zero-day attacks in the real world. In *ACM Conference on Computer and Communications Security*, 2012.

[4] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *IEEE Symposium on Security and Privacy*, 2006.

[5] J. Caballero, C. Grier, C. Kreibich, and V. Paxson. Measuring pay-per-install: The commoditization of malware distribution. In *USENIX Security Symposium*, 2011.

[6] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In *International World Wide Web Conference*, 2010.

[7] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. Zozzle: Low-overhead mostly static JavaScript malware detection. In *USENIX Security Symposium*, 2011.

[8] G. De Maio, A. Kapravelos, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Pexy: The other side of exploit kits. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2014.

[9] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. 2009.

[10] B. Eshete and V. N. Venkatakrishnan. Webwinnow: Leveraging exploit kit workflows to detect malicious urls. In *Conference on Data and Application Security and Privacy*, 2014.

[11] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *International Conference on Knowledge Discovery and Data Mining*, 1996.

[12] C. Grier, L. Ballard, J. Caballero, N. Chachra, C. J. Dietrich, K. Levchenko, P. Mavrommatis, D. McCoy, A. Nappa, A. Pitsillidis, et al. Manufacturing compromise: the emergence of exploit-as-a-service. In *ACM Conference on Computer and Communications Security*, 2012.

[13] K. Griffin, S. Schneider, X. Hu, and T.-C. Chiueh. Automatic generation of string signatures for malware detection. In *Recent Advances in Intrusion Detection*, 2009.

[14] D. Guido. A case study of intelligence-driven defense. *IEEE Security and Privacy*, 2011.

[15] J. Jones. The state of Web exploit kits. In *BlackHat US*, 2012.

[16] A. Kapravelos, Y. Shoshitaishvili, M. Cova, C. Kruegel, and G. Vigna. Revolver: An automated approach to the detection of evasive web-based malware. In *USENIX Security Symposium*, 2013.

[17] J. O. Kephart and W. C. Arnold. Automatic extraction of computer virus signatures. In *Virus Bulletin International Conference*, 1994.

[18] H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *USENIX Security Symposium*, 2004.

[19] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: Decloaking internet malware. In *IEEE Symposium on Security and Privacy*, 2012.

[20] V. Kotov and F. Massacci. Anatomy of exploit kits: Preliminary analysis of exploit kits as software artefacts. In *International Conference on Engineering Secure Software and Systems*, 2013.

[21] C. Kreibich and J. Crowcroft. Honeycomb: creating intrusion detection signatures using honeypots. *Workshop on Hot Topics in Networks*, 2003.

[22] Z. Li, M. Sanghi, Y. Chen, M.-Y. Kao, and B. Chavez. Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience. In *IEEE Symposium on Security and Privacy*, 2006.

[23] L. Lu, V. Yegneswaran, P. Porras, and W. Lee. Blade: an attack-agnostic approach for preventing drive-by malware infections. In *ACM conference on Computer and Communications Security*, 2010.

[24] A. Moshchuk, T. Bragin, S. D. Gribble, and H. M. Levy. A crawler-based study of spyware in the web. In *Network and Distributed System Security Symposium*, 2006.

[25] C. Nachenberg. Computer virus-antivirus coevolution. *Communications of the ACM*, 40(1):46–51, Jan. 1997.

[26] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *IEEE Symposium on Security and Privacy*, 2005.

[27] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Network and Distributed System Security Symposium*, 2005.

[28] M. Payer. Embracing the new threat: Towards automatically self-diversifying malware. In *The Symposium on Security for Asia Network*, 2014.

[29] R. Perdisci, W. Lee, and N. Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *USENIX Symposium on Networked Systems Design and Implementation*, 2010.

[30] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iFRAMEs point to us. In *USENIX Security Symposium*, 2008.

[31] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The ghost in the browser: Analysis of web-based malware. In *Workshop on Hot Topics in Understanding Botnets*, 2007.

[32] P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *USENIX Security Symposium*, 2009.

[33] K. Rieck, T. Krueger, and A. Dewald. Cujo: efficient detection and prevention of drive-by-download attacks. In *Annual Computer Security Applications Conference*, 2010.

[34] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *International Conference on Management of Data*, 2003.

[35] C. Seifert, V. Delwadia, P. Komisarczuk, D. Stirling, and I. Welch. Measurement study on malicious web servers in the .nz domain. In *Australasian Conference on Information Security and Privacy*, 2009.

[36] S. Singh, C. Estan, G. Varghese, and S. Savage. Earlybird system for real-time detection of unknown worms. Technical report, 2003.

[37] K. Wang, G. Cretu, and S. J. Stolfo. Anomalous payload-based worm detection and signature generation. In *Recent Advances in Intrusion Detection*, 2006.

[38] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. T. King. Automated web patrol with Strider HoneyMonkeys: Finding web sites that exploit browser vulnerabilities. In *Network and Distributed System Security Symposium*, 2006.

[39] J. Zhuge, T. Holz, C. Song, J. Guo, X. Han, and W. Zou. Studying malicious websites and the underground economy on the Chinese web. *Managing Information Risk and the Economics of Security*, 2008.