PlatPal: Detecting Malicious Documents with Platform Diversity

Meng Xu and Taesoo Kim, *Georgia Institute of Technology*

https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/xu-meng

This paper is included in the Proceedings of the
26th USENIX Security Symposium

August 16–18, 2017 • Vancouver, BC, Canada

ISBN 978-1-931971-40-9

Open access to the Proceedings of the
26th USENIX Security Symposium
is sponsored by USENIX

PLATPAL: Detecting Malicious Documents with Platform Diversity

Meng Xu and Taesoo Kim
Georgia Institute of Technology

Abstract

Due to the continued exploitation of Adobe Reader, malicious document (maldoc) detection has become a pressing problem. Although many solutions have been proposed, recent works have highlighted some common drawbacks, such as parser-confusion and classifier-evasion attacks.

In response to this, we propose a new perspective for maldoc detection: platform diversity. In particular, we identify eight factors in OS design and implementation that could cause behavioral divergences under attack, ranging from syscall semantics (more obvious) to heap object metadata structure (more subtle) and further show how they can thwart attackers from finding bugs, exploiting bugs, or performing malicious activities.

We further prototype PLATPAL to systematically harvest platform diversity. PLATPAL hooks into Adobe Reader to trace internal PDF processing and also uses sandboxed execution to capture a maldoc's impact on the host system. Execution traces on different platforms are compared, and maldoc detection is based on the observation that a benign document behaves the same across platforms, while a maldoc behaves differently during exploitation. Evaluations show that PLATPAL raises no false alarms in benign samples, detects a variety of behavioral discrepancies in malicious samples, and is a scalable and practical solution.

1 Introduction

Cyber attackers are turning to document-based malware as users wise up to malicious email attachments and web links, as suggested by many anti-virus (AV) vendors [39, 50, 54, 57]. Users are generally warned more on the danger of executable files by browsers, email agents, or AV products, while documents such as PDFs are treated with much less caution and scrutiny because of the impression that they are static files and can do little harm.

However, over time, PDF specifications have changed. The added scripting capability makes it possible for documents to work in almost the same way as executables, including the ability to connect to the Internet, run processes, and interact with other files/programs. The growth of content complexity gives attackers more weapons to launch powerful attacks and more flexibility to hide malicious payload (e.g., encrypted, hidden as images, fonts or Flash contents) and evade detection.

A maldoc usually exploits one or more vulnerabilities in its interpreter to launch an attack. Fortunately (or unfortunately), given the increasing complexity of document readers and the wide library/system component dependencies, attackers are presented with a large attack surface. New vulnerabilities continue to be found, with 137 published CVEs in 2015 and 227 in 2016 for Adobe Acrobat Reader (AAR) alone. The popularity of AAR and its large attack surface make it among the top targets for attackers [25], next to browsers and OS kernels. After the introduction of a Chrome-like sandboxing mechanism [2], a single exploit can worth as high as $70k in pwn2own contest [21]. The collected malware samples have shown that many Adobe components have been exploited, including element parsers and decoders [37], font managers [28], and the JavaScript engine [22]. System-wide dependencies such as graphics libraries [23] are also on attackers' radar.

The continued exploitation of AAR along with the ubiquity of the PDF format makes maldoc detection a pressing problem, and many solutions have been proposed in recent years to detect documents bearing malicious payloads. These techniques can be classified into two broad categories: static and dynamic analysis.

Static analysis, or signature-based detection [14, 27, 31, 33, 34, 36, 46, 52, 59], parses the document and searches for indications of malicious content, such as shellcode or similarity with known malware samples. On the other hand, dynamic analysis, or execution-based detection [45, 48, 58], runs partial or the whole document and traces for malicious behaviors, such as vulnerable API calls or return-oriented programming (ROP).

However, recent works have highlighted some common drawbacks of these solutions. Carmony *et al*. [11] show that the PDF parsers used in these solutions might have overly simplified assumptions about the PDF specifications, leading to an incomplete extraction of malicious payloads and failed analysis. It has also been demonstrated that machine-learning-based detection could potentially be evaded in principled and automatic ways [35, 53, 65]. In addition, many solutions focus only on the JavaScript parts and ignore their synergy with other PDF components in launching attacks. Therefore, even though modern AV products support PDF-exploit detection, they cannot quickly adapt to novel obfuscation techniques even if the latter constitute only minor modifications of existing exploits [55]. AV products also exhibit problems providing protection against zero-day attacks, due to the lack of attack procedures and runtime traces.

In this paper, we propose PLATPAL, a maldoc detection scheme that analyzes the behavioral discrepancies of malicious document files on different platforms (e.g., Windows or Macintosh (Mac)). Unlike the static and dynamic detection schemes that rely on existing malware samples to construct heuristics, PLATPAL is based on a completely different set of insights: 1) a benign document behaves the same (in a certain level) across platforms, while 2) a malicious document causes diverged behaviors when launching exploits on different platforms.

The first assumption can be empirically verified by opening many benign samples that use a variety of PDF features across platforms. To support the second assumption, we investigated the factors in OS implementation that could cause behavioral divergences when under attack and identified eight such factors, ranging from syscall semantics (more obvious) to heap object metadata structure (more subtle). We further show how they can be used to thwart attackers in finding bugs, exploiting bugs, or performing malicious activities.

PLATPAL is based on these insights. To detect whether a document has malicious payload, PLATPAL opens it with the same version of AAR instances, but running on top of different operating systems. PLATPAL records the runtime traces of AAR while processing the document and subsequently compares them across platforms. Consensus in execution traces and outputs indicates the health of the document, while divergences signal an attack.

Although the process sounds simple and intuitive, two practical questions need to be addressed to make PLATPAL work: 1) what "behaviors" could be potentially different on different platforms? and 2) how can they be universally traced? PLATPAL traces and compares two types of behaviors. Internal behaviors include critical functions executed by AAR in the PDF processing cycle, such as loading, parsing, rendering, and script execution. External behaviors include filesystem operations, network activities, and program launches. This aligns with typical malware analysis tools such as Cuckoo sandbox [44].

It is worth highlighting that PLATPAL should not be considered as a competitor to current malware analysis tools such as Cuckoo [44] as they rely on different assumptions. Current tools rely heavily on the availability of a blacklist (or whitelist) of OS-wide activities are already available such that a sample's behaviors can be vetted against the list. This approach works well for known malware but might lost its advantage against 0-day PDF exploits. On the other hand, PLATPAL does not require the such a list to function and only relies on the fact that it is difficult for an attacker to craft a malicious PDF that exploits AAR in exactly the same way in both Windows and Mac platforms.

PLATPAL is evaluated against 1030 benign samples that use various features in the PDF specifications and reports no discrepancies in their traces, i.e., no false alarms. For a collection of 320 maldoc samples exploiting 16 different CVEs, PLATPAL can detect divergences in 209 of them with an additional 34 samples crashing both AAR instances. The remainder are undetected for various reasons, such as targeting an old and specific version of AAR or failure to trigger malicious activities. PLATPAL can finish a scan of the document in no more than 24 seconds per platform and requires no manual driving.

Paper contribution. In summary, this paper makes the following contributions:

- We propose to execute a document across different platforms and use behavioral discrepancies as an indicator for maldoc detection.
- We perform in-depth analysis and categorization of platform diversities and show how they can be used to detect maldoc attacks.
- We prototype PLATPAL based on these insights. Evaluations prove that PLATPAL is scalable, does not raise false alarms, and detects a variety of behavioral discrepancies in malicious samples.

We plan to open source PLATPAL to prompt using platform diversity for maldoc detection and also launch a PDF maldoc scanning service for public use.

2 Maldoc Detection: A Survey

Existing maldoc detection methods can be classified broadly into two categories: 1) dynamic analysis, in which malicious code is executed and examined in a specially instrumented environment; and 2) static analysis, in which the detection is carried out without code execution. A summary of existing methods is presented in Table 1.

Category	Focus	Detection Technique	Parser ?	ML ?	Pattern ?	Evasion / Drawbacks
Static	JavaScript	Lexical analysis [27]	Yes	Yes	Yes	Heavy obfuscation, Code loading
	JavaScript	Token clustering [59]	Yes	Yes	Yes	
	JavaScript	API reference classification [14]	Yes	Yes	Yes	
	JavaScript	Shellcode and opcode signature [31]	No	No	Yes	
	Metadata	Linearized object path [36]	Yes	Yes	Yes	Mimicry [53], Reverse mimicry [35]
	Metadata	Hierarchical structure [33, 52]	Yes	Yes	Yes	
	Metadata	Content meta-features [46]	Yes	Yes	Yes	
	Both	Many above-mentioned heuristics [34]	Yes	Yes	Yes	
Dynamic	JavaScript	Shellcode and opcode signature [58]	Yes	No	Yes	Incompatible JS engine, Non-script based attacks
	JavaScript	Known attack patterns [45]	Yes	No	Yes	
	JavaScript	Memory access patterns [48]	Yes	No	Yes	
	JavaScript	Common maldoc behaviors [29]	No	No	Yes	Zero-day exploits
	Document	Violation of memory access invariants [62]	No	No	No	ROP and JIT-Spraying

Table 1: A taxonomy of malicious PDF document detection techniques. This taxonomy is partially based on a systematic survey paper [40] with the addition of works after 2013 as well as summaries parser, machine learning, and pattern dependencies and evasion techniques.

2.1 Static Techniques

One line of static analysis work focuses on JavaScript content for its importance in exploitation, e.g., a majority (over 90% according to [58]) of maldocs use JavaScript to complete an attack. PJScan [27] relies on lexical coding styles like the number of variable names, parenthesis, and operators to differentiate benign and malicious JavaScript code. Vatamanu *et al*. [59] tokenizes JavaScript code into variables types, function names, operators, etc. and constructs clusters of tokens as signatures for benign and malicious documents. Similarly, Lux0r [14] constructs two sets of API reference patterns found in benign and malicious documents, respectively, and uses this to classify maldocs. MPScan [31] differs from other JavaScript static analyzers in a way that it hooks AAR and dynamically extracts the JavaScript code. However, given that code analysis is still statically performed, we consider it a static analysis technique.

A common drawback of these approaches is that they can be evaded with heavy obfuscation and dynamic code loading (except for [31] as it hooks into AAR at runtime). Static parsers extract JavaScript based on pre-defined rules on where JavaScript code can be placed/hidden. However, given the flexibility of PDF specifications, it is up to an attacker's creativity to hide the code.

The other line of work focuses on examining PDF file metadata rather than its actual content. This is partially inspired by the fact that obfuscation techniques tend to abuse the flexibility in PDF specifications and hide malicious code by altering the normal PDF structure. PDF Malware Slayer [36] uses the linearized path to specific PDF elements (e.g., `/JS`, `/Page`, etc) to build maldoc classifiers. Srndic *et al*. [52] and Maiorca *et al*. [33] go one step further and also use the hierarchical structure for classification. PDFrate [46] includes another set of features such as the number of fonts, the average length of streams, etc. to improve detection. Maiorca *et al*. [34] focuses on both JavaScript and metadata and fuses many of the above-mentioned heuristics into one procedure to improve evasion resiliency.

All methods are based on the assumption that the normal PDF element hierarchy is distorted during obfuscation and new paths are created that could not normally exist in benign documents. However, this assumption is challenged by two attacks. Mimicus [53] implements mimicry attacks and modifies existing maldocs to appear more like benign ones by adding empty structural and metadata items to the documents with no actual impact on rendering. Reverse mimicry [35] attack, on the contrary, attempts to embed malicious content into a benign PDF by taking care to modify it as little as possible.

2.2 Dynamic Techniques

All surveyed dynamic analysis techniques focus on embedded JavaScript code only instead of the entire document. MDScan [58] executes the extracted JavaScript code on a customized `SpiderMonkey` interpreter and the interpreter's memory space is constantly scanned for known forms of shellcode or malicious opcode sequences. PDF Scrutinizer [45] takes a similar approach by hooking the `Rhino` interpreter and scans for known malicious code patterns such as heap spray, shellcode, and vulnerable method calls. ShellOS [48] is a lightweight OS designed to run JavaScript code and record its memory access patterns. During execution, if the memory access sequences match a known malicious pattern (e.g., ROP, critical syscalls or function calls, etc), the script is considered malicious.

Although these techniques are accurate in detecting malicious payload, they suffer from a common problem:

an incompatible scripting environment. AAR's JavaScript engine follows not only the ECMA standard [18], but also the Acrobat PDF standard [1] (e.g., Adobe DOM elements). Therefore, without emulation, objects like `doc`, `app`, or even `this` (which are very common in both benign and malicious documents) will not function correctly. In addition, malicious payload can be encoded as a font or an image object in the document [37], which will neither be extracted nor detected. Certain attacks might also exploit the memory layout knowledge such as the presence of ROP gadgets or functions available in AAR and its dependent libraries, which is hard to emulate in an external analysis environment.

Instead of emulating the JavaScript execution environment, Liu *et al*. [29] instruments the PDF document with context monitoring code and uses AAR's own runtime to execute JavaScript code and hence is not affected by the incompatibility problem. However, the instrumented code only monitors common and known patterns of malicious behavior such as network accesses, heap-spraying, and DLL-injection, etc, which are not fully generic and have to be extended when new anti-detection measures of malicious code come up. CWXDetector [62] proposes a W⊕X-like approach to detect illegitimate code injected by maldocs during execution. But similar to W⊕X, its effectiveness is compromised in the presence of ROP and JIT-spraying.

2.3 Summary and Motivations

Surveying the maldoc detection techniques yields several interesting observations:

Parser reliance. Since a document consists of both data (e.g., text) and executable (e.g., script) components, a common pattern is to first extract the executable components and further examine them with either static or dynamic analysis. To this end, a parser that is capable of parsing PDF documents the same way as AAR does is generally assumed. As shown in Table 1, all but three methods use either open-sourced or their home-grown parsers and assume their capability. However, Carmony *et al*. [11] shows that these parsers are typically incomplete and have oversimplified assumptions in regard to where JavaScript can be embedded, therefore, *parser confusion attacks* can be launched to easily evade their detection.

Machine learning reliance. Machine learning techniques are heavily used in maldoc detection, especially in static analysis, because of their ability in classification/clustering without prior knowledge of the pattern. As shown in Table 1, seven out of 13 methods use machine learning to differentiate benign and malicious documents, while another four methods can also be converted to use machine learning for heuristics mining. However, recently proposed adversarial machine learning techniques [20, 42, 65] raise serious doubts about the effectiveness of classifiers based on superficial features in the presence of adversaries. For example, Xu *et al*. [65] is capable of automatically producing evasive maldoc variants without knowledge about the classifier, in a way similar to genetic programming.

Structural/behavioral discrepancy. An implicit assumption in the surveyed methods is that structural/behavioral discrepancies exist between benign and malicious documents and such discrepancies can be observed. Since the document must follow a public format specification, commonalities (structural or behavioral) are expected in benign documents. If a document deviates largely from the specification or the common patterns of benign samples, it is more likely to be a malicious document. However, such an assumption is challenged by the Mimicus [53] and reverse mimicry [35] attacks in a way that a maldoc can systematically evades detection if an attacker knows the patterns used to distinguish benign and malicious documents. In addition, deriving the discrepancy patterns requires known malware samples. Therefore, all but one methods in Table 1 require known malware samples either to learn patterns automatically or to manually define patterns based on heuristics, expectations, or experience. This restricts their capabilities in detecting zero-day attacks where no prior knowledge can be obtained.

Full dynamic analysis. It is worth noting that only one dynamic detection method performs analysis on the entire file; instead, the rest of the methods perform analysis on the extracted JavaScript code only. This is in contrast with traditional sandboxed malware analysis such as Cuckoo [44] or CWSandbox [63], which executes the malware and examines its behavior and influence on the host operating system during runtime. One reason could be because the maldoc runs on top of AAR, which itself is a complex software and leaves a large footprint on the host system. The traces of maldoc execution are hidden in the large footprint, making analysis much harder.

Motivation. The development of PLATPAL is motivated by the above-mentioned problems in maldoc detection research. We design PLATPAL to: 1) share the same view of the document as the intended interpreter (i.e., AAR in this paper); 2) use simple heuristics that do not rely on machine learning; 3) detect zero-day attacks without prior knowledge; 4) capture the maldoc's influence on the host system; and 5) be complementary to the surveyed techniques to further raise the bar for maldoc attackers.

3 Platform Diversity

This section focuses on understanding why platform diversity can be an effective approach in detecting maldoc

attacks. We first present a motivating example and then list the identified factors that are important in launching attacks, but are different on Windows and Mac platforms. We further show how to use them to thwart attackers and concretize it with four case studies. We end by discussing platform-detection techniques that a maldoc can use and the precautions PLATPAL should take.

3.1 A Motivating Example

In December 2012, researchers published a public proof-of-concept exploit for AAR [37]. This exploit attacks a heap overflow vulnerability found in the PDF parser module when parsing an embedded BMP RLE encoded image (CVE-2013-2729). By simply opening the maldoc, the AAR instance on Windows platform (including Windows 7, 8 and 10) is compromised and the attacker can run arbitrary code with the privileges of the compromised process. During our experiment, we ran this exploit on the Windows version of AAR 10.1.4 and reproduced the attack. However, when we opened the same sample with the Mac version of AAR 10.1.4, the attack failed and no malicious activities were observed.

In fact, in the malware history, Windows has drawn more attraction from attackers than Mac, and the same applies to maldocs. The Windows platform tends to be more profitable because of its market share, especially with enterprise users [38], who heavily use and exchange PDF documents. Therefore, it is reasonable to expect that the majority of maldocs target primarily the Windows platform, as cross-platform exploits are much harder to develop due to the factors discussed later.

The mindset of maldoc attackers and the discrepancy in reacting to malicious payload among different platforms inspire us to use platform diversity as the heuristic for maldoc detection: a benign document "behaves" the same when opened on different platforms while a maldoc could have different "behaviors" when launching exploits on different platforms. In other words, cross-platform support, the power used to make the PDF format and AAR popular, can now be used to defend against maldoc attacks.

3.2 Diversified Factors

We identified eight factors related to launching maldoc attacks but are implemented differently on Windows and Mac platforms.

Syscall semantics. Both syscall numbers and the register set used to hold syscall parameters are different between Windows and Mac platforms. In particular, file, socket, memory, process, and executable operations all have non-overlapping syscall semantics. Therefore, crafting shellcode that executes meaningfully on both platforms is extremely difficult in practice.

Calling conventions. Besides syscalls, the calling convention (i.e., argument passing registers) for userspace function differs, too. While Windows platforms use `rcx`, `rdx`, and `r8` to hold the first three parameters, Mac platforms use `rdi`, `rsi`, and `rdx`. This makes ROP-like attacks almost impossible, as the gadgets to construct these attacks are completely different.

Library dependencies. The different sets of libraries loaded by AAR block two types of exploits: 1) exploits that depend on the existence of vulnerabilities in the loaded libraries, e.g., graphics libraries, font manager, or `libc`, as they are all implemented differently on Windows and Mac platforms; and 2) exploits that depend on the existence of certain functions in the loaded libraries, e.g., `LoadLibraryA`, or `dlopen`.

Memory layout. The offset from the attack point (e.g., the address of the overflowed buffer or the integer value controlled by an attacker) to the target point, be it a return address, GOT/PLT entry, vtable entry, or even control data, is unlikely to be the same across platforms. In other words, directing control-flow over to the sprayed code can often be blocked by the discrepancies in memory layouts across platforms.

Heap management. Given the wide deployment of ASLR and DEP, a successful heap buffer overflow usually leads first to heap metadata corruption and later exploits the heap management algorithm to obtain access to control data (e.g., vtable). However, heap management techniques are fundamentally different between Windows and Mac platforms. Therefore, the tricks to corrupt metadata structures maintained by `segment heap` [67] (Windows allocator) will not work in the `magazine malloc` [5] (Mac allocator) case and vice versa.

Executable format. While Windows platforms generally recognize COM, NE, and PE formats, Mac platforms recognize only the Mach-O format. Therefore, maldocs that attempt to load an executable after exploitation will fail. Although "fat binaries" that can run on multiple CPU architectures exist, we are not aware of an executable format (or any wrapper tools) that is capable of running on multiple platforms.

Filesystem semantics. Windows uses backslashes (\) as path separators, while Mac uses forward slashes (/). In addition, Windows has a prefixed drive letter (e.g., `C:\`) while Mac has a mount point (e.g., the root `/`). Therefore, hard-coded path names, regardless of whether they are in JavaScript or attacker-controlled shellcode, will break on at least one platform. Dynamically generated filenames rely on the fact that certain files exist at a given path, which is unlikely to hold true across platforms.

Expected programs/services. This is heavily relied upon by the dropper or phishing type of maldocs, for example, dropping a malformed MS Office document

that exploits MS Office bugs, or redirecting the user to a malicious website that attacks the Internet Explorer browser. As Mac platforms are not expected to have these programs, such attacks will fail on Mac platforms.

3.3 Attack Categorization

As shown in Figure 1, a typical maldoc attack consists of three steps: 1) finding vulnerabilities, 2) exploiting them to inject attacker-controlled program logic, and 3) profiting by performing malicious activities such as stealing information, dropping backdoors, C&C, etc. The identified diversity factors in §3.2 can help detect maldocs at different stages.

In terms of finding vulnerabilities, exploiting vulnerabilities on platform-specific components can obviously be detected by PLATPAL, as the vulnerable components do not exist on the other platform.

The exploitation techniques can be divided into two subcategories, based on whether an attack exploits memory errors (e.g., buffer overflow, integer overflow, etc) to hijack control-flow or exploits logic bugs (e.g., JavaScript API design flaws).

Memory-error based control-flow hijacking puts a high requirement on the memory content during exploitation. For example, ROP attacks, which are commonly found in maldoc samples, require specific gadgets and precise information on where to find them in order to make powerful attacks. However, these gadgets and their addresses in memory can be easily distorted by the discrepancies in loaded libraries and memory layouts.

On the other hand, exploiting features that are naturally cross-platform supported, e.g., JavaScript hidden API attacks or abusing the structure of PDF document to obfuscate malicious payload, are not subject to the intricacies of runtime memory contents and are more likely to succeed.

Finally, even if an attacker succeeds in the first two steps, the attack can be detected while the maldoc is performing malicious activities, such as executing a syscall, loading a PE-format executable on Mac platforms, or accessing a file that exists only on Windows platforms.

3.4 Case Studies

We use representative examples to show how platform diversity can be used to detect maldoc attacks in each step shown in Figure 1.

Platform-specific bug. One source of platform-specific bugs comes from system libraries that are used by AAR. An example is CVE-2015-2426, an integer overflow bug in the Windows Adobe Type Manager Library. A detailed study can be found at [28]. In this case, opening the maldoc sample on Windows platforms will trigger the exploitation, while nothing will happen when opening it on Mac platforms. In other words, maldocs that exploit bugs in dependent libraries will surely fail on other platforms.

Another source of bugs comes from the AAR implementation itself, and we also found a few cases where the implementation of the same function can be vulnerable on one platform but safe on the other. For example, CVE-2016-4119 is a use-after-free vulnerability in the `zlib` deflating algorithm used by AAR to decompress embedded images [30]. The Mac version of AAR is able to walk through the document and exit gracefully, while AAR on Windows crashes during the rendering stage. A closer look at their execution shows that the decoded image objects are different on these platforms.

Memory error. Due to the deployment of ASLR and DEP in modern operating systems, direct shellcode injection cannot succeed. As a result, attackers exploiting memory errors generally require some form of heap preparation to obtain read/write accesses to control data, and the most common target we observed is vtable.

In the case of [37], the maldoc sample exploits CVE-2013-2729, an integer overflow bug in AAR itself, to prepare the heap to obtain access to a vtable associated with an image object. In particular, it starts by allocating 1000 consecutive memory chunks, each of 300 bytes, a value carefully selected to match the size of the vtable, and subsequently free one in every 10 chunks to create a few holes. It then uses a malformed BMP image of 300 bytes to trigger the integer overflow bug and manages to override the heap metadata, which resides in an attacker-controlled slot (although the attacker does not know which slot before hand). The malformed BMP image is freed from memory, but what is actually freed is the attacker-controlled slot, because of the heap metadata corruption. Later, when the struct containing a vtable is allocated in the same slot (almost guaranteed because of heap defragmentation), the attacker gains access and hijacks control-flow by overriding vtable entries.

However, this carefully constructed attack has two assumptions, which do not hold across platforms: 1) the size of the vtable on Windows and Mac platforms is different; and 2) the heap object metadata structures are different. As a result, overriding the heap metadata on Mac platform yields no observable behaviors.

Logic bugs. Another common attack vector of AAR is the logic bugs, especially JavaScript API design flaws. Unlike attacks that exploit memory errors, JavaScript API attacks generally require neither heap constructions nor ROP-style operations. Instead, they can be launched with as little as 19 lines of JavaScript code, as shown in Figure 2. Gorenc *et al.* [22] further extends this technique to complete remote code execution attacks by abusing hidden JavaScript APIs.

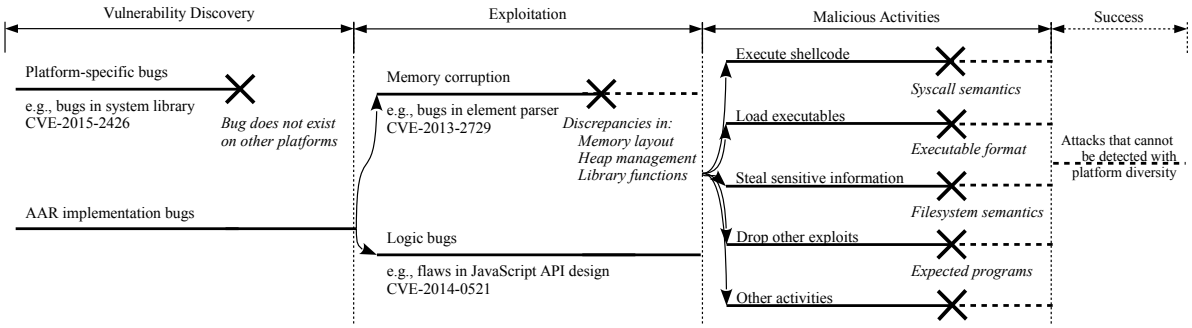

Figure 1: Using platform diversity to detect maldocs throughout the attack cycle. *Italic texts* near × refers to the factors identified in §3.2 that can be used to detect such attacks. A dash line means that certain attacks might survive after the detection.

```
1  var t = {};
2  t.__defineSetter__('doc', app.beginPriv);
3  t.__defineSetter__('user', app.trustedFunction);
4  t.__defineSetter__('settings', function() { throw 1; });
5  t.__proto__ = app;
6  try {
7    DynamicAnnotStore.call(t, null, f);
8  } catch(e) {}
9
10 f();
11 function f() {
12   app.beginPriv();
13   var file = '/c/notes/passwords.txt';
14   var secret = util.stringFromStream(
15       util.readFileIntoStream(file, 0)
16       );
17   app.alert(secret);
18   app.endPriv();
19 }
```

Figure 2: CVE-2014-0521 proof-of-concept exploitation

Besides being simple to construct, these attacks are generally available on both Windows and Mac platforms because of the cross-platform support of the JavaScript. Therefore, the key to detecting these attacks via platform diversity is to leverage differences system components such as filesystem semantics, expected installed programs, etc., and search for execution divergences when they are performing malicious activities. For example, line 15 will fail on Mac platforms in the example of Figure 2, as such a file path does not exist on Mac.

3.5 Platform-aware Exploitation

Given the difficulties of launching maldoc attacks on different platforms with the same payload, what an attacker can do is to first detect which platform the maldoc is running on through explicit or implicit channels and then launch attacks with platform-specific payload.

In particular, the Adobe JavaScript API contains publicly accessible functions and object fields that could return different values when executed on different platforms. For example, `app.platform` returns `WIN` and `MAC` on respective platforms. `Doc.path` returns file path to the document opened, which can be used to check whether the document is opened on Windows or Mac by testing whether the returned path is prefixed with `/c/`.

Another way to launch platform-aware attacks is to embed exploits on two platform-specific vulnerabilities, each targeting one platform. In this way, regardless of on which platform the maldoc is opened, one exploit will be triggered and malicious activities can occur.

In fact, although platform-aware maldocs are rare in our sample collection, PLATPAL must be aware of these attack methods and exercises precautions to detect them. In particular, the possibility that an attacker can probe the platform first before launching the exploit implies that merely comparing external behaviors (e.g., filesystem operations or network activities) might not be sufficient as the same external behaviors might be due to the result of different attacks. Without tracing the internal PDF processing, maldocs can easily evade PLATPAL's detection using platform-specific exploits, for example, by carrying multiple ROP payloads and dynamically deciding which payload to use based on the return value of `app.platform`, or even generating ROP payload dynamically using techniques like JIT-ROP [49].

However, we do acknowledge that, given the complexity of the PDF specification, PLATPAL does not enumerate all possible platform-probing techniques. Therefore, PLATPAL could potentially be evaded through implicit channels we have not discovered (e.g., timing side-channel).

3.6 Platform-agnostic Exploitation

We also identified several techniques that can help "neutralize" the uncertainties caused by platform diversity, including but not limited to heap feng-shui, heap spray, and polyglot shellcode.

Heap feng-shui. By making consecutive heap allocations and de-allocations of carefully selected sizes, an attacker can systematically manipulate the layout of the heap and predict the address of the next allocation or

de-allocation [51]. This increases the chance of obtaining access to critical data such as vtables even without knowing every detail of the system memory allocator.

Heap spray and NOP sled. By repeatedly allocating the attack payload and NOP sled in heap [13], an attacker is alleviated from using precise memory locations for control-flow hijacking; instead, an attacker only needs to ensure that control-flow is redirected to the sprayed area.

Ploylglot shellcode trampoline. Although not seen in the wild, it is possible to construct OS-agnostic shellcode in a similar manner as CPU architecture-agnostic shellcode [17, 64]. The key idea is to find operations that are meaningful in one platform and NOP on the other and use these operations to jump to different code for platform-specific activities.

Although these operations can succeed on both platforms, attacks using these techniques can still be detected by platform diversity. This is because these operations have to be paired with other procedures to complete an end-to-end attack. For example, heap manipulation can succeed but the resulting memory layout might not be suitable for both platforms to land the critical data in attacker-controlled memory because of the discrepancies in heap management, while ploylglot shellcode trampolines can run without crashing AAR, but the attack can still be detected by the malicious activities performed.

4 The PLATPAL Approach

This section presents the challenges and their solutions in designing PLATPAL that harvests platform diversity for maldoc detection.

4.1 Dual-level Tracing

Although the platform diversity heuristic sounds intuitive, two natural questions arise: 1) What "behaviors" could be potentially different across different platforms? and 2) How can they be universally traced and compared?

To answer the first question, "behaviors" must satisfy two requirements: 1) they are available and do not change across platforms and 2) they are the same for benign documents and could be different for maldocs. To this end, we identified two sets of "behaviors" that match these requirements: AAR's internal PDF processing functions (*internal behaviors*) and external impact on the host system while executing the document (*external behaviors*).

For *internal behaviors*, in AAR, PDF documents pass through the PDF processing functions in a deterministic order and trigger pre-defined callbacks sequentially. For example, a callback is issued when an object is resembled or rendered. When comparing execution across platforms, for a benign document, both function execution order and results are the same because of the cross-platform support of AAR, while for a maldoc, the execution trace could be different at many places, depending on how the attack is carried out.

In terms of *external behaviors*, because of the cross-platform nature of PDF specifications, if some legitimate actions impact the host system in one platform, it is expected that the same actions will be shown when opening the document on the other platform. For example, if a benign document connects to a remote host (e.g., for content downloading or form submission), the same behavior is expected on other platforms. However, if the Internet connection is triggered only upon successful exploitation, it will not be shown on the failing platform.

The architecture of PLATPAL is described in Figure 3. PLATPAL traces both internal and external behaviors, and we argue that tracing both types of behaviors is necessary. Tracing external behaviors is crucial to catch the behavioral discrepancy after a successful exploitation, i.e., the malicious activity step in Figure 1. For example, after a successful JavaScript hidden API attack [22], the attacker might want to execute shellcode, which will fail on Mac because of discrepancies in syscall semantics. The internal behaviors, however, all show the same thing: execution of JavaScript code stops at the same place.

The most compelling reason to have an internal behavior tracer is to defeat platform probing attempts, without which PLATPAL can be easily evaded by launching platform-aware attacks, as described in §3.5. Another reason to trace internal behaviors is to provide some insights on which AAR component is exploited or where the attack occurs, which helps the analysis of maldoc samples, especially for proof-of-concept (PoC) samples that simply crash AAR without any external activities.

4.2 Internal PDF Processing

PLATPAL's internal behavior tracer closely follows how AAR processes PDF documents. PDF processing inside AAR can be divided into two stages.

In the *parsing* stage, the underlying document is opened and the header is scanned to quickly locate the trailer and cross reference table (XRT). Upon locating the XRT, basic elements of the PDF document, called COS objects, are enumerated and parsed. Note that COS objects are only data with a type label (e.g., integer, string, keyword, array, dictionary, or stream). One or more COS objects are then assembled into PDF-specific components such as text, image, font, form, page, JavaScript code, etc. according to AAR's interpretation of PDF specifications. The hierarchical structure (e.g., which texts appear in a particular page) of the PDF document is also constructed along this process. The output, called PD tree, is then passed to the rendering engine for display.

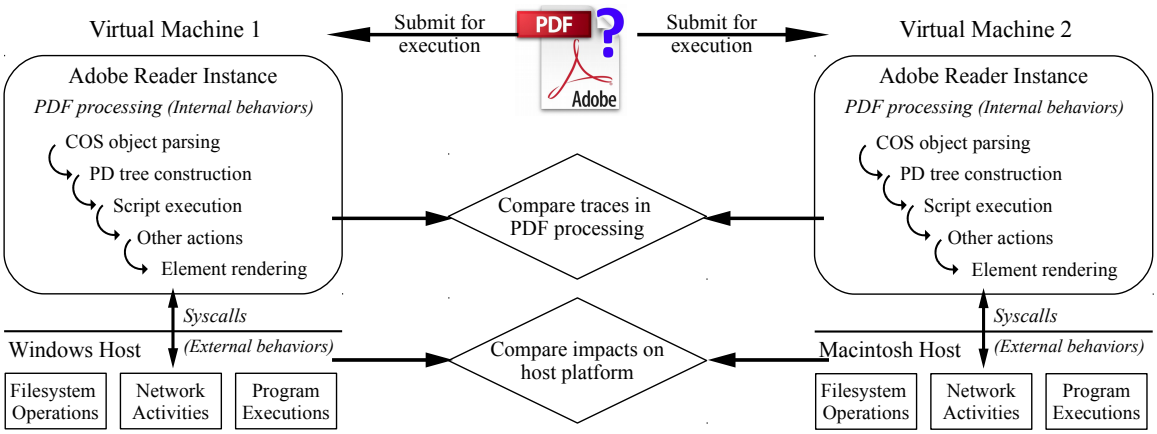

Figure 3: PLATPAL architecture. The suspicious file is submitted to two VMs with different platforms. During execution, both internal and external behaviors are traced and compared. Divergence in any behavior is considered a malicious signal.

The *drawing* stage starts by performing `OpenActions` specified by the document, if any. Almost all maldocs will register anything that could trigger their malicious payload in `OpenActions` for immediate exploitation upon document open. Subsequent drawing activities depend on user's inputs, such as scrolling down to the next page triggers the rendering of that page. Therefore, in this stage, PLATPAL not only hooks the functions but also actively drives the document rendering component by component. Note that displaying content to screen is a platform-dependent procedure and hence, will not be hooked by PLATPAL, but the callbacks (e.g., an object is rendered) are platform-independent and will be traced.

In addition, for AAR, when the rendering engine performs a JavaScript action or draws a JavaScript-embedded form, the whole block of JavaScript code is executed. However, this also enables the platform detection attempts described in §3.5 and an easy escape of PLATPAL's detection. To avoid this, PLATPAL is designed to suppress the automatic block execution of JavaScript code. Instead, the code is tokenized to a series of statements that are executed one by one, and the results from each execution are recorded and subsequently compared. If the statement calls a user-defined function, that function is also executed step-wise.

Following is a summary of recorded traces at each step:
COS object parsing: PLATPAL outputs the parsing results of COS objects (both type and content).
PD tree construction: PLATPAL outputs every PD component with type and hierarchical position in the PD tree.
Script execution: PLATPAL outputs every executed statement and the corresponding result.
Other actions: PLATPAL outputs every callback triggered during the execution of this action, such as change of page views or visited URLs.
Element rendering: PLATPAL outputs every callback triggered during the rendering of the PDF element.

4.3 External System Impact

As syscalls are the main mechanisms for a program to interact with the host platform, PLATPAL hooks syscalls and records both arguments and return values in order to capture the impact of executing a maldoc on the host system. However, for PLATPAL, a unique problem arises when comparing syscalls across platforms, as the syscall semantics on Windows and Mac are drastically different.

To ease the comparison of external behaviors across platforms, PLATPAL abstracts the high-level activities from the raw syscall dumps. In particular, PLATPAL is interested in three categories of activities:
Filesystem operations: including files opened/created during the execution of the document, as well as file deletions, renames, linkings, etc.
Network activities: including domain, IP address, and port of the remote socket.
External executable launches: including execution of any programs after opening the document.

Besides behaviors constructed from syscall trace, PLATPAL additionally monitors whether AAR exits gracefully or crashes during the opening of the document. We (empirically) believe that many typical malware activities such as stealing information, C&C, dropping backdoors, etc, can be captured in these high-level behavior abstractions. This practice also aligns with many automated malware analysis tools like Cuckoo [44] and CWSandbox [63], which also automatically generate a summary that sorts and organizes the behaviors of malware into a few categories. However, unlike these dynamic malware analysis tools that infer maliciousness of the sample based on the sequence or hierarchy of these activities, the only indication of maliciousness for PLATPAL is that the set of captured activities differs across platforms. Another difference is that the summary generated by Cuckoo and CWSandbox usually requires manual interpretation to

judge maliciousness, while the summary from PLATPAL requires no human effort in comparing behaviors across platforms.

5 Implementation

PLATPAL consists of three components: 1) an internal behavior tracer in the form of AAR plugin; 2) an external behavior tracer in the form of syscall tracer; and 3) a sandboxed environment for dynamic document examination based on VMware. We prototype PLATPAL to work on recent Windows (versions 7, 8 and 10) and Mac (versions Yosemite, El Capitan, and Sierra) platforms, and is compatible with all AAR versions from Adobe Reader X 10.0 to the latest version of Adobe Acrobat DC.

5.1 Internal Behavior Tracer

Given that AAR is closed-source software, it is not viable to hook AAR's PDF processing functions through source code instrumentation. Initially, we used dynamic binary instrumentation tools (i.e., Intel Pin [32] and DynamoRio [7]) to hook the execution of AAR and examine function calls at runtime. However, such an approach has two significant drawbacks: 1) These tools introduce a 16-20 times slowdown, which is not tolerable for practical maldoc detection. For example, executing a two-page document could take up to five minutes, and sometimes is even halted by the system; 2) The PDF processing logic is hidden in over 15000 functions (latest version of AAR) with no name or symbol information. It is difficult if not impossible to identify the critical functions as well as to construct the whole cycle.

To this end, PLATPAL chooses to develop an AAR plugin as the internal behavior tracer. The AAR plugin technology [3] is designed to extend AAR with more functionalities such as database interaction, online collaboration, etc. The compiled plugin takes the form of a loadable DLL on Windows and an app bundle on Mac, which is loaded by AAR upon initialization and has significant control over AAR at runtime. The AAR plugin provides a few nice features that suit PLATPAL's needs: 1) Its cross-platform support abstracts the platform-specific operations to a higher level; 2) It uses the internal logic of AAR in PDF processing and summarizes the logic into 782 functions and callbacks (nicely categorized and labeled), which enables PLATPAL to both passively monitor the execution of these functions and actively drive the document, including executing JavaScript code and rendering elements; 3) It is stable across AAR versions (only two functions are added since version 10, which are not used by PLATPAL); 4) Since the AAR plugin is in the form of a loadable module, it shortens the total document analysis time to an average of 24 seconds.

In recording behaviors discussed in §4.2, the COS objects and PD hierarchical information are extracted using the enumeration methods provided by the `CosDoc`, `PDDoc`, and `PDF_Consultant` classes. JavaScript code is first tokenized by a lexer adapted from `SpiderMonkey` and executed statement-by-statement with `AFExecuteThisScript` method from `AcroForm` class. The rest of the PDF-supported actions are launched with the `AVDocPerformAction` method. The PDF processing functions exposed to the AAR plugin can be hooked by the simple JMP-Trampoline hot-patching technique as summarized in [6].

5.2 External Behavior Tracer

As illustrated in §4.3, PLATPAL's external behavior tracer records syscall arguments and return values during document execution. On Windows, the tracer is implemented based on `NtTrace` [41]; on Mac, the tracer is a `Dscript` utilizing the `DTrace` [9] mechanism available on BSD systems. Both techniques are mature on respective platforms and incur small execution overhead: 15% to 35% compared to launching AAR without the tracer attached, which helps to control the total execution time per document. Constructing the high-level behaviors is performed in a similar manner as Cuckoo guest agent [44].

In PLATPAL, syscall tracing starts only after the document is opened by AAR. The AAR initialization process is not traced (as AAR itself is not a malware) and PLATPAL is free of the messy filesystem activities (e.g., loading libraries, initializing directory structures, etc) during the start-up, leaving the execution trace in a very short and clean state. In fact, a benign document typically has around 20 entries of filesystem traces and no network activities or external program launches. AAR uses a single thread for loading and parsing the document and spawns one helper thread during document rendering. Syscalls of both threads are traced and compared.

To compare file paths, PLATPAL further aggregates and labels filesystem operation traces into a few categories that have mappings on both platforms, including AAR program logic, AAR support file, AAR working caches, system library/framework dependencies, system fonts, and temporary files. Files outside these labels will go to the unknown category and will be compared based on filenames.

5.3 Automated Execution Sandbox

For PLATPAL, the purpose of having an automated execution sandbox is twofold: 1) to confine the malicious activities within a proper boundary and 2) to provide a clean execution environment for each document examination that is free from side effects by prior executions.

The virtual machine (VM) is initialized with a clean-slate operating system and subsequently provisioned with the necessary tools and settings, including AAR, the plugin, and the syscall tracer. The memory and disk snapshot is taken after the provision, and each subsequent document execution restores the states from this snapshot. PLATPAL uses VMware for the management of VMs and snapshots.

Workflow. PLATPAL can be started like `PLATPAL <file-to-check>`. After that, PLATPAL populates a Windows VM and a Mac VM and restores the memory and disk snapshots. The suspicious document is then uploaded to these VMs and AAR is started with the syscall tracer attached. After AAR is done with initialization, the control is transferred to the plugin (internal tracer), which opens the document for examination. After the examination finishes (or AAR crashes), logs from internal and external tracing are pulled from the respective VMs and compared on the host. PLATPAL reports whether discrepancies are detected among these logs.

6 Evaluation

In this section, we validate the fundamental assumption of PLATPAL: benign documents behave the same when opened across different platforms, while maldocs behave differently when doing exploitation on different platforms. We also evaluate PLATPAL's performance in terms of total time taken to finish a complete analysis.

Experiment setup. The experiments were conducted on a MacBook Pro (2016 model) with Intel Core i7 2.9GHz CPU and 16GB RAM running macOS Sierra. One VM is provisioned with Windows 7 Professional SP1 and the other VM is provisioned with OSX Yosemite 10.10.1. Each VM is further provisioned with 6 different versions of AAR instances[1] listed in Table 2. Each document sample is forced to be closed after one minute execution.

6.1 Benign Samples

The benign sample set consists of three parts: 1000 samples are collected by searching Google with file type PDF and no keywords. However, a majority of these samples do not use features that are typically exploited by maldocs. For example, only 28 files contain embedded fonts and 6 files contain JavaScript code. Therefore, we further collected 30 samples from PDF learning sites[2] that use advanced features in the PDF standard, including embedded JavaScript (26 samples), AcroForm (17), self-defined font (6), and 3D objects (2). All of the samples are submitted to VirusTotal and scanned by 48 AV products and none of them are flagged as malicious by any of the AV engine.

The samples are submitted to PLATPAL for analysis. In particular, each document is opened by all six versions of AAR instances on both platforms. This is to empirically verify that all AAR reader instances do not introduce non-determinism during the document executions. Pairwise behavior comparison is conducted per AAR version and no discrepancy is observed, for any AAR version tested. More importantly, the experiment results support the first part of PLATPAL's assumption: benign documents behave the same across platforms.

6.2 Maldoc Detection

The maldoc samples are all collected from VirusTotal. In particular, we collected samples with identified CVE numbers (i.e., the sample exploits a particular CVE) [3] as of Dec. 31, 2016. As a prototype, we restrict the scope by analyzing CVEs published after 2013 and further filter the samples that are obviously mislabeled (e.g., a 2011 sample exploiting a 2016 CVE) or of wrong types (e.g., a zip file or an image file) and obtained a 320-sample dataset.

The samples are submitted to PLATPAL for analysis. In addition, we select the AAR versions that are most popular based on the time when the CVE was published. In other words, each exploit is a zero-day attack to the AAR version tested. The per-CVE detection results are presented in Table 2 and the breakdown in terms of which behavior factor causes the discrepancy is listed in Table 3.

Interpretation. For any sample submitted to PLATPAL, only three outcomes are possible:

1) *Malicious*: At least one behavioral discrepancy is observed, including the case in which AAR crashes on both platforms but the internal behavior is different, i.e., they crash at different PDF processing stages.

2) *Suspicious*: AAR crashes on both platforms but no difference is observed in internal behaviors. Given that a benign document has no reason to crash AAR, PLATPAL considers these samples as suspicious.

3) *Benign*: No behavioral discrepancy can be observed and AAR exits gracefully on both platforms.

Overall result. Out of 320 samples, PLATPAL detected 209 (65.3%) *malicious* samples, 34 (10.6%) *suspicious* samples, and 77 (24.1%) *benign* samples.

Suspicious samples. Among the 34 suspicious samples, we are able to confirm that 16 are PoC samples, including 7 released on Exploit-DB [19], 3 in public blogs, and 6 inferred by their original filenames recorded by VirusTotal. These samples are likely obtained by fuzzing and upon

[1] Previous versions of AAR can be obtained from ftp://ftp.adobe.com/pub/adobe/reader

[2] The samples are mainly obtained from http://www.pdfscripting.com and http://www.planetpdf.com/

[3] VirusTotal labels a sample with CVE number as long as one of the hosted AV products flag the sample with the CVE label.

CVE	AAR Version	Num. Samples	Result	
			Both crash	Divergence
2016-6946	DC.16.45	51	8	40
2016-4204	DC.16.45	78	7	37
2016-4119	DC.10.60	1	0	1
2016-1091	DC.10.60	63	6	31
2016-1077	DC.10.60	1	0	1
2016-1046	DC.10.60	4	0	4
2015-5097	11.0.10	4	0	4
2015-2426	11.0.10	14	6	8
2015-0090	11.0.10	1	0	1
2014-0521	11.0.00	2	0	2
2014-0495	11.0.00	2	0	2
2013-3353	10.1.4	16	4	10
2013-3346	10.1.4	7	0	7
2013-2729	10.1.4	23	3	19
2013-0640	10.1.0	30	0	22
2013-0641	10.1.0	23	0	20
Total		320	34	209

Table 2: PLATPAL maldoc detection results grouped by CVE number. *Both crash* means AAR crashes on both platforms while executing the maldoc sample with no divergence on internal behaviors; *Divergence* means at least one behavioral discrepancy (either internal or external) is observed.

execution, will simply crash AAR. We expect it to apply to the rest of the suspicious samples as well.

Benign samples. We identified several reasons for the failed detection of these samples.

1) The maldoc targets old and specific AAR versions. Although a majority of maldoc samples exploit a wide range of AAR versions, we do find samples that target old AAR versions only, i.e., 9.X and 8.X, including 8 CVE-2013-0640 samples, 3 CVE-2013-0641 samples, and 1 CVE-2013-2729 sample. We also found that 13 CVE-2016-4204 samples and 10 CVE-2016-1091 samples seems to be exploiting AAR version 11.0.X and the exploits do not work on the AAR DC version used in the experiment. This is based on manual inspection of the JavaScript dump from these samples.

In total, they account for 36 out of the 77 samples classified as benign. This is also shows the drawback of PLATPAL, being a dynamic analysis approach, it requires proper setup of the execution environment to entice the malicious behaviors.

2) The maldoc sample could be mis-classified by AV vendor on VirusTotal. This could be true for 11 CVE-2016-4204 and 8 CVE-2016-1091 samples, as out of the 48 AV products hosted on VirusTotal, only one AV vendor flags them as malicious. In total, this accounts for 19 out of the 77 samples classified as benign.

3) The maldoc does not perform malicious activity. Not all malicious activities in the maldoc can be triggered. In particular, we observed two CVE-2013-3353 samples attempted to connect to a C&C server in JavaScript but did nothing afterwards because of the lack of responses, which results in no divergences in execution trace.

In the end, for the rest of the samples classified as benign (20 in total), we are unable to confirm a reason why no behavioral discrepancies are observed. It could be because of any of the aforementioned reasons (but we are unable to confirm) and we do not preclude the possibility that some samples could evade PLATPAL's detection. Given the scope and flexibility of PDF specification, it is possible that PLATPAL needs to hook more functions (e.g., per glyph to host encoding transformation performed a font) to capture finer-grained internal behaviors.

Behavior effectiveness. Table 3 also shows the effectiveness of various behaviors in detecting maldocs.

1) By the first row, it is possible to have only external behavior divergences, while internal behaviors are the same (e.g., due to pure JavaScript attacks). By the first column, it is also possible to have only internal behavior divergences, while external behaviors are the same (due to the powerful error-correction capability of AAR).

2) Crash/no crash is the most effective external indicator, as memory-error exploitation is the dominating technique for maldoc attacks among the samples. JavaScript execution is the most effective internal indicator, as almost all attacks involve JavaScript; even memory error exploits use it to prepare the heap.

Pinpointing attacks by internal tracer. One supplementary goal of the internal tracer is to provide insights on which AAR component is exploited or where the attack occurs given a maldoc sample. To evaluate how this goal is achieved, we performed a cross-check on where the internal behavior divergence occurs and the targeted AAR component of each CVE[4]. The result is shown in Table 4.

In four out of 7 cases, PLATPAL's internal tracer finds divergence during the invocation of the vulnerable components. In the CVE-2015-2426 case, since the vulnerable component is a font library, the divergence is first detected during the rendering process. In the CVE-2013-3346 case, the vulnerable component (ToolButton callback) is triggered through JavaScript code and hence, the first divergence occurs in the script engine. In the CVE-2013-2729 case, although the bug is in the parser component, the divergence is detected when the maldoc is playing heap feng-shui to arrange heap objects.

Resilience against automated maldoc generation. We test PLATPAL's resilience against state-of-the-art maldoc generation tool, EvadeML [65], which automatically produce evasive maldoc variants against ML-depended approaches in Table 1 given a malicious seed file. To do this, we selected 30 samples out of the 209 malicious samples which are also detected as malicious by PDFrate [46],

[4]Only CVEs which full details are publicly disclosed are considered

Internal Behavior	External Behavior						Total
	No difference	Both crash	One crash	Filesystem	Network	Executable	
No difference	77	34	0	6	3	0	120
COS object parsing	4	8	23	0	0	0	35
PD tree construction	0	0	2	4	2	0	8
JavaScript execution	5	5	47	18	12	4	91
Other actions	0	0	0	2	0	2	4
Element rendering	3	10	35	9	5	0	62
Total	89	57	107	39	22	6	320

Table 3: PLATPAL maldoc detection results grouped by the factor causing divergences. Note that for each sample, only one internal and one external factor is counted as the cause of divergence. E.g., if a sample crashes on Mac and does not crash on Windows, even their filesystem activities are different, it is counted in the crash/no crash category. The same rule applies to internal behaviors.

CVE	Targeted component	Divergence first occurs	Detects
2016-4119	Parser	Parser	Vuln. component
2016-1077	Parser	Parser	Vuln. component
2016-1046	Script engine	Script engine	Vuln. component
2015-2426	Library	Render	Exploit carrier
2014-0521	Script engine	Script engine	Vuln. component
2013-3346	Render	Script engine	Exploit carrier
2013-2729	Parser	Script engine	Exploit carrier

Table 4: Divergence detected by PLATPAL's internal tracer vs the actual buggy AAR component.

Item	Windows		Mac	
	Ave.	Std.	Ave.	Std.
Snapshot restore	9.7	1.1	12.6	1.1
Document parsing	0.5	0.2	0.6	0.2
Script execution	10.5	13.0	5.1	3.3
Element rendering	7.3	8.9	6.2	6.0
Total	23.7	8.5	22.1	6.3

Table 5: Breakdown of PLATPAL's analysis time per document (unit: seconds).

the default PDF classifier that works with EvadeML[5]. We then uses EvadeML to mutate these samples until all variants are considered benign. Finally, we send these evasive variants to PLATPAL for analysis and all of them are again marked as malicious, i.e., behavioral discrepancies are still observed. This experiment empirically verifies PLATPAL's resilience on automated maldoc generation tools. The main reason for the resilience is that EvadeML mainly focuses on altering the structural feature of the maldoc while preserves its exploitation logic and also the internal and external behaviors when launching the attack.

6.3 Performance

In PLATPAL, the total analysis time consists of two parts: 1) time to restore disk and memory snapshots and 2) time to execute the document sample. The latter can be further broken down into document parsing, script execution, and element rendering time. Table 5 shows the time per item and the overall execution time.

On average, document execution on both VMs can finish at approximately the same time (23.7 vs 22.1 seconds). Given that the VMs can run in parallel, a complete analysis can finish within 25 seconds. A notable difference is that script execution on the Windows platform takes significantly longer than on the Mac platform. This is because almost all maldoc samples target Windows platforms and use JavaScript to launch the attack. The attack quickly fails on Mac (e.g., wrong address for ROP gadgets) and crashes AAR but succeeds on Windows and therefore takes longer to finish. The same reason also explains why the standard deviation on script execution time is larger on the Windows platform.

[5]It is worthnoting that PLATPAL cannot be used as the PDF classifier for EvadeML as EvadeML requires a maliciousness score which has to be continuous between 0 and 1 while PLATPAL can only produce discrete scores of either 0 or 1. Therefore, we use PDFrate, the PDF classifier used in the EvadeML paper [65], for this experiment.

7 Discussion

7.1 Limitations

User-interaction driven attacks. Although PLATPAL is capable of simulating simple users' interactions (e.g., scrolling, button clicking, etc), PLATPAL does not attempt to explore all potential actions (e.g., key press, form filling, etc) or explore all branches of the JavaScript code. Similarly, PLATPAL cannot detect attacks that intentionally delay their execution (e.g., start exploitation two minutes after document open). These are common limitations for any dynamic analysis tool. However, we believe this is not a serious problem for maldoc detection, as hiding malicious activities after complex user interactions limits its effectiveness in compromising the victim's system.

Social engineering attacks. PLATPAL is not capable of detecting maldocs that aim to perform social engineering

attacks, such as faking password prompt with a JavaScript window or enticing the user to download a file and execute it. This is because these maldocs neither exploit bugs in AAR nor inject malicious payload, (in fact they are legit documents structural-wise) and hence will have exactly the same behaviors on both platforms.

Targeted AAR version. If a maldoc targets a specific version of AAR, its behaviors in PLATPAL will likely be either crashing both AAR instances (i.e., exploited the bug but used the wrong payload), or the document is rendered and closed gracefully because of error correction by AAR. In the latter case, PLATPAL will not be able to detect a behavioral discrepancy. This is usually not a problem for PLATPAL in practice, as PLATPAL will mainly be used to detect maldocs against the latest version of AAR. However, PLATPAL can also have a document tested on many AAR versions and flag it as suspicious as long as a discrepancy is observed in any single version.

Non-determinism. Another potential problem for PLATPAL is that non-deterministic factors in document execution could cause false alerts. Examples include return value of `gettime` functions or random number generators available through JavaScript code. Although PLATPAL does not suffer from such a problem during the experiment, a complete solution would require a thorough examination of the PDF JavaScript specification and identify all non-determinism. These non-deterministic factors need to be recorded during the execution of a document on one platform and replayed on the other platform.

7.2 Deployment

As PLATPAL requires at least two VMs, a large amount of image and memory needs to be committed to support the operation of PLATPAL. Our current implementation uses 60GB disk space to host the snapshots for six versions of AAR and 2GB memory per each running VM.

To this end, we believe that PLATPAL is best suited for cloud storage providers (e.g., Dropbox, Google Docs, Facebook, etc.) which can use PLATPAL to periodically scan for maldocs among existing files or new uploads. These providers can afford the disk and memory required to set up VMs with diverse platforms as well as enjoy economy of scale. Similarly, PLATPAL also fits the model of online malware scanning services like VirusTotal or the cloud versions of anti-virus products.

In addition, as a complementary scheme, PLATPAL can be easily integrated with previous works (Table 1) to improve their detection accuracy. In particular, PLATPAL's internal behavior tracer can be used to replace parsers in these techniques to mitigate the parser-confusion attack [11]. COS object and PD tree information can be fed to metadata-based techniques [33, 36, 46, 52], while the JavaScript code dump can be fed to JavaScript-oriented techniques [14, 27, 31, 45, 48, 58, 59] for analysis.

7.3 Future Works

We believe that PLATPAL is a flexible framework that is suitable not only for PDF-based maldoc detection but also for systematically approaching security-through-diversity.

Support more document types. MS Office programs share many features with AAR products, such as 1) supporting both Windows and Mac platforms; 2) supporting a plugin architecture which allows efficient hooking of document processing functions and action driving; 3) executing documents based on a standard specification that consists of static components (e.g., text) and programmable components (e.g., macros). Therefore, we do not see fundamental difficulties in porting PLATPAL to support maldoc detection that targets MS Office suites.

As another example, given that websites can also be viewed as HTML documents with embedded JavaScript, malicious website detection also fits into PLATPAL's framework. Furthermore, given that Chrome and Firefox browsers and their scripting engines are open-sourced, PLATPAL is capable of performing finer-grained behavior tracing and comparison with source code instrumentation.

Explore architecture diversity. Apart from platform diversity, CPU architecture diversity can also be harvested for maldoc detection, which we expect to have a similar effect in stopping maldoc attacks. To verify this, we plan to extend PLATPAL to support the Android version of AAR, which has both ARM and x86 variants.

8 Additional Related Work

In addition to the maldoc detection work, being an N-version system, PLATPAL is also related to the N-version research. The concept of the N-version system was initially introduced as a software fault-tolerance technique [12] and was later applied to enhance system and software security. For example, Frost [60] instruments a program with complementary scheduling algorithms to survive concurrency errors; Crane *et al*. [16] applies dynamic control-flow diversity and noise injection to thwart cache side-channel attacks; Tightlip [68] and Capizzi *et al*. [10] randomize sensitive data in program variants to mitigate privacy leaks; Mx [24] uses multiple versions of the same program to survive update bugs; Cocktail [66] uses multiple web browser implementations to survive vendor-specific attacks; and Nvariant [15], Replicae [8], and GHUMVEE [61] run program variants in disjoint memory layouts to mitigate code reuse attacks. Similarly, Orchestra [43] synchronizes two program variants which grow the stack in opposite directions for intrusion detec-

tion. In particular, Smutz *et al*. [47] attempts to identify and prevent detection evasions by constructing diversified classifiers, ensembling them into a single system, and comparing their classification outputs with mutual agreement analysis.

Although PLATPAL is designed for a completely different goal (i.e., maldoc detection), it shares the insights with N-version systems: an attacker is forced to simultaneously compromise all variants with the same input in order to take down or mislead the whole system.

Another line of related work is introducing diversity to the execution environment in order to entice and detect malicious behaviors. For example, HoneyClient [56], caches and resembles potentially malicious objects from the network stream (e.g., PDF files) and then send it to multiple emulated environments for analysis. Balzarotti *et al*. [4] detects "split personality" in malware, i.e., malware that shows diverging behaviors in emulated environment and bare-metal machines, by comparing the runtime behaviors across runs. Rozzle [26] uses symbolic execution to emulate different environment values malware typically checks and hence, entice environment-specific behaviors from the malware. to show diverging behaviors.

PLATPAL shares the same belief as these works: diversified execution environment leads to diversified behaviors, and focuses on harvesting platform diversity for maldoc detection.

9 Conclusion

Due to the continued exploitation of AAR, maldoc detection has become a pressing problem. A survey of existing techniques reveals that they are vulnerable to recent attacks such as parser-confusion and ML-evasion attacks. In response to this, we propose a new perspective: platform diversity, and prototype PLATPAL for maldoc detection. PLATPAL hooks into AAR to trace internal PDF processing and also uses full dynamic analysis to capture a maldoc's external impact on the host system. Both internal and external traces are compared, and the only heuristic to detect maldoc is based on the observation that a benign document behaves the same across platforms, while a maldoc behaves differently during exploitation, because of the diversified implementations of syscalls, memory management, etc. across platforms. Such a heuristic does not require known maldoc samples to derive patterns that differentiate maldocs from benign documents, which also enables PLATPAL to detect zero-day attacks without prior knowledge of the attack. Evaluations show that PLATPAL raises no false alarms in benign samples, detects a variety of behavioral discrepancies in malicious samples, and is a scalable and practical solution.

10 Acknowledgment

We thank our shepherd, Alexandros Kapravelos, and the anonymous reviewers for their helpful feedback. This research was supported by NSF under award DGE-1500084, CNS-1563848, CRI-1629851, CNS-1017265, CNS-0831300, and CNS-1149051, ONR under grant N000140911042 and N000141512162, DHS under contract No. N66001-12-C-0133, United States Air Force under contract No. FA8650-10-C-7025, DARPA under contract No. DARPA FA8650-15-C-7556, and DARPA HR0011-16-C-0059, and ETRI under grant MSIP/IITP[B0101-15-0644].

References

[1] Adobe Systems Inc. Document Management - Portable document format, 2008. http://wwwimages.adobe.com/content/dam/Adobe/en/devnet/pdf/pdfs/PDF32000_2008.pdf.

[2] Adobe Systems Inc. Introducing Adobe Reader Protected Mode, 2010. http://blogs.adobe.com/security/2010/07/introducing-adobe-reader-protected-mode.html.

[3] Adobe Systems Inc. Plug-ins and Applications, 2015. http://help.adobe.com/en_US/acrobat/acrobat_dc_sdk/2015/HTMLHelp/#t=Acro12_MasterBook/Plugins_Introduction/About_plug-ins.htm.

[4] Davide Balzarotti, Marco Cova, Christoph Karlberger, Christopher Kruegel, Engin Kirda, and Giovann Vigna. Efficient Detection of Split Personalities in Malware. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February–March 2010.

[5] Tyler Bohan. In the Zone: OS X Heap Exploitation. In *Proceedings of the 2016 Summercon*, New Yark, NY, July 2016.

[6] Jurriaan Bremer. x86 API Hooking Demystified, 2012. https://jbremer.org/x86-api-hooking-demystified/.

[7] Derek Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, MIT, 2004.

[8] Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi. Diversified Process Replica for Defeating Memory Error Exploits. In *Proceedings of the 2007 International Performance, Computing, and Communications Conference (IPCCC)*, New Orleans, LA, April 2007.

[9] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic Instrumentation of Production Systems. In *Proceedings of the 2004 USENIX Annual Technical Conference (ATC)*, Boston, MA, June–July 2004.

[10] Roberto Capizzi, Antonio Longo, V. N. Venkatakrishnan, and A. Prasad Sistla. Preventing Information Leaks Through Shadow Executions. In *Proceedings of the 2008 International Conference on Software Engineering (ICSE)*, Anaheim, CA, December 2008.

[11] Curtis Carmony, Mu Zhang, Xunchao Hu, Abhishek Vasisht Bhaskar, and Heng Yin. Extract Me If You Can: Abusing PDF Parsers in Malware Detectors. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2016.

[12] Liming Chen and Algirdas Avizienis. N-Version Programming: A Fault-Tolerance Approach To Reliability of Software Operation. In *Fault-Tolerant Computing, 1995*, Jun. 1995.

[13] Corelan Team. Exploit writing tutorial part 11 : Heap Spraying Demystified, 2011. https://www.corelan.be/index.php/

2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified.

[14] Igino Corona, Davide Maiorca, Davide Ariu, and Giorgio Giacinto. Lux0R: Detection of Malicious PDF-embedded JavaScript Code through Discriminant Analysis of API References. In *Proceedings of the Artificial Intelligent and Security Workshop (AISec)*, 2014.

[15] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-Variant Systems: A Secretless Framework for Security through Diversity. In *Proceedings of the 15th USENIX Security Symposium (Security)*, Vancouver, Canada, July 2006.

[16] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[17] daehee87. DEFCON 2014 Polyglot Writeup, 2014. http://daehee87.tistory.com/393.

[18] ECMA International. ECMAScript Language Specification, 2016. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf.

[19] Exploit Database. Offensive SecurityâĂŹs Exploit Database Archive, 2016. https://www.exploit-db.com.

[20] Joseph Gardiner and Shishir Nagaraja. On the Security of Machine Learning in Malware C&C Detection: A Survey. *ACM Computing Survey (CSUR)*, 49(3), September 2016.

[21] Dan Goodin. Pwn2Own Carnage Continues as Exploits Take Down Adobe Reader, Flash, 2013. https://arstechnica.com/security/2013/03/pwn2own-carnage-continues-as-exploits-take-down-adobe-reader-flash.

[22] Brian Gorenc, AbdulAziz Hariri, and Jasiel Spelman. Abusing Adobe Reader's JavaScript APIs. In *Proceedings of the 23rd DEF CON*, Las Vegas, NV, August 2015.

[23] Marco Grassi. [CVE-2016-4673] Apple CoreGraphics macOS/iOS JPEG memory corruption, 2016. https://marcograss.github.io/security/apple/cve/macos/ios/2016/11/21/cve-2016-4673-apple-coregraphics.html.

[24] Petr Hosek and Cristian Cadar. Safe Software Updates via Multi-version Execution. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, San Francisco, CA, May 2013.

[25] Kaspersky. Kaspersky Security Bulletin, 2015. https://securelist.com/files/2014/12/Kaspersky-Security-Bulletin-2014-EN.pdf.

[26] Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Rozzle: De-Cloaking Internet Malware. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[27] Pavel Laskov and Nedim Srndic. Static Detection of Malicious JavaScript-Bearing PDF Documents. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2011.

[28] Moony Li. Hacking Team Leak Uncovers Another Windows Zero-Day, Fixed In Out-Of-Band Patch, 2015. http://blog.trendmicro.com/trendlabs-security-intelligence/hacking-team-leak-uncovers-another-windows-zero-day-ms-releases-patch.

[29] Daiping Liu, Haining Wang, and Angelos Stavrou. Detecting Malicious Javascript in PDF through Document Instrumentation. In *Proceedings of the 44th International Conference on Dependable Systems and Networks (DSN)*, Atlanta, GA, 2014.

[30] Kai Lu and Kushal Arvind Shah. Analysis of Use-After-Free Vulnerability (CVE-2016-4119) in Adobe Acrobat and Reader, 2016. https://blog.fortinet.com/2016/06/06/analysis-of-use-after-free-vulnerability-cve-2016-4119-in-adobe-acrobat-and-reader.

[31] Xun Lu, Jianwei Zhuge, Ruoyu Wang, Yinzhi Cao, and Yan Chen. De-obfuscation and Detection of Malicious PDF Files with High Accuracy. In *Proceedings of the 46th Hawaii International Conference on System Sciences (HICSS)*, 2013.

[32] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analyis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL, June 2005.

[33] Davide Maiorca, Davide Ariu, Igino Corona, and Giorgio Giacinto. A Structural and Content-based Approach for a Precise and Robust Detection of Malicious PDF Files. In *Proceedings of the International Conference on Information Systems Security and Privacy (ICISSP)*, 2015.

[34] Davide Maiorca, Davide Ariu, Igino Corona, and Giorgio Giacinto. An Evasion Resilient Approach to the Detection of Malicious PDF Files. In *Proceedings of the International Conference on Information Systems Security and Privacy (ICISSP)*, 2016.

[35] Davide Maiorca, Igino Corona, and Giorgio Giacinto. Looking at the Bag is not Enough to Find the Bomb: An Evasion of Structural Methods for Malicious PDF Files Detection. In *Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Hangzhou, China, March 2013.

[36] Davide Maiorca, Giorgio Giacinto, and Igino Corona. A Pattern Recognition System for Malicious PDF Files Detection. In *Proceedings of the 8th International Conference on Machine Learning and Data Mining in Pattern Recognition (MLDM)*, 2012.

[37] Felipe Andres Manzano. Adobe Reader X BMP/RLE heap corruption, 2012. http://www.binamuse.com/papers/XFABMPReport.pdf.

[38] Net MarketShare. Desktop Operating System Market Share, 2017. https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=10&qpcustomd=0.

[39] Nexor. Preventing Document-based Malware from Devastating Your Business, 2013. https://www.nexor.com/wp-content/uploads/2016/02/Preventing-Document-Based-Malware-from-Devastating-Your-Business.pdf.

[40] Nir Nissim, Aviad Cohen, Chanan Glezer, and Yuval Elovici. Detection of Malicious PDF Files and Directions for Enhancements: A State-of-the-art Survey. *Computers & Security*, October 2014.

[41] Roger Orr. NtTrace - Native API tracing for Windows, 2016. http://rogerorr.github.io/NtTrace/.

[42] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. The Limitations of Deep Learning in Adversarial Settings. In *Proceedings of the 9th European Workshop on System Security (EUROSEC)*, 2016.

[43] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. Orchestra: Intrusion Detection Using Parallel Execution and Monitoring of Program Variants in User-space. In *Proceedings of the 4th European Conference on Computer Systems (EuroSys)*, Nuremberg, Germany, March 2009.

[44] Mark Schloesser, Jurriaan Bremer, and Alessandro Tanasi. Cuckoo Sandbox - Open Source Automated Malware Analysis. In *Black Hat USA Briefings (Black Hat USA)*, Las Vegas, NV, August 2013.

[45] Florian Schmitt, Jan Gassen, and Elmar Gerhards-Padilla. PDF Scrutinizer: Detecting JavaScript-based Attacks in PDF Documents. In *Proceedings of the 10th Annual International Conference on Privacy, Security and Trust (PST)*, 2012.

[46] Charles Smutz and Angelos Stavrou. Malicious PDF Detection

using Metadata and Structural Features. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2012.

[47] Charles Smutz and Angelos Stavrou. When a Tree Falls: Using Diversity in Ensemble Classifiers to Identify Evasion in Malware Detectors. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2016.

[48] Kevin Z. Snow, Srinivas Krishnan, Fabian Monrose, and Niels Provos. ShellOS: Enabling Fast Detection and Forensic Analysis of Code Injection Attacks. In *Proceedings of the 20th USENIX Security Symposium (Security)*, San Francisco, CA, August 2011.

[49] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2013.

[50] Sophps. The Rise of Document-based Malware, 2016. https://www.sophos.com/en-us/security-news-trends/security-trends/the-rise-of-document-based-malware.aspx.

[51] Alexander Sotirov. Heap Feng Shui in JavaScript. In *Proceedings of the 2007 Black Hat Europe Briefings (Black Hat Europe)*, Amsterdam, Netherlands, 2007.

[52] Nedim Srndic and Pavel Laskov. Detection of Malicious PDF Files Based on Hierarchical Document Structure. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2013.

[53] Nedim Srndic and Pavel Laskov. Practical Evasion of a Learning-Based Classifier: A Case Study. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2014.

[54] Symantec. Portable Document Format Malware, 2010. https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/the_rise_of_pdf_malware.pdf.

[55] Symantec. Internet Security Threat Reports, 2014. http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v18_2012_21291018.en-us.pdf.

[56] Teryl Taylor, Kevin Z. Snow, Nathan Otterness, and Fabian Monrose. Cache, Trigger, Impersonate: Enabling Context-Sensitive Honeyclient Analysis On-the-Wire. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2016.

[57] Trend Micro. Macro Malware: Here's What You Need to Know in 2016, 2016. http://blog.trendmicro.com/macro-malware-heres-what-you-need-to-know-in-2016/.

[58] Zacharias Tzermias, Giorgos Sykiotakis, Michalis Polychronakis, and Evangelos P. Markatos. Combining Static and Dynamic Analysis for the Detection of Malicious Documents. In *Proceedings of the 4th European Workshop on System Security (EUROSEC)*, 2011.

[59] Cristina Vatamanu, Dragoş GavriluŢ, and Răzvan Benchea. A Practical Approach on Clustering Malicious PDF Documents. *Journal in Computer Virology*, June 2012.

[60] Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Detecting and Surviving Data Races using Complementary Schedules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, October 2011.

[61] Stijn Volckaert, Bart Coppens, and Bjorn De Sutte. Cloning Your Gadgets: Complete ROP Attack Immunity with Multi-Variant Execution. *IEEE Transactions on Dependable and Secure Computing*, 13(4):437–450, July 2016.

[62] Carsten Willems, Felix C. Freiling, and Thorsten Holz. Using Memory Management to Detect and Extract Illegitimate Code for Malware Analysis. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2012.

[63] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox. In *Proceedings of the 28th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2007.

[64] Shane Wilton. One Shellcode to Rule Them All: Cross-Platform Exploitation, 2014. http://www.slideshare.net/ShaneWilton/one-shellcode-to-rule-them-all.

[65] Weilin Xu, Yanjun Qi, and David Evans. Automatically Evading Classifiers: A Case Study on PDF Malware Classifiers. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2016.

[66] Hui Xue, Nathan Dautenhahn, and Samuel T. King. Using Replicated Execution for a More Secure and Reliable Web Browser. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2012.

[67] Mark Vincent Yason. Windows 10 Segment Heap Internals. In *Black Hat USA Briefings (Black Hat USA)*, Las Vegas, NV, August 2016.

[68] Aydan Yumerefendi, Benjamin Mickle, and Landon P. Cox. Tightlip: Keeping applications from spilling the beans. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Cambridge, MA, April 2007.