Detecting Malicious Javascript in PDF through Document Instrumentation

Conference Paper · June 2014

DOI: 10.1109/DSN.2014.92

CITATIONS
28

READS
800

3 authors, including:

Haining Wang
University of Delaware
183 PUBLICATIONS **5,732** CITATIONS

SEE PROFILE

Detecting Malicious Javascript in PDF through Document Instrumentation

Daiping Liu
Department of Computer Science
College of William and Mary
dliu01@email.wm.edu

Haining Wang
Department of Computer Science
College of William and Mary
hnw@cs.wm.edu

Angelos Stavrou
Center for Secure Information Systems
George Mason University
astavrou@gmu.edu

***Abstract*— An emerging threat vector, embedded malware inside popular document formats, has become rampant since 2008. Owed to its wide-spread use and Javascript support, PDF has been the primary vehicle for delivering embedded exploits. Unfortunately, existing defenses are limited in effectiveness, vulnerable to evasion, or computationally expensive to be employed as an on-line protection system. In this paper, we propose a context-aware approach for detection and confinement of malicious Javascript in PDF. Our approach statically extracts a set of static features and inserts context monitoring code into a document. When an instrumented document is opened, the context monitoring code inside will cooperate with our runtime monitor to detect potential infection attempts in the context of Javascript execution. Thus, our detector can identify malicious documents by using both static and runtime features. To validate the effectiveness of our approach in a real-world setting, we first conduct a security analysis, showing that our system is able to remain effective in detection and be robust against evasion attempts even in the presence of sophisticated adversaries. We implement a prototype of the proposed system, and perform extensive experiments using 18623 benign PDF samples and 7370 malicious samples. Our evaluation results demonstrate that our approach can accurately detect and confine malicious Javascript in PDF with minor performance overhead.**

***Keywords*-Malcode bearing PDF; malicious Javascript; malware detection and confinement; document instrumentation.**

I. Introduction

Malware authors are constantly seeking for new ways to compromise computer systems. Recently, they have embarked to take advantage of popular forms of data exchange, focusing their attention on malcode-bearing PDF documents [1]. The PDF standard has several unique advantages when used as an attack vector: (1) it has replaced Microsoft Word as the most dominant document format; (2) it has been widely considered to be safe; (3) it is easy to craft a malicious PDF; and more importantly, (4) it supports Javascript. All of these features have made PDF one of the most attractive exploitation vehicles. This is clearly supported by the fact that the number of discovered PDF vulnerabilities has quadrupled in the last five years [2] with many attack cases having been reported [1] [3]. The most striking observation comes from Microsoft malware protection center, showing that the exploitation of old PDF vulnerabilities is on the rise [1].

Despite the increasing number of successful PDF infections and their impact on end users, thus far, only a few methods for detection of malicious PDF have been proposed as response to this emerging threat. Unfortunately, it appears that traditional signature and behavior based detection methods, which are favored by the majority of modern anti-virus software, cannot handle malicious PDF well. Recently, researchers exploit the structural differences between benign and malicious documents to detect malicious PDF [4] [5] [6] [7]. These methods have been proven to be simple, fast, and accurate. However, when attackers are aware of these static features, they can evade easily [8]. Another recent work extracts and tests malicious Javascript in an emulated interpreter [9]. Although it is more robust against evasion, attackers can still exploit syntax obfuscations to subvert Javascript extraction. Also it is very costly to emulate all PDF-specific Javascript objects. In 2009, Adobe announced the Protected Mode, a sandboxing mechanism that runs PDF reader in a confined environment. Although it raises the bar, Adobe Sandbox has its own drawbacks. An obvious one is that there exist vulnerabilities in the sandbox itself. Actually hackers have already discovered different ways to escape Adobe Sandbox [10] [11].

The detection of malicious PDF exhibits two distinct challenges. First, users tend to open multiple PDFs simultaneously. However, the runtime behaviors of a PDF reader can vary as different documents are opened, and both benign and malicious PDFs are processed by one single thread in the PDF reader. These can inevitably affect detection accuracy due to the interference among multiple open documents. Second, although it is straightforward to locate traditional malware once detected, it is non-trivial to pinpoint these malicious PDF documents since all open documents could be malicious.

In this paper, we introduce a *context-aware* approach to detect and confine malicious Javascript in PDF through static document instrumentation and runtime behavior monitoring. Our method is motivated by the fact that some essential operations of Javascript in malicious PDF rarely occur in benign documents. Our context-aware approach can efficaciously overcome the aforementioned two challenges. On one hand, context-aware approach can make detection features, like suspicious memory consumption, more effective in detection. On the other hand, the context information explicitly indicates which open documents are malicious.

There are different ways to achieve context-aware monitoring. One intuitive choice is to extract Javascript from documents [9] [14]. Alternatively, Javascript interpreters can be instrumented [15]. But these methods are neither robust nor easy to implement in practice. Instead, we choose to perform *static document instrumentation*. This method, to the best of our knowledge, has never been explored before for PDF malware detection and confinement. For each PDF Javascript snippet, we include a prologue and epilogue to inform our runtime detector for the entry to and exit from

Table I: Existing Methods to Detect and Confine Malicious PDF.

Method	Difficult to Evade	End-Host Deployment	Need Emulation	Low Overhead
Signature	No	Yes	No	Yes
Structural [5] [4] [6]	No	Yes	No	Yes
Extract-and-Emulate [9]	Neutral	No	Yes	No
Lexical Analysis of Javascript [7]	Neutral	Yes	No	Yes
Adobe Sandboxing [12]	Neutral	Yes	No	Yes
CWSandbox [13]	Neutral	No	Neutral	No
Our Method	Yes	Yes	No	Yes

Javascript context. The advantage of using static document instrumentation over the other two alternatives lies in three aspects. First and most important, it is immune to code and syntax obfuscations. Second, it does not need to emulate Javascript interpreters, resulting in much less development effort and minor computational overhead. Last but not least, it provides good portability and can be easily deployed at end hosts.

When an instrumented document is loaded, our runtime detector monitors the behaviors of a PDF reader process and identifies potential infection attempts from Javascript. The infection attempt manifests itself through a sequence of suspicious actions, such as exploiting to compromise systems, retrieving malware and executing it. By monitoring these suspicious behaviors as evidence of infection, we compute a weighted sum to detect malicious PDF.

Our system also defines five novel static features for detection. These features characterize the obfuscation techniques frequently used in malicious PDF. The combination of static and runtime features will be more effective and robust than existing methods, which are either fully static [5] [4] [6] or fully dynamic [9] [13]. A more thorough comparison between our method and others is presented in Table I.

For any new intrusion detection mechanism, we need to perform a security analysis—a task that in many cases is even more important than its detection performance. In principle, it is required that the defense system remains robust and secure even when its internal operation is exposed to attackers. To this end, we conduct a security analysis of our approach showing that our system is still effective in detection and robust against evasion attacks even in the presence of a sophisticated adversarial environment. In particular, a list of potential advanced attacks are discussed and mitigations for their impact are presented.

To validate the efficacy of our system, we conduct a series of experiments using a corpus of 18623 benign and 7370 malicious PDF documents. The experimental results show that our static and runtime features achieve very promising detection performance. No false positive and few (25 out of 942) false negatives are generated during the evaluation. It takes only 0.04 seconds on average to instrument a malicious sample and about 5.5 seconds to process a very large (20 MB) document. The slowdown caused by our runtime detector is 0.093 seconds for a single Javascript. Even when as many as 20 separate scripts are instrumented, the slowdown does not exceed 2 seconds. Overall, our system provides an effective defense against malicious PDF in practice.

The remainder of the paper is outlined as follows. In Section II, we survey related work. The system design is detailed in Section III. Then in Section IV, we analyze the robustness of our system under the assumption of an advanced attacker. The evaluation results are presented in Section V. Section VI discusses the limitations and future work. Finally, we conclude in Section VII.

II. Related Work

Existing research on malicious PDF detection has taken two directions, *static* methods which build statistical models from document content and classify unknown samples using machine learning, and *dynamic* methods which execute suspicious Javascript in some constrained environments.

Early static methods are based on n-gram analysis to detect universal malicious files [16] [17]. In 2011, Laskov et al. [7] presented PJScan, the first static method dedicated to the detection of malicious PDF. Using a patched SpiderMonkey, PJScan extracts lexical tokens of Javascript and trains an OCSVM (One Class Support Vector Machine) classifier to identify malicious PDF. Instead of analyzing Javascript, Malware Slayer [6] inspects the content of malicious PDF and counts the frequency of PDF keywords. Then, a set of keywords with high frequency are selected and fed into various machine learning algorithms for detection. PDFRate [4] extracts more structural features from PDF and thus builds a more accurate classifier. It can also detect targeted attacks. Srndic et al. [5] proposed a structural-path based method. They modeled a document as a set of structural paths and detected malicious PDF using Decision Tree and SVM (Support Vector Machine). Wepawet [18] uses JSAND [14], which leverages statistical and lexical features of Javascript, to detect malicious PDF. In general, static methods have been proven to be simple, fast, and effective. However, they are susceptible to mimicry attacks [8]. Our method differs from these fully static methods in that, besides static features, we also use runtime behaviors of malicious Javascript for detection.

Compared with static detection, dynamic approaches are more robust against mimicry attacks. Tzermias et al. [9] proposed MDScan, which extracts Javascript from documents and executes it in instrumented SpiderMonkey and Nemu [19]. However, such a method suffers several limitations. First, it requires reliable Javascript extraction, which can be subverted by syntax obfuscations. Attackers can hide shellcode at some weird places in a document, e.g., in the title, and reference it in forms like "`this.info.title`". In this case, the extracted Javascript will fail to execute in emulated environments. Moreover, it is required to emulate

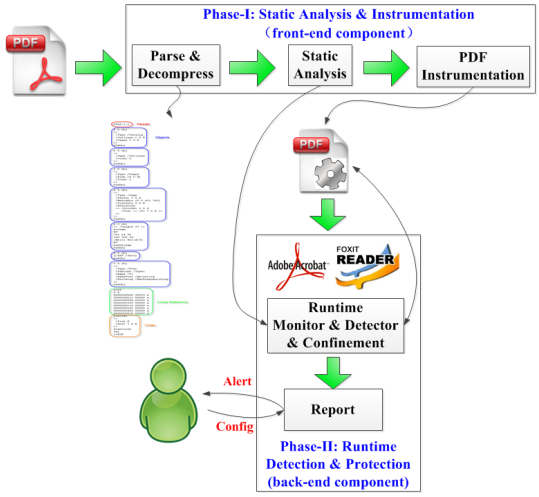

Figure 1: System Architecture

PDF-specific Javascript objects, both documented [20] and undocumented like `printSeps()`. Finally, the proposed defense cannot be readily deployed on a user's system.

Meanwhile, malicious Javascript detection on the Web is a well-studied topic and many methods have been proposed [14] [15] [21] [22]. However, these methods are specially designed for detecting malicious Javascript on the Web and they are mainly based on the analysis of Javascript code itself. Differently, our approach monitors suspicious *system*-level behaviors in the context of Javascript execution.

Similar to our approach, CWSandbox [13] and PEB heuristics [23] also detect suspicious runtime behaviors of document readers. However, CWSandbox [13] is used primarily for detecting traditional malware, and it can be easily evaded by event-triggering or environment-sensitive malicious Javascript. Polychronakis et al. [23] proposed to execute shellcode in a CPU emulator and detect suspicious memory accesses using four heuristics. Egele et al. [24] presented a similar method which identifies potential shellcode at runtime and tests it in libemu [25]. Compared with these methods, we use different and more robust runtime features, which characterize the essential operations required in the infection process. Moreover, we neither identify shellcode, which can be evaded by using English Shellcode [26], nor emulate CPU, which is heavyweight. Snow et al. [27] proposed to monitor system call sequences of document readers. However, they didn't model the infection process of malicious documents and their method is context-free.

III. System Design

A. Architecture

Our system consists of two major components, front-end and back-end, working in two phases. In Phase-I, the front-end component statically parses the document, analyzes the structure, and finally instruments the PDF objects containing Javascript. Then, in Phase-II when an instrumented document is opened, the back-end component detects suspicious behaviors of a PDF reader process in context of Javascript execution and confines malicious attempts. Figure 1 shows the architecture of our system.

Static Analysis and Instrumentation: For suspicious PDF, the front-end first parses the document structure and then decompresses the objects and streams. A set of static features are extracted in this process. When a document has been decompressed, the front-end will instrument it and add *context monitoring code* for Javascript. In some cases, if the document is encrypted using an owner's password, i.e., a mode of PDF in which the document is readable but non-modifiable, we need to remove the owner's password. With the help of PDF password recovery tools like [28], this can be done easily and very fast.

Runtime Detection: The back-end component works in two steps, runtime monitoring and runtime detection. When an instrumented PDF is loaded, the context monitoring code inside will cooperate with our runtime monitor, which tries to collect evidence of potential infection attempts. When Javascript executes to the end or a critical operation occurs, the runtime detector will compute a malscore. If the malscore exceeds a predefined threshold, the document will be classified as malicious.

B. Static Features

Several recent works have proposed to detect malicious PDF by statically analyzing document content [4] [5] [6]. Static methods are simple, and they have promising performance in detecting existing malicious documents. In this work, we define five novel static features to aid runtime detection by leveraging the obfuscation techniques used in malicious PDF. Although static features are vulnerable to evasion, their usefulness for detection lies in two aspects: (1) if malicious documents use obfuscations, our system can detect them with higher confidence; and (2) if not, then the unobfuscated documents can be processed more easily and accurately by our front-end component. In the following, we detail the static features used in our system.

Ratio of PDF Objects on Javascript Chain: In PDF, a labelled object is called an indirect object, which can be referred to by other objects [29]. Sometimes, there are several indirect objects between the root and the one containing real data. These PDF objects form a reference chain. In the sample PDF as shown in Figure 2, there are ten indirect objects. We extract every chain containing at least one Javascript object on the path. We call it a *Javascript chain*. This feature computes the ratio of the objects involved in Javascript chains to the total objects in a document. Normally, malicious documents contain few data and many of them have only one blank page. Thus, in malicious documents, the ratio should be relatively high.

PDF Header Obfuscation: The PDF specifications require only that the header appears somewhere within the first 1,024 bytes of the file [29]. Benign documents rarely have incentives to obfuscate PDF header, but malicious documents are more willing to do so. Actually a recent work has proposed to manipulate the file type identifiers to evade anti-virus software [30]. Another trick attackers can use is to specify an invalid version number in header. Our system checks if PDF header appears at the very beginning of a document and if the header format is valid.

The following three features are checked for objects on Javascript chains only.

Hexadecimal Code in Keyword: PDF standard allows any character except NULL to be represented by its 2-digit hexadecimal code, preceded by one or more number signs (#). Many malicious documents use this trick to hide keywords. For example, in object (4 0) in Figure 2, `/JavaScript` is encoded as `/JavaScr##69pt`.

Count of Empty Objects: Object (6 0) in Figure 2 shows a Javascript chain from a malicious PDF. In this document, the Javascript chain ends with an empty object. Actually, the real malicious Javascript is embedded in another chain. Our system counts the number of empty objects in a document.

Levels of Encoding: Encoding in PDF is used primarily for compression. Normally benign documents use only one level of encoding since multi-encoding brings little improvement. However, malicious documents tend to use multiple levels to evade anti-virus software.

Our system records the maximal encoding levels used on Javascript chains. Maximum, rather than average, is used for two reasons: on the one hand, maximum is more effective; on the other hand, average is susceptible to mimicry attacks. For example, attackers can deliberately insert many Javascript chains with one level of encoding. In this case, the average drops close to one.

C. Document Instrumentation

Due to its wide-spread adoption, simplicity, and strong expressiveness, Javascript is employed by the vast majority of malicious PDFs in the wild. Therefore, identifying and confining malicious Javascript in PDFs can effectively mitigate the risk they currently pose to Internet users. Motivated by the fact that malicious Javascript behaves significantly different from the benign one in system-level, we propose a context-aware detection and confinement approach. The core idea is to confine operations that are deemed suspicious based on the context of Javascript execution.

In order to implement the context-aware approach, one of the challenges is to identify when Javascript starts to execute and when it finishes. A simlpe solution is to extract Javascript from documents and execute it in an emulated environment. However, the extract-and-emulate method cannot guarantee reliable Javascript extraction, as demonstrated by an example shellcode in object (4 0) in Figure 2. Moreover, it can be very computationally expensive to emulate PDF-specific objects. An alternative option is to instrument a Javascript interpreter. For example, a snippet of monitoring code can be inserted at the entry and exit points of the Javascript interpreter. Although easy to implement, we do not choose this approach for two reasons. First, interpreter instrumentation is insecure and can be easily bypassed. Second, interpreter instrumentation has poor portability.

To overcome the aforementioned limitations, we propose to leverage static document instrumentation, which requires neither Javascript extraction nor environment emulation. Using our approach, a snippet of context monitoring code is inserted into the document statically. Every time Javascript gets executed and finishes execution, the context monitoring code takes control and informs our runtime detector.

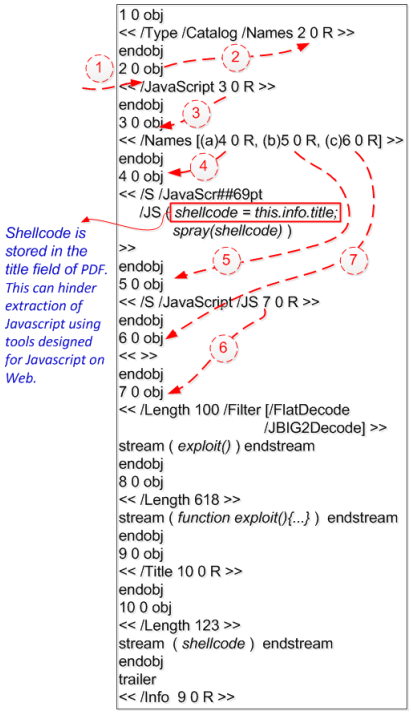

Figure 2: A Synthetic Sample of Malicious PDF. The start point can be object (2 0), (4 0), or (5 0). Any object can be selected as the start point, and here we assume (2 0) as the start point.

The first step of our method is to reconstruct all Javascript chains in a document. We use a similar technique described in previous works [7] [8] [9] to locate Javascript. Specifically, we scan the document for keywords `/JS` and `/JavaScript` that indicate a string or stream containing Javascript [29]. Next, we recursively backtrack to find the ancestors on a chain and forward search for the descendants. At the end of this process, we can extract a collection of Javascript chains. We only instrument the chains associated with some triggering actions, such as `/OpenAction` and `/AA`. Figure 2 illustrates the execution steps of the aforementioned algorithm. This algorithm is quite robust since it is immune to Javascript code obfuscation, and according to [29], the keyword `/JavaScript` should be plain text.

Javascript in PDF can be invoked either singly or sequentially (through `/Next` and `/Names`). The instrumentation process for single Javascript is shown in Figure 3. We first store the original code in a string which is passed as argument to `eval()` and then we prepend and append our context monitoring code to it. This process is quite simple and does not require sophisticated code analysis. The only operation we perform is to scan the code and add '\\' for ′ and ″ in the original Javascript code. When Javascript snippets are triggered, the context monitoring code, rather than the original script, gets executed first and it informs the runtime detector of the entrance and exit of Javascript context. During this process, the context monitoring code has to be able to communicate with the runtime detector. PDF provides three possible channels for communication: shared file, HTTP, and SOAP (Simple Object Access Protocol). Shared file is inefficient and insecure. The `Net.HTTP` method can be invoked only outside of a document [20],

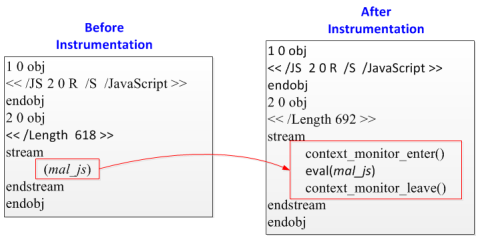

Figure 3: An Example to Illustrate Instrumentation

i.e., cannot work in our context monitoring code. We select SOAP for our implementation to avoid the pitfalls of the other communication options. To achieve that, a tiny SOAP server is built into the detector enabling the communication with the context monitoring code synchronously. A randomly generated key is used to protect the SOAP communications. The key has two parts, Detector ID and Instrumentation Key. Detector ID is generated when our system is installed. In case that an already instrumented document is downloaded, this field can be used to filter out communications from the invalid context monitoring code. The second field is randomly generated when instrumenting a document and it uniquely identifies an instrumented document. We also maintain a mapping between instrumented document and key. When instrumenting a file, we first ensure that no duplicate instrumentation is carried out on a single document. We further discuss the security of the key in Section IV.

For sequentially invoked scripts, the process is a little different. We can simply insert the context monitoring code for each separate Javascript listed in `/Names` dictionary or `/Next` field. However, this can incur intolerably high overhead. A better choice is to parse the chain and enclose all scripts invoked sequentially using one single context monitoring code, which is taken in our system.

Finally, attackers can also dynamically add Javascript using the methods listed in Table IV and delay the execution of Javascript using `setTimeOut()`. The two cases are specially handled in Section IV.

D. Runtime Features

When an instrumented PDF is opened, our stand-alone detector starts to monitor suspicious behaviors of the PDF reader and collect evidence of infection. We detect those essential operations that compromise target systems.

To improve the chance of successful exploits given various modern security enhancements, heap spraying has become the preferred weapon in hackers' arsenal. When heap is sprayed, a vulnerability like CVE-2008-2992 can be triggered to transfer the control to shellcode, which will execute the dropped malware, carry out drive-by-download, or establish a reverse bind shell. All of these operations should rarely occur in benign Javascript. Thus, any occurrence of these operations in the context of Javascript execution can be considered as suspicious. This is referred to as *JS-context* monitoring. In addition, we note that unlike browsers which normally work in multi-thread, PDF readers process documents in single-thread. That is, during the execution of Javascript, no other PDF objects in the same or another document will be processed. This fact simplifies our method and we do not need to consider the potential false positives caused by concurrency.

Table II: Runtime Behaviors Monitored in Two Contexts.

Context	Runtime Behaviors
Out-JS-Context	Process Creation and DLL Injection
JS-Context	Memory Consumption, Network Access, Mapped Memory Search, Malware Dropping, Process Creation, and DLL Injection

JS-context monitoring can effectively detect malicious documents that exploit the vulnerabilities in Javascript interpreters. However, attackers can also exploit other vulnerabilities like CVE-2010-3654 in Flash and CVE-2010-2883 in CoolType.dll. Javascript in such malicious documents is normally responsible for heap spraying and malformed data crafting. In such cases, probably the JS-context monitoring can detect only one suspicious operation, i.e., heap spraying, which is insufficient for accurate detection. To complement JS-context monitoring, we also monitor the runtime behaviors after Javascript finishes (*out-JS-context*).

Table II lists the runtime behaviors we monitor in the two contexts above. Each monitored behavior is defined as one runtime feature in our system. Essentially, these behaviors are modeled as sequences of system calls. While using system calls to detect anomaly is not new [27] [31] [32], our method differs in two aspects. First, most previous works focus on detecting the behavior deviations from expected execution. But we detect the infection attempts of malicious code. Second, although there exist works on modeling the behaviors of malware [32], our method relies on the context-aware monitoring which has not been explored in previous works. Below, we continue to explain the details of each monitored behavior.

Malware Dropping: A common practice of malicious PDF is to drop some malware to a user's file system. To monitor the malware dropping, we hook the APIs `NtCreateFile()`, `URLDownloadToFile*()`, and `URLDownloadToCacheFile*()` on Windows.

Suspicious Memory Consumption: In heap spraying, malicious code fills the heap with a NOP sled appended with shellcode. Subsequently, it attempts to divert the control flow to any address covered by the NOP sled that leads to the shellcode execution. In an effort to increase the probability of hitting a NOP, malicious code attempts to write a large area of memory, usually more than 100 MB [33].

Suspicious memory consumption can be very promising in detecting the presence of heap spraying, especially if monitored in JS-context. The context-free monitoring can cause many false positives, e.g., in a case that many documents are opened simultaneously. However, the context-aware monitoring in our method can effectively eliminate most noise. We check the `PROCESS_MEMORY_COUNTERS_EX` structure [34] at the entry/exit of JS context and when other in-JS sensitive APIs are captured.

Suspicious Network Access: Unlike on the Web, Javascript in PDF rarely connects to the Internet and its primary function is to dynamically render a document, which rarely relies on network communications. Actually,

the number of Javascript methods provided in PDF for network access is limited and most of them can be used only in restricted conditions. For example, `app.mailmsg()` and `app.launchURL()` establish network connections using third-party applications (email clients and browsers), which are not monitored by our runtime detector. And, the `Net.HTTP` object cannot be invoked by Javascript embedded in a document. Thus, any network connection generated in JS-context should be considered as suspicious. In our system, we hook all `connect` and `listen`. Note that we white-list the communications between the runtime detector and the context monitoring code.

Mapped Memory Search: Besides drive-by-download, attackers can also embed malware in a document. Such a technique is called Egg-hunt. In [35], a malicious sample using egg-hunt is analyzed. One challenge of egg-hunt is that attackers cannot know where malware is loaded in memory and they have to search the whole address space. However, some memory in the address space is unallocated, and dereferencing it can lead to segmentation fault. In order to prevent access violations, attackers have to employ some techniques to safely search the virtual address space. Several effective techniques, for both Linux and Windows, are described in [36]. In our implementation, `NtAccessCheckAndAuditAlarm()`, `IsBadReadPtr()`, `NtDisplayString()`, and `NtAddAtom()` are monitored.

Process Creation: The final step of an attack lies in execution of the dropped malware. Attackers can create a new process to execute the malware. In JS-context, this behavior can be a strong sign of infection attempt; while in out-JS-context, it can cause false positives. We observe that Windows error report programs and tools distributed with PDF readers, which obviously are benign, are usually invoked. So, we add them to a white-list. In implementation, we monitor `NtCreateProcess()`, `NtCreateProcessEx()`, and `NtCreateUserProcess()`.

DLL Injection: In the wild, usually attackers prefer to execute malware via DLL injection. This behavior should never occur in JS-context and rarely occur outside of JS-context. Thus, we monitor DLL injection in both JS-context and out-JS-context. In implementation, we monitor `CreateRemoteThread()`.

E. Runtime Detection and Confinement

Detection. The workflow of runtime detection and lightweight confinement is shown in Figure 4. The runtime detector works in three steps. Initially, all sensitive operations are ignored until at least one in-JS operation is captured from an unknown PDF. Although it may cause false negatives to discard out-JS operations at this step, we believe it is worthwhile for achieving a lower false positive rate and higher performance. Next, the detector starts to continuously record all sensitive operations. The core logic of the runtime detector is a weighted sum, as shown in Equation 1.

$$malscore = w_1 \sum_{i=1}^{7} F_i + w_2 \sum_{i=8}^{13} F_i. \quad (1)$$

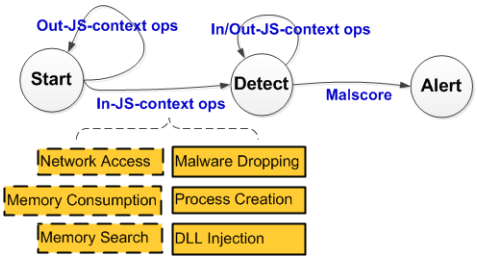

Figure 4: Workflow of Runtime Detection & Lightweight Confinement.

The first part represents the static and out-JS features. The second part denotes the in-JS features. The features are numbered from 1 to 13, and the runtime features are numbered in the order they appear in Table II. All these features are normalized to binary values. Instead of assigning a weight for each feature, we set a weight for each "part" in the equation. We also define a threshold and if the malscore exceeds it, the document is tagged as malicious. The feature normalization, weight and threshold setting are based on the statistical results of a large corpus of benign and malicious samples. We provide a detailed description in Section V-C.

In real world, users usually open many PDFs simultaneously, which must be correctly handled by the runtime detector. For each unknown open PDF which has carried out at least one in-JS operation, we maintain a separate malscore and a set of related operations. In-JS operations affect the corresponding malscore only, while out-JS operations contribute to every active malscore. Finally, in order to handle the case that multiple malicious PDFs work together to attack stealthily, we maintain a list of executables downloaded in JS context. When an in-JS operation invokes an executable in the list, we intentionally prepend a malware dropping operation for this PDF and append a malware execution operation for another PDF that downloads the file. Malscore is volatile, implying that it no longer exists when a PDF reader is closed. However, the maintained list of executables is persistently stored. When an alert is raised, we report the malscore, associated features, and the detected malicious PDFs to users.

Confinement. In Figure 4, the operations enclosed in solid border are confined. Our lightweight confinement, as well as runtime monitoring, is based on Windows API hooking. There are various ways to implement API hooking, e.g., modifying the system service dispatch table (SSDT) or the interrupt descriptor table (IDT). Our prototype adopts the import address table (IAT) hooking since it is simple, effective, and efficient. Although attackers could leverage `GetProcAddress()` or call kernel routines directly to bypass IAT hooking, it is quite uncommon [13]. In the future, we will use advanced kernel mode hooks to make it more difficult to evade.

An essential step of IAT hooking is to inject our hook DLL. There are two popular implanting techniques on Windows, i.e., remote thread injection and AppInit registry modification [37]. Our prototype adopts the latter approach. As AppInit modification can affect the whole system, which is undesirable, we utilize a similar technique introduced in [38]. The basic idea is to develop a trampoline DLL,

which further loads the IAT hook DLL if the host process is a PDF reader and otherwise does nothing. In this way, our confinement affects PDF readers only and thus incurs negligible overhead to the whole system.

Moreover, since API hooks execute in a PDF reader process, we need a channel for communications between API hooks and our stand-alone runtime detector. In our prototype, TCP socket is used. When the hook DLL is injected, its first job is to set up a TCP connection to the runtime detector. At runtime, it sends the captured API, API parameters, and memory usage (for suspicious memory consumption in §III-D) to the runtime detector.

Table III shows the pre-defined confinement rules executed by the runtime detector and Hook DLL. The rules are quite straightforward. The only issue that deserves attention is, in order to confine the created process, we use an existing sandbox tool, Sandboxie [39]. Currently, we just handle three sensitive operations. However, we can easily extend existing confinement rules.

F. De-instrumentation

In reality, it is common to open a document many times. In order to improve performance and scalability, we can monitor new documents only. We adopt an intuitive and simple approach, *document de-instrumentation*, to achieve this goal. When a document is identified as benign, our system removes the context monitoring code from it, i.e., de-instrumenting it. De-instrumentation is done in background after the PDF reader is closed. To facilitate de-instrumentation, our static instrumentation component will generate and export the corresponding de-instrumentation specifications when instrumenting a document. De-instrumentation significantly improves scalability while no security hole is introduced. Note that de-instrumenting at-once is a simple heuristic. A configurable parameter and randomization can be introduced to set the number of opens before de-instrumentation.

IV. Security Analysis

For any intrusion detection system, it is a must to enforce its own integrity and security. In this section, we first describe the threat model. Then, we present a list of potential advanced attacks and our countermeasures.

A. Threat Model

In our analysis, we assume an advanced attacker who can access our code and test it for unlimited times. Moreover, the attacker can embed some arbitrarily large shellcode in the document. The shellcode is able to: (1) identify the heap, stack, and code areas in memory; (2) scan the whole virtual address space; and (3) modify any memory content.

Meanwhile, we also assume that attackers can neither (1) understand the meaning of data in memory if there is no identifiable signature nor (2) manipulate our static instrumentation code since the instrumentation component gets executed before malicious code.

B. Potential Advanced Attacks and Countermeasures

Mimicry Attack: An obvious attack is the mimicry attack, targeting the messaging mechanism between the context monitoring code and the runtime detector. Attackers try to steal the key used in communications and send a fake message to the runtime monitor, mimicking the epilogue of the context monitoring code. Then, the shellcode can do anything without monitoring. An alternative approach is to search for our episode code and execute it before carrying out malicious operations. We argue that

> *our random key, context monitoring code randomization and duplication, and zero tolerance to fake message can effectively defeat such a mimicry attack.*

Attackers can use either signature-based [40] or test-based [41] methods to search for keys in memory. In many cases, the key is stored at some fixed addresses or somewhere near an identifiable string, e.g., "auth-password" or "MyPwd". Such a signature remains intact once software is released, and hence attackers can easily locate the key in memory. Our system avoids generating signatures through: (1) executing the context monitoring code using `eval()`; (2) generating the key randomly during static instrumentation; (3) randomizing the structure of the context monitoring code; and (4) creating copies of fake context monitoring code.

It is much easier to defeat the test-based cracking. We enforce that whenever a fake message is received, we tag the active document as malicious. Note that attackers cannot launch DoS attacks by pretending to be another PDF. As mentioned before, PDF readers work in single-thread and only one document is active at any time. From the key in the prologue, we can identify the active document, which is responsible for the fake message.

Runtime Patching Attack: Attackers can also carry out the runtime patching attack. There are two separate scripts in the document, so we instrument each of them independently. When the shellcode in the first script gets executed, it can locate the second script in memory and patch out the context monitoring code. Then, the second script can execute without monitoring. A variant attack is to distribute malicious Javascript in two separate documents.

To avoid the runtime patching attack, we ensure to take control at the beginning of each script. We apply encryption to enforce such control retaining. During instrumentation, an encryption scheme is randomly selected to encrypt the original script, and the decryption method is embedded in the prologue of the context monitoring code. In this way, malicious Javascript cannot get executed without our context monitoring code.

Moreover, several obfuscation methods are used to make it impossible for attackers to eliminate the context monitoring code but still keep the decryption code.

Staged Attack: An advanced attacker can split the exploit into multiple stages. Let us consider the simplest two-stage attack, as shown in Figure 5. In step 3, the Stage_2 code can be installed using Javascript methods listed in Table IV.

To defeat this kind of attack, we analyze the Javascript code and search for the methods in Table IV during static

Table III: Confinement Rules

Operation	Rules	
	Execute In Hook DLL	**Execute In Runtime Detector**
Malware Dropping	Before alert, call original API.	Before alert, maintain the list of downloaded executables; When alert, isolate.
Process Creation	Before alert, reject the call since it will be invoked by runtime detector.	Before alert, run target program in Sandboxie [39]; When alert, terminate and isolate the program.
DLL Injection	Always reject.	Isolate the injected DLL.

1) Instrument the target PDF.
2) Context monitoring code informs the enter of Javascript.
3) The Stage_1 shellcode setups Stage_2 code at runtime.
4) Context monitoring code informs the leave of Javascript.
5) Stage_2 shellcode is triggered by some event later.

Figure 5: Two-stage Attack

Table IV: Methods provided in PDF to add scripts at runtime.

Method	Trigger Event
Doc.addScript()	Open the document
Doc.setAction()	Close/Save/Print the document
Doc.setPageAction()	Open/Close a page
Field.setAction()	Operate on a form field
Bookmark.setAction()	Click the bookmark

instrumentation. Then, we instrument the dynamically added scripts that are stored in the parameters of these methods. A more robust solution we are working on is to hook these methods in Javascript interpreters and instrument dynamically inserted scripts on-the-fly. Since we only need to hook five methods, the development efforts and runtime overheads should be minor.

Delayed Execution: Another evasion approach is to delay the execution of Javascript. This can be achieved through `app.setTimeOut()` and `app.setInterval()` [20]. Our countermeasure is similar to the one for staged attack and we intentionally instrument the two Javascript methods above.

V. Evaluation

To validate the efficacy of our proposed approach, we implement a prototype on Windows. The front-end component is implemented in Python 2.7. The runtime monitor and detector in the back-end component are implemented in C and Java, respectively. And, the tiny SOAP server in the runtime monitor is built using the Web service framework JAX-WS. Based on a large corpus of real data, we first evaluate the effectiveness of our detection model and then examine the runtime overhead of our prototype.

A. Data Collection

We collected more than twenty thousand benign and malicious samples for this study. Table V summarizes the dataset used in our evaluation. The benign documents are from four trusted sources: (1) we collected thousands of documents from two users' file systems; (2) we downloaded hundreds of official forms and reports from large organizations like governments and well-known companies; (3) we collected a set of non-malicious PDF files from Contagiodump [42]; and (4) we randomly crawled over ten thousand of documents using Google and tested them using anti-virus software. The malicious samples are from Contagiodump and those containing no Javascript are excluded.

Table V: Dataset Used for Evaluation

Category	# of Samples	# with Javascript	Size
Known Benign	18623	994	11.84 GB
Known Malicious	7370	7370	172 MB
Total	25993	8364	12.01 GB

B. Feature Validation

Before measuring detection accuracy, we first validate the capability of our detection features to distinguish between benign and malicious documents. Here we present the statistical results of the features used in our system.

Static Features: We scanned all benign documents and found 994 samples containing Javascript. The following evaluation mainly relies on these 994 samples.

The first static feature we validate is the ratio of PDF objects on Javascript chains. Figure 6 shows the cumulative distribution function of the ratio in benign and malicious documents. As we can see, about 95% of malicious documents have a ratio over 0.2. We even found 64 samples with a ratio of 1. This is reasonable since malicious documents usually contain only one blank page. By contrast, the ratio in benign documents presents a quite different pattern. From the dotted line in Figure 6, we can clearly see that about 90% of benign documents have a ratio smaller than 0.2 and almost no document has a ratio over 0.6. The results indicate that this feature can effectively distinguish between benign and malicious documents.

The statistical results of the other static features in malicious documents are shown in Table VI. For boolean features, "False" is denoted as 0 and "True" as 1. We found that while empty objects can be found in malicious samples, no benign documents contain empty object. This complies with our intuition that people rarely have incentive to include these junk objects in documents and normally they tend to use automatic tools like `this.addscript()` and [43] to insert Javascript. These tools rarely generate empty objects. Unlike previous two features, more malicious samples use header obfuscation and hex code. As a comparison, we only found three benign documents with header obfuscation and no benign document contains hex code. We believe this is because usually PDF documents are created from other formats like Microsoft Word and LaTeX using automatic

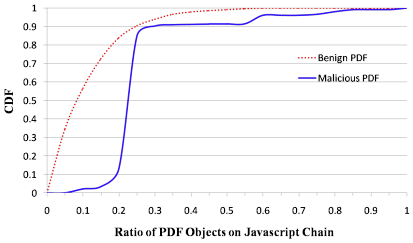

Figure 6: Ratio of PDF Objects on Javascript Chain in Malicious and Benign Documents

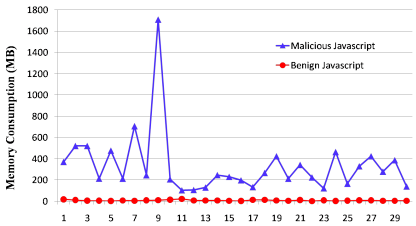

Figure 7: Memory Consumption of Malicious and Benign Javascripts

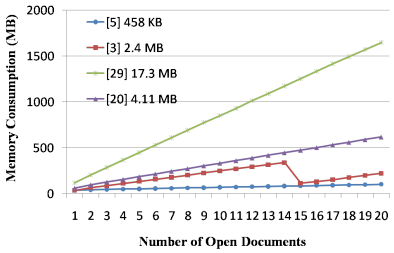

Figure 8: Memory Consumption of PDF Reader When Opening Many Documents

conversion tools. Such tools do not obfuscate document header or structure. Finally, only about 1% of malicious samples use multiple levels of encoding, and surprisingly about 3% of them do not use any encoding. In benign documents, we found that all of them use either zero or one level of encoding. Overall, these five features complement with the first feature and enable us to more accurately distinguish between benign and malicious documents.

Table VI: Statistics of Static Features of Malicious Documents.

Feature \ Value	0/False	1/True	2	3	6
Header Obfuscation	6792	578	-	-	-
Hex Code	6827	543	-	-	-
Empty Objects	7357	5	4	3	1
Encoding Level	233	7065	40	31	0

Memory Consumption: We randomly sampled 30 documents from each of two categories, "Known Benign" and "Known Malicious", respectively. All of the 30 selected benign documents contain Javascript. Then, we measured the memory consumption of the sampled 60 documents in JS-context and the results are shown in Figure 7. As we can see, one malicious sample can consume more than 1700 MB memory. On average, malicious samples consume about 336.4 MB memory while benign documents consume merely 7.1 MB. Moreover, the minimal memory consumed by malicious samples is 103 MB but the maximum by benign samples is only 21 MB. These results indicate that our context-aware monitoring of memory consumption could be an effective feature to differentiate between benign and malicious documents.

Context-aware v.s. Context-free. However, only if the monitoring is conducted in JS-context, will memory consumption be an effective feature. The context-free monitoring could be inaccurate. In order to demonstrate the deficiency of the context-free monitoring, we measure the memory consumption of a PDF reader when different number of documents are opened at the same time. Note that opening many documents simultaneously is a common practice in daily life. In our evaluation, we used Adobe Acrobat 9.0 and four documents with various size from our reference list, including [3] [5] [20] [29]. For each document, we made 20 copies and recorded the memory consumption of Acrobat when different number of copies were opened simultaneously. The results are shown in Figure 8. In most cases, the memory consumption increases linearly with the inceasing number of opened documents and it can grow up to 1600 MB. An exception is [3]. When the 15th copy is opened, the memory consumption drops to a lower level and then increases linearly again. We tested many times and this effect appeared in every test. Our speculation is that this specific document triggers some memory optimization mechanisms in Acrobat. From these results, we can see that it is almost impossible to set an appropriate threshold in the context-free monitoring. A high value could miss a large fraction of malicious documents while a low value may generate many false positives. Besides, as shown in Figure 8, the memory increase of [29] is also very large. Thus, in the context-free monitoring, the memory increase of a PDF reader is not a good feature either. By contrast, our context-aware monitoring is much more effective and accurate.

C. Detection Accuracy

We evaluate the detection accuracy of our prototype, in terms of false positive rate and false negative rate. We tested the malicious samples in VMware Workstation hosting Win XP SP1 with Adobe Acrobat 8.0/9.0 installed. We first describe the parameter configuration of our detector and then present the detection results.

1) Parameter Configuration

First, we normalize non-binary features, including F1, F4, F5, and F8. The normalization rules are listed in Table VII. According to Figures 6 and 7, we set F1 as 1 when the ratio ≥ 0.2 and F9 as 1 when the memory consumption ≥ 100 MB. Similarly, the values of F5 and F6 are set according to Table VI. In this way, all 13 features can be represented in binary values.

To set the weights and threshold, we need to meet the criterion that a document is tagged as malicious *iff* at least one JS-context feature and any other features have positive values. The basic idea is that if no suspicious behavior is detected in JS-context, the document contains no malicious Javascript and thus it is out of the scope of our detection. According to the criterion, we set w_1 as 1, w_2 as 9, and the threshold as 10, respectively.

2) Detection Results

We measured the false positive and false negative rates of the tuned detector over all benign documents with Javascript (994) and one thousand randomly selected malicious samples. The malicious samples cover vulnerabilities

Table VII: Parameter Configurations in Our System.

Parameter	Value
F1	If ratio $\geq$ 0.2, F1 = 1; else F1 = 0;
F4	If # of empty objects $\geq$ 1, F4 = 1; else F4 = 0;
F5	If encoding level $\geq$ 2, F5 = 1; else F5 = 0;
F8	If mem consumption $\geq$ 100 MB, F8 = 1; else F8 = 0;
w_1	1
w_2	9
Threshold	10

Table VIII: Detection Results

Category	Detected Malicious	Detected Benign	Noise	Total
Benign Samples	0	994	0	994
Malicious Samples	917	25	58	1000

in Javascript interpreter, Flash, U3D (Universal 3D), TIFF and JBIG2 image, etc. The detection results are shown in Table VIII.

It can be seen that no benign sample is misclassified as malicious, achieving zero false positive. There is only one sample with suspicious behavior in JS-context. However, since there is no other feature with positive value, this sample is still classified as benign. Afterwards, we checked the sample and confirmed that the script uses SOAP for network access. The rest 993 samples are tagged as benign simply because no suspicious JS-context behavior is monitored, although some samples have positive values in other features. Even though Javascript methods like SOAP and ADBC can generate network accesses, we are reluctant to white list them since we cannot decide the maliciousness of the target server.

During the test, 58 ($\sim$6%) of the malicious samples did nothing when opened. Inspecting those samples, we found that these samples exploited either CVE-2009-1492 [44] or CVE-2013-0640 [45] which do not work on Adobe Acrobat 8.0/9.0. As these samples failed to exploit, we excluded them when computing false negative rate. For the rest 942 samples, we successfully detected 917, with a detection rate of 97.3%. We examined the 25 undetected samples and we found two reasons that cause the misses. First, although malicious Javascripts in these samples spray the heap, the PDF reader process crashes when the scripts attempt to hijack the control flow. Second, the 25 undetected samples use no obfuscation and thus no static feature contributes to detection. Actually there are more than 25 samples that crash the PDF reader process, but the others are detected by our system via suspicious memory consumption and static features. Although false negatives are unavoidable when malicious PDF fails to exploit, it does not violate our primary goal, i.e., protecting users from damages of malicious PDF.

Table IX compares our method with previous countermeasures in terms of false positive rate and true positive rate. It is clear that our method is comparable with the best fully static methods [4] [5]. Since the malicious samples in our dataset are not the most recent (the latest was captured in Feb. 2013), we cannot fully demonstrate the superiority of our system over the fully static methods. Thus, we further compare our system with other methods by analyzing possible advanced attacks.

Table IX: Comparison With Existing Methods

Method	False Positive	True Positive
N-grams [17]	31%	84%
PJScan [7]	16%	85%
PDFRate [4]	2%	99%
Structural [5]	0.05%	99%
MDScan [9]	N/A	89%
Wepawet [18]	N/A	68% [9]
Ours	0	97%

- *Our approach v.s. Structural methods*: The mimicry attacks proposed in [8] can effectively bypass these structural methods [4] [5] [6] [7]. However, our approach is immune to the proposed attacks in that we detect the malicious attempts from Javascript rather than how malicious Javascript is stored in PDF.
- *Our approach v.s. Anti-virus Software*: There are a whole bunch of tricks available in the wild to evade anti-virus software [30] [46] [47]. Attackers can easily generate variants using these tricks to defeat anti-virus software. Compared with anti-virus software, our method can effectively detect new variants and zero-day malicious PDF in time because we use the inconcealable system-level behaviors of malicious PDF for detection.
- *Our approach v.s. Dynamic Analysis Tools*: Attackers can subvert existing dynamic analysis tools like CWSandbox [13] using event-triggering and environment-sensitive malcode. Our method does not suffer this limitation since we detect as real users operate on malicious documents.

Based on the analysis of potential advanced attacks, we can see that our method is more robust than existing defense against malicious PDF.

D. System Performance

To measure the runtime overhead of our method, we run our prototype on 32-bit Windows 7. We performed the tests on a laptop with a 2.53 GHz Intel Core 2 Duo CPU processor and 2 GB of RAM. The performance of each component in our system is presented below.

1) Static Analysis and Instrumentation

Overall, it took about 297.7 seconds to process all 7370 malicious samples, i.e., 0.04 seconds on average for each sample. We also measured the overhead when processing the files with various sizes. We randomly selected three benign and malicious documents, respectively. The sizes of these documents are shown in Table X. One of the malicious samples contains two scripts and the rest of five documents contain only one script.

The execution time of each step in static analysis and instrumentation is shown in Table X. We can see that the overhead is minor for both large and small documents. In particular, it took only about 5.5 seconds to process a 20 MB document. Considering that it could take 20 seconds to download the document (in case of 1 MB/s), the additional delay of 5.5 seconds for processing it is acceptable.

Table X: Execution Time (in seconds) of Static Analysis & Instrumentation.

PDF Size	Parse & Decompress	Feature Extraction	Instrumentation	Total
2 KB	0.0005	0.0255	0.0183	0.0444
9 KB	0.0008	0.0867	0.0138	0.1014
24 KB	0.0007	0.0726	0.0247	0.0981
325 KB	0.0569	0.0210	0.0236	0.1016
7.0 MB	0.8954	0.4023	0.0773	1.3750
19.7 MB	3.2219	2.0015	0.2761	5.4995

Table XI: Memory Overhead of Static Analysis & Instrumentation.

PDF Size	# of Python Objects	Memory Consumption
2 KB	74095	5.26 MB
9 KB	74085	5.26 MB
24 KB	74112	5.28 MB
325 KB	74616	5.63 MB
7.0 MB	366845	42.86 MB
19.7 MB	1081771	130.6 MB

Whereas most of the execution time is spent on feature extraction and instrumentation for small documents, the dominant overhead comes from parsing and decompressing as document size increases, which accounts for over 95% of the total execution time. Besides, for instrumentation, the overhead depends on the number of scripts. That is why it took more time to instrument the 2 KB file than the 9 KB file in Table X. The overhead increase is approximately linear. This is because during feature extraction, we have tagged the PDF objects containing Javascript code and our instrumentation component only needs to locate and instrument them.

In summary, the evaluation results indicate that the component of static analysis and instrumentation incurs minor overhead and can be used for end-host protection.

We also profiled memory overhead. Table XI presents the memory usage during static process. The memory overhead is a little bit high. However, since the front-end component works off-line and the RAM on modern systems can easily accomodate such a memory demand, the overhead is acceptable. Actually, for most documents, the memory overhead of our system is comparable with PDF readers like Adobe Acrobat. In the future work, we will optimize our program and use memory more efficiently.

2) Runtime Detector

The runtime detector with a tiny SOAP server requires about 19 MB memory. Although the detector maintains the state (i.e., all features) for each unknown open document, we found that the memory usage increases a little as the number of monitored documents increases. Thus, the overhead of our runtime detector is also minor.

We further evaluated the efficiency of our context monitoring code. We manually crafted a set of documents containing various copies of Javascript. The Javascript is from a randomly selected malicious sample. In total, we got 20 documents with 1 to 20 separate scripts in each document. For each crafted document, we measured the total execution time of Javascript before and after instrumentation. When one script is instrumented, the additional execution time incurred by our context monitoring code is about 0.093 seconds. Since most malicious documents in the wild contain only one script, this overhead represents the common case. Note that, although both benign and malicious documents can contain many scripts, in most cases these scripts are invoked sequentially via `/Names` and `/Next`. Thus, only one piece of the context monitoring code is inserted. Basically, the overhead grows linearly as the number of instrumented scripts increases. However, when there are 20 scripts, the overall overhead is still below 2 seconds. Benign documents may contain many singly invoked scripts, but in most cases these scripts are associated with some actions that probably are not triggered simultaneously. Therefore, when the overall overhead is distributed among each script, the performance degradation is still minor. In summary, our context monitoring code is efficient enough for online protection.

VI. Limitations and Future Work

Although the majority of existing malicious PDFs use Javascript to launch attacks, attackers can also have other options like ActionScript. Our approach cannot detect those malicious PDFs that do not use Javascript as the attack vector.

We have also not evaluated the effectiveness of our method for in-browser PDF viewers. The challenge lies in two aspects. First, in-browser PDF viewers usually start to render before a document has been completely downloaded. This brings difficulty to our static analysis and instrumentation. Moreover, runtime behaviors of a browser is much more complicated than those of a PDF reader. In our future work, we will create new runtime features for browsers and be able to detect malicious PDF in an in-brower PDF viewer.

Finally, we have not handled embedded PDF documents. In the future work, we will extract static features from both embedded and host PDFs. It would be also valuable to instrument embedded documents, and we plan to correlate the runtime behaviors from both embedded and host documents. In this way, we can effectively defeat the mimicry attacks proposed in [8].

VII. Conclusion

In this paper, we developed an effective and efficient hybrid approach—leveraging five novel static features and the context-aware behavior monitoring—for detection and confinement of malicious Javascript in PDF. The static features are designed to detect the obfuscation techniques that are widely used by malicious PDF but usually disregarded by benign documents. We also observed that the indispensable operations for malicious Javascript to compromise target systems rarely occur in JS-context. Based on this observation, we presented the static document instrumentation method to facilitate context-aware monitoring of potential infection attempts from malicious Javascript. The intrusive nature of instrumentation method endows our system with immunity to Javascript code and PDF syntax obfuscations. To validate the efficacy of our proposed approach, we conducted a security analysis given an advanced attacker, showing that

our method is much more robust than existing defense. The experimental evaluation based on over twenty thousand benign and malicious samples shows that our system can achieve very high detection accuracy with minor overhead.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their insightful and valuable comments. This work was partially supported by ONR grant N00014-13-1-0088 and AFRL Contract FA8650-10-C-7024.

REFERENCES

[1] "The rise in the exploitation of old pdf vulnerabilities," http://blogs.technet.com/b/mmpc/archive/2013/04/29/the-rise-in-the-exploitation-of-old-pdf-vulnerabilities.aspx.
[2] "http://www.cvedetails.com/product/497/adobe-acrobat-reader.html?vendor_id=53," accessed in June 2013.
[3] K. Selvaraj and N. F. Gutierrez, "The rise of pdf malware," Symantec, Tech. Rep., 2010.
[4] C. Smutz and A. Stavrou, "Malicious pdf detection using metadata and structural features," in *Proceedings of Annual Computer Security Applications Conference (ACSAC)*, 2012.
[5] N. Srndic and P. Laskov, "Detection of malicious pdf files based on hierarchical document structure," in *NDSS*, 2013.
[6] D. Maiorca, G. Giacinto, and I. Corona, "A pattern recognition system for malicious pdf files detection," in *Proceedings of International conference on Machine Learning and Data Mining in Pattern Recognition (MLDM)*, 2012.
[7] P. Laskov and N. Šrndić, "Static detection of malicious javascript-bearing pdf documents," in *Proceedings of Annual Computer Security Applications Conference (ACSAC)*, 2011.
[8] D. Maiorca, I. Corona, and G. 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 ACM SIGSAC symposium on Information, computer and communications security (AsiaCCS)*, 2013.
[9] Z. Tzermias, G. Sykiotakis, M. Polychronakis, and E. P. Markatos, "Combining static and dynamic analysis for the detection of malicious documents," in *Proceedings of European Workshop on System Security (EUROSEC)*, 2011.
[10] Z. Liu, "Breeding sandworms: How to fuzz your way out of adobe reader x's sandbox," in *Blackhat*, 2012.
[11] P. Vreugdenhil, "Adobe sandbox when the broker is broken," in *Cansecwest*, 2013.
[12] "Protected mode," http://www.adobe.com/devnet-docs/acrobatetk/tools/AppSec/protectedmode.html, 2013.
[13] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using cwsandbox," *IEEE Security and Privacy*, 2007.
[14] M. Cova, C. Kruegel, and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious javascript code," in *Proceedings of International conference on World wide web (WWW)*, 2010.
[15] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert, "Zozzle: fast and precise in-browser javascript malware detection," in *Proceedings of USENIX Security Symposium*, 2011.
[16] W.-J. Li, S. Stolfo, A. Stavrou, E. Androulaki, and A. D. Keromytis, "A study of malcode-bearing documents," in *Proceedings of International conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2007.
[17] M. Z. Shafiq, S. A. Khayam, and M. Farooq, "Embedded malware detection using markov n-grams," in *Proceedings of International conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2008.
[18] "Wepawet," http://wepawet.cs.ucsb.edu/.
[19] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos, "An empirical study of real-world polymorphic code injection attacks," in *LEET*, 2009.
[20] *JavaScript for Acrobat API Reference*, 2007.
[21] W. Xu, F. Zhang, and S. Zhu, "Jstill: mostly static detection of obfuscated malicious javascript code," in *Proceedings of ACM conference on Data and application security and privacy (CODASPY)*, 2013.
[22] C. Yue and H. Wang, "Characterizing insecure javascript practices on the web," in *Proceedings of the 18th International Conference on World Wide Web (WWW'09)*, 2009.
[23] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos, "Comprehensive shellcode detection using runtime heuristics," in *Proceedings of Annual Computer Security Applications Conference (ACSAC)*, 2010.
[24] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda, "Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks," in *Proceedings of International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2009.
[25] "libemu," http://libemu.carnivore.it/.
[26] J. Mason, S. Small, F. Monrose, and G. MacManus, "English shellcode," in *Proceedings of ACM conference on Computer and communications security (CCS)*, 2009.
[27] K. Z. Snow and F. Monrose, "Automatic hooking for forensic analysis of document-based code injection attacks: Techniques and empirical analyses," in *Proceedings of the European Workshop on System Security (EuroSec)*, 2012.
[28] "Free pdf password remover," http://www.4dots-software.com/pdf-utilities/free-pdf-password-remover/.
[29] *PDF Reference (sixth edition)*, 2006.
[30] S. Jana and V. Shmatikov, "Abusing file processing in malware detectors for fun and profit," in *IEEE Symposium on Security and Privacy (SP)*, 2012.
[31] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff, "A sense of self for unix processes," in *Proceedings of IEEE Symposium on Security and Privacy*, 1996.
[32] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang, "Effective and efficient malware detection at the end host," in *Proceedings of USENIX Security Symposium*, 2009.
[33] A. Sotirov, "Heap feng shui in javascript," in *Blackhat Europe*, 2007.
[34] *PROCESS_MEMORY_COUNTERS_EX structure*, http://msdn.microsoft.com/en-us/library/ms684874%28v=vs.85%29.aspx, accessed in June 2013.
[35] A. Schneider, "Whos looking for eggs in your pdf?" http://labs.m86security.com/2010/11/whos-looking-for-eggs-in-your-pdf/, 2010.
[36] skape, *Safely Searching Process Virtual Address Space*, 2004.
[37] "Working with the appinit_dlls registry value," http://support.microsoft.com/kb/197571.
[38] "Loaddllviaappinit," http://blog.didierstevens.com/2009/12/23/loaddllviaappinit/.
[39] "Sandboxie," http://www.sandboxie.com/.
[40] S. Davidoff, "Cleartext passwords in linux memory," 2008.
[41] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: cold boot attacks on encryption keys," in *Proceedings of Usenix Security*, 2008.
[42] "Contagiodump collection," http://contagiodump.blogspot.com/, accessed in June 2013.
[43] "Add javascript to existing pdf files (python)," http://blog.rsmoorthy.net/2012/01/add-javascript-to-existing-pdf-files.html, accessed in June 2013.
[44] "Cve-2009-1492," http://www.cvedetails.com/cve/CVE-2009-1492/, accessed in August 2013.
[45] "The number of the beast," http://vinsula.com/cve-2013-0640-adobe-pdf-zero-day-malware/, 2013.
[46] "Making malicious pdf undetectable," http://www.signal11.eu/en/research/articles/malicious_pdf.html, 2009.
[47] "Another nasty trick in malicious pdf," https://blog.avast.com/2011/04/22/another-nasty-trick-in-malicious-pdf/, 2011.