Multi-Variant Execution to Protect Unpatched Software

Kevin Bauer, Veer Dedhia, Richard Skowyra, William Streilein, and Hamed Okhravi
MIT Lincoln Laboratory

***Abstract*—For a variety of economic and practical reasons, security patches often cannot be deployed immediately after a patch's release. To mitigate attacks against unpatched software, we present the design and evaluation of a Moving Target technique that uses a form of software diversity called multi-variant execution. Our technique decomposes the software's behavior into its low-level system calls and compares unpatched and patched execution traces to identify malicious behavior in the unpatched software. We evaluate our approach on benign and malicious document samples and our results indicate that multi-variant execution can detect real exploits with low false positives.**

I. Introduction

In today's complex cyber threat ecosystem, static and predictable defenses are insufficient to effectively counter determined cyber attackers. While traditional defenses such as signature-based anti-virus tools can be useful at detecting a significant fraction of malicious campaigns, a recent Microsoft Security Intelligence Report indicates that even when security patches and best-practice defense tools are used, adversaries continue to develop advanced attack techniques and infections still occur with startling frequency [1].

One important yet often overlooked reason why cyber attackers continue to have the strategic advantage is due to the delayed deployment of security-critical software patches. In response to vulnerability disclosures, software vendors typically release patches as quickly as possible in order to mitigate the potential damage caused by live exploits. However, there is a cost associated with patching a system, in terms of disruption of services (patching a system often requires a reboot) or the potential unreliability of the patch (patches often receive minimal testing). As a result, system administrators may not apply all security-relevant patches immediately [2]. This is especially relevant with respect to key high-availability services (e.g. authentication servers, DNS, web proxies), long-running tasks which cannot be easily check-pointed (e.g. high-performance computing workloads), and legacy systems which are not compatible with a new software version and cannot be upgraded without breaking other applications (e.g. old operating system distributions or browsers). In August 2014, for example, 16.37% of PCs were still running Windows XP, despite support for it ending in April 2014 [3]. Even when a system can be patched within a relatively short window, it has been shown that exploits can be generated quickly and automatically based on a newly released patch [4]. This unfortunate reality leaves systems vulnerable to attack through the relatively easy exploitation of known vulnerabilities.

We propose a novel Moving Target (MT) technique to provide resilience to services relying on unpatched systems during the window of vulnerability between the time a patch is made available and when it is deployed. Our approach uses dynamic software variants to compare the behavior of unpatched and patched software on document inputs in order to detect attacks in the wild before unpatched systems can be compromised.

Moving Target and Its Applications. In response to the constantly evolving threat landscape, *active defense* has been proposed as a way to restore the advantage to the defender. Such defenses may involve real-time threat detection in combination with a coordinated response that seeks to change the cyber environment in a way that thwarts a particular attack or forces the adversary to change tactics or target other systems. MT is a form of active defense that specifically seeks to introduce diversity, dynamism, or randomness into the computing environment in order to achieve attack resilience [5].

Numerous MT techniques have been proposed in the literature with a variety of security goals, including to *protect* systems [6]–[8], *detect* known and unknown threats [9]–[11], *react* to successful exploits [12], [13], and/or *deceive* the adversary [14]. In this paper, we design and evaluate a new MT technique based on multi-variant execution to detect exploits against unpatched systems for which a known security patch is available but not yet deployed on target systems. Our approach explores the general concept of MT in terms of software diversity, which is not as well covered by the literature as techniques that leverage randomization or uncertainty.

Multi-variant Execution for Detection. Our approach to malicious document detection applies a dynamic software-based MT technique known as *multi-variant execution* to compare the execution of two versions of a document viewer: an *unpatched* variant that contains a security-related software bug and a *patched* variant that fixes the vulnerability. Discrepancies in the execution traces of the two variants are used to expose and detect malicious documents. The use case for our technique is an enterprise network with numerous machines. When a patch becomes available for a software application used in the enterprise, it can be immediately applied to some of the machines (*e.g.* end-host workstations, but not others (*e.g.* high-availability servers or legacy systems). To protect the unpatched machines, the untrusted documents sent to them (for example, via email attachments) are forwarded to a dedicated, sandboxed scanner machine which runs both the patched and unpatched variants of the application. If enough discrepancies are observed between the two variants when opening the document, the document is marked as malicious and is quarantined. Otherwise, it is forwarded to the unpatched destination machine. Note that the scope of our work does not include detecting every malicious document; rather, we strive to detect the malicious documents that use the vulnerability fixed by the security patch in order to ensure that the unpatched machines are secure against those documents during the window of time before they are patched. This ensures that the enterprise network can remain resilient to attack even with vulnerable systems, at the cost of potentially not executing all documents sent to unpatched machines.

Evaluation and Results. We analyze the effectiveness of our

This work is sponsored by the Department of Defense under Air Force Contract #FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

978-1-4799-8594-4/15/$31.00 ©2015 IEEE

approach in detecting malicious Portable Document Format (PDF) files. We evaluate it in a Windows environment using a corpus of 500 benign documents and both a custom data set of synthetic malware and 50 live, publicly available malicious PDFs from the Open Malware repository. We target malicious PDF detection due to the inclusion of PDF file format exploits in popular exploit kits such as Black Hole, Phoenix, and others [15]. Furthermore, during the year 2013 malicious PDF documents were reportedly the most common type of file format exploit in the wild, according to Microsoft [1]. Our initial findings illustrate that naïvely comparing the execution traces is infeasible since the normal execution of the document viewer application, itself, results in many variant executions (for example, due to signal handling, threading, and unpredictable GUI events). This is contrary to the intuitive belief that opening the same benign document with the same variant of the application should result in similar execution traces. However, we show that by performing differential analysis on the set of Windows API calls, we can accurately detect all of the malicious documents with a low false positive rate.

II. Background and Related Work

Moving Target Techniques. MT techniques have been proposed as a way to re-balance the cyber environment in favor of defense. The goal is to make computer systems less static (a.k.a. dynamism), less deterministic (a.k.a. randomness) and less homogeneous (a.k.a. diversity) [16]. One study categorizes MT techniques into five large domains according to the layer of the software stack where the movement happens: dynamic networks, dynamic platforms, dynamic runtime environment, dynamic software, and dynamic data. Interested readers should see Okhravi *et al.* for a survey of MT research [17].

Multi-Variant Execution. Multi-variant execution techniques are a form of dynamic software MT techniques in which multiple diversified copies of a software application are executed, and their outputs are compared using a monitor. The focus in multi-variant techniques is on diversity, rather than randomness or dynamism, although some techniques in this area incorporate some of those aspects.

The most similar related work to our approach is Orchestra [11] which compares the behavior of two software variants to detect attacks. The variants are created using the Reversed Stack method. However, there are major differences with our approaches. First, Orchestra creates the variants for the software, thus requiring access to the source code. In our approach, the variants already exist (one being the unpatched version of the software and the other one the patched version), so it can be used to protect proprietary, closed-source applications. In fact, our analysis in this paper is on Adobe Reader which is a closed-source application. Second, the goal of Orchestra is different than ours. In Orchestra, the goal is to detect intrusion attempts against the software, while our goal is protection of unpatched machines. One major implication of this difference is that in Orchestra, the two variants have to run at all times, whereas in our approach, the two variants only run during the period when a patch is available for the software and before it is patched. Third, the comparison algorithms are also different. Orchestra relies on system call level synchronization, while our approach relies on differential set analysis of Windows API calls.

III. Approach and System Design

The modern IT enterprise is large and complex, both in terms of number and diversity of systems and users. In such an environment, deploying security-critical software patches to all affected machines in a timely manner is a challenging task. The goal of this work is to develop techniques and tools to protect unpatched systems during the vulnerability window between the release of the security-relevant patch by software vendors and the deployment of the patch throughout the enterprise.

Specifically, we consider the scenario in which a malicious data payload (e.g. a PDF file) is downloaded to an enterprise network. Prior to its execution on any unpatched system, the file is examined by DEA inside of a protected malware sandbox on a separate machine. Online interactions with unpatched systems (e.g. HTTP requests) are thus out of scope, as is execution of the malicious payload directly on the vulnerable machine.

Our approach to protecting unpatched software leverages a form of MT defense based on the concept of software diversity and multi-variant execution. Multi-variant execution has proven to be a powerful tool for detecting exploit attempts [11], [18]. In general, multi-variant execution approaches utilize two or more diversified software variants, where each variant may have a distinct memory layout, for example. By executing each variant and comparing the results of the executions, it may be possible to observe differences in execution among the variants and infer that an exploit has been attempted.

A. Multi-variant Execution for Detection

We propose an approach to multi-variant execution called *differential execution analysis* (DEA) that compares precisely two software variants and relies on a distance function to quantify how different the executions of the respective variants are. Using DEA, we develop an execution monitor that can be deployed within a complex enterprise network to detect attacks against unpatched systems.

Software Variants. With DEA, the software is executed in two distinct variants. First, the *unpatched* variant consists of an executable that contains a known software vulnerability for which a security patch is available but has not been applied. Second, the *patched* variant is the result of applying the security patch to the unpatched variant. Each variant is executed with the same inputs and the results of each variant's execution are compared to identify differences that may be the result of malicious behavior.

Relative to prior work on multi-variant execution [11], our work is the first that employs variants that execute slightly different code (i.e., the application of the patch). In fact, the patched variant could potentially contain code modifications beyond the minimum necessary to fix the vulnerability in question (features may be added or removed in addition to the security fix). This makes the task of comparing execution traces particularly challenging, as we cannot necessarily execute the two variants in lock-step.

Differential Execution Analysis. The behavior of an executing application can be expressed as the sequence of system calls that execute over the course of the application's run time. Prior work on multi-variant execution simultaneously executes each of n diversified variants in lock-step, using monitors to follow the execution of system calls between each variant [11]. We adopt a similar approach of monitoring and comparing each variant's system calls, but because the variant code is not consistent, it is not feasible to execute each variant in lock-step. Furthermore, detecting intrusions may not be as straightforward as identifying differences in execution traces, as different software versions may exhibit a certain amount of variability even when processing benign inputs (no intrusion).

We consider each variant's execution trace as a set of

system calls made during the execution.[1] These system calls could merely be the name of the function or the name of the function in addition to the argument values used in the call. (We evaluate both approaches in Section IV and also describe our approach to identifying useful argument values from those that have arbitrary values.) Let S_p and S_u be the set of system calls within the patched and unpatched execution traces, respectively. To quantify the difference in behaviors between the patched and unpatched executions, we apply the Jaccard distance metric J, as shown in Equation 1.

$$J = 1 - \frac{|S_p \cap S_u|}{|S_p \cup S_u|} \quad (1)$$

J varies from 0 to 1, where a value of 1 indicates that the two sets are disjoint and a value of 0 indicates that the sets are the same.

To identify an exploit in the unpatched variant, we apply a threshold on the distance metric J. If J is larger than a threshold τ, then we consider the differences between execution traces to be too large to have occurred under benign circumstances and, thus, the unpatched variant is assumed to be under attack.

While the Jaccard set distance captures differences in system calls between patched and unpatched variants, it may be the case that there exists some degree of variability between patched and unpatched execution traces that occurs even on benign inputs. In this case, the Jaccard set distance does not distinguish between variability that is normal and less common differences due to an attack. We introduce a *weighted* Jaccard distance metric to emphasize system calls in execution that have been observed less frequently or never (in a modest-size corpus of execution traces) and de-emphasize system calls that are more common. The weighted Jaccard distance J_w is defined in Equation 2.

$$J_w = 1 - \frac{\sum_{s \in S_p \cap S_u} w(s)}{\sum_{s \in S_p \cup S_u} w(s)} \quad (2)$$

where $w(s)$ is a function that weights the system calls according to frequency across a corpus of benign executions. (S_p and S_u are the same as those used in Equation 1.) This weighting function is the Inverse Document Frequency, which is commonly used in natural language processing for tasks such as document classification (defined in Equation 3).

$$w(s) = \log\left(\frac{N + 0.5}{n_s + 0.5}\right) \quad (3)$$

In the above, s is a system call, n_s is the number of times s occurs in the corpus, and N is the number of execution traces in the corpus. The corpus is a collection of execution traces for both the unpatched and patched variants processing benign inputs (e.g., training data).

B. Prototype Monitor Implementation

Our monitor prototype targets Windows-based exploits and is built on top of the open source Cuckoo Sandbox malware analysis framework [19]. Cuckoo sets up a virtualized environment for safe execution of potentially malicious payloads. These include executable binaries, file formats with executable content (*e.g.* Microsoft Office documents or PDF files), or inputs such as URLs.

[1]Due to nondeterminism arising from, *e.g.*, signal handling, treating an execution trace as a sequence of system calls introduces unnecessary variance between executions on the same input. In addition, each variant is running different code, so comparing executions in lock-step may not be possible. Thus, we treat execution traces as sets.

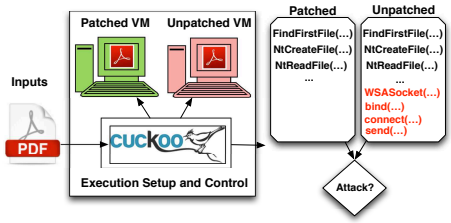


Fig. 1: A high-level view of the multi-variant execution-based monitor. Inputs are aggregated, and each variant execution is orchestrated by the Cuckoo analysis framework. After execution, Windows call traces are produced and analyzed. A hypothetical exploit attempt is shown in the unpatched trace (in red).

Our prototype execution monitor's system architecture is depicted in Figure 1. The prototype executes each variant within its own VirtualBox VM running Windows XP Service Pack 3. The prototype executes the variants by first restoring each variant's VM from a consistent snapshot, copying any necessary input data into the VM, and finally executing each variant. At variant execution time, a dynamic-link library (DLL) is injected to instrument all Windows API calls. This produces a trace of all Windows API calls, complete with function names and argument values.[2] The monitor prototype consists of 800 lines of source code and is written in Python.

The initial version of the monitor is geared toward the detection of exploits in the free and widely deployed Adobe Reader PDF rendering application. The unpatched variant is a version of Adobe Reader with a known security vulnerability; the patched variant is an updated version that contains a security patch to fix the vulnerability.

Beyond providing instrumentation of the execution, the Cuckoo framework also provides user emulation to drive the application interface (by automatically moving the mouse, clicking on dialog boxes, and driving the keyboard). Also, since malware may attempt to detect reverse-engineering and analysis environments (such as a honeypot or the Cuckoo Sandbox), measures are taken to hide the fact that the variant is executing within a virtualized system (for example, by eliminating common Windows registry entries associated with virtualization software).[3]

IV. Evaluation and Results

In order to demonstrate the efficacy of our approach to detecting attacks on unpatched software, we conduct experiments with both synthetic and real data sets. In this section, we describe the data sources used to support an empirical evaluation of exploit detection through differential execution analysis and present the experimental results.

A. Experiment Setup

Since our approach to malicious document detection uses a threshold on the differences in execution behavior as a detector, it is necessary to understand how the behavior (in terms of system call sets) varies between different versions of the Adobe Reader application across many input documents. In addition, in order to evaluate true detection of exploits, we need a representative sample of malicious PDF documents.

[2]Note that we use the terms "system call" and "Windows API call" interchangeably throughout this paper.

[3]We consider the problem of preventing the malware from detecting the VM/honeypot environment (and thereby attempting to evade analysis) to be out of scope for this work.

TABLE I: Adobe Reader version numbers used as "unpatched" and "patched" variants in our evaluation.

Experiment	Unpatched	Patched
1	9.0	9.1
2	9.1	9.3
3	9.3	9.3.3

TABLE II: Summary of the exploits, bugs, and software versions affected in our synthetic malware data set.

Exploit	Vulnerability	Version
Adobe Collab.getIcon()	Stack Overflow (CVE-2009-0927)	9.0
Adobe U3D	Array Overrun (CVE-2009-3953)	9.1
Adobe Bundled LibTIFF	Integer Overflow (CVE-2010-0188)	9.3

Unpatched and Patched Variants. To evaluate our proposed multi-variant execution scheme, we need to select versions of the Adobe Reader software for both the unpatched and patched variants. For the purposes of evaluation, we select versions of Adobe Reader that contain well-known vulnerabilities and exploits (these are considered "unpatched"). We also select a version of the application for each unpatched version that fixes a vulnerability. We select three pairs of version numbers that we evaluate within the unpatched and patched roles of our multi-variant execution scheme, which are given in Table I.

Benign Data Source. In order to quantify the differences in execution behavior between unpatched and patched versions of Adobe Reader on benign documents, we select PDF documents from the `Govdocs1` data set [20] as inputs.[4] This data set consists of freely available files (image formats, MS Office documents, PDFs, and text files) scraped from the .gov web domain. We randomly select 500 PDF documents to use as our benign samples.[5]

We also randomly select 75 documents to use to as a training corpus for the weighted Jaccard set distance metric.[6] The corpus of benign execution traces is constructed simply by rendering all 75 documents in each version of Adobe Reader. This produces an execution trace for each document, which is used to construct the weighting function from Equation 3.

Malware Data Sources. Obtaining real labeled malware traces is more challenging. In particular, in order to locate real malware samples that exploit a vulnerability within a specific version of Adobe Reader (e.g., the unpatched version), it is necessary to reverse-engineer the malware sample to learn precisely what vulnerability is being exploited. Similarly, we need to understand the exploited vulnerability in order to determine the correct patched version to use.

In order to evaluate our approach on malware samples with ground-truth vulnerability information, we first generate a data set of realistic yet synthetic PDF malware using the Metasploit framework [21]. Metasploit simplifies the process of developing exploits against a set of known vulnerabilities by providing tools for selecting a specific exploit and embedding a payload (the post-exploit behavior such as establishing a reverse shell to an attacker's host). We treat this as a *closed world* evaluation, where ground-truth is available and we can quantify detection performance. Our custom malware data set is summarized in Table II.

While we are aware of no publicly available malware repository that contains the necessary reverse-engineering information for our evaluation, we select 50 PDF samples from the Open Malware repository [22] for an *open world* evaluation. These malware samples have been scanned by anti-virus tools and produce a positive match to a malware signature. Although, these real samples do not come with ground-truth labels on the underlying vulnerability being exploited, the repository does maintain the VirusTotal signature that detects the malware and provides the exact date when the sample was added to the public database. In an effort to obtain samples that exploit the vulnerabilities in Adobe Reader 9.0–9.3, we select traces from the 2009–2010 time frame, which corresponds to the release of these versions of Adobe Reader.[7]

B. Results

We next turn our attention to evaluating the multi-variant execution technique. In particular, we first measure the expected amount of difference in the proposed execution trace comparison metrics between unpatched and patched variants under benign inputs. Next, we use the characteristic differences to optimally tune a threshold parameter that detects malware with a high true detection rate and low false alarms.

Parameter Tuning. Before we attempt to detect malicious documents, we first characterize the expected differences in execution traces between the unpatched and patched variants. Using the 500 benign PDF documents described in Section IV-A, each document is rendered in both unpatched and patched variants, and the execution traces are compared using three distance metrics.

Figure 2a shows the distribution of Jaccard distances on sets of Windows API function call names (no arguments). For each pair of variants, the expected difference in call names is small, particularly so for the comparison between Adobe Reader 9.3 and 9.3.3 (a minor version difference).

Figure 2b shows the impact of computing the Jaccard metric using both Windows API names and argument values. Including argument values may help to identify function calls that are common to both variants, but with different argument values. We observe that several Windows API functions take arguments whose values may not be consistent between each variant. For example, these arguments include pointers, file/socket handles, process/thread IDs, graphical-user window parameters, and others. We exhaustively enumerate and ignore these argument values that are unlikely to have consistent values between variants. Despite this, we see that there is greater variance in the set distances for Experiments 1 and 2. (Experiment 3 traces still exhibit little variability.)

Figure 2c shows the distribution of the weighted Jaccard set distances for each pair of variants. We note that half of all distances are close to 0 relative to the training sets. We conjecture that malware traces may utilize system calls that are rare or never occur in the corpus, which would result in a high weighted distance.

Given the expected execution differences between the unpatched/patched variants under benign inputs, we can derive a threshold value τ for each pair of variants and each distance metric for an expected, fixed false positive rate. For example, for the variant pair from Experiment 3 using the Jaccard set distance on API call names, the threshold is $\tau = 0.22$. We derive thresholds for each variant pair and distance metric, which we apply next to detect malicious documents and evaluate the false positive rates.

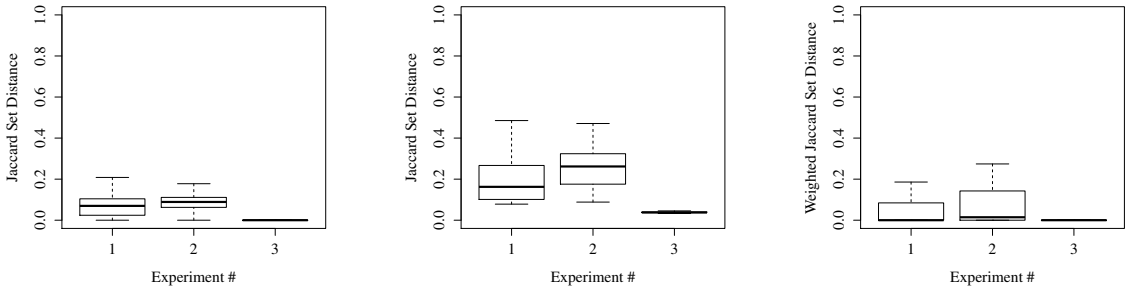
Detecting Synthetic Malware. We next evaluate our multi-variant execution technique on a set of custom malware

[4]Freely available at http://digitalcorpora.org/corpora/files.

[5]Note that some of the samples within the `Govdocs1` data set are known to contain malware (according to an anti-virus scan). While we deliberately reject the samples that are detected to maintain a clean data set, there may still be malicious samples that cannot be detected by AV tools.

[6]Note that the training corpus documents are disjoint from the 500 documents used for the evaluation.

[7]Note, however, that even though we select 50 samples, some may be false positives while others may not impact the versions of Adobe Reader considered in this paper.

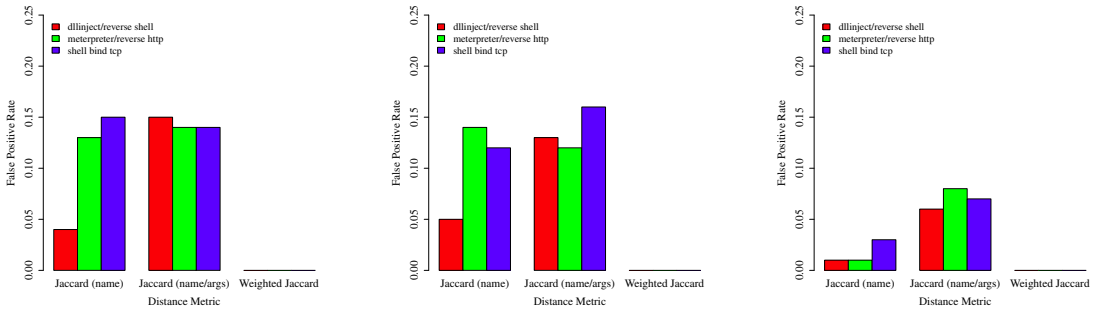


(a) Jaccard distance on Windows API function names.

(b) Jaccard distance on Windows API function names and argument values

(c) Weighted Jaccard distance on Windows API function names

Fig. 2: The distribution of set distances for three distance metrics for the three multi-variant experiments listed in Table I.



(a) Adobe Reader 9.0 Exploit

(b) Adobe Reader 9.1 Exploit

(c) Adobe Reader 9.3 Exploit

Fig. 3: False positive rates for each exploit and distance metric to a achieve true detection.

generated by Metasploit. This can be regarded as a closed world evaluation, where we have ground-truth knowledge of the underlying vulnerability, exploit, and post-exploit payload. For each exploit summarized in Table II, we generate malicious PDF documents with three Metasploit payloads, each of which attempts to achieve persistence and establish a control channel back to the attacker's remote host.

The payloads we use for evaluation range in complexity from simple and easy to detect to complex and stealthy. The payloads are selected to provide a useful set of post-exploit behaviors that an attacker likely would need to execute in order to compromise a target host and achieve a primary objective such as data theft or bringing the compromised host into a botnet or spam campaign. The malware samples in our data set are constructed by combining each exploit with each of three post-exploit payloads from Metasploit:

1) `shell_bind_tcp`: a relatively simple shellcode that binds a socket to a command shell and listens for connections from a remote attacker;
2) `meterpreter/reverse_http`: a more sophisticated payload that embeds the "meterpreter" attack framework binary into the malicious PDF and establishes a reverse command shell connection to an attacker's remote host over HTTP; and
3) `dllinject/reverse_shell`: a reverse shell payload that uses DLL injection, which allows the exploit to execute a payload without ever touching the compromised host's hard disk.

We execute both variants and report the false positive rates necessary to detect the malware, for each distance metric.

Figure 3a shows the false positive rates for detecting instances of the Adobe Collab.getIcon() exploit with each payload across the three proposed distance metrics. For the standard Jaccard set distance metrics that use function names and names plus arguments, the false positive rates are relatively high for all payloads (0.04–0.15). However, the weighted Jaccard distance metric achieves a false positive rate of 0.0 while detecting all of the true malware samples, which is optimal.

The false positive rates for detecting the Adobe U3D exploits are shown in Figure 3b. Similar to the Adobe Reader 9.0 exploit, the false positive rates are relatively high for the non-weighted Jaccard distance metrics, regardless of the payload used. The weighted Jaccard distance still achieves a 0.0 false positive rate while detecting the malware samples.

Figure 3c shows the false positive rates for detecting the Adobe Bundled LibTIFF exploits. Interestingly, the false positive rates for the non-weighted Jaccard distances are relatively low compared to the other two experiments (0.01–0.03 for the Jaccard distance on function names and 0.06–0.08 for Jaccard distance on function names and argument values). This higher performance relative to the other two experiments is likely due to the fact that the patched and unpatched variants are very close in version number (9.3 and 9.3.3), and likely have relatively fewer code differences than the variant pairs used in the other two experiments. This demonstrates the importance of choosing a patched variant that consists of the minimal code changes needed to fix the vulnerability present in the unpatched variant (when possible). In addition, the weighted Jaccard distance achieves a 0.0 false positive rate while detecting all

instances of the exploit.

Detecting Real Malware. We lastly turn our attention to detecting real, live malware traces using multi-variant execution. In contrast to the previous results where we had complete knowledge of the underlying software vulnerabilities, exploit techniques, post-exploit payloads, and precise set of software versions affected, we now evaluate our techniques in an open world setting. As such, we have little ground-truth knowledge about the vulnerabilities exploited in the samples, and in particular, we do not know exactly which version of Adobe Reader is "unpatched" (e.g., vulnerable to the exploit) and which is "patched" (e.g., fixes the vulnerability). Despite these challenges, we attempt to discover which malware samples affect the versions of Adobe Reader we use in our evaluation using publicly available malware samples (as described in Section IV-A).

We run all 50 suspected malware samples through each of the three multi-variant execution pairs. Due to its high true detection performance with a low false positive rate, we use the weighted Jaccard set distance to identify potential exploit attempts in Adobe Reader versions 9.0, 9.1, and 9.3. We then manually examine the traces with the highest distance scores between variant executions in order to find empirical evidence of an exploit attempt. This allows us to validate the detection as a likely true positive or a possible false positive.

In Adobe Reader 9.0, the multi-variant execution technique identifies seven samples with a weighted Jaccard distance score higher than any observed in the baseline comparison between Adobe Reader 9.0 and 9.1 rendering benign inputs. Of these samples, on manual inspection of the execution traces for version 9.0, six contain Windows API calls that exist neither in version 9.1 nor in the baseline of executions with benign documents. The previously unobserved Windows API calls are weighted highly, which contributes to the high score for the execution comparisons. The functions in question are primarily related to opening network sockets and establishing communication with a remote host: `WSAStartup`, `socket`, `select`, `connect`, `getaddrinfo`, and `CreateRemoteThread`. We also noted several instances where executable files are written to disk for automatic execution at boot-time (for example, `C:\autoexec.bat`). Since these six samples contain unusual system calls similar to those in our synthetic malware corpus in addition to other unusual and suspicious behavior, we conclude that our technique detects six likely exploit attempts and one potential false positive.

For Adobe Reader versions 9.1 and 9.3, we detect five and three malware samples, respectively. For these versions, all of the detected sample traces also contain function calls related to establishing communication with remote hosts. Thus, we conclude that our technique is adequate for detecting exploit attempts that deviate from the expected behavior of the application. We leave it as future work to expand our evaluation to investigate and characterize the wide range of malicious behaviors that may be harder to distinguish from legitimate behavior.

V. Conclusion

We propose an MT-based approach to protect unpatched software applications against attacks that target the vulnerabilities fixed by an already released patch. Our approach relies on running two variants of the software (patched and unpatched) on a monitor and performing differential set analysis of execution traces. Our technique, called differential execution analysis (DEA), is appropriate for protecting machines in situations where the machine cannot be immediately patched in order to avoid breaking its functionality. The goal of our MT technique is exposure and detection of malicious documents.

References

[1] "Microsoft security intelligence report," www.microsoft.com/security/sir, 2014.

[2] S. Beattie, S. Arnold, C. Cowan, P. Wagle, and C. Wright, "Timing the application of security patches for optimal uptime," 2002, pp. 233–242.

[3] K. Lab. (2014) Kaspersky security network report: Windows usage and vulnerabilities. Kaspersky Lab.

[4] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, 2008.

[5] H. Okhravi, T. Hobson, D. Bigelow, and W. Streilein, "Finding focus in the blur of moving-target techniques," *IEEE Security & Privacy*, vol. 12, no. 2, pp. 16–26, Mar 2014.

[6] H. Okhravi, A. Comella, E. Robinson, and J. Haines, "Creating a cyber moving target for critical infrastructure applications using platform diversity," *Elsevier International Journal of Critical Infrastructure Protection*, vol. 5, pp. 30–39, Mar 2012.

[7] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (aslp): Towards fine-grained randomization of commodity software," in *Proc. of ACSAC'06*. Ieee, 2006, pp. 339–348.

[8] Z. Liang, B. Liang, L. Li, W. Chen, Q. Kang, and Y. Gu, "Against code injection with system call randomization," in *Networks Security, Wireless Communications and Trusted Computing, 2009. NSWCTC '09. International Conference on*, vol. 1, April 2009, pp. 584–587.

[9] T. Jackson, B. Salamat, G. Wagner, C. Wimmer, and M. Franz, "On the effectiveness of multi-variant program execution for vulnerability detection and prevention," in *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, ser. MetriSec '10, 2010, pp. 7:1–7:8.

[10] T. Jackson, C. Wimmer, and M. Franz, "Multi-variant program execution for vulnerability detection and analysis," in *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, ser. CSIIRW '10. New York, NY, USA: ACM, 2010, pp. 38:1–38:4.

[11] B. Salamat, T. Jackson, A. Gal, and M. Franz, "Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space," in *Proceedings of the 4th ACM European Conference on Computer Systems*, ser. EuroSys '09. New York, NY, USA: ACM, 2009, pp. 33–46.

[12] P. Sousa, A. N. Bessani, M. Correia, N. F. Neves, and P. Verissimo, "Resilient intrusion tolerance through proactive and reactive recovery," in *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing*, ser. PRDC '07, 2007, pp. 373–380.

[13] S. Sidiroglou, O. Laadan, A. D. Keromytis, and J. Nieh, "Using rescue points to navigate software recovery," in *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, ser. SP '07, 2007, pp. 273–280.

[14] D. Kewley, J. Lowry, R. Fink, and M. Dean, "Dynamic approaches to thwart adversary intelligence gathering," in *Proceedings of the DARPA Information Survivability Conference and Exposition II*, 2001.

[15] V. Kotov and F. Massacci, "Anatomy of exploit kits: Preliminary analysis of exploit kits as software artefacts," in *Proceedings of the 5th International Conference on Engineering Secure Software and Systems*, ser. ESSoS'13. Springer-Verlag, 2013, pp. 181–196.

[16] "Cybersecurity game-change research & development recommendations," NITRD CSIA IWG, Strategic Context, May 2010.

[17] H. Okhravi, M. Rabe, T. Mayberry, W. Leonard, T. Hobson, D. Bigelow, and W. Streilein, "Survey of cyber moving targets," *Massachusetts Institute of Technology Lincoln Laboratory Technical Report 1166*, September 2013.

[18] E. D. Berger and B. G. Zorn, "Diehard: Probabilistic memory safety for unsafe languages," in *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '06, 2006, pp. 158–168.

[19] "Automated malware analysis – cuckoo sandbox," http://www.cuckoosandbox.org.

[20] S. Garfinkel, P. Farrell, V. Roussev, and G. Dinolt, "Bringing science to digital forensics with standardized forensic corpora," *Digit. Investig.*, vol. 6, pp. S2–S11, Sep. 2009.

[21] Rapid7, "The metasploit framework," http://www.metasploit.com.

[22] "Open malware," http://oc.gtisc.gatech.edu:8080.