Comprehensive Analysis and Detection of Flash-based Malware

Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck

Institute of System Security, TU Braunschweig, Germany

Abstract. Adobe Flash is a popular platform for providing dynamic and multimedia content on web pages. Despite being declared dead for years, Flash is still deployed on millions of devices. Unfortunately, the Adobe Flash Player increasingly suffers from vulnerabilities, and attacks using Flash-based malware regularly put users at risk of being remotely attacked—most prominently highlighted by numerous exploits made public during the past months. As a remedy, we present GORDON, a method for the comprehensive analysis and detection of Flash-based malware. By analyzing Flash animations at different levels during the interpreter's loading and execution process, our method is able to spot attacks against the Flash Player as well as malicious functionality embedded in ActionScript code. To achieve this goal, GORDON combines a *structural analysis* of the container format with *guided execution* of the contained code, a novel analysis strategy that manipulates the control flow to maximize the coverage of indicative code regions. In an empirical evaluation with 26,600 Flash samples collected over 12 consecutive weeks, GORDON significantly outperforms related approaches when applied to samples shortly after their first occurrence in the wild, demonstrating its ability to provide timely protection for end users.

Keywords: Adobe Flash, Malware, Classification

1 Introduction

Adobe Flash is a widespread platform for providing multimedia content on web pages—despite being declared dead for years and the recent standardization of HTML5. According to Adobe, the Flash Player is still deployed on over 500 million devices across different hardware platforms, covering a large fraction of all desktop systems [42]. Furthermore, a significant number of web sites employs Flash for advertising, video streaming and gaming, such that every *fourth* web site in the top 1,000 Alexa ranking still makes use of Flash-based content [22].

Unfortunately, the implementation of Flash is continuously suffering from security problems. During the last ten years over 690 different vulnerabilities have been discovered in the Adobe Flash Player [32]. In the year 2015 alone, 314 new vulnerabilities have been made public, 268 of which enable remote code execution and require a user to merely visit a web page to be infected. This growing attack surface provides a perfect ground for miscreants and has lead to a large variety of Flash-based malware in the wild.

Three factors render the Flash platform particularly attractive for attackers: First, the large number of vulnerabilities considerably increases the chances for compromising a wide range of systems. Second, the ability to execute ActionScript code as part of an attack allows to probe the target environment and carry out sophisticated exploit strategies. Finally, the Flash platform provides several means for obstructing the analysis of attacks—most notably the capability to execute downloaded or dynamically assembled code. As a result of such obfuscation, the analysis of Flash-based attacks is difficult and time-consuming. Often, signatures for virus scanners are only available with notable delay such that end users remain unprotected for a considerable period of time.

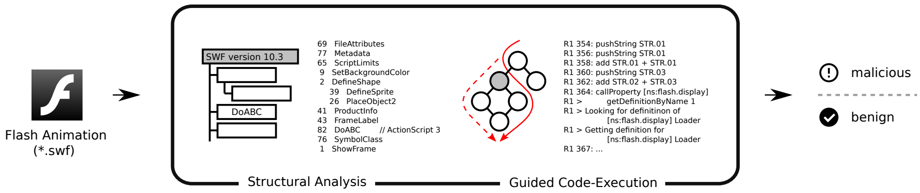
In this paper, we present GORDON, a method for the automatic analysis and detection of Flash-based malware. Our method combines a *structural analysis* of the Flash container format with *guided execution* of ActionScript code, a lightweight and pragmatic form of multi-path exploration. While related approaches orient analysis to normal execution [27, 44, 48] or external triggers [6, 14, 30], GORDON actively guides the analyzer towards interesting code regions to maximize the coverage thereof. This equips us with a comprehensive view on a sample, including downloaded and dynamically assembled code. By additionally inspecting the container format, we are able to construct a detection method capable of spotting malicious ActionScript code as well as exploits targeting the Flash Player directly.

To cope with the large diversity of Flash files in practice, GORDON implements support for all versions of Flash animations, including all versions of ActionScript code. To the best of our knowledge, we are the first to provide a generic method for the analysis and detection of Flash-based malware that enables a comprehensive view on the behavior and structure of a Flash animation across all versions. The efficacy of GORDON in practice is demonstrated in an evaluation with 26,600 Flash samples collected over a time of 12 consecutive weeks. GORDON detects 90% of the malicious samples shortly after their appearance in the wild with a false-positive rate of at most 0.1%. Consequently, our method provides an excellent starting point for fending off Flash-based malware more efficiently.

In summary we make the following contributions:

- **Guided code-execution.** We propose a lightweight and pragmatic approach for exploring ActionScript code in Flash-based malware that guides analysis towards large or otherwise characteristic code regions automatically.
- **Comprehensive analysis of Flash.** With the combination of a *structural analysis* of Flash containers and a *guided execution* of embedded code we provide a fine-grained view on samples across all versions of ActionScript code and Flash.
- **Effective detection of Flash-based attacks.** Based on this analysis, we develop a detection method that accurately identifies Flash-based exploits and malware shortly after their occurrence, providing a good starting point to bootstrap signature-based approaches.

The rest of the paper is structured as follows: In Section 2 we introduce GORDON, our method for the analysis and detection of Flash-based malware, followed by a detailed description of the employed structural analysis in Section 3, our guided code-execution in Section 4 and GORDON's detector in Section 5. Our evaluation is presented in Section 6. We discuss limitations and related work in Section 7 and Section 8, respectively. Section 9 concludes the paper.



```
69   FileAttributes
77   Metadata
65   ScriptLimits
 9   SetBackgroundColor
 2   DefineShape
   39   DefineSprite
   26   PlaceObject2
41   ProductInfo
43   FrameLabel
82   DoABC        // ActionScript 3
76   SymbolClass
 1   ShowFrame
```

```
R1 354: pushString STR.01
R1 356: pushString STR.01
R1 358: add STR.01 + STR.01
R1 360: pushString STR.03
R1 362: add STR.02 + STR.03
R1 364: callProperty [ns:flash.display]
R1 >          getDefinitionByName 1
R1 > Looking for definitinon of
              [ns:flash.display] Loader
R1 > Getting definition for
              [ns:flash.display] Loader
R1 367: ...
```

Fig. 1. Schematic depiction of the analysis and detection process of GORDON with a Flash-based malware as input, the two-step analysis of the profiler and the classification of our method's detector as output.

2 System Overview

The diverse nature of attacks based on Flash requires an analysis method to inspect these animations on different levels. To this end, we implement our method GORDON by integrating it into different processing stages of two Flash interpreters, thereby blending into existing loading and execution processes. This allows us to make use of data generated directly during execution, such as dynamically constructed code or downloaded files. We achieve this analysis using the following two-step procedure (see Figure 1): First, we instrument the processing unit of the Flash interpreter in order to profile a malware's structure as well as the execution of contained code. Second, we combine these profiles into a common representation to power a classifier based on machine learning techniques, that allows to effectively discriminate malicious from benign Flash animations.

Profiling the Malware. GORDON's profiler is implemented on the basis of two popular and mature open-source implementations of the Flash platform that are complementary with respect to the versions they support: *Gnash* [20] and *Lightspark* [34]. While Gnash provides support for Flash up to version 9, Lightspark enables processing version 9 and higher. As a result, GORDON is able to analyze all currently relevant versions and file formats of Adobe Flash animations, including all versions of ActionScript code. The profiling implemented for both interpreters features two kinds of analyses, that in turn make use of data arising during an interpreter's regular loading and execution process [1]:

- First, the profiler of GORDON inspects the hierarchical composition of the Shockwave Flash (SWF) format. This can be done during the loading phase when the interpreter parses the file for further processing (Section 3).
- Second, the control flow of embedded ActionScript code is analyzed in order to determine *indicative regions*. By strategically changing the control flow at branches in the code, GORDON guides execution along paths covering as much indicative regions as possible (Section 4).

Detecting Flash-based Malware. Based on the output of these different analyses, we are then able to decide whether a particular Flash animation is malicious or not. To this end we translate the structural report of a file and the execution trace of contained ActionScript code into a representation that allows to train a machine learning classifier (Section 5).

3 Structural Analysis

We begin our analysis by breaking down Flash animations into *tags*, the primary containers employed by the SWF file format [2] to store ActionScript code as well as data of various kinds, including audio, video, image and font data. Due to the large number of different types of tags Flash files expose a huge attack surface for memory corruption exploits. As a consequence, many exploits rely on very specific types and arrangements of tags to succeed, and thus, the sequence of tags alone can already serve as a strong indicator for malware.

For the structural analysis as employed by GORDON only tag identifiers and structural dependencies are of interest, contained data on the other hand is not considered. Consequently, GORDON does not need to know about the format of individual tags and hence can be applied to unknown tags, e.g., tags introduced in future versions. However, to further enhance the overall detection our method may be combined with approaches to specifically target data formats that can be included in a Flash animation's tags. Moreover, exploits often rely on corrupt or incomplete tags. To better account for these, we additionally include two specific tag identifiers that mark (a) *incomplete* tags, i.e. tags that are known to the interpreter, but could not be correctly parsed and (b) tags that contain *additional data* beyond their specified limits. The latter occurs, for instance, whenever the file contains data at the end that is not fully contained in its last tag.

As a result of this structural analysis, we obtain a sequence of container types including their nestings for each Flash file. Figure 2 shows the resulting container listing for a sample[1] of the LadyBoyle malware using `CVE-2015-323`.

```
69   FileAttributes
77   Metadata
 9   SetBackgroundColor
 2   DefineShape
     39   DefineSprite
     26   PlaceObject2
86   DefineSceneAndFrameLabelData
43   FrameLabel
87   DefineBinaryData  // Payload
87   DefineBinaryData  // Payload
82   DoABC             // ActionScript 3
76   SymbolClass
 1   ShowFrame
```

Fig. 2. Excerpt of the structural report for a LadyBoyle malware sample[1].

[1] md5: `cac794adea27aa54f2e5ac3151050845`

In comparison to many other Flash animations, the content of this file is rather short. However, for this specific sample the presence of the `DefineBinaryData` and `DoABC` tags is crucial. The first contain the malware's payload as binary data, which in turn gets extracted by ActionScript 3 code embedded in the latter. These tags in combination comprise the malicious functionality of the sample. While in this particular case the structure alone is only an indicator for the malicious behavior, that needs to be backed up by an analysis of the embedded ActionScript code, other types of malware rely on corrupt tags that allow to distinctively distinguish these. Some containers, such as the `DefineShape` tag, allow to enclose an arbitrary number of other containers. We include these in the listing as children of the parent tag. Note that the `DefineShape` tag and its children are not present in the original sample and have been added for illustration purposes only.

For convenience, the structural report can also be represented as a sequential list of identifiers, where nested containers are indicated by brackets:

```
69 77 9 2 [ 39 26 ] 86 43 87 87 82 76 1
```

It is important to note, that this representation already encodes the complete hierarchy and relations of the tags to each other. This condensed form is particularly suitable for automated approaches that do not require a textual description of the tags. We revisit this topic when discussing the implementation of Gordon's detector in Section 5.

4 Guided Code-Execution

When analyzing a sample with Gordon we aim at observing as much indicative behavior of a Flash animation as possible—ideally the analysis covers all possible execution paths and corner cases. However, as extensively discussed in computer security literature in the past [e.g., 27, 30] this is not feasible due to the potentially exponential number of different paths, making it necessary to revert to approximations and heuristics in practice.

While related approaches orient analysis to normal execution [27, 44, 48] or external triggers [6, 14, 30], our method guides execution towards *indicative code regions*: Each branch is chosen such that the execution corresponds to the path that covers the most indicative ActionScript code not observed so far. In particular, we are interested in exploring paths containing security-related objects and functions as well as branches that contain more code than others. Figure 3 exemplarily shows the selected paths of two consecutive runs. During the first, Gordon's profiler guides execution towards the `loadMovie` function, which enables Flash animations using ActionScript 2 to dynamically load code in form of another SWF file. The second run then directs the interpreter along the path covering the most bytecode instructions. This strategy can hence be seen as a way to not only maximize code coverage locally (within the sample itself), but globally, including all code that is loaded dynamically.

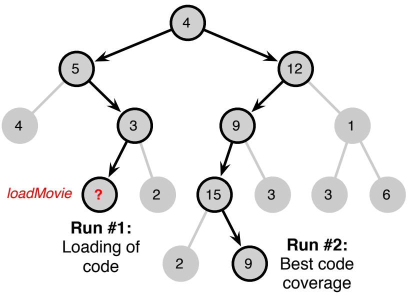


Fig. 3. Illustration of the path-selection strategy. Node labels correspond to the amount of bytecode instructions in each basic block. Black lines indicate chosen execution paths.

This is made possible by inspecting the control flow of the ActionScript code contained in a Flash file with the aim of learning a) how much code can be covered along a specific path and b) where security-related objects and functions such as the aforementioned `loadMovie` are located. To this end, we first derive the control-flow graph (CFG) of the ActionScript bytecode in question and remove cycles induced by loops and recursive function calls (Section 4.1). Second, the resulting graph is annotated with locations of indicative functionality and the number of instructions contained in each branch, which in turn enables us to efficiently determine the overall code coverage of individual paths (Section 4.2). The results of this analysis is then used for the actual execution of the Flash animation, allowing Gordon to navigate through the code in a targeted way (Section 4.3).

4.1 Control-Flow Analysis

A control-flow graph (CFG) as shown in Figure 4 contains basic code blocks as its nodes and directed edges for branches connecting them [see 3]. As part of the Adobe Flash Player's verification phase, the ActionScript VM already checks certain control flow properties when bytecode is loaded into the interpreter [1]. Our control-flow analysis can thus be thought of as a natural extension to the examinations conducted by Flash interpreters. We, however, make use of this information only as a starting point for the following analysis.

Upon the generation of a CFG, we are ready to find execution paths that maximize code coverage. To easily determine these paths, the graph needs to first undergo a few modifications. In particular, it is necessary to eliminate cycles that occur due to loop statements in the code. Once these cycles are removed we obtain an *acyclic control-flow graph (ACFG)* which allows us to efficiently determine the code size of complete paths in the graph. To this end, we rewrite all *back-edges* (edges pointing backwards with respect to the control flow) by linking them to the first code block after the loop. Figure 4 demonstrates this for a simple while loop. All conventional loop, nested loops and their special cases such as *unnatural loops* can be efficiently resolved using the *dominance* relations of the individual nodes [see 3].

```
var a:int = 294;
var b:int = 1722;

while (b != 0)
{
  var h:int = a % b;
  a = b; b = h;
}
trace(a);
```

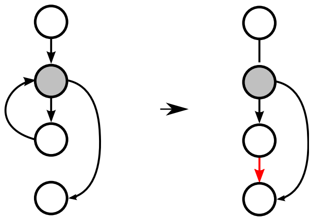


Fig. 4. An ActionScript 3 snippet, the corresponding control flow graph (CFG) and its acyclic transformation (ACFG). Dark nodes represent loop headers, bright nodes generic code blocks; newly inserted edges are shown in red.

4.2 Annotating Control-Flow Edges

Once an ACFG has been generated, we annotate each of its edges with the number of bytecode instructions covered by the following code block. We artificially increase the weight of individual instructions, if they correspond to security-related objects and functions. For example, to pinpoint the dynamic loading of code, we set the weighting for calls to the `loadMovie` function (ActionScript 2) and the `Loader` object (ActionScript 3) to the maximum to ensure the analyzer targets these first. Both are frequently used by Flash-based malware to load code downloaded from the Internet or dynamically assembled at runtime. Similarly, it is possible to emphasize other security-related functions and objects in ActionScript, such as `readBytes` and `ByteArray` which are often used for obfuscated code.

Given the annotated graph, the search for the most indicative code regions can be rephrased as a *longest-path problem*. For arbitrary graphs determining the longest path is NP-hard. Fortunately, for *directed acylic graphs* such as the ACFG extracted previously, this is possible [see 11, 38].

4.3 Path Exploration

With the annotated ACFG at hand, we can now guide the interpreter to execute security-related or large code regions by stopping at every conditional jump and choosing the branch corresponding to the path with the highest weight. In order to avoid executing indicative code unnecessarily often, we constantly update visited regions within the ACFG. Moreover, Gordon enables multiple executions based on the coverage analysis of previous runs. Hence, a different path is taken and different code regions are visited in each run, thereby challenging adversarial attempts to hide payload in paths not covered initially. As analysis output of the guided execution, we obtain all covered ActionScript instructions across multiple execution runs. Figure 5 shows a short excerpt of the instructions executed by a malware to facilitate the `CVE-2015-03-313` exploit^2 in the first run (R1).

Instructions at offset 973 to 983 show how the malware obfuscates the usage of the `ByteArray` object at offset 984. This object is frequently used to construct malicious payloads at run-time. The complete listing shows how the encrypted payload is composed out of different parts, decrypted and finally loaded.

```
R1 973:  pushString   "fla"
R1 975:  pushString   "sh.uti"
R1 977:  add          "fla" + "sh.uti"
R1 978:  pushString   "ls.Byt"
R1 980:  add          "flash.uti" + "ls.Byt"
R1 981:  pushString   "eArray"
R1 983:  add          "flash.utils.Byt" + "eArray"
R1 984:  callProperty [ns:flash.utils] getDefinitionByName 1
R1 >     Looking for definition of [ns:flash.utils] ByteArray
R1 >     Getting definition for [ns:flash.utils] ByteArray
R1 987:  getLex: [ns:] Class
```

Fig. 5. Excerpt of behavioral report[2].

In the following we address certain implementation details of GORDON's guided code-execution with a special focus on the characteristics of Flash-based malware and potential adversarial attempts to avoid analysis.

Reducing Branch Candidates. Although GORDON is capable of pursuing all branches in ActionScript code, narrowing down the candidates speeds up the process and limits the possibility of breaking the semantics of a sample. Often, web-based attacks are tailored towards specific browser environments and thus only trigger malicious activity upon checking for the correct target environment [27, 44]. The conditional jumps underlying these checks provide excellent candidates for our guided execution, as they usually lead to a malware sample's payload and are likely to be mutually exclusive, therefore reducing the risk of semantic side-effects.

To restrict our analysis to these conditional jumps, we implement a taint-tracking mechanism that propagates taint from environment-identifying data sources to conditional jumps. In the scope of Flash-based malware, such data typically originates from the `System.capabilities` and `flash.system.Capabilities` data structures available in ActionScript 2 and 3, respectively. To track the data flow across built-in functions, we conservatively taint the result whenever at least one of the input arguments is tainted. Note that for simplicity, we do not consider implicit data-flow and control dependencies in our implementation [see 8, 31] but leave this for future work.

Countering Obfuscation. To account for dynamically loaded code, we additionally hook the interpreter's loading routines. All such code then passes through the same analysis steps as the host file, allowing to analyze files downloaded from the Internet as well as potentially encrypted code embedded in the Flash animation itself equally thoroughly. This scheme is applied recursively to ensure that all code is covered by our analysis.

Furthermore, GORDON implements an *adaptive timeout* mechanism rather than a fixed period of time as utilized in previous works [13, 19, 44]. In particular, we reset a 10s timer each time the sample attempts to load code, giving the sample time to react to this event. This may increase the analysis duration for certain files but significantly reduces the effort for those that do not load data or do not contain ActionScript code at all. On average a sample is executed for 12.6 seconds with a maximum duration of 3 minutes, reducing the analysis time by 93% compared to a fixed timeout.

[2] md5: `4f293f0bda8f851525f28466882125b7`

We also take precautions for the possibility that an execution path is not present in the statically extracted ACFG. In these rare cases, we switch to determining the size of the branch in an online manner: Gordon looks ahead in order to inspect the instructions right after the branching point and passively skips over instructions to determine the sizes of the branches. This analysis in principle is the same as performed earlier (Section 4.1) but applied to the newly discovered piece of bytecode only.

Lastly, we have observed an increase in the use of event-based programming in recent malware—presumably to circumvent automatic detection—and thus incorporate the automatic execution of such events into Gordon's profiler. Immediately after an event listener is added the specified function gets passed an appropriate dummy event object and is executed without waiting for the actual event to happen.

Updating the ACFG. Our method is designed to run a sample multiple times. To this end, we update the edge labels of the ACFG during execution to reflect the visited code and recompute the largest path in an online manner. Consequently, our method implements a lightweight variant of multi-path exploration that executes different code during each run. Since we decide on each condition at runtime and identical code regions (functions) may be referenced multiple times we not only cover the code of the single largest path in the graph but potentially a combination of a number of paths. This softens the definition of such a path as used in graph theory but makes a lot of sense for this application especially.

5 Learning-based Detection

In order to demonstrate the expressiveness of our analysis, we implement a learning-based detector that is trained on known benign and malicious Flash animations. This approach spares us from manually constructing detection rules, yet it requires a comprehensive dataset for training (see Section 6.1). However, as most learning algorithms operate on vectorial data, we first need to map the analysis output of Gordon to a vector space.

Vector Space Embedding. To embed the structural and behavioral reports generated by Gordon in a vector space, we make use of classic *n-gram models.* These models have initially been proposed for natural language processing [9, 41] but are also used in computer security for analyzing sequential data [e.g., 24, 28, 33, 39, 47].

In particular, we extract *token n-grams* from both kinds of analysis outputs by moving a sliding window of length n over the tokens in the reports. While the compact output representation of Gordon's structural analysis already is in a format that can be used to extract such tokens, the reports generated by the guided code-execution need to be normalized first: We extract all instructions, including their names and parameters. Moreover, we replace values passed as parameters with their respective type, such as `INT`, `FLOAT` or `STR`. To avoid loosing relevant information we however preserve all names of operations, functions and objects. Finally, we tokenize the behavioral reports using white-space characters.

High-order n-grams compactly describe the content, implicitly reflect the structure of the reports and can be used for establishing a joint map to a vector space. To this end, we embed a Flash animation x in a binary vector space $\{0,1\}^{|S|}$ spanned by the set S of all observed n-grams in the analysis output. Each dimension in this vector space is associated with the presence of one n-gram $s \in S$. Formally, this mapping ϕ is given by

$$\phi : x \longmapsto \big(b(s,x) \big)_{s \in S}$$

where the function $b(s,x)$ returns 1 if the n-gram s is present in the analysis output of the file x and 0 otherwise.

Classification. Based on this vector space embedding, we apply a *linear Support Vector Machine* (SVM) for learning a classification between benign and malicious Flash animations. While several other learning algorithms could also be applied in this setting, we stick to linear SVMs for their excellent generalization capability and very low run-time complexity, which is linear in the number of objects and features [37].

In short, a linear SVM learns a hyperplane that separates two classes with maximum margin—in our setting corresponding to vectors of benign Flash animations and Flash-based malware. The orientation of the hyperplane is expressed as a normal vector w in the input space and thus an unknown sample can be classified using an inner product as follows

$$f(x) = \langle w, \phi(x) \rangle - t$$

where t is a threshold and $f(x)$ the orientation of $\phi(x)$ with respect to the hyperplane. That is, $f(x) > 0$ indicates malicious content in x and $f(x) \leq 0$ corresponds to benign content.

Due to the way the mapping of n-grams is defined, the vector $\phi(x)$ is sparse: Out of millions of possible token n-grams, only a limited subset is present in a particular sample x. These vectors can thus be compactly stored in memory. Also, the inner product to determine the final score can be calculated in linear time in the number of n-grams in a sample

$$f(x) = \sum_{s \in S} w_s \, b(s,x) = \sum_{s \text{ in } x} w_s - t$$

We integrate this classifier into Gordon, such that it can be applied to either the analysis outputs individually or to the joint representation of both.

6 Evaluation

We proceed to empirically evaluate the capabilities of Gordon in different experiments. In particular, we study the effectiveness of the guided execution in terms of code covered (Section 6.2), compare the detection performance with related approaches (Section 6.3) and further demonstrate the effectivity of Gordon in a temporal evaluation (Section 6.4). Before presenting these experiments, we introduce our dataset of Flash-based malware and benign animations.

6.1 Dataset Composition

The dataset for our evaluation has been collected over a period of 12 consecutive weeks. In particular, we have been given access to submissions to the VirusTotal service, thereby receiving benign and malicious Flash files likewise. Since many web crawlers are directly tied to VirusTotal, the collected data reflects the current landscape of Flash usage to a large part.

We split our dataset into malicious and benign Flash animations based on the classification results provided by VirusTotal two months later: A sample is marked as malicious, if it is detected by at least 3 scanners and flagged as benign, if none of the 50 scanners hosted at VirusTotal detects the sample. Samples that do not satisfy one of the conditions are discarded. This procedure enables us to construct a reasonable estimate of the ground truth, since most virus scanners refine their signatures and thus improve their classification results over time. The resulting dataset comprises 1,923 malicious and 24,671 benign Flash animations, with about half the samples being of version 8 or below and the other half of more recent versions, therefore handled by the ActionScript VM version 1 and 2 respectively. A summary of the dataset is given in Table 1.

Table 1. Overview of the evaluation dataset

Classification	**AVM1**	**AVM2**	**Total**
Malicious	864	1,059	1,923
Benign	12,046	12,625	24,671
Total	12,910	13,684	26,594

To account for the point in time the samples have been observed in the wild, we group the samples in buckets according to the week of their submission to VirusTotal. Consequently, we obtain 12 sets containing benign and malicious Flash animations corresponding to the 12-week evaluation period. These temporal sets are used during the evaluation to construct temporarily disjoint datasets for training and testing to conduct our experiments in strict chronological order: For our experiments the performance is determined only on samples that have been submitted to VirusTotal after any sample in the training data. This ensures an experimental setup as close to reality as possible and demonstrates the approach's effectivity of providing timely protection.

6.2 Coverage Analysis

In our first experiment, we evaluate the effectiveness of the proposed guided code-execution strategy. To this end, we investigate the code coverage of malware samples in our 12 week dataset. We apply Gordon to the malware and inspect the output of the interpreter. Due to obfuscation techniques employed by malware, the amount of *statically* contained code of a Flash file often is not a reliable measure in this setting. Hence, we compare the number of executed instructions with respect to a regular execution of the samples. With the path-exploration strategy employed by GORDON, we manage to oberserve over 50% more ActionScript code than during a naive execution, and unveil crucial information not provided otherwise. We mainly credit this leap in coverage to the recursive analysis of dynamically loaded code and code assembled at runtime.

6.3 Comparative Evaluation

We continue to evaluate the detection performance of GORDON, showing its ability to correctly classify Flash-based malware and specifically compare our method with FLASHDETECT[3] [44]. In particular, we evaluate the approaches on the complete set of 12 consecutive weeks, where we use *weeks 1–6* for training and *weeks 7–9* for validation to calibrate the parameters of the detectors. We then combine these two sets for final training and apply the detectors to *weeks 10–12* for testing the detection performance. Table 2 summarizes the results as the true-positive rates and the corresponding false-positives rates of the methods.

Table 2. Detection rates of FLASHDETECT & GORDON.

Method	FLASHDETECT[3]	GORDON-1%	GORDON-0.1%
False-postive rate	1%	1%	0.1%
True-positive rate	26.5%	95.2%	90.0%

GORDON. As described in Section 5 GORDON's detector can be applied to either the analysis outputs individually or to the joint representation of both. The relation thereof is shown in Figure 6(a) as a ROC curve with the detection performance as true-positive rate on the y-axis over the false-positive rate on the x-axis. To map the reports of GORDON's profiler to the vector space we make use of 4-grams. Each representation and the combination of both are plotted as different curves.

At a false-positive rate of 0.1% the individual representations attain a detection rate of 60–65%. The combination of both (GORDON-0.1%) increases the detection performance significantly and enables spotting 90.0% of the Flash-based attacks. If the false-positive rate is increased to 1%, our method even detects 95.2% of the malicious samples in our dataset (GORDON-1%). Additionally we break down this results by CVE numbers. Figure 6(b) shows the detection performance as true-positive rate over the years of appearance of the particular vulnerabilities in our dataset. The average performance is slightly below the overall detection rate, indicating that we also detect malware that does not carry exploits itself, but facilitates a different attack or uses obfuscation to obscure the presence of an exploit. This perfectly demonstrates the capabilities of our approach: First, the complementary views on the behavior and structure of Flash animations provide a good basis for analyzing attacks and, second, this expressive representation can be effectively used for detecting malware in the wild.

[3] Versions not supported by FLASHDETECT (version 8 and below) have been excluded.

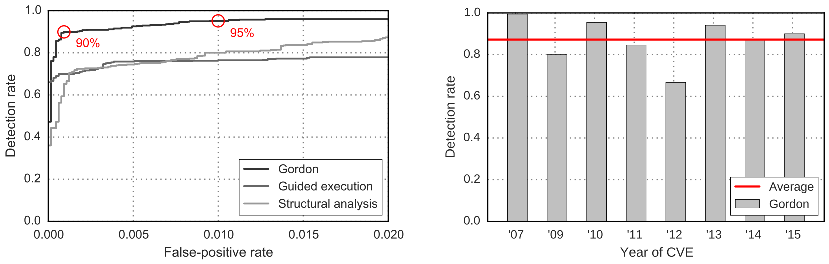


Fig. 6. Detection performance of Gordon as ROC curve and sorted by CVE numbers.

FlashDetect. For the related method FlashDetect we slightly modify the setting and exclude Flash animations of versions below 9 from the evaluation, as this detector is dedicated to the analysis of ActionScript 3 malware only. Nevertheless, FlashDetect only identifies 26.5% of the malicious Flash samples at a false-positive rate of 1%.

Although FlashDetect employs a heuristic for eliciting malicious behavior during the execution of a Flash animation, it misses 3 out of 4 attacks. We attribute this low performance to two issues: First, compared to our method the employed branch selection strategy is less effective and second, the method has been tailored towards specific types of attacks which are not prevalent anymore. Gordon in contrast does not rely on manually selected features, but models the underlying data using n-grams. Therefore it can better cope with the large diversity of today's malware. Due to the low performance of FlashDetect, we omit it from the ROC curve in Figure 6(a).

AV Engines. We finally determine the detection performance of 50 virus scanners on the testing dataset. The 5 best scanners detect 82.3%–93.5% of the malicious samples. However, due to the very nature of signature-based approaches they provide detection with practically no false positives. If we parametrize Gordon to zero false positives only 47.2% of the malware is detected. This clearly shows, that Gordon cannot compete with manually crafted signatures in the long run, but provides solid detection of Flash-based malware shortly after its first occurrence in the wild *without* the need for manual analysis.

As a consequence, we consider our method a valuable tool for improving the analysis of Flash-based malware in the short run and a way to provide traditional approaches with a good starting point in day-to-day business to efficiently craft signatures for AV products.

6.4 Temporal Evaluation

To demonstrate the use of Gordon as a fast, complementary detector, we study its performance over several weeks of operation. We again make use of 4-grams and 12 consecutive weeks of collected Flash data. This time we however apply the detector one week ahead of time, that is, we classify one week after the other, based on the previous weeks.

We start off with *week 1* as training, *week 2* as validation and *week 3* as first test dataset. Over the course of the experiment we shift the time frame forward by one week and likewise increase the training dataset. This can be seen as expanding a window over the experiment's period of time. Hence, Gordon's detector accumulates more and more data for training—just as a system operating in practice would. In order to optimally foster complementary approaches we choose a rather liberal false-positive rate of 1%. Figure 7 shows the true-positive rates achieved by our method during 10 weeks of operation. Gordon starts off below its average performance and takes time till *week 5* to perform well, reaching detection rates between 80% and 99% for the remaining weeks. As our method makes use of machine learning techniques, the detector requires a certain amount of training data before it is fully operational and reaches its optimal performance. If parametrized to 0.1% false positives, Gordon still reaches detection performances of 82% on average.

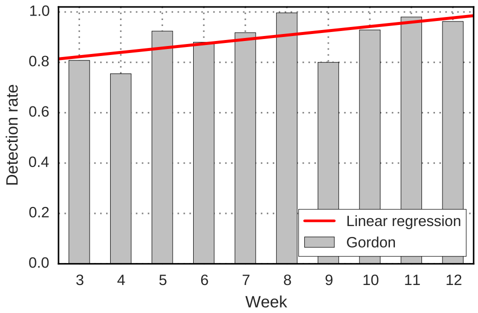


Fig. 7. Gordon's performance over 12 consecutive weeks. The red line illustrates the detector's progression over time, showing a clear uptrend towards its optimal performance.

Overall, this experiment shows Gordon's potential to improve on the detection performance shortly after a malware's appearance in the wild. We consider the number of false-positives—benign samples that need to be additionally analyzed without directly resulting in a malicious signature—as tolerable trade-off for the leap taken in short-term detection performance. In practice, one may start off with a rather strict configuration, accept a lower gain and scale up the interval according to available resources.

7 Limitations

The experiments discussed in the previous section demonstrate that our method provides an effective solution for the analysis and detection of Flash-based malware. Nonetheless, our approach has some limitations which are discussed in the following.

Breaking Code Semantics. With GORDON we make a trade-off between completeness and simplicity of analysis. By pragmatically forcing the execution of specific branches the analyzer avoids expensive computions at execution-time, but may—similar to previous approaches [27, 48]—break semantics of underlying code. Our experiments however show that restricting GORDON to branches which depend on environment-identifying data (c.f. Section 4) reduces the impact of such inconsistencies and that the overall effectiveness of the detector is not influenced in a negative way. Note that, GORDON's path-exploration strategy of guiding analysis towards indicative code regions can also be used in combination with symbolic execution—an adaption worth exploring in future work.

Analysis-aware Malware. Experience has shown that successful analysis systems have repeatedly been subject to dedicated evasion techniques of various types [10]. For GORDON two particular variations come to mind: First, a malware author may leverage differences in implementation of Lightspark and Gnash compared to the Adobe Flash Player. While this is true, the underlying concepts of GORDON can be easily transferred to any interpreter when used in production, possibly using instrumentation [21].

Second, malware might hide its payload in a seemingly irrelevant, low-weighted branch, veiled by branches containing more instructions—potentially across multiple stages. By maximizing the code coverage over multiple executions, GORDON systematically restricts the available space for hiding malicious code. The number of executions thereby is a parameter that allows to strike a balance between coverage and analysis time. Furthermore, the proposed weighting of the annotated ACFG can be refined to better characterize indicative code regions and adapted to malware trends. This can be deployed without the need to change the underlying analysis system and enhanced as analysis-aware malware evolves.

Dynamic Loading of Other File Formats. Although GORDON inspects dynamically loaded code in the form of Flash animations, we do not currently track and analyze other file formats such as audio, video and image containers. These have shown to be a possible attack vector in the scope of Flash malware in the past and have been considered in other malware analysis systems [19, 44]. The detection of embedded malware, however, is a research field of its own and ranges from statically matching shellcode signatures to finding suspicious code in different file containers [4, 39, 40, 50]. For GORDON, we thus consider the analysis of other file formats mainly an engineering effort of integrating other successful approaches.

Interaction with JavaScript and the DOM. Similarly to malware families that make use of ActionScript to set the grounds for exploiting a particular vulnerability in the browser, there also exist attack campaigns that utilize JavaScript for heap spraying, for instance, in order to exploit a vulnerability in the Flash Player. This cannot be handled with the current prototype of GORDON as we solely focus on the Flash part in this paper. Bringing together our method with systems that have proven effective for detecting JavaScript malware [e.g., 13, 16, 35] may close this gap elegantly.

Machine Learning for Malware Detection. Finally, as GORDON's detector is based on machine learning, it may be vulnerable to mimicry attacks [18, 46, 47]. For n-gram models, Fogla and Lee [17] show that generating a polymorphic blending attack is NP-hard, but can be approximated for low-order n-grams. While non-trivial in practice, such attacks could theoretically be conducted against GORDON. However, the use of high-order n-grams elevates complexity to a level where such attacks become impractical. In addition to mimicry, a powerful attacker may systematically introduce samples to shift the classification boundary to her advantage [5, 23]. These attacks can have an effect on GORDON's detector, but require access to large portions of the training data to be effective. As a consequence, such attacks can be alleviated if data from different sources is mixed and subsequently sanitized [15].

8 Related Work

A large body of research has dealt with the detection and analysis of web-based attacks, yet Flash-based malware has received only little attention so far. In this section, we discuss work related to GORDON, focusing on two strains of research: (1) Flash-based malware and (2) multi-path exploration.

Note that the implementations of JavaScript and ActionScript interpreters are fundamentally different, making an application of detection approaches for malicious JavaScript *source-code* unlikely to operate on Flash-based malware available in *bytecode*. Consequently, we do not discuss approaches for malicious JavaScript code in this paper and refer the reader to a wide range of research [7, 13, 26, 27, 35]. Nonetheless, combining detection methods for malicious JavaScript and Flash can be of considerable value. Also, the work by Šrndić and Laskov [45] is of particular interest, since they have been the first to show the practicality of using hierarchical document structure for detection.

Flash-based Attacks and Malware. Only few works have studied means to fend off malware targeting the Adobe Flash platform [19, 44]. ODOSWIFF [19], focuses on detecting malicious ActionScript 2 Flash advertisements based on expert knowledge of prevalent attacks. In contrast to ODOSWIFF, our method employs machine learning to automatically produce a classifier based on benign and malicious Flash animations. FLASHDETECT [44], the successor of ODOSWIFF also makes use of machine learning techniques and, similar to GORDON, employs an instrumented interpreter to dump dynamically loaded code. However, FLASHDETECT only pursues one level of staged-execution, focuses solely on ActionScript 3 and employs a simple heuristic for subverting environmental checks that has proven insufficient for modern Flash-based malware. By contrast, GORDON aims at maximizing the coverage of indicative code regions independent of particular attacks, across multiple stages and versions. As a result, our method allows to uncover vitally more code than FLASHDETECT and thereby attains a better basis for detecting attacks. Furthermore, by not relying on hand-crafted features GORDON can better cope with the large diversity of today's malware.

Industry research has mainly focused on instrumenting Flash interpreters for analysis purposes. Wook Oh [49], for instance, presents methods to patch ActionScript bytecode to support function hooking, and more recently, Hirvonen [21] introduces an approach for instrumenting Flash based on the Intel Pin Platform. These systems complement Gordon and may be used to implement our method for other platforms.

Aside from Flash-based malware and therefore, orthogonal to Gordon several authors have inspected the malicious use of Flash's cross-domain capabilities [25], its vulnerability to XSS attacks [43] and the prevention of such [29].

Multi-path Exploration. Ideally an analysis covers all possible paths and corner case, which however is not feasible due to the potentially exponential number of different execution paths. Most notably in this context is the work by Moser et al. [30], who propose to narrow down analysis to paths influenced by input data such as network I/O, files or environment information. While this effectively decreases the number of paths to inspect it still exhaustively enumerates all paths of this subset under investigation. A second strain of research has considered symbolic execution for the analysis of program code and input generation [e.g., 12, 36]. Brumley et al. [6] combine dynamic binary instrumentation with symbolic execution to identify malware behavior triggered by external commands. Similarly, Crandall et al. [14] use symbolic execution to expose specific points in time where malicious behavior is triggered. Equally to the enumeration of paths, symbolic execution shares the problem of an exponential state space.

With Rozzle, Kolbitsch et al. [27] also make use of techniques from symbolic execution. However, instead of generating inputs, data in alternative branches is represented symbolically and, upon subsequent execution of both branches, merged. In doing so, Kolbitsch et al. except to break existing code due to the execution of infeasible paths. Based on the symbolic representation Rozzle likewise is subject to an exponential state space that is dealt with by limiting the depth of the symbolic trees used. Limbo [48] avoids this kind of state explosion and reverts to a more simple strategy of forcing branching conditions to monitor execution. Limbo however again exhaustively enumerates paths and thus does not address the underlying problem in the first place.

All these methods are either driven by the original execution path [27, 48] or focus on external triggers [6, 14, 30]. Gordon on the other hand, first identifies indicative code regions and guides the interpreter towards these, enabling a payload-centric analysis.

9 Conclusions

In light of an increasing number of vulnerabilities in Flash, there is an urgent need for tools that provide an effective analysis and detection of Flash-based malware. As a remedy, we present Gordon, a novel approach that combines a *structural analysis* of the Flash container format with *guided execution* of embedded ActionScript code—a lightweight and pragmatic form of multi-path exploration. Our evaluation on 26,600 Flash samples shows that Gordon is able to cover more code than observed with other approaches. Moreover, this increase of coverage exposes indicative patterns that enable Gordon's detector to identify 90–95% of malware shortly after its appearance in the wild.

Our method can be used to bootstrap the current process of signature generation and point an analyst to novel malware samples. Gordon thereby provides a valuable step towards the timely protection of end users. Furthermore, the guided execution of code is a simple yet effective strategy for studying malicious code that might also be applicable in other branches of malware analysis, such as for JavaScript and x86 inspection.

Acknowledgments The authors would like to thank Emiliano Martinez of VirusTotal for supporting the acquisition of malicious Flash files. Furthermore, we gratefully acknowledge funding from the German Federal Ministry of Education and Research (BMBF) under the projects APT-Sweeper (FKZ 16KIS0307) and INDI (FKZ 16KIS0154K) as well as the German Research Foundation (DFG) under project DEVIL (RI 2469/1-1).

References

1. Adobe Systems Incooperated. ActionScript virtual machine 2 (AVM2) overview. Technical report, Adobe System Incooperated, 2007.
2. Adobe Systems Incooperated. SWF file format specification. Technical report, Adobe System Incooperated, 2013.
3. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley, 2006.
4. P. Baecher and M. Koetter. libemu - x86 Shellcode Emulation, 2008.
5. B. Biggio, B. Nelson, and P. Laskov. Poisoning attacks against support vector machines. In *Proc. of International Conference on Machine Learning (ICML)*, 2012.
6. D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*, pages 65–88. Springer, 2008.
7. D. Canali, M. Cova, G. Vigna, and C. Kruegel. Prophiler: a fast filter for the large-scale detection of malicious web pages. In *Proc. of the International World Wide Web Conference (WWW)*, pages 197–206, Apr. 2011.
8. L. Cavallaro, P. Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *Proc. of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 143–163, 2008.
9. W. Cavnar and J. Trenkle. N-gram-based text categorization. In *Proc. of SDAIR*, pages 161–175, Las Vegas, NV, USA., Apr. 1994.
10. X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Proc. of Conference on Dependable Systems and Networks (DSN)*, pages 177–186, 2008.
11. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3rd Edition)*. MIT Press, 2009.
12. M. Cova, V. Felmetsger, G. Banks, and G. Vigna. Static detection of vulnerabilities in x86 executables. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, pages 269–278, 2006.

13. M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In *Proc. of the International World Wide Web Conference (WWW)*, pages 281–290, 2010.
14. J. R. Crandall, G. Wassermann, D. A. S. Oliveira, Z. Su, S. F. Wu, and F. T. Chong. Temporal search: Detecting hidden malware timebombs with virtual machines. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 25–36, 2006.
15. G. Cretu, A. Stavrou, M. Locasto, S. Stolfo, and A. Keromytis. Casting out demons: Sanitizing training data for anomaly sensors. In *Proc. of IEEE Symposium on Security and Privacy*, pages 81–95, 2008.
16. C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. Zozzle: Fast and precise in-browser JavaScript malware detection. In *Proc. of USENIX Security Symposium*, pages 33–48, 2011.
17. P. Fogla and W. Lee. Evading network anomaly detection systems: formal reasoning and practical techniques. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, pages 59–68, 2006.
18. P. Fogla, M. Sharif, R. Perdisci, O. Kolesnikov, and W. Lee. Polymorphic blending attacks. In *Proc. of USENIX Security Symposium*, pages 241–256, 2006.
19. S. Ford, M. Cova, C. Kruegel, and G. Vigna. Analyzing and detecting malicious flash advertisements. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, pages 363–372, 2009.
20. gnash. GNU Gnash. `https://www.gnu.org/software/gnash`, visited April 2016.
21. T. Hirvonen. Dynamic Flash instrumentation for fun and profit. In *Proc. of Black Hat USA*, 2014.
22. httparchive. `http://www.httparchive.org`, visited April 2016.
23. L. Huang, A. D. Joseph, B. Nelson, B. I. P. Rubinstein, and J. D. Tygar. Adversarial machine learning. In *Proc. of ACM Workshop on Artificial Intelligence and Security (AISEC)*, pages 43–58, 2011.
24. J. Jang, A. Agrawal, , and D. Brumley. ReDeBug: finding unpatched code clones in entire os distributions. In *Proc. of IEEE Symposium on Security and Privacy*, pages 48–62, 2012.
25. M. Johns and S. Lekies. Biting the hand that serves you: A closer look at client-side flash proxies for cross-domain requests. In *Proc. of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 85–103, 2011.
26. A. Kapravelos, Y. Shoshitaishvili, M. Cova, C. Kruegel, and G. Vigna. Revolver: An automated approach to the detection of evasive web-based malware. In *Proc. of USENIX Security Symposium*, pages 637–651, Aug. 2013.
27. C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. In *Proc. of IEEE Symposium on Security and Privacy*, pages 443–457, 2012.
28. P. Laskov and N. Šrndić. Static detection of malicious JavaScript-bearing PDF documents. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, pages 373–382, 2011.
29. M. T. Louw, K. Thotta, and V. N. Venkatakrishnan. AdJail: Practical enforcement of confidentiality and integrity policies on web advertisments. In *Proc. of USENIX Security Symposium*, pages 371–388, 2010.
30. A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proc. of IEEE Symposium on Security and Privacy*, pages 231–245, 2007.

31. S. K. Nair, P. N. D. Simpson, B. Crispo, and A. S. Tanenbaum. A virtual machine based information flow control system for policy enforcement. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 197(1):3–16, 2008.
32. S. Özkan. CVE Details. `http://www.cvedetails.com`, visited April 2016.
33. R. Perdisci, D. Ariu, P. Fogla, G. Giacinto, and W. Lee. McPAD: A multiple classifier system for accurate payload-based anomaly detection. *Computer Networks*, 5(6): 864–881, 2009.
34. A. Pignotti. Lightspark. `https://github.com/lightspark`, visited April 2016.
35. P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *Proc. of USENIX Security Symposium*, pages 169–186, 2009.
36. P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *Proc. of IEEE Symposium on Security and Privacy*, pages 513–528, 2010.
37. B. Schölkopf and A. J. Smola. *Learning with Kernels.* MIT Press, Cambridge, MA, 2002.
38. R. Sedgewick and K. Wayne. *Algorithms (4th Edition).* Addison-Wesley, 2011.
39. M. Z. Shafiq, S. A. Khayam, and M. Farooq. Embedded malware detection using markov n-grams. In *Proc. of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 88–107, 2008.
40. S. J. Stolfo, K. Wang, and W.-J. Li. *Towards Stealthy Malware Detection*, volume 27 of *Advances in Information Security*, pages 231–249. Springer US, 2007.
41. C. Suen. N-gram statistics for natural language understanding and text processing. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 1(2):164–172, Apr. 1979.
42. A. Systems. Adobe Flash runtimes: Statistics. `http://www.adobe.com/products/flashruntimes/statistics.html`, visited April 2016.
43. S. van Acker, N. Nikiforakis, L. Desmet, W. Joosen, and F. Piessens. FlashOver: Automated discovery of cross-site scripting vulnerabilities in rich internet applications. In *Proc. of ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2012.
44. T. van Overveldt, C. Kruegel, and G. Vigna. FlashDetect: ActionScript 3 malware detection. In *Proc. of International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 274–293, June 2012.
45. N. Šrndić and P. Laskov. Detection of malicious PDF files based on hierarchical document structure. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2013.
46. D. Wagner and P. Soto. Mimicry attacks on host based intrusion detection systems. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, pages 255–264, 2002.
47. K. Wang, J. J. Parekh, and S. J. Stolfo. Anagram: A content anomaly detector resistant to mimicry attack. In *Proc. of International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 226–248, 2006.
48. J. Wilhelm and T. Chiueh. A forced sampled execution approach to kernel rootkit identification. In *Proc. of International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 219–235, 2007.
49. J. Wook Oh. AVM inception - how we can use AVM instrumentation in a beneficial way. In *Shmoocon*, 2012.
50. C. Wressnegger, F. Boldewin, and K. Rieck. Deobfuscating embedded malware using probable-plaintext attacks. In *Proc. of Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, pages 164–183, Oct. 2013.