A Study of Malcode-Bearing Documents

Wei-Jen Li, Salvatore Stolfo, Angelos Stavrou, Elli Androulaki, and Angelos D. Keromytis

Computer Science Department, Columbia University
{weijen,sal,angel,elli,angelos}@cs.columbia.edu

Abstract. By exploiting the object-oriented dynamic composability of modern document applications and formats, malcode hidden in otherwise inconspicuous documents can reach third-party applications that may harbor exploitable vulnerabilities otherwise unreachable by network-level service attacks. Such attacks can be very selective and difficult to detect compared to the typical network worm threat, owing to the complexity of these applications and data formats, as well as the multitude of document-exchange vectors. As a case study, this paper focuses on Microsoft Word documents as malcode carriers. We investigate the possibility of detecting embedded malcode in Word documents using two techniques: static content analysis using statistical models of typical document content, and run-time dynamic tests on diverse platforms. The experiments demonstrate these approaches can not only detect known malware, but also most zero-day attacks. We identify several problems with both approaches, representing both challenges in addressing the problem and opportunities for future research.

Key words: Intrusion Detection, N-gram, Sandbox Diversity

1 Introduction

In this paper, we focus on *stealthy and targeted* attacks where malcode is delivered to a host in an otherwise normal-appearing document. Modern documents and the corresponding applications make use of embedded code fragments. This embedded code is capable of indirectly invoking other applications or libraries on the host as part of document rendering or editing. For example, a pie chart displaying the contents of a spreadsheet embedded in a Word document will cause Excel components to be invoked when the Word document is opened. As a result, documents offer a convenient means for attackers to penetrate systems and reach third-party host-based applications that may harbor vulnerabilities which are not reachable, and thus not directly exploitable, remotely over the network. Disturbingly, attackers are simply exploiting deliberate features that are critical to the way modern document-handling applications operate, instead of some temporary vulnerabilities or bugs.

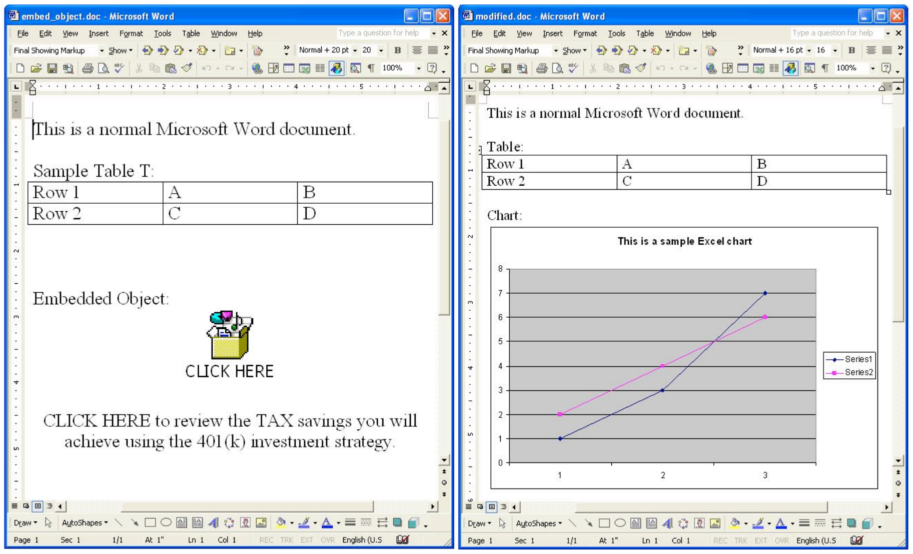
Several cases have been reported where malcode has been embedded in documents (e.g., PDF, Word, Excel, and PowerPoint [1–3]) transforming them into a vehicle for host intrusions. These trojan-infected documents can be served up by any arbitrary web site or search engine in a passive "drive by" fashion, transmitted over email or instant messaging (IM), or even introduced to a system by other media such as CD-ROMs and USB drives, bypassing all the network firewalls and intrusion-detection systems. Furthermore, the attacker can use such documents as a stepping stone to reach other systems, unreachable via the regular network. Hence, any machine inside an organization with the ability to open a document can become the spreading point for the malcode to reach any host within that organization. Indeed, a recent attack of this nature was reported in [4] using Wikipedia. There is nothing new about the presence of viruses in streams, embedded as attached documents, nor is the use of malicious macros a new threat [5, 6], e.g., in Word documents. However, simply disabling macros does not solve the problem; other forms of code may be embedded in Word documents, for which no easy solution is available other than not using Word altogether.

The underlying problem is that modern document formats are essentially object-containers (e.g., Object Linking and Embedding (OLE) format for Word) of any executable object. Hence, one should expect to see any kind of code embedded in a document. Since malcode is code, one cannot be entirely certain that a piece of code detected in a document is legitimate or not, unless it is discovered and embedded in an object that typically does not contain code. Simply stated, **modern document formats provide a convenient object-container format and constitute thus a convenient and easy to use "code-injection platform."**

To better illustrate the complexity of the task of identifying malcode in documents through a concrete study, we limit our investigation to Microsoft Word document files; Word documents serve as a "container" for complex object embeddings that need to be parsed and executed to render the document for display. In addition to the well known macro viruses, two further possible scenarios are introduced bellow:

Execution strategies of embedded malcode: From the attackers perspective, the optimal attack strategy is to architect the injected code as an embedded object that would be executed automatically upon rendering the document. In addition to automated techniques such as the *WMF, PNG and JPEG vulnerabilities*, an attacker can also use social engineering whereby an embedded object in a document, appearing as an icon, is opened manually by the user, launching an attack including attacks against third-party vulnerable applications. The left-side screen shot of Fig. 1 is an example of a Word document with embedded malcode, in this case a copy of the Slammer worm, with a message enticing a user to click on the icon and launch the malcode.

Dormant malcode in multi-partite attacks: Another stealth tactic is to embed malcode in documents that does not execute automatically nor by user intervention when the document is opened, but rather lies dormant in the file store of the target environment awaiting a future attack that would extract the hidden malcode. This multi-partite attack strategy could be used to successfully embed an arbitrarily large and sophisticated collection of malcode components across multiple documents. The right screen shot in Fig. 1 demonstrates another simple example of embedding a known malicious code sample, in this case also Slammer, into an otherwise normal Word document. The document opens entirely normally, with Slammer sitting idly in memory. Both infected files can open normally in a Windows environment. However, the right one appears with no discernible differences from a normal document while a different document could incorporate this Slammer-laden document when it is opened, and invoke the malcode contained therein. Although real-world attacks identical to our example have not appeared, similar scenarios that combine multiple attacks have been studied. Bontchev [5] discussed a new macro attack that can be created by combining two known malicious macros. (e.g., a macro virus resides on a machine, another macro virus reaches it, and "mutates" into a third virus.) Filiol *et al.* [7] analyzed the complexity of another type of viruses named k-ary viruses, which combine actions of multiple attacks.


embed_object.doc - Microsoft Word
modified.doc - Microsoft Word
CLICK HERE
This is a sample Excel chart
Series1
Series2

This is a normal Microsoft Word document.

Sample Table T:

Row 1	A	B
Row 2	C	D

Embedded Object:

CLICK HERE to review the TAX savings you will achieve using the 401(k) investment strategy.

This is a normal Microsoft Word document.

Table:

Row 1	A	B
Row 2	C	D

Chart:

Fig. 1. Left: A screen shot of an embedded executable object to entice the user to click and launch malcode. Right: Example of malicious code (Slammer) embedded in a normal document.

Our aim is to study the effectiveness of two techniques that have been applied in the context of "traditional" network worms: statistical analysis of content to identify portions of input that deviate from expected normal content as estimated from a training corpora, and detection of malicious behavior by dynamic execution on multiple, diverse platforms. The challenge is to find a method to inspect the binary content of any document file before it is opened to determine whether it is suspicious and may indeed be infected with malicious code without *a priori* knowledge of the specific code in question or where it may be embedded in the document.

Initially, we explore the detection capabilities of statical analysis techniques. More specifically, we investigate the application of statistical modeling techniques to characterize the typical content of documents. Our goal is to determine whether we can detect embedded malcode using statistical methods on the binary file content. Furthermore, we introduce novel dynamic run-time tests that attempt to expose the attackers' actions through application diversity: we open the files using a set of different implementations of document processing application in a sandboxed environment. To quantify the detection capabilities of statistical analysis, we perform a series of experiments where statistical analysis is applied to labeled training documents to characterize both normal and malicious document content. Our experiments show that statistical analysis techniques outperform generic COTS Anti-Virus (AV) scanners. To further improve our detection capability, we designed novel tests that harness the application diversity to expose malicious byte-code. In these tests, documents are opened in a diverse set of sandboxed and emulated environments exposing malicious code execution. We show that in most cases, malicious code depend on operating system or program characteristics for successful completion of its execution. In the process of our experimentation, we discovered that attackers use existing benign documents as vehicles for their attack. Thus, we can further improve our classification if we use benign documents from the Web to train our detectors since even small deviations from normality can expose an attack.

Our results indicate that both static statistical and dynamic detection techniques can be employed to detect malicious documents. However, there are some weaknesses that make each method incomplete if used in isolation. For statistical analysis, we would like to be able to determine the "intent" and "effect" of the malicious code. On the other hand, dynamic tests may fail to detect the presence of stealthy malcode that is designed to hide its actions. Hence, neither technique alone will solve the problem in its entirety. We posit that a hybrid approach integrating dynamic and static analysis techniques will likely provide a suitable solution.

Paper Organization: The next section discusses related work and research reported in the literature. Section 3 describes the static statistical approach including an overview of the byte-value n-gram algorithm, the SPARSEGui program and the experimental results. We introduce the dynamic run-time tests and the use of application diversity in Section 4. Section 5 concludes the paper with suggestions that perhaps collaborative detection methods may provide a fruitful path forward.

2 Background and Related Work

2.1 Binary Content File Analysis

Probabilistic modeling in the area of content analysis mainly involves n-gram approaches [8–10]; the file binary contents are measured and the distribution of the frequency of 1-gram, as well as each fixed size n-gram, is computed. An early research effort in this area is the Malicious Email Filter [11], using a naive Bayes classifier algorithm applied to the binary content of email attachments known to be viral. The classifier was trained on both "normal" executables and known viruses to determine whether emails likely included malicious attachments that should be filtered.

Others have applied similar techniques including, for example, Abou-Assaleh *et al.* [12, 13] to detect worms and viruses. Furthermore, Karim *et al.* suggest that malicious programs are frequently related to previous ones [14]. They define a variation on n-grams called "n-perms" An n-perm represents every possible permutation of an n-gram sequence, and n-perms can be used to match possibly permuted malicious code. McDaniel and Heydari [15] introduce algorithms for generating "fingerprints" of file types using byte-value distributions of file content. However, instead of computing a set of centroid models, they compute a single representative fingerprint for the entire class. This strategy may be unwise. Mixing the statistics of different subtypes and averaging of the statistics of an aggregation of examples may tend to loose information. A report from AFRL proposes the Detector and Extractor of Fileprints (DEF) process for data protection and automatic file identification [16]. By applying the DEF process, they generate visual hashes, called fileprints, to measure the integrity of a data sequence, compare the similarity between data sequences, and to identify the data type of an unknown file. Goel [17] introduces a signature-matching technique based on Kolmogorov Complexity metrics, for file type identification.

2.2 Steganalysis

There exists a substantial literature on the subject of steganography, the means of hiding secret messages embedded in otherwise normal appearing objects or communication channels. We do not provide an analysis of this area since it is not exactly germane to the topic of identifying embedded malcode in documents. However, many of the steganalysis techniques that have been under investigation to detect steganographic communication over covert channels may bear resemblance to the techniques we applied during the course of this research study. For example, Provos' work on defeating steganalysis [18] highlights the difficulty of identifying "foreign" material embedded cleverly within media objects that defeats statistical analysis while maintaining what otherwise appears to be a completely normal-appearing objects, e.g., a sensible image.

The general class of steganographic embedding of secret messages may be viewed as a "mimicry" attack, whereby the messages are embedded in such a fashion as to mimic the statistical characteristics of the objects in which the messages are embedded. Our task in this project was a more limited view of the problem, to identify embedded "zero day malcode" inside documents. The conjecture that drives our analysis is that code segments may be limited to a specific set of statistical characterizations so that one may be able to differentiate code from other material in which it is embedded; i.e., code embedded in an image may appear to have a significantly different statistical distribution to that of the class of images used to transport it. Unfortunately, this tends not to be true especially with a crafty adversary capable of generating obfuscation techniques that shape the appearance of the code's binary content to have a user-chosen statistical distribution. One presumes that the attacker knows the particular statistical modeling and testing technique applied while shaping their embedded code to pass the test. Such techniques are being honed by adversaries fashioning polymorphic attack engines that change a code segment and re-shape it to fit an arbitrary statistical distribution, to avoid inspection and detection.

2.3 Polymorphic Code Generation Tools

Polymorphic viruses are nothing new; "1260" and the "Dark Avenger Mutation Engine" were considered the first two polymorphic virus engines, written in the early 90s. Early work focused on making detection by COTS signature scanners less likely. Polymorphic worms with vulnerability-exploiting shellcode, e.g., ADMutate [19] and CLET [20], are primarily designed to fool signature-based IDSes. CLET features a form of padding, called cramming, to defeat simple anomaly detectors. However, cram bytes are derived from a static source, i.e., instructions in a file included with the CLET distribution; while this may be customized to approach a general mimicry attack, it must be done by hand. An engine crafted by Lee's team at Georgia Tech [21] had this purpose in mind; an attack vector was morphed by padding bytes guided by a statistical distribution learned by sniffing the environment in which the code would be injected, hence allowing the code to have a "normal" appearing statistical characterization. This engine targeted the 1-gram distributions computed by the PAYL anomaly detector; the obfuscation and evasion technique was subsequently countered by the Anagram sensor that implements higher-order n-gram analysis. The core algorithm in the Anagram sensor is the basis of the zero-day malcode detection algorithm employed in SPARSEGui as described briefly later, and which we consider to be related to the Shaner algorithm [22] devised to classify files into their respective types. During the course of our tests using thousands of Word documents provided, we found that performance was hard to improve without carefully redistributing training data. In addition, Song *et al.* [23] also suggest it is futile to compute a statistical model of malicious code, and hence identifying malcode embedded in a document may not be the wisest strategy. Hence, we also applied a dynamic test approach to compare against the static analysis approaches, implemented as the VM-based test facility described in Section 4.

2.4 Dynamic Sandbox Tests

Sandboxing is a common technique for creating virtual environments where it is safe to execute possibly unsafe code. For example, Norman Sandbox [24] simulates an entire machine as if it were connected to a network. By monitoring the Windows DLLs activated by programs, it stops and quarantines programs that exhibit abnormal behavior. Since this is a proprietary commercial system, it is unknown exactly how abnormal behavior is determined. Willems *et al.* present an automated system call analysis in a simulated environment, the CWSandbox [25]. They use API hooking: system calls to the Win32 API are re-routed to monitoring software that gathers all the information available to the operating system. Instead of using a virtual environment, TTAnalyze [26] runs a CPU emulator, QEMU, which runs on many host operating systems. Recently, Microsoft Research developed BrowserShield [27], a system that performs dynamic instrumentation of embedded scripts. Similar in spirit to our approach, BrowserShield is designed to detect embedded malcode implemented as HTML scripts which would otherwise be undetectable using static analysis alone.

In this paper, we employ virtual machines running Word-processing applications on diverse platforms; in one case, the native implementation on Windows, in another, a Windows emulation environment running under Linux hosted by a virtual machine. This architecture is easy to implement, and provides a safe means of learning expected behavior of Word document processing under different implementations, using the multiple platform diversity as an additional source of information to identify malcode.

3 Statistical Analysis

As a first effort to identify malcode-infected documents, we used *static inspection of the statistical byte sequences of binary content*. Our intuition is that the binary content of malicious Word documents contains substantial portions of contiguous byte sequences that are significantly different (abnormal) from typical/benign Word documents. Our approach is reminiscent of corpus-based machine learning in natural language processing of human-generated content. The goal is to explore the detection capabilities and limitations of statistical characterization given the available training data. We will start by introducing the tools we used to perform the static analysis and experiments.

3.1 The POI Parser and SPARSEGui

A document may contain many types of embedded objects. To achieve any reasonable level of detection performance, we found it necessary to "parse" the binary file format into its constituent embedded object structure and extract the individual data objects, in order to model instances of the same types together, without mixing data from multiple types of objects.

We used the open-source Apache POI [28] application, a Java implementation of the OLE 2 Compound Document, to decompose Word files into their exact, correct constituent structures. The parsed object structures in the binary content of the files will be referred "sections." We further modified the POI software so that the location of each object within a Word file is revealed. These sections include header information, summary information, word document, CompObj, 1Table, data, pictures, PowerPoint document, macros, *etc.* Fig, 2 displays the histograms of byte content of four common sections whose differences are easy to observe.

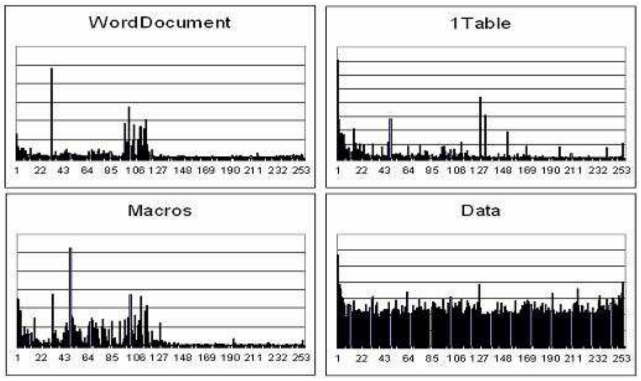


Fig. 2. Byte occurrence frequency of WordDocument, 1Table, Macros, and Data. The byte values were parsed from 120 benign Word documents containing macros. In these plots, the byte value 0 and FF were removed because they occurred relatively much more frequent than the others and will mess up the display.

SPARSEGui includes a number of the modeling techniques described and calls upon the POI parser to provide the means of displaying detailed information about the binary content of Word files as well as presenting experimental results to the user. The experimental results in the remainder of this paper were produced using this toolkit. This program was designed not only to implement the methods described herein but also to provide a user-friendly interface which can extend to analyst information for deeper inspection of a suspect Word file. A screen shot is shown in Fig. 3 in Section 3.4.

3.2 Statistical Content-Based Detection

To evaluate whether statistical binary content detectors can effectively detect malcode embedded in documents, we used the Anagram [8] algorithm. Although Anagram was originally designed to identify anomalous network packet payloads, it is essentially an *efficient approximation* of Shaner's algorithm [22] enabling us to detect malicious binary content. Anagram extracts and models high-order n-grams (an n-gram is a sequence of contiguous n byte values) exposing significant anomalous bytes sequences. All necessary n-gram information is stored to highly compact and efficient Bloom filters [29] reducing significantly the space complexity of the detection algorithm. Contrary to the original 1-class modeling technique applied to the PAYL algorithm [30], we introduce the same mutual-information strategy as suggested by Shaner. Hence, we utilize both "good" and "bad" models that are generated using labeled benign and malicious datasets, respectively. In this way, we train one benign and one malicious model and classify the files by comparing the score computed against both models.

As a next step, we had to determine the optimal n-gram size that best captures the corpus of our documents. To that end, we evaluated the detection performance and storage requirements of all Anagram models with gram size from 4 to 8 bytes. Although larger sized grams can capture more fine-grained information, they can significantly increase the space requirements of Anagram both in terms of runtime memory and in terms of storage. Therefore, for higher ordered grams, a larger Bloom filter is required to avoid having collision that can lead to false positives. The detailed discussion of the size of grams and the use of Bloom filters is beyond the scope of this paper since it depends on the memory usage, the type of data analyzed, and the implementation of Bloom filters [31]. Based on the results of our experiments, we selected the 5-gram model, which consumed reasonable memory and accurately detects attacks.

However, the performance of our statistical methodology was also dependent upon the amount and quality of our training set: without a sufficient training corpus, the detector may produce too many false positives. On the other hand, using a very broad set of documents can produce an augmented and under-trained normality model increasing our false negative rate. To minimize these issues, we generated a model of what we considered as "normal" behavior using Anagram on benign documents. Our aim is to then use Anagram in testing mode on unknown documents to ferret out documents with abnormal content, indicative of zero-day malcode embedded within the document in question. We posit that by generating organization or group specific benign and malicious behavior models, we can further facilitate the detection. The assumption is that documents being created, opened and exchanged within a specific group should be similar, and malicious documents' byte content should be significantly different from them. For the benign data corpus, we collected 4825 samples from two anonymous organizations using *wget* over their public facing web sites (1775 and 3050 documents for each group). In addition, we downloaded 2918 real-world malicious documents from VX Heavens [1] [32].

We used two different approaches to build the normality models. The first method is more coarse-grained and involves scanning and storing all the n-grams from the collected documents creating two separate training sets: one for the benign and one for the malicious model. In the testing/detection phase we compare the n-grams obtained from the documents under test with both the benign and malicious models generating a "similarity score." This score is the ratio of the number of testing n-grams that exist in the training models to the total number of testing n-grams. Testing documents are classified according to the similarity scores they receive from the two models. Documents that receive the same score for both models are deemed malicious. The other approach involves generating multiple normality models corresponding to different document sections instead of using just a single model. Thus, the training documents are parsed and the models are created using the parsed sections, one model for each section. Different sections are text, tables, macros, and other more rare data objects. For each of the section, a weight is assigned. During testing phase, we compare the grams of each of the section from the unclassified document to the ones generated during training. The final similarity scores for each testing document are computed by summing up all of the scores for the individual sections. Thus, we categorize

[1] These experimental datasets can be reached from our web site for interested readers: http://www1.cs.columbia.edu/ids/SPARSE/Spring07TestFiles/

the document under question based on how similar it is to the benign and the malicious section models. The advantage of this method is that different types of embedded objects are not mixed together, so the classification will be more accurate.

For this second method, it is essential to discover the appropriate weights for each section. Although we can easily parse the documents into different sections, we cannot identify which of the sections are malicious even when we know that the whole document is malicious since a document can have section interdependencies. As a result, an appropriate weight for each section cannot be "learnt" by repetitive training/testing. To address this problem, we use the normalized byte size of each testing section as weight.

3.3 Performance Evaluation

We evaluated our statistical content-based techniques using the data we collected on Web. Furthermore, we compared our approach to a COTS AV scanner to both verify and measure our detection performance. In our experiments, we used a standard 5-fold cross-validation scenario, in which data were equally split into five groups, and when each group was tested, the other four were used as training data. All of the pre-mentioned 4825 benign and 2918 malicious documents were tested. In all of our experiments we used an Intel(R) Xeon(TM) CPU 2.40GHz, with 3.6GB memory, running Fedora 3. Depending on the file size, the overhead varied when training/testing a document. The average time to parse or test a file was 0.226 seconds, and the standard deviation was 0.563 seconds. Table 1 presents the experimental results of both methods. The overall performance of Method 2, taking advantage of the parser, was highly accurate and superior to the performance of Method 1. However, the false positive rate of Method 2 was slightly higher than that of Method 1 because Method 2 provided a more detailed comparison. Unfortunately, Method 2 created a more sensitive classifier leading to a slight increase in false positives.

Table 1. Detection results of 5-fold cross-validation. Method 1: Train one single model without parsing. Method 2: Train multiple models for the parsed sections.

	Method 1	Method 2
TP/FN	92.32% / 7.68%	98.69% / 1.31%
FP/TN	0.02% / 99.98%	0.15% / 99.85%
Total Accuracy	95.79%	99.22%

The most recently patched AV scanners have the signatures of all of the malware collected from VX Heavens rendering our dataset inappropriate for further comparing our approach to general COTS AV scanners. Hence, we prepared a second malicious dataset consisting of 35 (10 benign and 25 malicious) carefully crafted files where ground truth was unknown to us before the test. A third-party evaluator created this set of test files for a complete "blind test" of our analysis results. For this test, we trained one model over the 2918 malicious documents and another model using one group of the benign documents we collected (both groups of data had the same final result). After verifying the testing results of these 35 files when ground truth was disclosed, 28 were correctly classified, shown in the first column of Table 2. Note that the numbers in the table are the actual numbers instead of percentages. We achieved zero false positive, but a significant number of false negatives appeared.

Table 2. Detection results of the 35 files. Stat.1: Statistical test, Stat.2: Statistical test with improved strategy, AV: COTS AV scanner

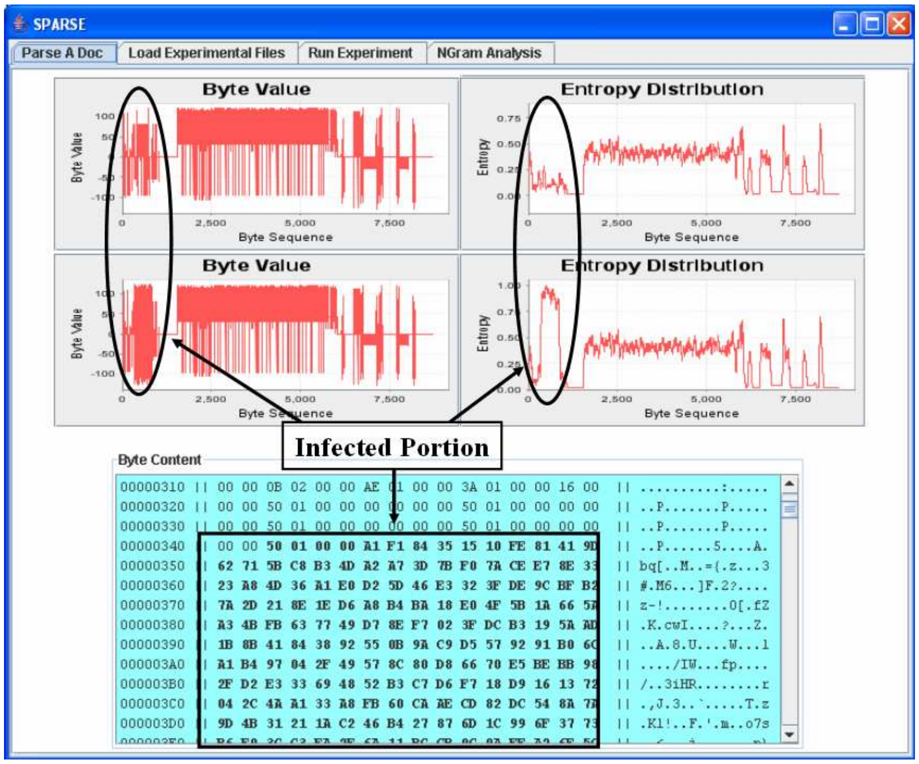
	Stat. 1	Stat. 2	AV
TP/FN	18/7	23/2	17/8
FP/TN	0/10	0/10	0/10
Total Correct	28	33	27

Our statistical analysis technique performed slightly better than a COTS AV scanner, whose result is shown in the third column of Table 2. Our success can be attributed to the fact that we were able to model enough information from the malicious document training set to at least “emulate” the AV scanners with up-to-date signatures. Additionally, the strategy of computing normal models over an organization’s specific set of normal documents produced a model that could detect anomalous events, i.e., the zero-day attacks. However, in this 35-file test, some small, well crafted malcode cleverly embedded in malicious documents were very hard to detect mainly because of their relatively small size in comparison to the whole documents. Statistics based detection mechanism performs poorly in this case. Therefore, we introduce a further strategy.

3.4 File Content Differences Identify embedded malcode

It is possible that an adversary may carefully craft malcode and embed it within a document chosen from a public source. In their effort to blend, attackers would rather use existing public documents since crafting their own documents could contain private or proprietary information that might identify them. Furthermore, an attacker may devise an attack without paying particular attention to the viewable readable document text portion. Malicious documents may contain what might be regarded as “gibberish” material. To generate a benign-looking document, an attacker can mine random benign documents from the Web and embed malcode in them. In that case, comparing the test document to the original or even similar benign document found on the Web can narrow down the detection region and increase the detection accuracy.

Fig. 3 presents a screen shot of SPARSEGui comparing an original benign host document and the infected version of the same document. (The host document was a document accessible on the Internet.) The left two charts represent the byte values, ranging from -128 to 127, of these two documents, the original and infected one respectively. In addition to the byte values, the n-gram entropy values, defined as the number of distinct byte values over an n-gram, are shown in the right two charts. To clearly exhibit the anomalous portion, n is assigned to 50 (i.e., 50-gram) in this figure. In this case, the infected document has a clearly discriminable high entropy portion. Having observed several similar cases, we also discovered that such portions bearing malcode usually contain foreign grams, i.e., never-seen-before grams, which is displayed by using bold characters in the bottom panel of Fig. 3. To evaluate if foreign grams or entropy of foreign grams can provide information to locate suspicious code remains an item for future work.



```
00000310 || 00 00 0B 02 00 00 AE 01 00 00 3A 01 00 00 16 00  || ..........:.....
00000320 || 00 00 50 01 00 00 00 00 00 00 50 01 00 00 00 00  || ..P.......P.....
00000330 || 00 00 50 01 00 00 00 00 00 00 50 01 00 00 00 00  || ..P.......P.....
00000340 || 00 00 50 01 00 00 A1 F1 84 35 15 10 FE 81 41 9D  || ..P......5....A.
00000350 || 62 71 5B C8 B3 4D A2 A7 3D 7B F0 7A CE E7 8E 33  || bq[..M..={.z...3
00000360 || 23 A8 4D 36 A1 E0 D2 5D 46 E3 32 3F DE 9C BF B2  || #.M6...]F.2?....
00000370 || 7A 2D 21 8E 1E D6 A8 B4 BA 18 E0 4F 5B 1A 66 5A  || z-!........O[.fZ
00000380 || A3 4B FB 63 77 49 D7 8E F7 02 3F DC B3 19 5A AD  || .K.cwI....?...Z.
00000390 || 1B 8B 41 84 38 92 55 0B 9A C9 D5 57 92 91 B0 6C  || ..A.8.U....W...l
000003A0 || A1 B4 97 04 2F 49 57 8C 80 D8 66 70 E5 BE BB 98  || ..../IW...fp....
000003B0 || 2F D2 E3 33 69 48 52 B3 C7 D6 F7 18 D9 16 13 72  || /..3iHR........r
000003C0 || 04 2C 4A A1 33 A8 FB 60 CA AE CD 82 DC 54 8A 7A  || .,J.3..`.....T.z
000003D0 || 9D 4B 31 21 1A C2 46 B4 27 87 6D 1C 99 6F 37 73  || .K1!..F.'.m..o7s
```

Fig. 3. SPARSEGui screen-shot: parse a benign document and compare its infected version.

In the prior experiments, the detector produced 7 false negatives. It appears that some of them were crafted based on random benign documents found on the Web. Such "mimic attacks" could evade our statistical content-based detector. Therefore, we developed the following detection strategy: we first parse the inspected document (D1) by using SPARSEGui and take a portion of the text as tokens in a Google search. In case where a document (D2) is found on the Web to have at least 90% of its content in common with D1 (but less than 100%), we extract the n-grams from D1 that *do not appear* in D2. Then, they are computed against the trained Bloom filters and classified to which class it is close, i.e., benign or malicious. Without increasing false positives, this strategy detected 5 malicious documents that were misclassified in the previous 35-file test. Shown in the second column of Table 2, the result was superior to the tested COTS AV scanner.

The 35 test files were purposefully chosen and crafted to avoid detection by statistical means. Even so, we were able to detect almost all of the malicious documents without false positives. However, we did misclassify two malicious documents. Given our performance under such an adverse testing set, we believe that our results demonstrate that our approach has merit and warrants further investigation to improve upon detection performance.

Additionally, our experiments reveal another principle: to validate whether some portion of a document has embedded malcode, mutual collaboration across sites could help identify hidden malcode. Sites that cooperate to detect malcode-laden documents and that share suspect documents could validate that indeed malcode hidden in documents has been discovered. The privacy-preserving sharing of suspect documents among sites is posited as a useful next step in reducing the threat posed by malicious documents appearing openly on the Internet.

A general observation for all the static statistical approaches is that they exhibit inherent limitiations in detecting malicious documents. First, any machine learning or statistical method for that matter, is entirely dependent upon the amount and quality of the training corpus. In general, it is not known a priori if the sample of data available to train and test models is truly representative of a particular empirical distribution from which test cases will be drawn. Without sufficient and representative training data, it is both practically and theoretically infeasible to compute a meaningfull statistical model. In addition, malicious content can be crafted with extremely small portions of malcode spreading throughout a section of a document. Compared to the entire file, which is usually substantially larger, the embedded malcode is very hard to detect. The shorter the sequence of code, the higher the likelihood that a static-analysis detector will miss it. On the other hand, far too many false alarms may be generated if the sensitivity of the detector is raised too high. Lastly, statistical tests may indeed find portions of a document that contain code, but the binary content of the detected foreign code may not identify the "intent" of embedded code. Hence, we investigated an alternative dynamic run-time technique that can improve upon the statistical content-based analysis.

4 Dynamic Run-Time Tests Using Environment Diversity

In this section, we introduce a series of dynamic tests that exploit the diversity of emulation environments to expose malicious documents. Our goal is to determine whether opening a malicious document under an emulated environment can force the malicious code to exhibit easily discernible behavior which deviates from normal and hence identify malicious documents. We show that this behavioral deviation clearly indicates the existence of malicious code inside the file under inspection.

In this case, we did not implement complex instrumentation nor did we apply API hooking to monitor the execution of Word; the implementation of the experimental test bed we built is straightforward: we open documents using the same Microsoft Word executable on different environments both emulated and non-emulated. To avoid damaging our system and to be consistent in applying the same test environment to each file, we ran experiments in a virtual machine (i.e., sandbox) with an identical setup for each test. After each file is tested, we reload the VM image and test the next document. For our prototype, we used VMware WorkStation software installed on the same host machine. For the VM hosted operating system, we installed Linux (Fedora). In that hosted Linux we installed CrossOver Office Standard 5.0.1, a Windows binary translator and emulator. In addition, we had another VM hosting Windows XP and the same version of Microsoft Office (Word 2003) that was used for CrossOver. Based on the observables, we introduce a series of three tests which are referred to as Test 1 (OS crash), Test 2 (Unexpected changes), and Test 3 (Application failure).

4.1 Test 1 – OS Crashes

Applications need to interact with the operating system via libraries and system calls in order to perform even the simplest tasks such as reading or writing a file. In Windows, this happens through the loading of Dynamic Linked Libraries (DLLs), which are loaded both at the beginning of the application's execution and on demand. In large programs, such as Microsoft Word or other applications in the MS Office suite, the number of required DLLs is very large (two to three dozen, depending on the application and the features used by the file loaded). Some of these DLLs are necessary for the program to startup. Most of the rest of the DLLs that the application loads at runtime are required to execute and render the embedded objects and macros after the document is opened. We use the emulated Windows environment on Linux as a concept of changing the loading order of DLLs from the original Windows. Then, the code exceptions depending on this exact order can be revealed. We conjecture that such exceptions, which lead to program and system crashes, are indicative of malcode: normal, non-attacking objects and macros should not depend on the loading order of the DLLs but only on whether the needed DLLs are actually loaded. Based on this hypothesis, a document is opened under the emulated environment to determine if it crashes the application or the underlying operating system. If it does, we declare the document as malicious.

4.2 Test 2 – Unexpected Changes to the Underlying Environment

Test 1 limits our ability to identify malicious documents because most malicious documents may succeed in executing the embedded malcode yet may not crash the test environment.

In the second test, we expand the set of what we deem as abnormal behavior to include all easily observable malicious changes to the hosted operating system. We run the second test after applying the first test and only to documents that fail to be labeled malicious in the first test. Thus, if the document can be opened without any fault or catastrophic error, we examine all the platform files generated or modified by the Word process, i.e., we compare the system right before and after opening a testing document. Our goal is to determine if there are unexpected differences recorded when the malcode embedded in some malicious documents are executed but do not terminate the Word process with a failure or crash.

However, executing the application on multiple environments by opening benign documents may also produce differences in runtime behavior. Hence, there is a small probability that a benign Word document might exhibit different execution behavior (but not failure) under an emulated platform. To minimize such false positive errors, we first train 1000 benign and 1000 malicious documents and gather all of the changes observed to the underlying systems after opening the files. We then generate a list of expected (benign) and unexpected (malicious) changes based on the nature of the document examined. These changes include temporary file creation, data file change such as index.dat, and registry modification. All changes, or the lack of changes, can be used to identify malcode execution. Currently, 27 registry keys are checked in our model, some of them are shown in Table 3, and a Java program is used to automatically verify the changes.

Table 3. The list of registry keys that may be modified after opening a Word document.

[HARDWARE//DESCRIPTION//System//CentralProcessor//0]
[Software//Classes//Interface//A4C46780-499F-101B-BB78-00AA00383CBB //TypeLib]
[Software//Classes//TypeLib//00020430-0000-0000-C000-000000000046//2.0//0 //win32]
[Software//Microsoft//Windows NT//CurrentVersion//Fonts]
[System//CurrentControlSet//Control//Print//Environments//Windows NT x86 //Drivers//PS Driver]
[Software//Microsoft//Office//11.0//Word]
[Software//Microsoft//Office//11.0//Common//LanguageResources]
[Software//Microsoft//Office//Common//Assistant]
[Software//Wine//Fonts]
[Software//Microsoft//Office//11.0//Word//Text Converters//Import//MSWord8]
[Software//Microsoft//VBA//6.0//Common]

When we applied Test 2 to classify the same documents in Test 1, we observed a substantial increase in the true positive rate. This was something we expected since we increased the set of what we deemed as abnormal behavior. However, some malcode may be considerably more “quiet” and “stealthy” and not produce any observable malicious changes to the underlying system. Hence, we apply a third and final test to determine if any easily discernible application behavior indicates the execution of malcode.

4.3 Test 3 – Non-Fatal Application Errors

When we first tested to the set of the malicious documents available, we discovered some types of pop-up messages generated by Microsoft Word. These messages do not cause the OS or emulation environment to fail, but they are clear indicators of malcode execution causing the application to gracefully terminate only some part of the application execution. Users are presented with pop-up windows requesting their input or intervention before they can proceed viewing the document. We use these pop-up messages as the last useful information we can extract from the execution of Word and utilize it for dynamic detection of malicious documents. If both Test 1 and Test 2 fail to label a document as malicious, we apply this final test to the document: we open the document and observe the application output. If one of the known pop-up messages appears on the screen, we mark the document as malicious. However, some benign documents, without embedded malcode, can spawn the same pop-up messages because of improper macro design, rare embedded objects using different versions of applications, or any incorrect use of embedded objects in a Word document.

4.4 Experiments and Analysis

We performed the first experiment by randomly choosing 400 benign and 400 malicious documents which were not in the set of the 1000 benign and 1000 malicious training documents mentioned in Section 4.2. We used the "successive" strategy as the following: Test 1 is first performed, Test 2 is performed only if the testing document is labelled benign in Test 1, and Test 3 is performed only if the testing document is labelled benign in Test 2. The terms "Test 1," "Test 1+2," and "Test 1+2+3" represent the three steps of this successive strategy, respectively. In addition to the successive strategy, we also evaluated indivisual Test 2 and Test 3 as shown in the last two columns of Table 4. Only a few malicious documents were detected by Test 1, but Test 1+2 dramatically increased the true positive rate. The best result we obtained - 97.12% accuracy - was when we employed Test 1+2+3, in which 777 documents out of 800 were correctly classified. However, when performing Test 1+2+3 after Test 1+2, the number of false positives increased by one. This false positive was actually a benign document that requires a border-control feature which is not a default feature in Word. Though Test 1+2+3 doesn't improve the performance much after Test 1+2, it provides further coarse-grained analysis given that it determines if a document contains macros that run automatically when the document is opened.

Table 4. Detection results of 400 benign and 400 malicious documents.

	Test1	Test1+2	Test1+2+3	Only Test2	Only Test3
TP/FN %	4/397 1%/99%	380/20 95%/5%	381/19 95.25%/4.75%	376/24 94%/6%	239/161 59.75%/40.25%
TN/FP %	400/0 100%/0%	397/3 99.25%/0.75%	396/4 99%/1%	397/3 99.25%/0.75%	399/1 99.75%/0.25%
Total Accuracy	50.5%	97.12%	97.12%	96.62%	79.75%

The overhead to test a document is approximately 170 seconds, including cleaning up the image of previous test (1 sec.), duplicating the system image for next comparison (30 sec.), delaying for the observation of Test 1 and 3 (40 seconds), and comparing the image for Test 2 (130 sec. because the size of the compared image is 416MB). Although tests can be performed in parallel, the current overhead is only acceptable to offline analysis.

The next experiment is the blind test using the 35 files. In this test, shown in Table 5, we achieved 100% accuracy; all of the 35 files were correctly classified. However, we do not believe that these tests constitute a complete set of dynamic execution tests that are able to cover all possible malicious documents. First, we have some false negatives when testing the dataset we collected. Second, a stealthy successful attack may be crafted such that it produces no discernible and easily viewable change to the execution environment, i.e., a logic bomb or a multi-partite attack. In cases where malcode may attempt to install a rootkit without any discernible external failures or message pop-ups, an additional test would be necessary to compare the "dormant" virtual machine to its original image and to compare each for possible rootkit installs. Alternatively, running a fully instrumented "shadow" version of the binary of the Word application might identify anomalous program execution at the very point the malcode attempts to exploit a vulnerability or otherwise exhibits anomalous program behavior.

Table 5. Detection results of the 35 files. AV: COTS AV scanner.

	AV	Test1	Test1+2	Test1+2+3	Only Test2	Only Test3
TP/FN	8/17	8/17	25/0	25/0	17/8	12/22
TN/FP	0/10	0/10	0/10	0/10	0/10	0/10
Total Correct	27	18	35	35	27	22

Overall, neither static statistical content-based analysis nor dynamic runtime testing provides a 100% detection accuracy. A combination of both approaches will likely provide more accurate performance. For example, as a preliminary stage, objects embedded in a document can be extracted by the static parser and then subjected to the dynamic tests so the specific malicious objects can be detected. Moreover, these detected malicious objects or malcode can be sent back to patch the static content-based detection models to improve accuracy.

5 Conclusion

Our intention was to provide a better understanding of the issues involved in detecting zero-day malcode embedded in modern document formats. Although the analysis we present is applicable to several other popular file formats, we focused on Word files. Word documents constitute a large percentage of the files exchanged globally both in private and in public organizations. Furthermore, their rich semantics and the large variety of embedded objects make them an ideal attack vehicles against a wealth of applications. Unfortunately, modern proprietary document formats, exemplified by Microsoft Word, are highly complex and fundamentally provide a convenient and easy to use "code injection platform".

In an effort to explore our detection capabilities, we designed sensors using two complementary detection strategies: static content-based statistical-learning analysis and coarse-grained dynamic tests designed to exploit run-time environment diversity. For the static analysis, we employed Anagram an algorithm that generates byte-code n-gram normality models of the training set. Anagram can effectively generate a similarity score between the tested document and the normality model. We compare each document to both benign and malicious normality models and classify the documents based on their similarity scores to those models. We found it necessary to "parse" the binary file format into its constituent embedded object structure in order to extract the individual data objects. Otherwise, the statistical characterizations of different kinds of objects would be blended together, producing poor characterizations of what may be "normal" content. Having a separate weighted model for each of the section of a document increased our total accuracy to 99.22% from 95.79%. Unfortunately, statistical anomaly detection techniques have some inherent limitations: they are dependent on the training set, they require the malicious content to be significantly large and they cannot reveal the "intent" of the malicious documents.

To address these issues, we performed several experiments where we imposed a dynamic run-time environment using multiple COTS implementations of the Word application encased in virtual machines. In some cases, it was immediately obvious and trivial to observe that the document was poisoned; In other cases, validation that the document harbored malcode was dependent upon the actions of the application changing local files or registries. With well specified policies that define unwanted, malicious, dynamic events have a high chance of detecting malicious "intent" (or, at least, behavior) of the code embedded in documents. However, detecting malicious documents by observing runtime behavior also has weaknesses. On the one hand, improperly designed benign macros may cause false alarms in Test 3; on the other hand, logic bombs or stealthily multi-partite attacks may exhibit no abnormal runtime behavior either and thus would be a false negative under dynamic tests. Hence, a deep file inspection using static analysis is still warranted for such stealthy attack cases.

The experimental results indicate that no single static model, nor a single approach, will likely reach the gold standard of 100% detection accuracy, and 0% false positive rate by static analysis alone and do so with a minimum of computational expense (e.g., a small overhead while simply opening the document). However, a combination of techniques, combining statistical analysis and dynamic testing will likely provide reasonable operational value. For example, to amortize the costs of testing documents, perhaps a preliminary stage (static parsing) that extracts suspect embeddings in a document that are then subjected to dynamic tests, which can be performed in parallel among instrumented application instances, may achieve high accuracy and reasonable computational performance objectives. Furthermore, malcode detected by runtime dynamic tests can and should be integrated in a feedback loop. Malcode that is extracted should be used as training data to update static detection models to improve accuracy.

Finally, we conjecture that malcode crafted for a particular version of Word may be reused in a number of publicly available documents. Hence, a collaborative detection process may provide greater benefit. It may be harder for an adversary to craft an attack that is undetectable by all such detectors. Thus, collaboration among a large number of sites that each attempts by a variety of different means to detect malcode embedded in documents would benefit each other by exchanging suspicious content to correlate for common instances of attack data. An alternative strategy might be to create a server farm running many different versions of document applications and that are coordinated to identify documents that harbor malcode, similar in spirit and scope to the Strider Honeymonkey [35] project for collaborative malicious web site detection.

References

1. John Leyden: Trojan exploits unpatched Word vulnerability. The Register, May 2006
2. Joris Evers: Zero-day attacks continue to hit Microsoft. News.com, September 2006
3. David Kierznowski: Backdooring PDF Files. September 2006
4. Matthew Broersma: Wikipedia hijacked by malware. Techworld, Nov 2006 `http://www.techworld.com/news/index.cfm?RSS\&NewsID=7254`
5. Vesselin Bontchev: Possible Virus Attacks Against Integrity Programs and How to Prevent Them. Proc. 2nd Int. Virus Bull. Conf., 1992, pp.131-141.
6. Vesselin Bontchev: Macro Virus Identification Problems. Proc. 7th Int. Virus Bull. Conf., 1997, pp. 175-196.
7. Eric Filiol, Marko Helenius, Stefano Zanero Open Problems in Computer Virology Journal in Computer Virology (2006), pp.55-66.
8. Ke Wang, Janak Parekh, and Salvatore J. Stolfo: Anagram: A Content Anomaly Detector Resistant to Mimicry Attack. Proc. Int. Conf. on Recent Advanced in Intrusion Detection, RAID06, Sept 2006
9. Salvatore J. Stolfo, Wei-Jen Li, Ke Wang: Fileprints: Identifying File Types by n-gram Analysis. 2005 IEEE Information Assurance Workshop
10. Wei-Jen Li, Ke Wang, Salvatore J. Stolfo: Towards Stealthy Malware Detection. Malware Detection Book, Springer Verlag, (Jha, Christodorescu, Wang, Eds.), 2006
11. Matthew G. Schultz, Eleazar Eskin, Erez Zadok, Salvatore J. Stolfo: Data Mining Methods for Detection of New Malicious Executables. IEEE Symposium on Security and Privacy, Oakland, CA, May 2001
12. Tony Abou-Assaleh, Nick Cercone, Vlado Keselj, Ray Sweidan: Detection of New Malicious Code Using N-grams Signatures. Proceedings of Second Annual Conference on Privacy, Security and Trust, October 13-15, 2004
13. Tony Abou-Assaleh, Nick Cercone, Vlado Keselj, and Ray Sweidan: N-gram-based Detection of New Malicious Code. In Proceedings of the 28th IEEE Annual International Computer Software and Applications Conference, COMPSAC 2004. Hong Kong. September 28-30, 2004

14. Md. Enamul Karim, Andrew Walenstein, and Arun Lakhotia: Malware Phylogeny Generation using Permutations of Code. Journal in Computer Virology, 2005
15. McDaniel and M. Hossain Heydari: Content Based File Type Detection Algorithms. 6th Annual Hawaii International Conference on System Sciences (HICSS'03)
16. Andrew J. Noga: A Visual Data Hash Method. Air Force Research report, October 2004
17. Sanjay Goel: Kolmogorov Complexity Estimates for Detection of Viruses. Complexity Journal, vol. 9, issue 2, Nov-Dec 2003
18. Steganalysis, `http://niels.xtdnet.nl/stego/`
19. K2. ADMmutate. 2001 Available from: `http://www.ktwo.ca/security.html`
20. T. Detristan, T. Ulenspiegel, Y. Malcom, and M. Underduk: Polymorphic Shellcode Engine Using Spectrum Analysis. Phrack 2003
21. Oleg Kolesnikov, and Wenke Lee: Advanced Polymorphic Worms: Evading IDS by Blending in with Normal Traffic. USENIX Security Symposium. 2006, Georgia Tech: Vancouver, BC, Canada
22. Shaner: US Patent No. 5,991,714, Nov 1999
23. Yingbo Song, Michael E. Locasto, Angelos Stavrou, Angelos D. Keromytis, Salvatore J. Stolfo: On the Infeasibility of Modeling Polymorphic Shellcode for Signature Detection Tech. report cucs-00707, Columbia University, Feb 2007
24. Kurt Natvig: SandboxII: Internet Norman SandBox Whitepaper, 2002
25. Carsten Willems, Felix Freiling, and Thorsten Holz: Toward Automated Dynamic Malware Analysis Using CWSandbox. IEEE Security and Privacy Magazine, April 2007, Vol. 5, No. 2, pp. 32-39
26. Fabrice Bellard: QEMU, a Fast and Portable Dynamic Translator. In proceedings of the USENIX 2005 Annual Technical Conference, pages 41-46, 2005
27. Charlie Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir: BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML. OSDI 2006, Seattle, WA
28. POIFS: `http://jakarta.apache.org/`
29. Burton H. Bloom: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, v.13 n.7, p.422-426, July 1970
30. Ke Wang, Gabriela Cretu, Salvatore J. Stolfo: Anomalous Payload-based Worm Detection and Signature Generation. Proceedings of the Eighth International Symposium on Recent Advances in Intrusion Detection(RAID 2005)
31. Andrei Broder, Michael Mitzenmacher: Network Applications of Bloom Filters: A Survey. Allerton Conference 2002.
32. `http://vx.netlux.org/`
33. Eric Totel, Frederic Majorczyk, and Ludovic Me: COTS Diversity Intrusion Detection and Application to Web Servers. RAID 2005
34. James C. Reynolds, James Just, Larry Clough, and Ryan Maglich: On-line intrusion detection and attack prevention using diversity, generate-and-test, and generalization. In Proceedings of the 36th Hawaii International Conference on System Sciences, 2003
35. Yi-Min Wang, Doug Beck, Xuxian Jiang, Roussi Roussev: Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities. NDSS 2006