SPARSE: A Hybrid System to Detect Malcode-Bearing Documents

Wei-Jen Li and Salvatore J. Stolfo
Department of Computer Science
Columbia University
{weijen,sal}@cs.columbia.edu

Abstract

Embedding malcode within documents provides a convenient means of penetrating systems which may be unreachable by network-level service attacks. Such attacks can be very targeted and difficult to detect compared to the typical network worm threat due to the multitude of document-exchange vectors. Detecting malcode embedded in a document is difficult owing to the complexity of modern document formats that provide ample opportunity to embed code in a myriad of ways. We focus on Microsoft Word documents as malcode carriers as a case study in this paper. We introduce a hybrid system that integrates static and dynamic techniques to detect the presence and location of malware embedded in documents. The system is designed to automatically update its detection models to improve accuracy over time. The overall hybrid detection system with a learning feedback loop is demonstrated to achieve a 99.27% detection rate and 3.16% false positive rate on a corpus of 6228 Word documents.

1. Introduction

Modern document formats are fundamentally object containers that provide a convenient "code-injection platform." One can embed many types of objects into a document, not only scripts, tables, and media, but also arbitrary code used to render some embedded object of any type. Many cases have been reported where malcode has been embedded in documents (e.g., PDF, Word, Excel, and PowerPoint [1,2,3]) transforming them into a vehicle for host intrusions., Malcode bearing documents can be easily delivered and bypass all the network firewalls and intrusion detection systems when posted on an arbitrary website as a passive "drive by" Trojan, transmitted over emails, or introduced to systems by storage media such as CD-ROMs and USB drives. Furthermore, attackers can use such documents as a stepping stone to reach other systems, unreachable via the regular network. Consequently, 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.

In this study, we focus on Microsoft Word document files. Microsoft Office documents are implemented in Object Linking and Embedding (OLE) structured storage format, in which any arbitrary code could be embedded and executed [4,5]. There is nothing new about the presence of viruses in email streams, embedded as attached documents, nor is the use of malicious macros a new threat [6,7], (e.g., in Word documents). However, neither signature-based state of the art detection nor simply disabling macros solves the problem; any form of code may be embedded in Word documents, for which no easy solution is available other than not using Word altogether.

To better illustrate the complexity of the task of identifying malcode in documents, we first briefly introduce two possible attack scenarios:

Execution strategies of embedded malcode: The most common attack strategy is to architect the injected code that would be executed automatically when the document is opened, closed, or saved. In addition to such techniques, some selective attacks are crafted to appear as a button or an icon with a phishing-like message that tricks a user to manually launch the malcode. The screen shot in Figure 1-a 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 is not activated when rendering the document nor by user intervention, bur rather the

malcode waits quietly in the target environment for another future attack that would retrieve the document from disk and then launches the embedded 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 screen shot in Figure 1-b demonstrates an example of embedding a known malicious code, in this case Slammer, into an otherwise normal Word document. There is no indication of the presence of Slammer, not even from the host AV scanner that includes signatures for Slammer. With the malcode sitting idly in memory, the document opens entirely normally and causes no discernible unusual system behavior from normal documents. Similar scenarios that combine multiple attacks have been studied. Bontchev [6] 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. [8] analyzed the complexity of another type of viruses named k-ary viruses, which combine actions of multiple attacks.

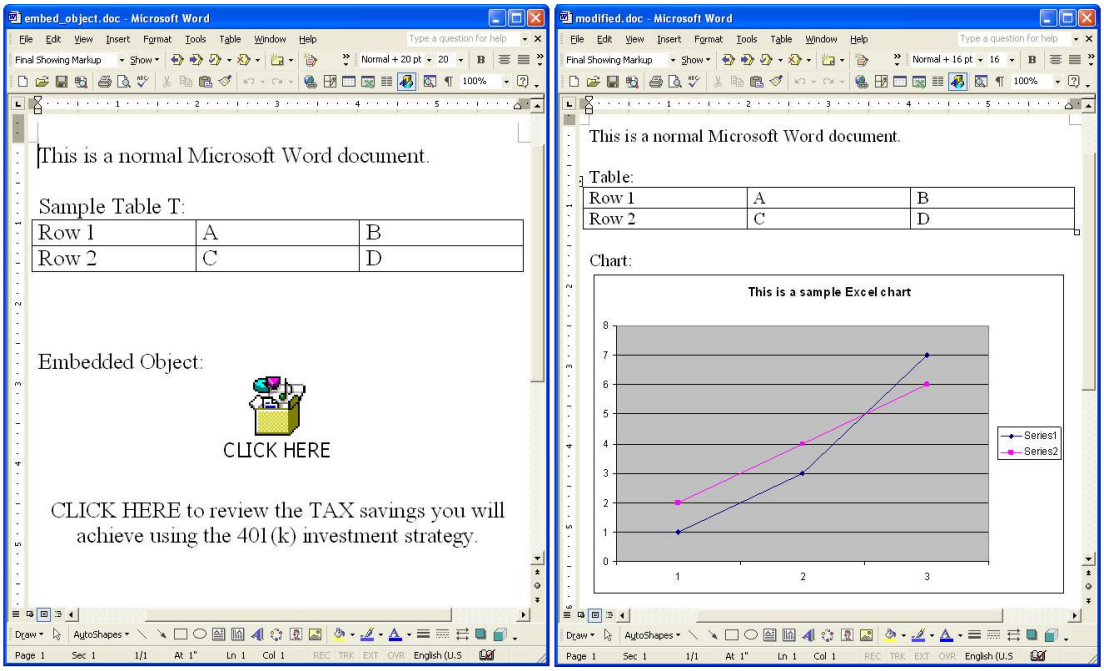

Figure 1. Left (1-a): A screen shot of an embedded executable object to entice the user to click and launch malcode. Right (1-b): Example of malicious code (Slammer) embedded in a normal document.

Approaches to malware detection can be roughly categorized into two areas: static and dynamic. Static approaches analyze the binary content of malware without executing the code [9-18], while dynamic approaches execute the malware in emulated environments and observe the run-time behavior [19-24]. However, both approaches, static and dynamic, have weaknesses. Current work on polymorphism [25-28] suggests it is futile to compute a signature model of all possible malicious code segments. AV Scanners will likely be obsolete. Dynamic detection approaches may also be thwarted by stealthy mimicry attacks [29, 30] that may be crafted such that it produces no discernible and easily viewable change to the execution environment. Hence, neither technique alone will solve the problem in its entirety. However, their combination may substantially improve the ability to detect a wide variety of malcode bearing documents.

In our prior work [37], we proposed two directions to detect malicious Word documents: a static and a dynamic approach, but several open issues remained that we now address. In this paper, we present a

hybrid system we call SPARSE[1] that combines multiple detectors and various detection strategies to detect malicious documents as well the location of the embedded malcode. Specifically, we present a static detector that characterizes the binary contents of document files and a dynamic event detector system that models the system's run-time behavior.

We provide an effective means of combining both static and dynamic detection methodologies in an integrated system and present several new detection methods that improve upon the sandboxing approaches previously developed. In our prior work VM images were compared before and after opening a document. This limited view of the dynamic execution of a document rendering application could easily miss important system events indicating the presence of stealthy malicious code execution.

To activate and examine passive embedded objects that require human action to launch, we implemented a mechanism that automatically interacts with the document to simulate user click behavior to purposely launch embedded objects that may harbor malcode. Furthermore, to detect stealthy embedded malcode that may mimic normal execution behavior, we introduce a technique inspired by instruction set randomization techniques, that thwart code injection attacks, that randomly changes data values in certain sections of Word documents to cause malcode failures. Finally, we introduce a method to locate the malicious portion embedded in a document by removing data sections one at a time and testing for the presence of malcode in each. This strategy not only locates the malcode but the extracted section with the malcode can be used as data for an integrated feedback loop to update the models to improve the accuracy of the detector over time.

We report a series of experiments to quantify the detection accuracy of the hybrid SPARSE system. The result shows that the integrated system outperforms each of the individual constituent detectors and achieves an overall 99.27% detection rate and 3.16% false positive rate using a sample of 6,228 Word documents acquired from public sources. We do not argue that our approach is perfect; however, we demonstrate that a hybrid detection system combining various detection strategies can enhance the level of protection, certainly beyond signature-based methods. The result is that attackers who craft documents with embedded malcode will expend far more effort to create undetectable malcode bearing documents.

The contributions of this paper are:

1. A hybrid detection system that integrates a static file binary content detector and a dynamic run-time system event detector. The system not only accurately detects malicious documents but also can automatically generate new data to update the detection model.
2. A malcode locating mechanism that indicates the malicious portion of test documents.
3. A data randomization method to disable and detect embedded stealthy malcode.
4. An automaton simulating a user's interaction with documents to launch embedded objects for test by the dynamic analysis component.
5. A static analysis of sections harboring malcode using entropy analysis that may provide a useful forensic tool to inspect document content.

The paper is organized as follows. In Section 2, we discuss related work and in Section 3 we detail our techniques, including the integrated detection system and, subsequently, each detector and detection method. We evaluate our approach in Section 4 and conclude the paper in Section 5.

2. Background and Related Work

2.1 Binary Content File Analysis

[1] Initial work on this problem was primarily focused on Statistical PARSing of the binary content of documents leading to the name of the system as SPARSE. Document may also be quite sparsely populated with malcode compared to the rest of a document's benign content.

Statistical analysis of file binary contents has been studied in recent years including statistical n-gram modeling techniques in which the distribution of the frequency of 1-gram, as well as a mixture of higher order n-grams, are computed to model file content of various types [9, 10, 11]. Shaner's work [12] is probably among the earliest work of this type applied to the binary content of files. Abou-Assaleh et al. [14] detect worms and viruses based on the frequency of certain n-grams, and Karim et al. [15] define a variation on n-grams called "n-perms," which represents every possible permutation of an n-gram sequence that may be used to match possibly permuted malicious code. Both appear later than related work on payload anomaly detection and malcode baring document detection [9, 10, 11].

Some work has been done to automatically identify the type of an unknown file based upon a statistical model of its content. Goel [16] introduces a signature-matching technique based on Kolmogorov complexity metrics. McDaniel and Heydari [17] introduce algorithms for generating "fingerprints" of file types using byte-value distributions by computing a single model for the entire class. However, instead of computing multiple centroid models, they mix the statistics of different subtypes and compute loose information by averaging the statistics of examples. AFRL proposes the Detector and Extractor of Fileprints (DEF) process for data protection and automatic file identification [18]. They identify the data type of unknown files by generating visual hashes, called fileprints, and measuring the integrity of a data sequence.

2.2 Dynamic System behavior detection

Anomaly detection based on system calls has been studied for years. Forrest et al. introduced the earliest approach [31], while others improved upon the technique by incorporating system call arguments [32, 33]. On the other hand, recent studies have shown that mimicry attacks [29, 30] that utilize legitimate sequence of system calls, or even system calls with arguments can evade detection if crafted properly.

To monitor and collect the system run-time behavior such as system calls, sandboxing is a common technique where it is safe to execute possibly unsafe code. For example, the Norman Sandbox [19] 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. CWSandbox [20] employs an API hooking technique that hooks onto the Win32 API to gather system call information. TTAnalyze [21] runs a CPU emulator, QEMU, which runs on many host operating systems. BrowserShield [22], developed by Microsoft Research, instrument embedded scripts to protect against HTML-based Web attacks that is similar in spirit to our approach to detect embedded malcode in documents.

2.3 Steganalysis, Polymorphism, and Mimicry Attacks

Steganography is a technique that hides secret messages embedded in otherwise normal appearing objects or communication channels. Provos [34] studies cleverly embedded "foreign" material within media objects that evades statistical analysis while maintaining what otherwise appears to be completely normal-appearing objects (e.g., a sensible image). Similarly, polymorphic techniques have been exploited to deceive signature-based IDSes. ADMutate [25] and CLET [26] craft polymorphic worms with vulnerability-exploiting shellcode to defeat simple static anomaly detectors. According to the statistical distribution learned by sniffing the environment, Lee et al. [27] inject morphed padding bytes into the code allowing the code to have a "normal" appearing statistical characterization. Song et al. [28] suggest it is futile to compute a set of signature models of malicious code, and hence identifying malcode embedded in a document using signature-based approaches may not be the wisest strategy.

Steganography and polymorphism are convenient techniques for "mimicry" attack; the malcode is shaped to mimic the statistical characteristics of the embedded host or the normal objects, in order to avoid inspection and detection. Furthermore, there exist mimicry attacks [29, 30] that mimic the legitimate dynamic system behavior to evade system call based detection. The file data randomization technique introduced in this paper may counter these obfuscation techniques since either the encrypted

code or the decoder hiding in documents is randomly modified and its functionality may be disabled potentially leading to its detection if the modification results in a system crash.

3. The Detection System

In this section, we first describe the design of our detection system. Section 3.1 provides an overview of the entire system and how the components are related. Section 3.2 introduces the document parser and Section 3.3 describes the static detector. The dynamic sandbox and the data randomization technique to detect stealthy embedded malcode are presented in 3.4, while the method used to locate the malicious portion of documents is presented in Section 3.5.

3.1 System Design

The SPARSE detection system, shown in Figure 2, includes the document parser, the static detector, the dynamic run-time system event detector, and the malcode locator. First, the document parser parses documents into constituent object embedding structures and extracts the individual data objects, in order to model instances of the same types together, without mixing data from multiple types of objects. Second, the static detector, which is based on analyzing the file binary content, employs a 2-class mutual information detection strategy and contains both a benign model (whitelist) and a malicious model (blacklist). The third component, the dynamic system event detector, is a virtual machine in which we execute Word and observe the system's behavior when opening documents. Finally, the malcode locator is also a virtual machine running parallel to the other two detectors and is designed to test suspicious code detected by the previous components and to further indicate where the malcode is located. The host control system, the document parser, and the three detectors are primarily implemented in Java; other functions, such as connecting the components and operating the VM, are programmed in script languages (e.g., batch script and AutoIt [35]). The main program shown in Figure 3 is named SPARSEGUI, which is a graphical user interface that parses documents, provides basic statistical analysis, and performs all of the experiments.

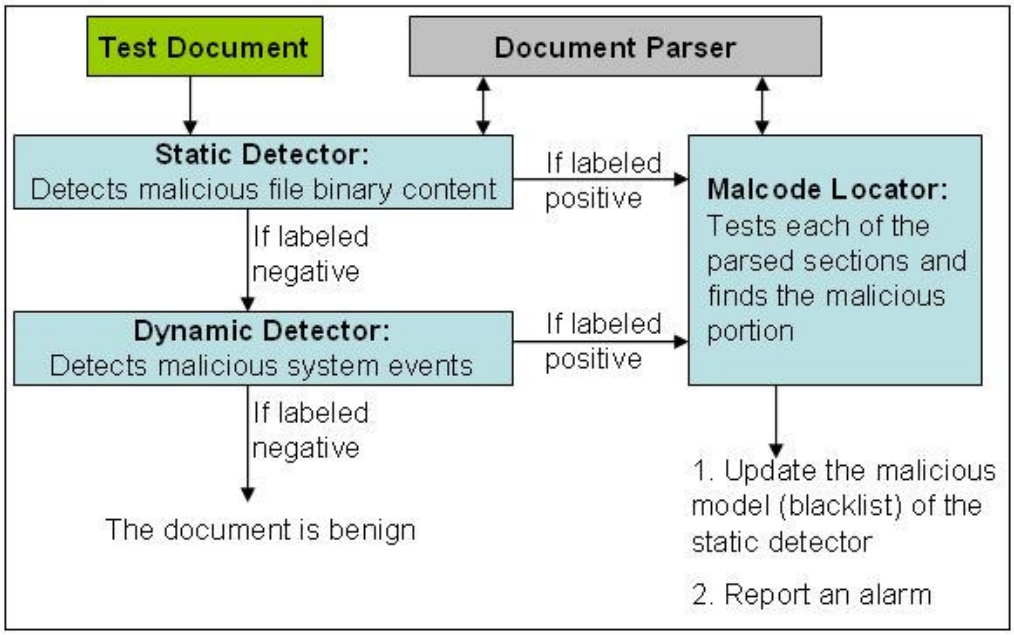

Figure 2: The SPARSE system design.

The entire system is operated as a series of detectors. The test document is sent to the static detector, and the static detector calls the parser to separate the document into individual sections and compares the byte information against its set of models each conditioned on distinct object types. If labeled negative (benign), the document is subjected to further tests using the dynamic detector. Subsequently, the dynamic detector renders the document and monitors the system's behavior. If the dynamic detector also labels it negative, the document is deemed benign. On the other hand, if either detector labels the document positive, it is labeled malicious and it is subsequently processed by the malcode locator

One may set low thresholds using the SPARSEGUI so the false positive (FP) rate is minimized in both detectors. Moreover, the false negative (FN) rate is reduced by integrating the detection results of the two detectors. Specifically, the test document is deemed malicious if it is labeled positive by either detector, and it is deemed benign if both detectors say negative. The malcode locator parses the document into sections and tests them in the VM to determine which is (are) malicious. Furthermore, the malicious portion(s) found in this malcode locator are used to update the malicious model in the static detector. The detailed techniques are described in the following sections.

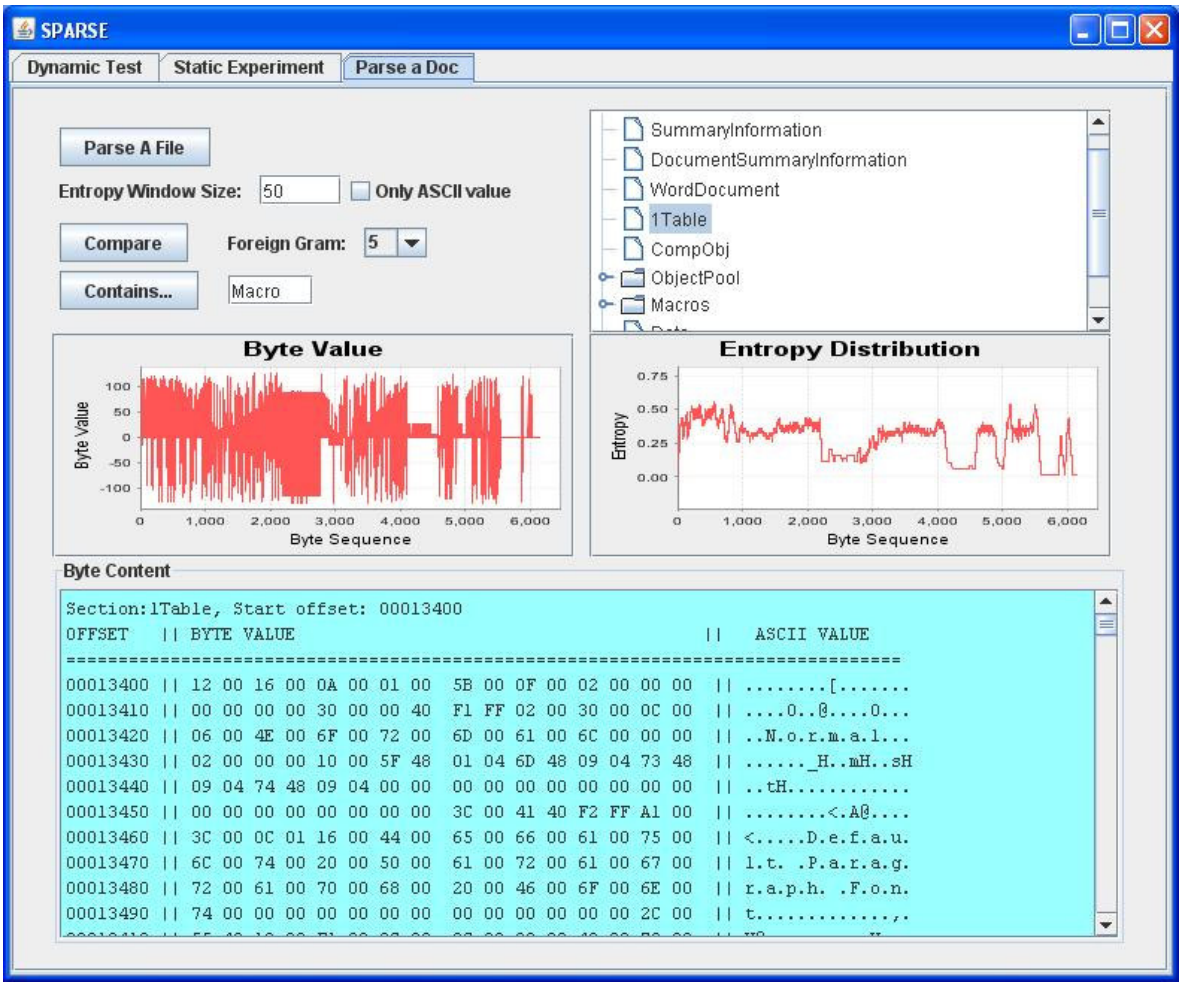

Figure 3: A screenshot of SPARSEGUI

3.2 Document Parser

The MS Office format – OLE (Object Linking and Embedding) storage structure, shown in Figure 4 – is a complex proprietary format that contains many object types such as scripts, images, and arbitrary code. This vast array of different kinds of embedded objects one may encounter in Word files led us

immediately to consider methods to extract the different kinds of objects, and then to model each object type separately; otherwise the statistical characterizations of different kinds of objects would be blended together likely producing poor characterizations of what may be "normal" content. This finer-grained approach also provides the opportunity to hone in on the precise location where the malicious component may be embedded. Furthermore, isolation of embedded objects would help in identifying vulnerable third-party programs and libraries used to render these objects, e.g. proprietary media objects containing malcode.

For the static detector, we parse the documents into independent "sections" and model them separately. (The nodes shown in Figure 4 are referred to as "sections" in the rest of this paper.) By the means of the parser, we can build multiple static models for the parsed sections instead of using a single model for the whole file. A set of training documents are parsed and the models are computed using all of the parsed sections, one model for each section type, such as text, tables, macros, and other data objects. A weight is computed for each model that is proportional to the data length of each section.

Test files are compared against the models after being parsed using the same process. Each section of a test document is compared against the trained models producing a similarity score for that section. A final weighted similarity score is computed by summing the section scores. Moreover, the dynamic malcode locator also takes advantage of the parser. The malcode locator extracts the parsed sections and tests them individually as described in section 3.5.

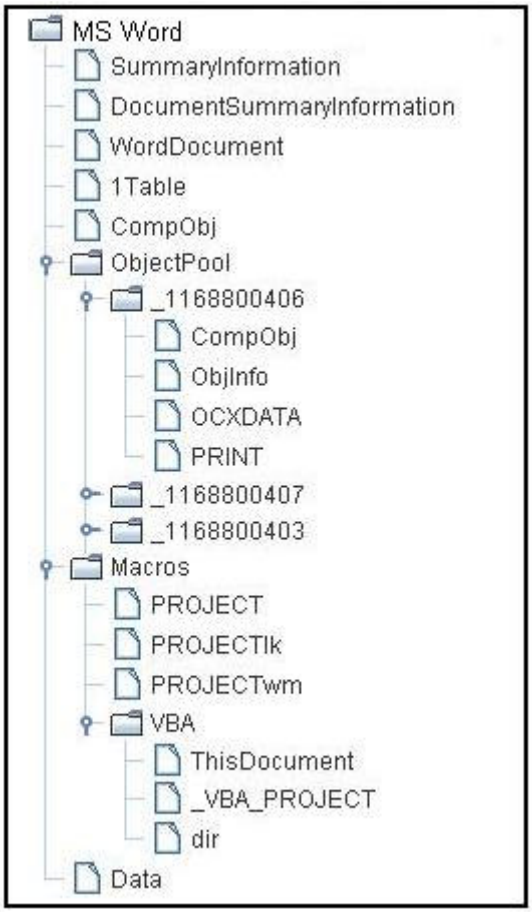

Figure 4: An example of a parsed document in OLE format

3.3. Static Detector

The static detector is based on the Anagram algorithm [9] which was originally devised for detecting

malicious exploits in network traffic. Anagram extracts and models high order n-grams exposing significant anomalous byte sequences. All necessary n-gram information is stored in highly compact and efficient Bloom filters [36] reducing significantly the space complexity of the detection algorithm. The detailed algorithm used to detect malicious documents is described in [37] in which a 2-class mutual information modeling technique (i.e. one benign model and one malicious model) is suggested. Note that both the benign and the malicious model referred in the rest of this paper actually represent a set of sub-models in which each contains the n-gram information of a specific parsed section.

The 2-class technique classifies the test document, either benign or malicious, depending on which model returns a higher similarity score, and this tactic has higher accuracy than 1-class anomaly detection that may suffer from a high False Positive rate. However, the 2-class strategy may fail to detect zero-day attacks – a zero-day attack may be similar to neither the benign nor the malicious model. There is little hope in increasing the detection accuracy without introducing more false alarms by using this single approach.

Hence, a hybrid detection system that utilizes dynamic information may provide a better chance to detect embedded malcode. We minimize the FP rate in this (static) detector and leave the labeled negative test documents to the next dynamic detector. To lower the FP rate, we train the malicious model using a significant number of known malcode samples and set the threshold so that only malcode that is very close to the model will create an alarm. In other words, we purposely set the threshold logic favoring benign documents, and possibly zero-day malware, that would have a low similarity score computed against the malicious model. The malicious model is updated by feedback from the malcode locator (detailed in Section 3.5) in which we extract the malicious sections and use them to train the static malicious model, while ignoring the benign portions.

3.4 Dynamic Sandbox

3.4.1 Dynamic System Event Detector

In this section, we introduce our dynamic detector that monitors the system run-time behavior. Several types of sandboxes exist for analyzing a programs' dynamic behavior. Some approaches utilize controlled environments [21], while others take a snapshot on only a specific state [24] or compare the image before and after executing the malware [37]. The disadvantage of these techniques is that only a limited view of the system is monitored. A certain number of run-time system events do not reside in memory after the emulation is completed, and some system events will not be produced if the test environments are strictly controlled. In both cases, the malicious behavior may not be observed. Since document formats are not only vulnerable applications (e.g. attackers exploit the bugs in Winword.exe) but also convenient malcode containers (e.g. any type of objects can be embedded by using trivial techniques), our detection system must be able to monitor all possible run-time behavior. As a result, the entire virtual machine became the best choice for our study. Although an entire virtual machine is more expensive than emulators and may be vulnerable to some attacks [38], our approach has significant advantages. The dynamic system behavior can be observed in the virtual machine; no activity is missed. Furthermore, an entire virtual machine with a complete operating system allows us to craft agents to simulate human interaction with the test application in the sandbox (e.g., the passive attached objects can be activated). Not only can our system detect malware, but it can also report malicious events in other third party applications. All of these processes are automated by the SPARSEGUI and script programs.

We implemented the system in VMware Workstation, under which we install Microsoft Windows XP SP2 and MS Office with no patches installed. To capture the run-time system behavior, we use Process Monitor [39]. Process Monitor is a process hooking monitoring tool for Windows that captures various system events. Similar to modeling system calls with arguments, we monitor more specific system information including files created/modified, modules loaded, registry accessed/changed, and process activity. We refer to all of these as "system events" in the rest of this paper.

A system event, appearing like a system call along with its argument, is defined as the concatenation of the operation type and the file path. For example, a model is shown in Figure 5, in which the system events are recorded by rendering a Word document on Windows. In this figure, "CreateFile, c:\a.reg" means that the operation is "CreafeFile" and the file path is "c:\a.reg." Figure 5 is a simplified example; however, when a Word document is opened in Windows, there are thousands or tens of thousands of individual system events. Among these system events, there are 90 distinct operations and hundreds of distinct system events.

The system events, displayed in Figure 5, were captured by executing a Word Virus. It is clear that the virus replaced the original system file regedit.exe, created some DLLs, and changed the security setting. Apparently, opening normal documents does not execute regedit.exe and does not exhibit these unusual events. Hence, this particular virus would be trivially detected as an abnormal execution of Word.

Figure 5: A sample system behavior profile when opening a document.

To model the dynamic system events, we also chose to use the Anagram algorithm, but here considering a distinct system event as a distinct token. Hence, the algorithm is used to model n-grams representing the behavior of Word while opening documents. For example, a 3-gram is a contiguous sequence of three system events. In Figure 5, each system event is represented by a long string, which is very space inefficient. To reduce the model space, we tokenize the system event by assigning each a unique index. For example, if we assign "RegCreateKey, hkcu\software\microsoft\office" the index 1, "RegSetValue, hkcu\software\microsoft\office\xp" as 2, "CreateFile, c:\a.reg" as 3,then this sequence of three events is a 3-gram, <1, 2, 3>.

The n-gram system events recorded by opening a set of documents D in the VM is denoted as $g(D,n)$. Therefore, when training a number of documents D_{train}, the model is denoted as $g(D_{train},n)$. In the same fashion that Anagram uses a Bloom filter model, we do not compute the frequency of n-

grams; instead, we only record whether or not the n-grams exist in $g(D_{train}, n)$. When testing the document d_{test}, we collect $g(d_{test}, n)$ and compute the number of $g(d_{test}, n)$ that exist in $g(D_{train}, n)$, denoted as N_{exist}. Finally, the similarity score is the ratio of N_{exist} to the total number of n-grams in $g(d_{test}, n)$, denoted as *T*. That is, $Score = \frac{N_{exist}}{T}$

We implemented a 1-class anomaly detection strategy using the Anagram algorithm; that is, we only train a "benign model." A single model is computed representing all system events recorded when opening thousands of normal training documents. A test document is opened and the system events exhibited are tested against this model. If the test document produces a similarity score lower than a preset threshold, it is deemed anomalous. The detector also checks whether there is any abnormal popup window (e.g., some attacks display popup windows, and Word also has popup information when it encounters exceptions). A timeout is set on the detector in cases where malcode crashes the OS in the VM; the detector will report this time out as a malicious document as well.

In addition to simply opening the document and observing system behavior, we also implemented an automaton that executes the embedded objects in documents. We instrumented AutoIt [35], a script language for automating Windows GUI applications, to simulate user actions such as hot-keys and cursor activities. We use AutoIT to exploit the functions listed on the Word menu bar to select and execute the embedded objects. For example, the object icons on the document can be reached by selecting the *Go To* command in Word (i.e. Ctrl+G) and be executed by clicking *Activate Content* in the *Edit* drop-down list on the menu bar. Moreover, AutoIt can also emulate some simple user actions such as saving and closing the document. In our experiments, we only utilize a few common user operations; a complete exploration of simulating a complete set of human interactions remains an open problem for future research [40, 41].

3.4.2 Document Data Randomization

One of the primary concerns of the dynamic system event detection is the exploitation by mimicry attacks [29, 30]. The approach described in 3.4.1 models sequences of pairs of system operations and the file path, using the Anagram algorithm. This dynamic detector could be evaded if the malcode utilized exactly the same sequences of operations (with arguments) as captured by the benign document model. Such information would generally be available to an attacker by training their own model on other publicly available documents. To counter this mimicry technique, we introduce an approach that modifies possible embedded malcode by randomly changing the data values of certain content. This strategy was inspired by the technique of instruction set randomization (ISR) [42, 43] for thwarting code injection attacks. Whereas ISR randomly maps the instruction set of a program, the strategy we use in SPARSE is to randomize data values in the document that may render any embedded malcode inoperative.

Before detailing the approach, we briefly describe how malcode may be embedded and launched in documents[2]. Word itself may harbor a vulnerability that is easily exploited. The attacker may craft data that exploits the vulnerability which returns an invalid memory address and causes a buffer overflow. Subsequently, Word jumps to a specific location to execute the code which is usually 4 bytes long (i.e. this could be an exception handling function or just a bug). Since not much can be accomplished in a 4-byte code, the attacker has to craft this as a jump that points to another portion of the document in which the malcode is embedded, and this space is either a padding area or a normal data area. In fact, the 4-byte code sequences exploited in the attacks that were available to us for study are also in the data area.

[2] This is different from traditional macro attacks. Since macros are VBA code and are usually located in the "Macros" or "WordDocument" section, identifying malicious macros is easier than the recent exploits of MS Office vulnerabilities

The proprietary Microsoft document format is intricate. In general terms, we may categorize the byte content of a Word document into two types: the data and the pointers to the data (or the length of data). When processing a document, Word first looks for the pointers which are at absolute or relative locations; these pointers tell Word where to find the data that specifies the document rendering function by its type (e.g., text, images, or tables). The data values are either exact data values or information telling Word how to display (e.g., size and style). In addition, there are some "magic numbers" in some cases which are keywords and cannot be changed without crashing Word. For example, Microsoft Office reference schemas [44] are streams that cannot be arbitrarily modified.

Embedding malcode in the pointers is difficult because they are usually short (i.e. from 1 bit to a few bytes) and it is hard to exploit these sections without being fairly easily noticed (e.g., by introducing high entropy values in a certain area or significantly increasing the document size or crashing the system.) As a result, an attacker would likely find it most convenient and safe to embed their malicious shellcode in the data or the padding areas of a document.

To counter this type of attack, we randomly change the data portions to slightly different values, for all of the non-zero data values that can be changed. Specifically, for all of the byte values that can be changed (i.e. neither keywords nor pointers), we randomly increase or decrease by some arbitrary value x (e.g., changing the character "A" to "B"). In our test cases, the value of x ranged from 1 to 3. In many cases, the rendering of the document display will likely be distorted, but this is what we expect. For example, the images in Word documents are placed in the data section. Randomly changing the value of the image content won't break the system but will damage the appearance of the image. On the other hand, stealthy embedded malcode residing in the data portion, if there is any, will also be changed, and subsequently either Word will crash or the malcode will be disabled when an attempt is made to execute it.

For example, the hexadecimal Opcode value "6A" and "EB" represent the *push* and *jmp* X86 instructions, respectively. If the byte values are increased by 1, they become "6B" and "EC" which are not correct Opcodes. Even though sometimes the changed code is valid, it can become another completely unintended instruction. As a result, the program or the OS will not be able to correctly execute the attackers' shellcode and will either crash or terminate the process. Hence, whether the shellcode exhibits obvious or mimicry behavior, our system can detect it by the data randomization process, as long as the increment value x is randomly chosen and kept secret from the attacker.

On the other hand, some normal ASCII data used in the "1Table" section may appear like "Times New Roman" whose corresponding byte values are "54 69 6D 65 73 20 4E 65 77 20 52 6F 6D 61 6E." Probably these are the data that describe the text type to display. Changing any of these values to another random value, including the extended ASCII characters, would never crash Word. The worst case is Word displays a blank page or displays some strange characters. There also exist a few byte sequences that cannot be changed such as keywords [44]. Indeed, some changes may cause Word to be unable to display the content at all, and these documents are considered as FPs. However, Word does not crash in these cases; instead, it displays a message indicating that the document may be damaged. We doubt a true attack would cause this message to be displayed. If the attack were crafted to display this error message (but not crash the system), it would appear before we apply the data randomization process, but not after.

3.5 Malcode Locator

The last component, the malcode locator, is also executed in a virtual machine sandbox. Once a document is deemed malicious by one of the previous detectors, we hone in on the section responsible for the malicious behavior. The locator statically parses the test document into separate sections and tests each of them in the sandbox. However, this method may be flawed – the malcode may not reside in only one single section. For example, there could be a pointer in one section directing Word to execute code in another section. Testing either section alone may not reveal the malicious behavior.

Thus, we introduce an alternative strategy: we remove a section and test the rest together. Assuming a document Doc0 contains three sections Sec1, Sec2, and Sec3, we generate three new documents, Doc1, Doc2, Doc3, and each contains two of the three sections. In other words, each new document has one

section removed from the original document. We then execute the original and the three new documents in the sandbox. If only Doc0 and Doc1 show malicious behavior (e.g., both crash the system, or Doc1 has a similar set of malicious system events to Doc0), then we know that Sec2 and Sec3 contain the malcode. Doc2 and Doc3 behave normally because Sec2 and Sec3 are removed, respectively. After this examination, the labeled malicious section, Sec2 and Sec3, in this case, will be used to update the static malicious model, the blacklist. One may argue that, for example, the pointer section should not be considered malicious because it doesn't harm the system. However, since the malcode will not be executed without this pointer, we consider both as malicious.

In practice, the sandbox setup of the malcode locator is identical to the dynamic detector. They are operated separately so when the malcode locator extracts the malcode of a document, the dynamic detector can test the next one.

In addition to the malcode locator using dynamic behavior, we also evaluated a method to find the malcode using static byte value n-gram entropy analysis. Figure 6 shows an original benign document and its infected version. The left two charts at the top of the display compare the byte sequence values, where the upper one represents the original document and the lower one is the infected version. The right two charts at the top of the display analyze the difference between the two by computing entropy values of the file contents. In this figure, the entropy is the number of distinct byte values within a 50-gram window. Apparently, the tainted portion has high entropy values and is easily distinguished from data appearing in normal text.

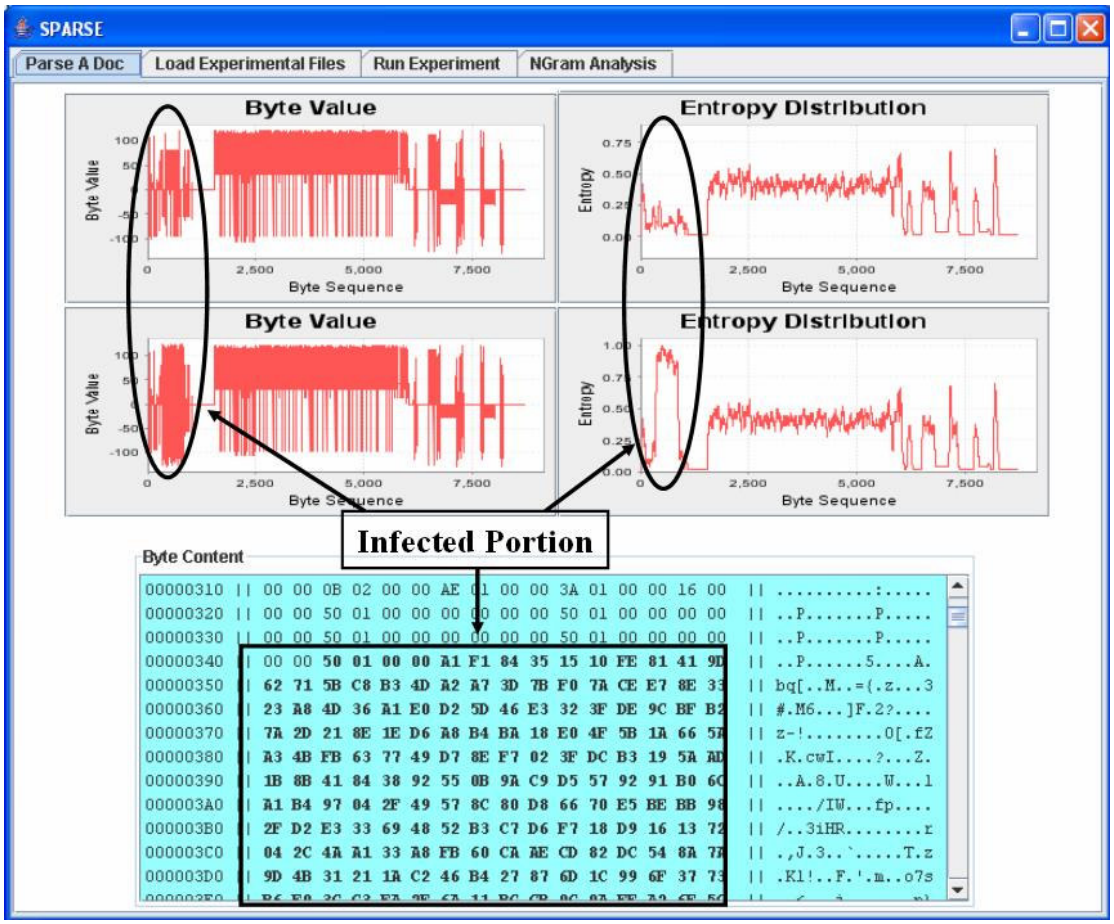

Figure 6: SPARSEGUI screen-shot: parse a benign document compared to its infected version

To evaluate whether examining the entropy value can detect malcode, we performed a 5-fold crossover-validation test of benign and malicious documents. For each test document, we computed the

50-gram entropy. If the document had 50-gram entropy values higher than the threshold, it was deemed malicious. By varying the threshold, we draw a ROC curve shown in Figure 7. Apparently, this method cannot detect malicious documents with an acceptable FP rate. This is because Word document format is complex and contains various types of objects themselves represented with high entropy content. Not only the malicious documents but also the benign documents contain high entropy data. As a result, there is little chance to set a proper classification threshold on either the entire document or a specific section to distinguish benign and malicious documents. We also tested some other n-gram sizes from 20 to 100, and the results were similar. Hence, we conclude that examining entropy value is not appropriate to detect malicious documents.

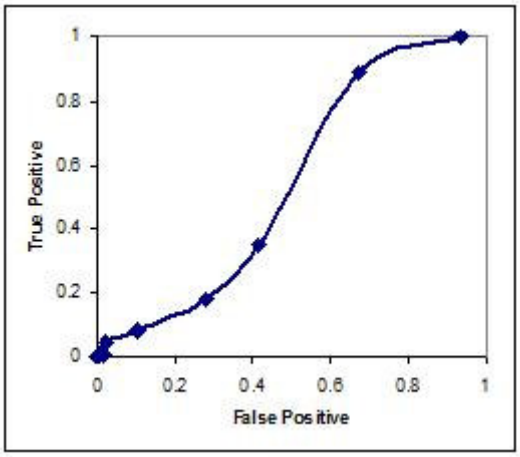

Figure 7: ROC curve of malicious document detection by using entropy value.

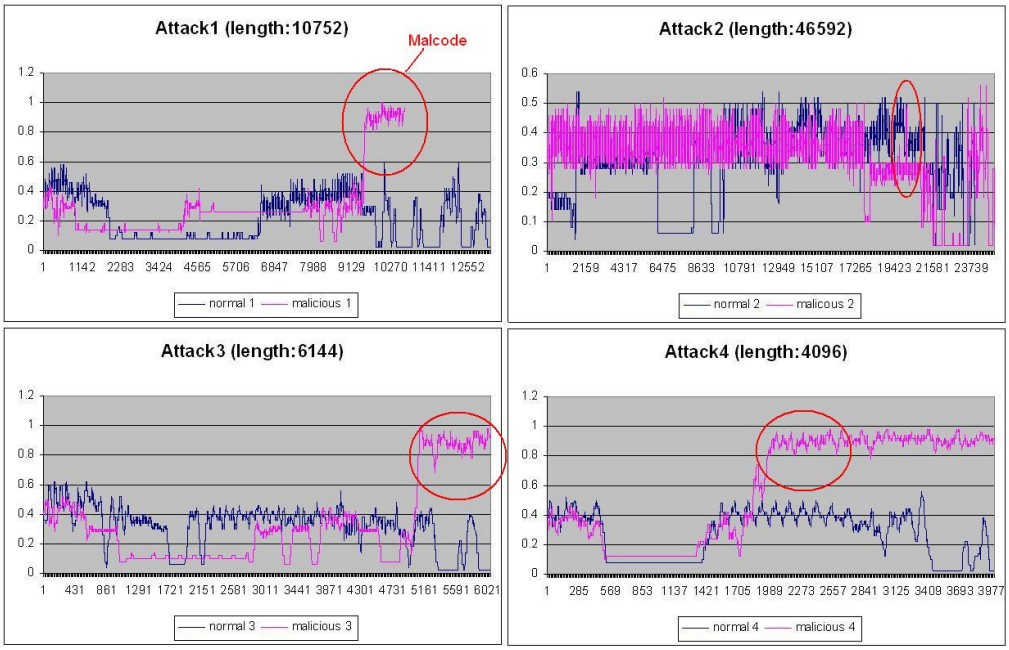

Figure 8: Entropy value of the 1Table section of four known attacks.

However, we can still search for the high entropy area for potential embedded code once a document is labeled malicious[3]. In Figure 8, we computed the entropy value of four known attacks in which we

[3] Currently the examination of entropy values is not integrated in our detection system. This is only utilized as additional information when manually verifying malcode for forensic purposes.

know exactly what and where the malcode is. In this figure, the circled curves are entropy values of malicious documents. All of the malicious contents are located in the high entropy area. For comparison, the lower curves on the same scale are benign documents with the same length.

4. Evaluation

4.1 The Experimental Data

This section describes the data used for our evaluations,as summarized in Table 1. We selected 6228, including both benign and malicious Word documents. The labeled data was provided by external sources. The performance of a statistical detection model is dependent upon the quality of the training data. An insufficient training corpus will lead to a weak model; overtraining the model, on the other hand, may cause false negatives. In other words, collecting and training all possible variations of legitimate documents is likely infeasible, and it is difficult to determine if the training data is sufficient for the statistical model to perform accurate examination. Thus, we generate models for a detector that is biased to a set of data generated by a distinct organization, i.e. we compute group specific models. The assumption is that documents being created, opened and exchanged within a specific group are likely to be similar (through shared templates, or exchange of incremental versions of the same documents, proposals, papers, etc.). Furthermore, we fundamentally assume that the behaviors of malicious documents are significantly different from the behavior of the training corpus. To this end, we collected benign documents publicly posted using *wget* from two specific sites, http://www.columbia.edu and http://www.nj.gov. In addition, we also downloaded malicious documents from VX Heavens [45] and 29 carefully crafted documents with embedded malcode that were provided by a third party for our evaluation.

To demonstrate the ability of SPARSE to detect zero-day attacks, we sorted the documents by the date that they were created and equally split the documents into two groups (i.e. training and testing) before and after a specific date. In the experiments, we selected 2003-11-21, which was a date that roughly split data into two sets with similar sizes. This was a reasonable methodology to simulate the zero-day detection scheme because all of the documents created after 2003-11-21 could be considered as zero-day where new malcode is introduced and never seen in any document from an earlier date.

Category	Malicious Training	Malicious Testing	Benign Training	Benign Testing
# of docs	1449	1498	1715	1516
File size (k byte)	Mean:49kb max:1060kb min:6kb	Mean:49kb max:317kb min:6kb	Mean:105kb max:541kb min:2kb	Mean:148kb max:903kb min:5kb
Collected from	VX Heavens	VX Heavens	From the Web	From the Web
Timeframe	1996-02-08 to 2003-11-21	2003-11-22 to 2006-11-26	1996-12-10 to 2003-11-21	2003-11-21 to 2007-1-24

Table 1: The experimental dataset. Approximately 94% of the documents were created after 2000-01-01.

4.2 Evaluation of Document Data Randomization

To evaluate the data randomization strategy, we tested five "1Table" attacks exercising vulnerabilities known to exist in earlier versions of Word that have since been patched in the most recent versions. In this case we knew exactly what the malicious behaviors were. For each of the test documents, we automatically applied the randomized data method, and subsequently we observed and compared the dynamic behaviors when executing both the test documents and the modified versions in the VM. All of the attacks were successfully detected – the data modification forced a system crash. A summary of the observations is shown in Table 2. We tested three memory corruption attacks, one attack that opened the

calculator (i.e. cale.exe), and an encrypted attack showing a popup window with a "system hacked" message.

Among the three memory corruption documents, Word couldn't display the documents, instead, it either showed a "Your system is low on virtual memory" or ran forever. A sample code is shown in Table 3. This shellcode, which was an infinite loop to corrupt the memory, was embedded in the data area, and by changing any of the byte values, except "01" and "02," would disable the function (i.e. the code would become meaningless). After applying the data randomization method to these three attacks, we forced Word to terminate immediately [4] after opening the documents. It appeared Word found incorrect embedded code and stopped the process. For the other two attacks, the situation was similar – Word was terminated immediately.

	Attack type	Behavior after randomization
Attack 1	Memory corruption	Word was terminated
Attack 2	Memory corruption	Word was terminated
Attack 3	Memory corruption	Word was terminated
Attack 4	Opened system32/calc.exe	Word was terminated
Attack 5	Encrypted malcode	Word was terminated

Table 2: A summary of the tested known attacks

Byte value	Code	Comment
BB XX XX XX XX	mov ebx, XX XX XX XX	XX XX XX XX is the attack
6A 01	push 01	Add argument of the attack
6A 02	push 02	Add argument of the attack
FF D3	call ebx	Call the attack
EB F8	Jmp F8	Jump 8 bytes backward, which is 6A 01

Table 3: The shellcode example of a memory corruption attack.

For comparison, we also performed the same test scenario on 1516 benign documents. Most of them behaved normally after the data randomization process – they did not cause the system or Word to crash; however, we could not apply the strategy to some complex benign documents. Among these 1516 test documents, 35 caused Word to display an error message after applying the data randomization process (i.e. a popup window displayed "Word was unable to read this document. It may be corrupt."). Although there was an error message, the system didn't crash. Since the stealthy embedded malcode would never cause Word to display this message, this could be considered as benign. Nevertheless, in this study, we measured them as false alarms. This does not invalidate the utility of this strategy; rather it demonstrates our lack of sufficient knowledge of the complex binary format of Word documents that inhibits the general application of the technique. We believe those with the deep knowledge of Word formats can apply this method safely to increase the security of Word documents.

Overall, we applied this data randomization process after the dynamic system event detector, and only applied to the labeled negative documents. This technique was used to detect stealthy embedded malcode that otherwise did not reveal any abnormal behavior when examined by the dynamic system event detector.

4.3 Evaluation of the Malcode Locator

For this experiment, we parsed the test documents to create new documents, in which each had a single section removed. We observed and compared the system behaviors produced by executing all of the

[4] This was tested in Windows XP SP2 and Word 2002 with no patches. Other versions of Windows and Word may exhibit different error handling mechanism.

"original" documents to their "modified" versions. The detection scenario was the same as the dynamic detector described in Section 3.4 – we opened the modified test document in the VM and observed the system events. Because all of the test documents here were malicious, if a section was removed and the document behaved benign, the removed section is labeled as malicious. In addition to the 5 documents we tested in 4.2, we also included 13 macro attacks which were either attached in the "Macros" section or in the "WordDocument" section, so there were 18 test attacks in total.

By using the malcode locator, all of the malicious sections were successfully found. In some cases, the malcode was embedded in only one section; in other cases, the malcode appeared not in a single section. For example, a trigger was in the "WordDocument" section, but the malcode was in the "Macros" section. In such cases, both sections were determined malicious. Within the experiments described in section 4.2 and 4.3, we only presented the test of a few malicious documents since these were the attacks that we manually verified the location of the malcode and observed the differences before and after removing or changing data on the test documents.

In our study, the malcode locator only indicated which section was malicious but didn't hone in on the specific pieces of code. We believe we can improve this strategy by combining the technique with the data randomization technique by modifying the section content piecemeal until failure is detected. We leave this as future work.

4.4 Evaluation of the Detection System

In this section, we first evaluate each of the individual detectors and then present the final results of the integrated system. To evaluate the static detector, we trained and tested on all of the 6228 documents and each of the test documents received two similarity scores, one from the benign model and one from the malicious model. A document was labeled benign or malicious according to which score was higher. A model contained five sub-models that represented the "WordDocument," "1Table," "Data," "Macros," and "Others" section. Each sub-model was a 2^{28}-bit (32MB) Bloom filter so the entire model size was 160MB. We chose a large size to ameliorate the inherent False Positive rate of Bloom filters as they saturate.

Table 4 summarizes the relative sizes of the Bloom filter models. The numbers are the ratio of used bits to the Bloom filter size. The used bits on the malicious model was less than the bits on the benign model because we trained the entire file of the benign documents but only trained the malicious portions (i.e. determined by the malcode locator) of the malicious documents. However, the malicious "Macros" sub-model was larger than the benign "Macros" sub-model. Among our datasets, 73% of the malicious documents contained the "Macros" section; on the other hand, only 4% of the benign documents contained the "Macros" section.

While training or testing a document, SPARSE scans through the file and extracts and compares n-grams. The overhead varied depending on the file size. The average time to process a document was approximately 0.23 seconds on a Intel(R) Xeon(TM) CPU 2.66GHz machine with 3.6GB memory, running Fedora 6.

	Malicious model	Benign model
WordDocument	0.98 %	21.47 %
1Table	0.15 %	3.34 %
Data	0.15 %	50.87 %
Macros	2.7 %	0.24 %
Others	0.46 %	12.78 %

Table 4: The proportion of bits used in the model after training documents before 2003-11-21.

After testing, we had a 0.06% FP rate (i.e. only 1 benign document was labeled malicious) and 8.41% FN rate. This was what we expected, since we wanted to minimize the FP rate and leave the documents labeled negative to be tested by the next detector.

For the dynamic system event detector, since we tended to observe all system behavior and to simulate user interaction with the documents, testing a document needed 60 to 80 seconds. In the first 30 seconds, the VM opened the document and waited for Word to display and to execute everything embedded; the rest of the time included waiting for AutoIt to perform some actions and Process Monitor to export the system information.

The first test was to examine the influence of the length of the system event n-gram. Figure 9 shows the ROC curves of a test using different lengths of n. The two charts represent the same results, but the right one is on a smaller scale (the X scale is from 0.02 to 0.1 and the Y scale is from 0.8 to 1.0). Notice that when the FP rates are about 3%, the 1-gram had the highest TP rate[5], which was about 96%. The reason is that most of the tested malicious documents in our dataset caused at least one never before seen system event (i.e. a never before seen system event is referred to a "foreign gram"); however, the benign documents usually did not introduce foreign 1-grams. More specifically, our experiment showed that the documents used and exchanged within a specific group contained similar objects (or code); therefore, Word did not launch unusual system events to display these documents.

On the other hand, higher ordered grams introduced more FPs when the TP rates were the same. Because the higher the order, the more possible permutation of grams we would see. For example, if there were in total N unique system events, there would be N possible 1-grams, N^2 possible 2-grams, N^3 possible 3-grams, and N^4 possible 4-grams. In theory, the range of possible grams grows exponentially. As a result, the benign documents containing objects that were arranged in a "never-been-seen-before order" would introduce anomalous sequences of system events against the high-ordered gram model.

Though 1-gram performed the best in this experiment, mimicry attacks could still be carefully shaped so that no foreign grams were introduced. Therefore, we chose a 4-gram for the rest of the evaluation[6], and set the threshold to 0.994 which had TP rate around 85% as displayed in Figure 9.

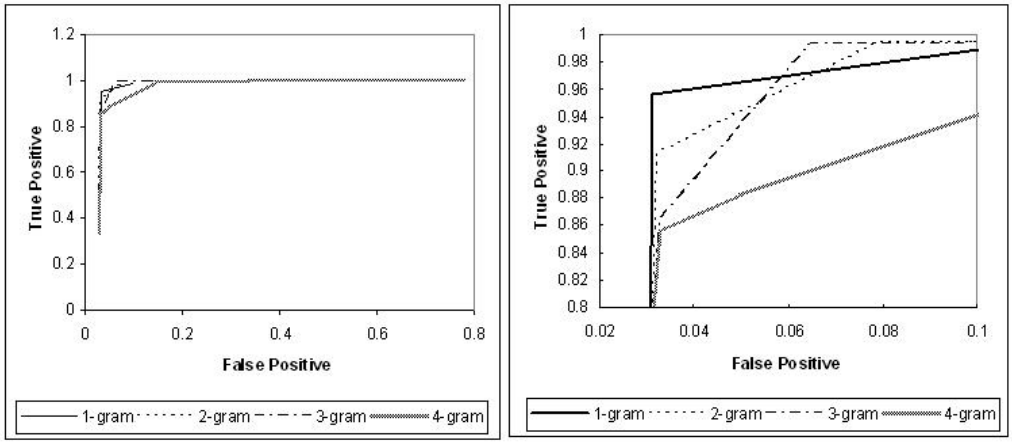

Figure 9: Evaluation of the dynamic system event detector by cross-validation over different n-gram length.

After testing the same scheme we did for the static detector, the dynamic system event detector performed with a 3.1% FP rate (i.e. 47 documents) and 14.1% FN rate. We couldn't avoid a number of

[5] An FN is a malicious file labeled benign, while a TP is a malicious file correctly labeled malicious. In our evaluation, FP + TP = 100% which is the total number of tested malicious documents.

[6] In this paper, we experimented different gram lengths up to 4-gram to evaluate the methods. Others may be interested in evaluating longer n-grams.

FPs because many of them actually behaved abnormal. We verified each of them manually; some appeared like complex forms that consumed too much memory and some even caused exceptions on Word (i.e. a popup window displayed "Word was unable to read this document. It may be corrupt."). These were either recklessly designed or damaged documents, but since the intent was apparently not malicious, we considered them FPs.

For the integrated system, we combined the results of both the static and the dynamic detectors. Since we set thresholds to minimize the FP rate on each detector, we simply used "OR" for the integrated detector. That is, if a document was labeled positive by either detector, it is deemed malicious. Therefore, the overall FP rate was the sum of the FP rate of both detectors, which was 3.16%, and the FN rate, which represented the number of test malicious documents that were labeled negative by both detectors, was 1.06% (16 attacks). Among these FNs, there were three Trojan-Droppers that actually did nothing in the sandbox, a logic attack that only launched on the first day of the month, and several worms that only did a query on the system but did not have any destructive payload.

The overall results are shown in Table 5. Apparently, using the hybrid system (the third row) was superior to any single approach (the first and the second rows). We further tested an incremental scenario to demonstrate that the detection performance benefited from using the information produced by the malcode locator. In this experiment, when testing each test document, if the document was labeled positive by either the static detector or the dynamic detector, we ran the malcode locator to find the malicious sections that were then updated into the static malicious model. The result is shown in the fourth row in Table 5 labeled as tactic 1. The TP rate increased; however, the FP rate was doubled. This was because the dynamic detector produced 3.1% FP rate; these false alarmed documents were somehow damaged but still contained legitimate data (there were false alarms by the dynamic detector but not by the static detector). Updating the signature definition by using these documents, consequently, we introduced legitimate information into the malicious model. To avoid this mistake, we decided to update the model more carefully. We only updated the malicious model if the test document was labeled positive by both detectors; this strategy was referred to "tactic 2." By using this tactic, shown in the last row in Table 5, we increased the TP rate without increasing the FP rate. Nevertheless, we could not avoid a few FNs.

	FP rate	FN rate	TP rate
Only static detector (without updating)	0.06%	8.41%	92.2%
Only dynamic detector	3.1%	14.1%	85.9%
Hybrid system (without updating)	3.16%	1.06%	98.94%
Hybrid system (with updating tactic 1)	7.38%	0.4%	99.6%
Hybrid system (with updating tactic 2)	3.16%	0.73%	99.27%

Table 5: the overall results of testing the system and each component

In general terms, it is difficult to define "malicious intent." For example, should we consider recklessly designed, damaged documents malicious? In our detection system, a malicious activity is actually a set of anomalous system events that have not been seen in the model. Thus, any legitimate but anomalous code will be determined malicious. On the other hand, stealthy malcode which does not produce sufficient amount of anomalous events will not trigger the alarm such as the Trojan-droppers we observed. Detecting malware laden documents remains a hard problem only partially solved by the hybrid detection system described in this paper.

5. Conclusion

Modern document formats provide a convenient means to penetrate systems, since every kind of object or any arbitrary code can be embedded in a document. We presented a study using various approaches to detect embedded malcode in Word documents. We implemented a hybrid detection system that integrates various detectors and detection schemes. First, we presented a static detector that employs statistical

analysis of byte content using the Anagram model with the 2-class detecting strategy. Second, together with an automaton that simulates user interaction with documents and a data randomization method that randomly modifies data values to disable embedded malcode, a dynamic run-time system event detector was presented that is resistant to attacks spread out over multiple sections in a document. We also presented a strategy to hone in on the section harboring the malcode in a malicious document by removing and testing each portion of a document in turn. The overall hybrid detection system was demonstrated to achieve a 99.27% detection rate and 3.16% false positive rate on a corpus with 6228 documents. This hybrid detection system can not only accurately detect malicious documents but can also automatically update its detection model.

Although our detection system could label a document as malicious or benign, the system cannot reveal the intent of any embedded code. An abnormally behaved code could be a new type of legitimate use that is buggy or a recklessly designed object that could harm the system although it was not intended to do so. On the other hand, attacks could be selective and stealthy so there would be no discernible and easily viewable anomalous behavior to be noticed. Hence, there is no easy way to reach the gold standard of 100% detection accuracy, and 0% false positive rate by any analysis alone and do so with a minimum of computational expense, yet. We do not argue that our approach is perfect; however, in this paper, we demonstrate that by combining various detectors and strategies, a hybrid detection system provides improved security to detect malcode embedded in documents.

References

1. Leyden, J.: Trojan exploits unpatchedWord vulnerability. The Register (May 2006)
2. Evers, J.: Zero-day attacks continue to hit Microsoft. News.com (September 2006)
3. Kierznowski, D.: Backdooring PDF Files (September 2006)
4. http://www.microsoft.com/technet/security/Bulletin/MS07-043.mspx
5. http://www.zerodayinitiative.com/advisories/ZDI-06-034.html
6. Bontchev, V.: Possible Virus Attacks Against Integrity Programs and How to Prevent Them. In: Proc. 2nd Int. Virus Bull. Conf. pp. 131–141 (1992)
7. Bontchev, V.: Macro Virus Identification Problems. In: Proc. 7th Int. Virus Bull. Conf. pp. 175–196 (1997)
8. Filiol, E., Helenius, M., Zanero, S.: Open Problems in Computer Virology. Journal in Computer Virology, pp. 55–66 (2006)
9. Wang, K., Parekh, J., Stolfo, S.J.: Anagram: A Content Anomaly Detector Resistant to Mimicry Attack. In: Zamboni, D., Kruegel, C. (eds.) RAID 2006. LNCS, vol. 4219, Springer, Heidelberg (2006)
10. Stolfo, S.J., Li, W.-J., Wang, K.: Fileprints: Identifying File Types by n-gram Analysis. In: 2005 IEEE Information Assurance Workshop (2005)
11. Li, W.-J., Wang, K., Stolfo, S.J.: Towards Stealthy Malware Detection. In: Jha, Christodorescu, Wang (eds.) Malware Detection Book, Springer, Heidelberg (2006)
12. Schultz, M.G., Eskin, E., Zadok, E., Stolfo, S.J.: Data Mining Methods for Detection of New Malicious Executables. In: IEEE Symposium on Security and Privacy, Oakland, CA (May 2001)
13. Shaner: US Patent No. 5,991,714 (November 1999)
14. Abou-Assaleh, T., Cercone, N., Keselj, V., Sweidan, R.: Detection of New Malicious Code Using N-grams Signatures. In: Proceedings of Second Annual Conference on Privacy, Security and Trust, October 13-15, 2004 (2004)
15. Karim, M.E., Walenstein, A., Lakhotia, A.: Malware Phylogeny Generation using Permutations of Code. Journal in Computer Virology (2005) McDaniel, Heydari, M.H.: Content Based File Type Detection Algorithms. In: 6th Annual Hawaii International Conference on System Sciences (HICSS'03) (2003)
16. Goel, S.: Kolmogorov Complexity Estimates for Detection of Viruses. Complexity Journal 9(2) (2003)
17. McDaniel, Heydari, M.H.: Content Based File Type Detection Algorithms. In: 6th

Annual Hawaii International Conference on System Sciences (HICSS'03) (2003)
18. Noga, A.J.: A Visual Data Hash Method. Air Force Research report (October 2004)
19. Natvig, K.: SandboxII: Internet Norman SandBox Whitepaper (2002)
20. Willems, C., Freiling, F., Holz, T.: Toward Automated Dynamic Malware Analysis Using CWSandbox. IEEE Security and Privacy Magazine 5(2), 32–39 (2007)
21. Bellard, F.: QEMU, a Fast and Portable Dynamic Translator. In: proceedings of the USENIX 2005 Annual Technical Conference, pp. 41–46 (2005)
22. Reis, C., Dunagan, J., Wang, H.J., Dubrovsky, O., Esmeir, S.: BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML. OSDI, Seattle, WA (2006)
23. K2. ADMmutate (2001) Available from http://www.ktwo.ca/security.html
24. **Litterbox:** http://www.wiul.org
25. K2. ADMmutate (2001) Available from http://www.ktwo.ca/security.html
26. Detristan, T., Ulenspiegel, T., Malcom, Y., Underduk, M.: Polymorphic Shellcode Engine Using Spectrum Analysis. Phrack (2003)
27. Kolesnikov, O., Lee, W.: Advanced Polymorphic Worms: Evading IDS by Blending in with Normal Traffic. USENIX Security Symposium, Georgia Tech: Vancouver, BC, Canada (2006)
28. Song, Y., Locasto, M.E., Stavrou, A., Keromytis, A.D., Stolfo, S.J.: On the Infeasibility of Modeling Polymorphic Shellcode for Signature Detection Tech. report cucs-00707, Columbia University (February 2007)
29. Kruegel C., Kirda E., Mutz D., Robertson W, and Vigna G.: Automating mimicry attacks using static binary analysis. Proceedings of the 14th conference on USENIX Security Symposium
30. Wagner D. and Soto P. Mimicry attacks on host-based intrusion detection systems: Proceedings of the 9th ACM conference on Computer and communications security, CCS 2002
31. Forrest S, Hofmeyr S, Somayaji A, and Longstaff T.: A sense of self for unix processes. Proceedings of the 1996 IEEE Symposium on Security and Privacy
32. Kruegel C., Mutz D., Valeur F., and Vigna G.: On the detection of anomalous system call arguments. In Europeon Symposium on Research in Computer Security. October 2003
33. Tandon G. and Chan P.: Learning rules from system call arguments and sequences for anomaly detection. ICDM Workshop on Data Mining for Computer Security, 2003
34. Steganalysis http://niels.xtdnet.nl/stego/
35 AutoIt, Automate and Script Windows Tasks: http://www.autoitscript.com/autoit3/
36. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM 13(7), 422–426 (1970)
37. Li W., Stolfo S., Stavrou A., Androulaki E., and Keromytis A.: A Study of Malcode-Bearing Documents. DIMVA, 2007
38. Ferrie P.: Hidan and Dangerous. W32/Chiton (Hidan), Virus Bulletin, March 2007, page 4-
39. Process Monitor: http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx
40. http://www.captcha.net/
41. Chellapilla K, Larson K., Simard P., and Czerwinski M.: Computers beat Humans at Single Character Recognition in Reading based Human Interaction Proofs (HIPs). In Proceedings of CEAS'2005
42. Gaurav S. Kc Angelos D. Keromytis Vassilis Prevelakis: Countering CodeInjection Attacks With InstructionSet Randomization. Proceedings of the 10th ACM conference on Computer and communications security
43. Barrantes E, Ackley D, and Forrest S.: Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. CCS, 2003
44. http://rep.oio.dk/Microsoft.com/officeschemas/SchemasIntros.htm
45. http://vx.netlux.org/