Testing and Evaluation of Virus Detectors for Handheld Devices

Jose Andre Morales, Peter J. Clarke, Yi Deng
School of Computing and Information Sciences
Florida International University
Miami, Fl 33199
{ jmora009, clarkep, deng } @cis.fiu.edu

ABSTRACT

The widespread use of personal digital assistants and smartphones should make securing these devices a high priority. Yet little attention has been placed on protecting handheld devices against viruses. Currently available antivirus software for handhelds is few in number. At this stage, the opportunity exists for the evaluation and improvement of current solutions. By pinpointing weaknesses in the current antivirus software, improvements can be made to properly protect these devices from a future tidal wave of viruses. This research evaluates four currently available antivirus solutions for handheld devices. A formal model of virus transformation that provides transformation traceability is presented. Ten tests were administered; nine involved the modification of source code of a known virus for handheld devices. The testing techniques used are well established in PC testing; thus the focus of this research is solely on handheld devices. The test results produced high false negative rates for the antivirus software and an overall false negative rate of 42.5%. This high rate shows that current solutions poorly identify modified versions of a virus. The virus is left undetected and capable of spreading, infecting and causing damage.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.8 [**Software Engineering**]: Metrics – *performance measures;* D.4.6 [**Operating Systems**]: Security and Protection – *Invasive Software*

General Terms

Measurement, Performance, Reliability, Security, Verification.

Keywords

Anti-virus, malware, black-box testing, virus, worm, handheld, pda, windows mobile, smartphones, windows ce

1. INTRODUCTION

On June 14, 2004, the first computer virus infecting handheld devices was identified [25]. The first virus to infect handhelds[*] running Windows Mobile operating system was released July 17, 2004 [21]. This was the beginning of a new era for the virus and antivirus community. At the time there were little if any antivirus solutions available. An overwhelming majority of users were vulnerable to any possible viral attack. In a reactionary effort, security companies released antivirus solutions for the infected devices that only protected against these specific viruses. Still today many handhelds do not have some form of antivirus software installed.

This research evaluates current antivirus solutions for handhelds with the objective of identifying problems in their detection mechanisms. To achieve this objective we introduce a formal model to represent virus transformations and use the model in the generation of test cases. This model provides detailed traceability of the transformations produced by a virus. The transformed viruses can be precisely ordered by creation time and transformation type. The approach taken was to create test cases that are modifications of an already identified virus and load them into the handheld running the antivirus software. That is, we wanted to test the detection accuracy of the antivirus software against virus modifications. Specifically, the tests were designed with the goal of producing false negatives, which occur when an infected object is not detected as infected, by the virus detectors. Testing virus detectors for production of false negatives has been extensively performed in PCs [1, 26] and is well documented. Therefore this research focuses only on testing handheld devices. A high false negative rate would reveal virus detection weaknesses in the software. The test environment consisted of a Pocket PC running the Microsoft Windows Mobile operating system and the antivirus software. The tested antivirus software is specifically designed for this platform and currently available to the public.

To our knowledge, this research is the first to evaluate current antivirus solutions for the Windows Mobile Platform and for handheld devices in general. The flaws and problems discovered by this research can help lay the foundation for future study and work in virus detection for handheld devices. The results of this

[*] Smartphones and personal digital assistants will be collectively referred to as handheld devices or handhelds throughout this paper.

work can be made public via vulnerability databases, such as the National Vulnerability Database [19]. This research also provides insight on the application of testing methodologies to a new platform in the emerging area of handheld devices. *Currently there is no standard set of test cases for virus detectors on this platform*. Testing related organizations like Eicar.com and av-test.org also have not yet addressed this issue. The test cases created here can be applied to the development of a standardized set of test cases for this platform and these devices.

In the next section we overview the terminology used in the paper. Section 3 describes related work on testing virus detectors. Section 4 describes a formal model for virus transformation and the test categories used to generate the test cases. Section 5 describes the tests we performed and Section 6 our results. Finally we conclude in Section 7.

2. BACKGROUND

1. Computer Viruses: A computer virus is defined as a program that copies a possibly evolved version of itself [26]. Computer viruses have become very sophisticated in detection avoidance, fast spreading and causing damage. A highly populated taxonomy of viruses exists with each classification having its own challenges for successful detection and removal [26]. Today viruses are regarded as a real global threat and viewed as a weapon usable by those bent on creating large scale interruption of everyday life [4, 10].

2. Virus Detectors: The problem of viral detection was studied by Cohen which showed that detecting a virus is not decidable [2]. Many detection algorithms have been presented [24], each with its advantages and disadvantages. Virus detection can be classified as one of two forms: *signature based* and *behavior based* [26]. Signature based detectors work by searching through objects for a specific sequence of bytes that uniquely identify a specific version of a virus. Behavior based detectors identify an object as being viral or not by scrutinizing the execution behavior of a program [23]. Behavior based detection is viewed by many including the authors as key to the future of virus detection [3, 15, 17] because of its ability to detect unknown viruses.

3. Handheld Devices: A handheld device can be described as a pocket sized device with computing capabilities. Two types of handheld devices are relevant to this paper: the personal digital assistant, also called pda, and the smartphone. A pda is used as a personal organizer that includes a contact list, calendar of events, voice recorder, notes, and more. A smartphone can be viewed as a cellular phone integrated with a pda. Both of these types of handhelds share some basic limitations such as: limited screen size, variable battery life, small storage space, operating system installed with limited resources and reduced processing capabilities [8, 27]. These limitations may not allow for antivirus software to be as powerful as those found in desktop PC's. Signature databases and detection functionalities are limited in size and scope. This can possibly result in more viruses being able to easily spread and avoid detection in an environment with weak security. Some handheld device security issues have been previously addressed in [5, 6, 7, 13, 29].

4. Evolution of Virus Detectors: The evolution of virus detectors has moved parallel with the release of viruses in a reactionary manner [12]. As new viruses with new techniques were identified, antivirus researchers rushed to include these new tactics in their software [18, 26]. This evolution has produced a learning curve, with virus authors and antivirus researchers as both teacher and student. Antivirus companies need to develop security solutions for these devices that defend against the types of viruses seen in the past without having to go through the same learning curve for a second time.

5. Software Testing: In this paper we use a black-box approach to test the antivirus solutions for handheld devices. Black-box testing is an approach that generates test data solely from the application's specification [16]. Since the software under test is proprietary, we employ the end-user view of the software as our specification. This specification is *the detection of objects infected with a virus*. There are several techniques used to generate test cases based on the specification of a software system [30]. Two of these techniques are *input space partitioning, and random testing [30]*. Partition testing uses domain analysis to partition the input-output behavior space into subdomains such that any two test points chosen in a subdomain generates the same output value [20]. Random testing involves the selection of test points in the input space based on some probability distribution over the input space [16]. To generate the input data for our test cases we used a combination of input space partitioning and random selection of test points. Due to the limited access to the full specification of the antivirus software, we informally apply partition testing and random testing. We intuitively apply these techniques using the results of previous studies in testing antivirus software.

3. RELATED WORK

This research is motivated by the work done by Christodorescu and Jha [1]. Their research proposed methods of testing malware detectors based on program obfuscation [26]. They used previously identified viruses to test the resilience of commercially available antivirus software for PCs. Christodorescu and Jha address two questions in their work; (1) the resistance of malware detectors to obfuscations of known malware, (2) can a virus author identify the algorithm used in a malware detector based on obfuscations of the malware. The approach they used to answer these questions involved: the generation of test cases using program obfuscation, the development of a signature extraction algorithm, and the application of their methodology to three commercial virus scanners. The results of their work indicated that the commercial virus scanners available for PCs are not resilient to common obfuscation transformations. We use a similar approach to test the virus detection ability for handheld devices. Unlike the work by Christodorescu and Jha [1], we are limited by the number of viruses available for handheld devices. This limitation is based on the fact that virus authors have just only started to write viruses targeting handheld devices. Our experiments use similar transformations on the source code of the malware to generate test cases.

Marx [14] presents a comprehensive set of guidelines for testing anti-malware software in the "real world". Marx claims that many of the approaches used to test anti-malware software in research do not translate into appropriate testing strategies for small business and home office use. He further states that the focus of testing for the real world should be to create tests that are as exact as possible. That is, tests that focuses on on-demand, on-access, disinfection and false positive testing of the anti-malware

software products. Although his article is targeted for data security managers and professional testers, he outlines procedures that should be taken when performing anti-virus software testing in any environment. The work done by Marx [14] was used as a reference guideline for this research. Other relevant research on the subject of testing virus detectors can be found in [9, 11].

4. TESTING AND EVALUATION

In this section we present a formal model for the transformation of viruses and show how this model is used to generate the test cases for our study. Descriptions of each of the five test categories are also given.

4.1 Formal Model of Virus Transformation

As previously stated, a virus is defined as a program that copies a possibly evolved version of itself [26]. A virus $v \in V$ where V is the set of all possible viruses, enters during its execution a transformation stage R where one or more possibly evolved copies of v written v', are produced and copied to some location (see equation(1)). Successful transformation occurs when v' has preserved the original intended execution behavior XB of v (see equation (2)). Thus we have the following:

$$R_i (p_j, v, s) \equiv p_{ij} (v, s) = v' \qquad (1)$$

R_i is the currently running transformation instance. $p_{ij} \in P$ is the specific type of transformation where P = {T, H, B, L, C}, for example B means substitution (see section 4.2 for descriptions of these values). i holds a value representing the number of transformations that have occurred, the current value of i is the ith transformation to have taken place. j holds the value representing the number of times, jth occurrence, a specific transformation type p has occurred, if p = H and j = 3 that means that the transformation type H has been used in 3 transformations up to this point. v is the virus to be transformed. s is an element that provides p the details for a specific transformation. For example if p = B then s may contain the line numbers to substitute and the new lines to use for substitution (see section 4.2 for details of s for each transformation type). v' is the transformed version of v. When R_i occurs, the operation is always independent from every other occurrence of R. The virus v used as input by R is always the same; it is the virus currently executing that invokes R. The output of R, written v', is always a possible evolution of v. The number of v's that is produced is equal to the value of i. In each occurrence of R, the only input that may change is the information held in s. Thus the output v' of two occurrences of R may be the same if s was unchanged in both operations and the same transformation type p was used.

$$\text{If } (XB(v') = XB(v)) \text{ Then } R_i (p_j, v, s) = \text{Success}$$
$$\text{Else } R_i (p_j, v, s) = \text{Failure} \qquad (2)$$

v' can equivalently be written as v_{ijk} where k is the symbol for the transformation type used in a specific transformation R_i. k is added to differentiate the value of j for each transformation type p. This is necessary to illustrate that there are multiple instances of j, one for each transformation type p that is used. Each j has its own value representing the j number of times p has been used. Therefore, if j = 2 and k = C, we know that this is the second time compression is used. Assume virus v has finished one execution of itself. During this execution 5 transformations occurred. The transformation types used were: 1 substitution of source code, 2 compressions, 1 insertion of trash source code and 1 label renaming. Using the notation above, we can formalize this as follows:

$$R_1 (B_1, v, s) \equiv B_{11} (v, s) = v_{11B}$$
$$R_2 (C_1, v, s) \equiv C_{21} (v, s) = v_{21C}$$
$$R_3 (C_2, v, s) \equiv C_{32} (v, s) = v_{32C}$$
$$R_4 (H_1, v, s) \equiv H_{41} (v, s) = v_{41H}$$
$$R_5 (L_1, v, s) \equiv L_{51} (v, s) = v_{51L}$$

We can see from this notation that placing the outputs v' in order of creation is simple. The notation facilitates identifying each virus v' by order of creation and input transformation type. Note that virus v_{21C} and v_{32C} may have been transformed the same or differently from one another. This is, as previously noted, dependent on the information held in s.

A virus detector written D, is a software program meant to detect and remove viruses before infecting a computer system [26]. When detection is complete only one of two outcomes can result. The detection was successful or there was a failure. A successful detection implies the correct identification of a virus infected object O_v. This implies that the object O is infected with a virus v. That is, the sequence of bits representing v is contained within the sequence of bits representing O. Thus v becomes a subsequence of O. The object could be a file, an address in memory, or some other information stored in a computer system. All objects O are assumed non-viral before detection starts. We express this idea as follows:

v *is a subsequence of* O *iff* O *is infected with* v (3)

if v *is a subsequence of* O *then* O *transforms to* O_v (4)

$D(O)$ = *Success implies* v *is a subsequence of* O (5)

A failed detection produces one of two outcomes: a false positive, FP, or a false negative, FN. A false positive occurs when a non viral object is detected as being viral. A false negative occurs when a virus infected object is not detected as being viral. A small amount of false positives is tolerable, but false negatives must be avoided always. Therefore:

$D(O) = FP$ *falsely implies* v *is a subsequence of* O *for some virus* v (6)

$D(O_v) = FN$ D *fails to recognize that* v *is a subsequence of* O *for a specific virus* v (7)

Note (7) assumes that the object is already infected with a virus thus justifying the use of the symbol O_v.

4.2 Test Categories

The test cases generated, using a non-strict approach to input space partitioning and random testing, can be classified in five categories. These are transposition of source code, insertion of trash source code, substitution of source code, label renaming and compression of the virus executable. These categories were chosen due to the facilitation each one gives virus detectors to produce a false negative [1]. These categories are also characteristic of polymorphism [18, 26] and metamorphism [26], powerful techniques used by virus authors. Test case implementations of each category are presented in section 5.2.

1. Transposition of Source Code: Transposition is the rearrangement of statements in the source code. This makes the virus look differently by reorganizing its physical appearance. It still preserves the original intended execution behavior. Transposition can be done randomly or in specific areas. The whole body of the source code or only pieces of it can be transposed as long as the original intended execution behavior is preserved. Applying (1) we have:

$$R_i\,(T_j,\ v,\ s) \equiv T_{ij}\,(v,\ s) = v_{ijT} \qquad (8)$$

where p = T indicates transposition and s provides the line numbers of the source code to transpose. Transposition can result in changing the area of source code that is used as the signature by virus detectors. This is a result of a change in the byte sequence of the executable version of the virus. The transposition can also result in an increase in the byte size of the virus executable. This is due to the addition of commands that preserve the original intended execution behavior. These changes make transposition of source code a possible cause of a virus detector producing a false negative.

2. Insertion of Trash Source Code: This category inserts new code into the original source code. This new code consists of instructions that do nothing to change, alter or affect the intended behavior of the original source code. It does, in some cases, change the byte size of the executable version of the virus. By changing the byte size of the executable, some virus detectors may produce a false negative more easily. This occurs in the case where the detector uses the length of the entire virus as part of the detection process. Thus a change in this length could result in the detector misreading the virus. What the newly inserted code does is inconsequential as long as it does not change the original intended behavior of the source code. Using rule (1) trash source code insertion is expressed as:

$$R_i\,(H_j,\ v,\ s) \equiv H_{ij}\,(v,\ s) = v_{ijH} \qquad (9)$$

where p = H denotes trash insertion. s holds the trash code to be inserted and source code locations of where to insert them.

3. Substitution of Source Code: The removal of lines of source code is replaced with different lines of code. The lines of code used for replacement are not copied from other areas of the code body. The replacement lines can be the same size as the original. They can also be deliberately shortened or lengthened. This is done to manipulate the overall byte size of the virus executable. The lines that are to be replaced cannot be in an area that can disrupt the original intended execution behavior. This implies that this process cannot be random. Careful selection of lines to replace can assure preservation of execution behavior. Applying (1) produces as follows:

$$R_j\,(B_j,\ v,\ s) \equiv B_{ij}\,(v,\ s) = v_{ijB} \qquad (10)$$

p = B specifies substitution and s details which lines to replace and the lines to replace them with. A virus detector can produce a false negative under this category for one of two possible reasons. First, the substituted lines can change the source code used as a signature by the detector for a given virus. Second, as discussed before, if the byte size is not preserved it could cause the detector to identify it as benign. This occurs in cases where the length of the virus is used in detection.

4. Label Renaming: This category involves the substitution of label names in the source code for new names. A label is synonymous with a procedure or function name in a high level language. The label is a pointer to an address space where the instructions to be executed are located. A label therefore points to a set of instructions that are always executed when the label is referenced. The new labels can be kept the same byte size as the original one and also can be purposely changed to a different size. In addition, the corresponding calls to these labels must be updated to ensure original intended execution behavior. The label names chosen for substitution should be those that reference blocks of instructions essential to the virus execution such as: finding a file to infect, opening a file for infection and infecting the file. A virus detector can produce a false negative in this category only when a signature includes a label or a call to a label that has been modified. If no labels are included in the virus signature and the length of the entire virus is not used for detection, the possibility of a false negative is greatly reduced. This category is expressed as follows from (1):

$$R_i\,(L_j,\ v,\ s) \equiv L_{ij}\,(v,\ s) = v_{ijL} \qquad (11)$$

where p = L signifies label renaming and s holds a list of the label names to replace and the new names to replace them with.

5. Compression of a Virus Executable: This category is the compression of the original virus executable. Compression is done by a commercial product or private software belonging to the virus author. The original intended execution behavior is fully preserved. When a virus transforms it can evolve into a new version of itself that is self compressed. This new version makes no modifications to alter the execution as it is originally intended. Virus detectors can produce a false negative under this category by failing to match the virus signature. The compression may create a new byte sequence in achieving an overall byte size reduction. This in turn may cause the source code used for the virus signature to be completely modified and thus detection is almost impossible. Virus compression can be simply expressed as follows:

$$R_i\,(C_j,\ v,\ s) \equiv C_{ij}\,(v,\ s) = v_{ijC} \qquad (12)$$

p = C represents compression and s holds the file name for the compressed version.

5. TEST IMPLEMENTATION

As of the writing of this paper there were only two known viruses for the Windows Mobile platform: WinCE.Duts.A and Backdoor.Brador.A [21, 22]. Of these two viruses we were only able to conduct testing with one of them, WinCE.Duts.A. Though the source code for both of these is readily available to the public [21, 22], Duts is the only one whose available source code can be assembled and executed. The Duts virus consists of 531 lines of source code. This virus was created as a proof of concept code by virus author Ratter formerly of the virus writers group 29A. It exposes some of the vulnerabilities already present in the Windows Mobile platform. It is written in the ARM processor assembly language.

5.1 Testing Environment

Four commercially available antivirus products for handheld devices were tested: *Norton*, *Avast!*, *Kaspersky*, and

Airscanner.com. The handheld device used for testing was a Toshiba 2032SP Pocket PC running Windows Mobile 2002 (version 3.0.11171, build 11178) with full phone functionality provided by Sprint PCS. The central processing unit is the ARM processor SA1110. The Operating System of the PC used was Windows XP service pack 2. Before administering the test cases a control test was given. The original virus was tested for detection to assure each antivirus product properly identified it. *Each of the ten test cases were allowed to fully execute to assure that infection of the system was occurring. Thus showing the original intended execution behavior of the virus had been preserved after modifications was made.*

5.2 Description of Test Cases

The test cases were introduced to the handheld device via the synchronization functionality from a PC. The version used here was Microsoft ActiveSync version 3.7.1 build 4034. The antivirus software performed a complete virus scan with every test. Before testing commenced the antivirus software was checked for updates from the software company's website including the latest virus signature database. Due to the page limit of this paper we are unable to show the complete code listing for the test cases. However, we show relevant segments of code for several test cases.

1. Transposition of Source Code

Test Case 1.1: We took a set of blocks of source code and inserted labels to each of these blocks. The area of the source code chosen for this is the area where the actual file infection takes place, thus assuring probable execution of the transposed source code. Then with the use of branch statements each labeled block branched to the next block in the set thus preserving the original execution order. As a final step, all the blocks were rearranged and taken out of its original physical order. The following is an implementation of this starting at line 308 of the virus source code:

Original Source Code	*Modified Source Code*
ldr r8, [r0, #0xc] *add r3, r3, r8* *str r3, [r4, #0x28* *sub r6, r6, r3* *sub r6, r6, #8* *mov r10, r0* *ldr r0, [r10, #0x10]* *add r0, r0, r7* *ldr r1, [r4, #0x3c]* *bl _align_*	*section19* *ldr r8, [r0, #0xc]* *add r3, r3, r8* *str r3, [r4, #0x28]* *bl section20* *section21* *mov r10, r0* *ldr r0, [r10, #0x10]* *add r0, r0, r7* *ldr r1, [r4, #0x3c]* *bl _align_* *bl section22* *section20* *sub r6, r6, r3* *sub r6, r6, #8* *bl section21*

Test Case 1.2: This involved manipulation of values held in various registers at a given moment during the execution. In assembly language, registers are used extensively to hold values and addresses. The manipulation of these values was done via addition and/or subtraction of a value in a particular register. Moving the value to other registers was also used. The result was an extended piece of source code that took a value, modified it via 2 to 5 instructions and finished by placing back the original value in the original register. This transformation preserved the execution order of the virus and the intended values held in the registers at a given instant in execution. The following is an implementation starting at line 80 of the virus source code:

Original Source Code	*Modified Source Code*
mov r0, r5 *mov r1, r4* *mov lr, pc* *ldr pc, [r11, #-20]* *cmp r0, #0* *bne find_files_iterate*	*mov r0, r5* *mov r1, r4* *add r0, r0, #2* *add r0, r0, #4* *add r1, r1, #6* *sub r0, r0, #6* *sub r1, r1, #4* *sub r1, r1, #2* *mov r4, r1* *mov r5, r0* *mov lr, pc* *ldr pc, [r11, #-20]* *cmp r0, #0* *bne find_files_iterate*

2. Insertion of Trash Source Code

Test Case 2.1: This involved a copy of an original single line of code. The line was pasted back into the source code immediately following the original one. This did not change the behavior because the line of source code chosen consists of the instruction DCB which defines a byte with a string value. This insertion only increased the byte size of the file by the size of the line of code.

Test Case 2.2: In this test, the same instruction as in test case 2.1 was inserted right after five lines of source code. The five lines were not in successive order and deliberately chosen to cover the whole body of the source code. Each chosen line represented an essential part of the execution sequence such as: finding a file to infect and reading the stack pointer. The insertion did not affect the intended execution of the code and increased the file's byte size by length of the insert line multiplied by five.

DCB " just looking "

Inserted after each of the following lines

Line 18 mov r11, sp
Line 64 ldr pc, [r11, #-24] ; find first file
Line 228 cmp r0, #0
Line 303 ldr r6, [r4, #0x28] ; gimme entrypoint rva
Line 361 mov lr, pc

3. Substitution of Source Code

Test Case 3.1: Here we replaced line 514 of the virus source code:

DCB "This is proof of concept code. Also, i wanted to make avers happy."

With

DCB "This is foorp fo tpecnoc code. Also, i wanted to make avers happy."

The substitution preserved the length of the original line while making a modification to a subsection of it. This was done to make a modification that did not affect the byte size of the virus. This substitution did not affect the intended execution of the virus. Finally, it is worth noting that the format of the two lines is indeed identical with respect to spaces and character alignments.

Test Case 3.2: This test is similar to test case 3.1. We replaced the same line 514 of the virus source code with an almost identical one. This new line also had a modification to a subsection of it. The modification was not the same as that of the first test. This modification made the length of the line smaller than the original and thus also decreased the overall byte size. Also the character and space alignment was not preserved. The following is the performed line substitution:

DCB "This is proof of concept code. Also, i wanted to make avers happy."

Changed to

DCB "This is poc code. Also, i wanted to make avers happy."

Test Case 3.3: Here we again substituted line 514 of the virus source code with a new one. The new line of code was maximally modified while still preserving the ability to assemble the source code. The line used for replacement was the same length as the original line but space and character alignment were purposely not preserved. The following is the actual substitution:

DCB "This is proof of concept code. Also, i wanted to make avers happy."

Changed to

DCB "dkfjvd dkfje dkfdsfg kd934,d kdick 3949rie jdkckdke 345r dlie4 vhg"

4. Label Renaming

The labels that were used for substitution were purposely kept the same byte size and also made different sizes in the tests. Also the corresponding calls or branches to these labels were also modified to ensure original execution behavior. The label names chosen for substitution referenced blocks of instructions essential to the virus execution such as: finding a file to infect, opening a file for infection and infecting the file.

Test Case 4.1: This test was a simple reversal of four label names found throughout the source code. The byte size was preserved. Also character alignment was preserved. Two of the labels, appearing in lines 79 and 397 of the virus source code were renamed as follows:

Line Number	Original Source Code	Modified Source Code
79	find_next_file	next_file_find
397	open_file	file_open

Test Case 4.2: In this test, the label names were purposely made longer thus increasing the byte size. In this test the character and space alignment were not preserved. Two of these labels, located at lines 79 and 482 of the virus source code were renamed as follows:

Line Number	Original Source Code	Modified Source Code
79	find_next_file	next_file_to_find_for_use
482	ask_user	user_ask_question_to_continue

5. Compression of a Virus Executable

Test Case 5.1: Compression of the virus executable was done by compressing the executable version of the original virus using commercially available software. The software PocketRAR [28] was chosen for this task. This choice was made based on the experience of using the software and there is a version available for Windows Mobile. The compressed file was placed in the handheld device and opened to view its contents. Then the virus scan was performed. This was done to find out if the antivirus software would not only detect the virus in compressed form but also delete it or at a minimum keep it from executing.

6. TEST RESULTS

Table 1 shows results of applying the tests described above. Column 1 is the test categories. Column 2 is the individual tests in the order described in Section 5. Columns 3 through 6 contain the individual tests results for the antivirus software used in the test executions. The last row shows the false negative rate of each of the software tested. A value of 0 represents detection failure, thus the virus was not detected and deleted and was still capable of execution. A value of 1 represents detection success and deletion of the infected file. A value of 2 denotes successful detection but not deletion, this value was added for the special case of compression. Clearly a value of 0 is a false negative.

Norton had the highest false negative rate with Avast! having the lowest. Not including scanning the original virus, a total number of 40 tests were performed. Of these, 23 tests were successful detections, leaving 17 as failures. This is an overall 42.5% false negative rate, very high and unacceptable. In the test for compression of source code, a special note should be taken regarding the behavior of the virus. The compression software apparently creates a temporary copy of the contents of a compressed file when the files are viewed. The virus scan detects and deletes this temporary copy, however, the original virus file can still be executed from within the compressed file view. Thus the compression software does not allow the antivirus to delete

Table 1 Virus scanner test results and false negative percentage by software					
		Norton	Avast!	Kaspersky	Airscanner.com
Original virus		1	1	1	1
Transposition	Test 1.1	0	1	0	0
	Test 1.2	0	1	1	0
Trash Insertion	Test 2.1	0	1	1	1
	Test 2.2	0	0	0	0
Substitution	Test 3.1	1	1	1	1
	Test 3.2	0	1	1	1
	Test 3.3	1	1	0	0
Label Renaming	Test 4.1	1	1	1	1
	Test 4.2	1	1	1	1
Compression	Test 5.1	2	2	2	2
False Negative %		60%	20%	40%	50%

Table 2 False negative percentage by individual test and category					
		Successful Detection	Failed Detection	Per Test False Negative %	Test Category False Negative %
Transposition	Test 1.1	1	3	75%	62.50%
	Test 1.2	2	2	50%	
Trash Insertion	Test 2.1	3	1	25%	62.50%
	Test 2.2	0	4	100%	
Substitution	Test 3.1	4	0	0%	25%
	Test 3.2	3	1	25%	
	Test 3.3	2	2	50%	
Label Renaming	Test 4.1	4	0	0%	0%
	Test 4.2	4	0	0%	
Compression	Test 5.1	0	4	100%	100%

the contents of a compressed file. We count this as a failure because the virus is still in the handheld device, even though it was detected, and can still be executed. Table 2 shows false negative rates with columns 1 and 2 similar to Table 1, Columns 3 and 4 shows successful and failed detections, and Columns 5 and 6 show false negative rates by individual test and test category.

Compression had the highest false negative rate followed by transposition of source code and insertion of trash source code. In the individual test results, the second test of trash insertion caused all the antivirus software to produced false negatives. Yet the first test only caused one false negative. This shows the insertion of trash source code within actual lines of instruction code is enough to cause the detector to incorrectly identify the file as viral. The transposition test category, the first test caused the most false negatives. The insertion of branch statements in the source code results in a different physical appearance while maintaining the same execution behavior proved to be very effective in avoiding detection.

In the substitution of source code category the false negative produced in test two hints that a slight decrease in the byte size of the virus executable may cause the virus to go undetected. In test three of the same category, we purposely made space and character alignments different than the original line of source code while keeping the byte size the same which caused some false negatives to occur.

In the label renaming category preserving and purposely changing the byte size of the labels did not affect the virus detectors. This implies that changing the byte size may have the affect of avoiding detection if the byte size reduction is done in certain areas of the source code. Also one can infer that labels may not be used by the virus signatures. When a byte size reduction causes a false negative, the modified area might be of critical importance to the detector deciding if the code is viral or not. During the test case creation, we were not aware if the signature used by a detector was modified. Many of the successful detections could have occurred because the transformation did not affect the virus signature. Overall, with a 42.5% false negative rate, there is clearly room for improvement.

7. CONCLUSION

We have presented a technique of testing handhelds based on a formal model of virus transformation. The results show multiple

flaws in current virus detectors for handheld devices. The tests led to high false negative rates for each antivirus product and an extremely high overall false negative rate of 42.5%. These results suggest that current virus detectors are purely simple signature based detection. The formal model shows how detailed traceability of the virus transformations can be done. Future work includes the detailed study of false negative productions in any of the given tests. Byte size changes, substitution and transposition of source code and compression require further study to improve virus detection under these conditions. Currently we have a great archive of knowledge of viruses for PCs. This information can be used to produce sophisticated virus scanners for handheld devices given their limitations. Ideally, this will occur expeditiously and preemptively to help avoid infections of future viruses for handheld devices.

8. ACKNOWLEDGEMENTS

This was supported in part by the National Science Foundation under Grant No. HRD-0317692. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements either expressed or implied by the above agencies. The authors thank Gonzalo Argote-Garcia, Konstantin Beznosov and Mihai Barbulescu for their contributions to this research.

9. REFERENCES

[1] Christodorescu M. and Jha S. Testing Malware Detectors. International Symposium on Software Testing and Analysis (ISSTA) 2004.

[2] Cohen F. A short course on computer viruses. Wiley Professional Computing, 1994.

[3] Conry-Murray A. Behavior Blocking Stops Unknown Malicious Code. Network Magazine, June 2002.

[4] Denning D. Cyberterrorism. Testimony before the Special Oversight Panel of Terrorism Committee on Armed Services, US House of Representatives, 23 May 2000.

[5] Fogie S. Pocket PC Abuse: To Protect and Destroy. Black Hat USA 2004

[6] Foley S. and Dumigan R. Are Handheld Viruses a Threat? Communications of the ACM, January 2001, Vol. 44, No. 1.

[7] Ford R. The Wrong Stuff?, IEEE Security & Privacy, 2004.

[8] Francia G. Embedded System Programming. Journal of Computing Sciences in Colleges, Dec 2001, Vol. 17 Issue 2.

[9] Gordon S. and Howard F. Antivirus Software Testing for the New Millennium. Proceedings of National Information Systems Security Conference (NISSC) 2000.

[10] Gordon S. and Ford R. Cyberterrorism?. Symantec Security Response White Paper, 2003.

[11] Gordon S. and Ford R. Real World Anti-Virus Product Reviews and Evaluations - The current state of Affairs. Proceedings of the 1996 National Information Systems Security Conference.

[12] IBM Research. Virus Timeline. http://www.research.ibm.com/antivirus/timeline.htm.

[13] Mackey D., Gossels J. and Johnson B.C. Securing your handheld devices. The ISSA Journal, April 2004.

[14] Marx A. A guideline to anti-malware-software testing. European Institute for Computer Anti-Virus Research (EICAR) 2000 Best Paper Proceedings, pp.218-253.

[15] Messmer E. Behavior blocking repels new viruses. NetworkWorldFusion, January 28, 2002.

[16] Myers G. J. The Art of Software Testing. John Wiley & Sons, second edition, June 2004.

[17] Nachenberg C. Behavior Blocking: The Next Step in Anti-Virus Protection. Security Focus, March 19, 2002. http://www.securityfocus.com/infocus/1557.

[18] Nachenberg C. Computer Virus-Antivirus Coevolution. Communications of the ACM, January 1997, Vol. 40 No. 1.

[19] National Vulnerability Database. http://nvd.nist.gov/.

[20] Ntafos S. C. On random and partition testing. In Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis 1998 (ISSTA'98) pp. 42-48, Clearwater Beach FL, Mar 1998. ACM Press.

[21] Peikari C., Fogie S. and Ratter/29A. Details Emerge on the First Windows Mobile Virus,. http://www.informit.com/articles/article.asp?p=337069.

[22] Peikari C., Fogie S., Ratter/29A and Read J. Reverse Engineering the first Pocket PC Trojan. http://www.samspublishing.com/articles/article.asp?p=340544.

[23] Schneider F. Enforceable Security Policies. ACM Transactions on Information and System Security. Vol. 2, No. 1, February 2000, pages 30-50

[24] Singh P. and Lakhotia A. Analysis and Detection of Computer Viruses and Worms: An Annotated Bibliography. ACM SIGPLAN Notices, February 2002.

[25] Symantec Antivirus Research Center. http://securityresponse.symantec.com/avcenter/venc/data/symbos.cabir.html

[26] Szor P. The Art of Computer Virus Research and Defense, Addison-Wesley, 2005.

[27] Vahid F. and Givargis T. Embedded System Design a Unified Hardware/Software Introduction. Wiley 2002.

[28] WinRAR, http://www.win-rar.com/.

[29] Wireless Handheld and Smartphone Security, Symantec Security White Paper, http://www.symanctec.com.

[30] Zhu H., Hall P. A. V. and May J. H. R. Software Unit Test Coverage and Adequacy. ACM Computing Surveys, 29(4), pp. 366 – 427,1997. ACM Press