Generic Unpacking Techniques

Komal Babar
Department Of Computer Science
Military College of Signals
National University of Science and Technology
Pakistan
komal@mcs.edu.pk

Faiza Khalid
Department Of Computer Science
Military College of Signals
National University of Science and Technology
Pakistan
Faiza@mcs.edu.pk

***Abstract*—**Traditional signature-based malware detection techniques rely on byte sequences, called signatures, in executable for signature-matching. Modern malware authors can bypass signature-based scanning by employing the recently emerged technology of code obfuscation for information hiding. Obfuscation alters the byte sequence of the code without effectively changing the execution behavior. A commonly used obfuscation technique is packing. Packing compresses and/or encrypts the program code. Actual code stays hidden till runtime (when the executable is unpacked) making it immune to static analysis. Since every packer has its associated unpacker to undo packing, a successful generic unpacker is difficult to come by. A few automated unpacking techniques have been published so far that attempt to unpack packed binaries without any specific knowledge of the packing technique used. In this paper, we aim to provide a comprehensive summary of the currently published prevalent generic unpacking techniques and weigh their effectiveness at dealing with the spreading nuisance of packed malware.

Dynamic analysis is a promising solution to the packing problem as every packed binary has to inevitably unpack itself for execution. Emulation (running code in a virtual environment) is an effective and powerful technique for generic unpacking. We will be reviewing various unpacking techniques based on emulation and a few other hybrid and alternative approaches.

Index Terms – obfuscation, generic unpacking, malware, dynamic analysis, emulation, virtual machines

I. INTRODUCTION

One of the most prevalent features of modern malware is obfuscation. Obfuscation is the process of modifying something so as to hide its true purpose. Obfuscation increases the complexity of a program to make reverse engineering harder. Three of the most important practical obfuscations are packing, code reordering, and junk insertion. This paper only discusses packing, which is the most commonly used anti-reverse engineering technique. The packing obfuscation replaces a binary (code and data) sequence with a data block containing the binary sequence in packed form (encrypted or compressed) and a decryption routine that, at runtime, recovers the original binary sequence from the data block. The result of the packing obfuscation is a program that dynamically generates code in memory and then executes it. There are a large number of tools available for this purpose commonly known as executable packers [29, 30, 31, 32, 33]. Packing describes the process of encrypting a program and adding a runtime decryption routine to it, such that the behavior of the original program is preserved. Programs obfuscated by packing consist of a decryption routine (an instruction sequence that generates code and data), a trigger instruction that transfers control to the generated code, an unpacked area (the memory area where the generated code resides), and a packed area (the memory area from where the packed original binary is read) [10]. Packers embed an unpacking stub into the packed program and modify the program entry point to point to the unpacking stub. When the packed program executes, the operating system reads the new entry point and initiates execution of the packed program at the unpacking stub. The purpose of the unpacking stub is to restore the packed program to its original state and then to transfer control to the restored program. Packers vary significantly in their degree of sophistication. The most basic packers simply perform compression of a binary's code and data sections. More sophisticated packers not only compress, but also perform some degree of encryption of the binary's sections. Finally, many packers will take steps to obfuscate a binary's import table by compressing or encrypting the list of functions and libraries that the binary depends upon. In this last scenario, the unpacking stub must be sophisticated enough to perform many of the functions of the dynamic loader, including loading any libraries that will be required by the unpacked binary and obtaining the addresses of all required functions within those libraries [1].

Packing is applied on legitimate software to reduce the size of executable files and to protect the intellectual property that is distributed with the code. Malware writers use packing to bypass signature-based detection as packing completely modifies the binary foot-print of a program. The malicious code resides in the executable file in an encrypted form, and is not exposed until the moment the executable is run. A static analysis of a packed program will view the obfuscated block as non-instruction data or omit its analysis entirely, thereby hiding the program's true intentions. The percentage of new malware that is packed is on the rise, from 29% in 2003 to 35% in 2005 up to 80% in 2007. This situation is further made complex by the ease of obtaining and modifying the source code for various packers. Modifications to the source code can introduce changes in the compression or encryption algorithm, create multiple layers of encryption and/or add protection against reverse engineering. Currently, new packers are created from existing ones at a rate of 10–15 per month [21]. According to WildList 03/2006, over 92% of malware file out there are runtime packed. Only 54 out of 739 files are not packed [17].

Unpacking is the recovering of the original program that has the same relevant behavior as the packed program. Unpacking consists of constructing a program instance which contains the embedded program, contains no code-generating routine, and behaves equivalently to the self-generating program [12]. Malware authors understand that analysts will attempt to break through any obfuscation, and as a result they design their malware with features designed to make de-obfuscation difficult. De-obfuscation can never be made truly impossible since the malware must ultimately run on its target CPU; it will always be possible to observe the sequence of instructions that malware execute. In all likelihood, the malware author's goal is simply to make analysis sufficiently difficult that a window of opportunity is opened for the malware in which it can operate without detection [1]. Dedicated decryption routines can be developed to detect any packed virus but writing such a routine requires that the virus be analyzed completely. A thorough analysis of the malware and then developing and testing a specific decryption routine could take a lot of work and time to accomplish. Moreover, dedicated routines fail to detect modifications to the packing routine. Generic unpacking attempts to unpack obfuscated binaries without determining the exact packing technique used to pack the program. In this paper, we intend to identify, compare and contrast various existing generic unpacking techniques and highlight their strengths and weaknesses.

II. Detecting Packing

It can be useful to first detect that an executable has been packed using some encryption or compression technique before setting to the task of unpacking it generically. One of the heuristic methods used for packer detection is to see how the byte distribution (entropy) is changed by the packers as well as check import tables of the executable under observation [8]. Analyzing byte distribution involves determining the frequency of occurrence of the byte distributions of the file contents. Such a frequency analysis is advantageous in detecting compressed data as effective compression techniques tend to increase the entropy of byte distributions in the file. This is done without unpacking data in the file from its compressed form and therefore helps in detecting compressed files without actually decrypting its contents which otherwise would make the system vulnerable to potentially malicious executables [9]. However, this technique alone cannot be used as a criterion to identify a packed executable as legitimate copyright protected software also use packing for information hiding of an executable to evade disassembly of binary code and reduce size for distribution over the Internet.

III. Dynamic Analysis Techniques

Once the packed executable has been detected, the executable then needs to be unpacked correctly. Dynamic analysis is a promising answer to the problem of hidden code extraction because it does not depend on signatures. Dynamic analysis techniques make use of the fact that no matter what packing technique is applied to the executable, the actual code or its equivalent will ultimately be available in memory and sooner or later, it will execute at some point at run-time. This innate property of a packed executable is the key to extracting the hidden binary code or its equivalent as a raw memory dump. However, it is not certain where the hidden binary code lies in the memory and when the execution flow jumps to the hidden code. Apart from this, another essential piece of information for analysis of an executable is the original entry point (OEP). *The original entry point is the first hidden instruction being executed when the program control flow is transferred from the decryption/unpacking routine to the hidden code* [18]. Dynamic analysis is less susceptible to being tricked by the use of obfuscation or self-modifying code. When using dynamic analysis techniques, the issue arises in the choice of environment in which the sample should be executed. The use of a sacrificial lamb (*a dedicated standalone machine that is reinstalled after each dynamic test run*) is not an efficient solution because of the overhead involved. In addition to determining the type of environment to be used for dynamic analysis, one can also discern the different types of information that can be captured during the analysis process [15].

A. Run and Dump Unpacking

Generally, a packed program upon executing first unpacks the program in memory, loads the required libraries, and accesses the imported functions by scanning the import address table of the executable. The structure of the original program is exposed and a snapshot of the memory image can be taken at this point and stored in a file (called dumping). This file can then be analyzed for signature analysis by virus scanners. The advantage of this technique is that the critical and complex task of unpacking is done by the stub itself. The paradox of this technique is figuring out exactly when the snapshot of the memory should be taken. If the snapshot of the memory is taken prematurely (before the program has been completely unpacked) the entire hidden code will not be obtained for analysis. And if the snapshot is delayed for too long, the program will start executing the unpacked code, making the system vulnerable to malicious attacks [1]. Furthermore, the dumped executable requires some additional fixing of header structures, but the code itself is visible in its original form and available for reverse engineering and static analysis. This simple method works for most kinds of executable packers and encryptions, as the unpacking function typically extracts the complete program right at the start, and does not interfere with later computations (but this is not always the case). The biggest drawback of this method is that the executable must be loaded, which might not be acceptable in all cases as it cannot always be guaranteed that the program is terminated before any malicious function is called. Sandbox environments can be used to avoid the potential damage [11]. Sandbox is a virtual environment provided to the executables to run, so that they cannot exploit the actual system while they are being analyzed [2]. *Sandboxes are usually found in a kid's playground. Kids use it to play in, building and tearing down structures. A sandbox inside a scanner engine is also a playground – for computer files* [19]. Sandboxing can be performed in two ways: Sandboxing using Virtual Machine-where the executable runs on a subset of the

actual system in a constrained controlled environment and Sandboxing using Emulation-where the sandbox is a virtual world where everything is emulated drawing a concrete wall between the real and the emulated environment.

B. Virtual Machine

Running the executable in a virtual machine (i.e. a virtualized computer), such as one provided by VMware [24], is a popular choice. In this case, the malware can theoretically only damage the virtual PC and not the real one. After performing a dynamic analysis, the infected hard disk image is simply discarded and replaced by a clean one (i.e., so called *snapshots*). Most (or all) code is run directly in an isolated hardware environment which can be done using software solutions (VMware) or using hardware features (new Intel/AMD processors, IBM z/VM). It requires support from the OS or kernel-level modifications (drivers). In virtualization, code is not (usually) analyzed or cached. It is just run in an isolated environment [27]. Virtualization solutions are sufficiently fast. There is almost no difference to running the executable on the real computer, and restoring a clean image is much faster than installing the operating system on a real machine. The running code inside of a self contained environment can be more closely controlled than raw hardware [15]. Unfortunately, a significant drawback is that the executable to be analyzed may determine that it is running inside a virtual machine and may become inactive or execute differently in order to evade the virtual setup. All current virtual machines exhibit identifiable features and detecting them is one of the most common methods available to a malware author to protect malicious code from analysis [5].

C. Emulation

A PC emulator is a piece of software that emulates a personal computer (PC), including its processor, graphic card, hard disk, and other resources, with the purpose of running an unmodified operating system [15]. Generic code emulation is a very potent de-obfuscation technique. A virtual machine is implemented to simulate the CPU and memory management systems to impersonate the code execution. The packed executable is replicated in the virtual environment and no actual code (which may contain malicious content) is executed by the real processor. The purpose of the code emulation is to mimic the instruction set of the CPU using virtual registers and flags. It is also important to define memory access functions to fetch 8-bit, 16-bit, and 32-bit data. Furthermore, the functionality of the operating system must be emulated to create a virtualized system that supports system APIs, file and memory management [2]. All the hardware resources are virtualized using data structures.

When the packed executable runs in the emulator, each instruction triggers some software routines that update the respective data structures in such a way that the program gets the same response it would get if run on an actual processor. Each instruction is first decoded to find the instruction type, length, operands and other information that need to be updated. Once the necessary information is available, the respective emulation routines are called which update any emulated hardware resources (which are actually data structures), if required. The address of the next instruction is obtained either as result of instruction decoding or computed by the emulation routine.

Generally, Program's Entry Point marks the beginning of emulation which executes instructions sequentially. Complexity of the unpacking process posses little difficulty to the emulation environment provided that the unpacking stub is available in the executable and the emulator has enough resources to complete unpacking. Since a program would use only a small subset of all the resources of system, it is quite affordable to provide emulator with these resources.

On the other hand, code emulation is significantly slower than running the code on an actual processor. Decoding a single instruction in a program requires hundreds of instructions to be executed at the back-end. After instruction decoding, several routines are called for updating data structures, find where the next instruction lies in memory, followed by many other steps to simulate a correct response. All these factors make emulation considerably slower and inefficient. This difference in speed of execution is one of the very strong tools used by anti-emulation techniques which could be embedded inside an executable being analyzed. However, such anti-emulation technique can as easily be fooled by emulator by providing incorrect clock readings so that the system appears faster to the program.

Another problem is that, for some packers, the program is not completely unpacked at one time, or some parts of its code may have been moved around by the packer. Also, if the emulation engine does not emulate correctly, error tracing becomes very complex since emulator executes a lot more instructions than an actual processor. One of the inherent problems of dynamic analysis techniques is deciding when to stop the emulation process. Heuristic checks can be used to help make this decision [16]. Occurrence of an event, which may be statically defined, could be used to stop emulation. In addition, some form of resource exhaustion limit, for example number of emulated instructions or emulation time, is needed in order to avoid infinite-loop execution [6]. Some other common ways to trick emulation is using fake API calls, using complex program logic (which can greatly slow down the emulation process). Since one of the major weaknesses of emulation is its speed, the goal of anti-emulation techniques is that the emulator quits without finishing the unpacking process (usually a maximum time-out is predefined in emulator) [25].

D. Difference between Emulators and Virtual Machines

It is important to differentiate emulators from virtual machines. Like PC emulators, Virtual Machines can run an unmodified operating system, but they execute a statistically dominant subset of the instructions directly on the real CPU. This is in contrast to PC emulators, which simulate all instructions in software. Because all instructions are emulated in software, the system can appear exactly like a real machine to a program that is executed, yet keep complete control. Thus, it is more difficult for a program to detect that it is executed inside a PC emulator than in a virtualized environment. A PC emulator has complete control over the sample program. It can intercept and analyze both native Windows operating system calls as well as Windows API calls while being invisible to

malicious code. The complete control offered by a PC emulator potentially allows the analysis that is performed to be even more fine grain [15]. On the other hand virtual machines are much faster than PC emulation; they are almost up to the native speed of the system being used.

E. Unpacking Methods Using Emulation:

1. MALWARE NORMALIZATION

The method for malware normalization attempts to unpack malware generically. Unpacking by malware normalization consists of two basic steps. First, execute the program in a controlled environment to identify the control-flow instruction that transfers control into the generated-code area. i.e. execute the program in an emulator, collect all the memory writes (retaining for each address only the most recently written value) and monitor execution flow. If the program attempts to execute code from a memory area that was previously written, capture the target address of the control flow transfer (i.e., the trigger instruction) and terminate execution. By emulating the program and monitoring each instruction executed, the moment when execution reaches a previously written memory location can be identified. Second, with the information captured in the previous step, construct a normalized program that contains the generated code. In the second step, construct a non-self-generating program. Using the captured data, an equivalent program can be constructed that does not contain the code generator. The data area targeted by the trigger instruction is replaced with the captured data. The memory write captured contain both dynamically generated code and the execution specific data e.g. the state of the program stack and heap. The executable file of the new program is a copy of the executable file of the old program with the byte values in the virtual memory range set from the captured data. The program location where execution was terminated is used as the entry point for the new program.

This technique has some major drawbacks. The unpacked executable is not ready-to-run. Although the packed code can be successfully unpacked and produced in the normalized executable, but since its not the actual file, the import table which lists the dynamically linked libraries and API calls used by the program may not be recovered, since most packing obfuscations replace it by a custom dynamic loader. This approach is open to resource consumption attacks and can have false negatives since the execution time in the sandbox often has to be heuristically restricted for performance reasons.

2. RENOVO

The Renovo emulation technique is a fully dynamic method which monitors currently executed instructions and memory writes at run-time. Renovo uses an approach similar to malware normalization but with a few customizations. The approach maintains a shadow memory of the memory space of the analyzed program, observes the program execution, and determines if newly generated instructions are executed. Then it extracts the generated code and data. Assuming nothing about the binary compression and encryption techniques, it provides a means to extract the hidden code and information, which is robust against anti-reverse-engineering techniques.

After the packed executable starts, its attached decryption routine performs transformation procedures (also called hidden layers) on the packed data, and then recovers the original code and data. After this, the decryption routine sets up the execution context for the original program code to be executed. This involves initializing the CPU registers and setting the program counter to the entry point of the newly-generated code in memory.

A packed executable may have multiple hidden layers, making it even more difficult to analyze. But irrespective of the packing method and the hidden layers, the original program code and data will ultimately be available in memory. Also, the instruction pointer should jump to the OEP (Original Entry Point) of the restored program code which has been written in memory at run-time. Making use of these properties of packed executables, an algorithm to dynamically extract the hidden original code and the OEP from the packed executable has been suggested in [18] which examines whether the current instruction has been generated at run-time, after the program binary was loaded.

The advantages of this approach are threefold: Firstly, nothing is assumed about the packing methods except the inevitable fact that the original hidden code should eventually be written and executed at run-time. Therefore, the approach is able to handle any sort of packing techniques applied to the binaries. Secondly, the approach can determine the exact memory regions accommodating the code or data generated at run-time. Since the information about memory writes are kept at byte-level, it is possible to efficiently extract the newly-generated code and data. Lastly, this approach does not rely on any information on the code and data sections of the binary.

When analyzing an executable, it is run in an emulated environment. The emulated environment facilitates simulating CPU instructions in a fine-grained manner, in particular the instructions that perform memory writes. This technique also suffers from the weaknesses of emulation and can easily be evaded by anti-emulation techniques. Packers also use anti-memory dumping technique that involves the deletion of a section of code immediately after it is executed.

IV. HYBRID APPROACHES

Detection of packed malware requires the use of emulation and sandboxing technologies which are open to resource consumption attacks since the execution time in a sandbox has to be restricted for performance reasons [10].

If an executable is packed with a complex packer, it is quite useful to use a combination of emulation and specific unpacking routines for known packer. Although using specific routines is an efficient approach but integrating it with emulation introduces additional complexity to the system. [6].

As emulation based unpacking is too slow and static unpacking is too specific, a hybrid approach is a solution which combines the advantages of both. The static unpackers do not handle heavily polymorphic code. It is handled by slow generic emulation instead. Advantage of this technique is that it is universal by generic emulation and fast as the specific unpacking is independent of the code being emulated. It is

easily expandable and can be implemented and deployed incrementally, as initially small number of specific unpacking routine are written and with time, the results of emulations can be used to widen the scope [14].

1. PolyUnpack

Reference [22] gives a hybrid unpacking approach called PolyUnpack. It is a behavior-based approach that uses a combination of static and dynamic analysis to automate the process of extracting the hidden-code of packed malware. The core emphasis of this technique is on the results (i.e., runtime-generated code execution) of unpack-execution rather than the unpacking mechanism used. It supercedes other approaches as prior knowledge about the packer or explicit programming of the semantic behavior capturing all instances in an unpacking class is not required. It first generates a static code view of the packed program (i.e. the code sequence of program does not produce any code at run-time) in memory using static analysis. The static code model is then forwarded to dynamic analysis engine (i.e. emulation or any virtual environment). This dynamic analysis differs from the usual run-time analysis as it has the ability to verify if the observed sequence of execution matches any part of the static model. While the stub (restoration routine) is being executed, the code will follow the same sequence as the stub's static view. After execution of each instruction, the execution context is compared to the static model. The point in run-time when the code deviates from the static view indicates that the code has been unpacked (since unpacked code sequence is not available in static model). This is the stopping condition of analysis and provides the unpacked code of the executable under analysis. This technique uses the fact that the hidden code which has been obfuscated is not available to the static model. Thus this approach gives a wider scope to the static analysis which otherwise is too specific and provides a mechanism which is independent of packing mechanism.

This is not all the unpacker has to do as it is not always the case that absence of a code sequence in static model will be the hidden code. The dynamic link libraries (DLLs) loaded during Windows binary execution also result in execution of code that is not available in the static model. So, whenever, a DLL is loaded, the memory it occupies is noted. During single-step execution, the program's program counter (pc) is continually compared against all known memory areas. If the pc points to memory occupied by a DLL, the return address from the stack is read and a breakpoint set there, allowing single-step execution to resume after the call's return.

To deal with the increased program complexity due to multiple packed layers, the PolyUnpack technique can be extended to proceed iteratively. Every time the resultant hidden code is attained, the static model of the code is generated. If at any step the binary sequence of executed code does not match any part of static view, the next iteration begins. The final unpack code is the one whose static model is totally consistent with its run-time execution sequence. This technique like most instrumentation tools is not transparent to the malware being processed. Therefore, there exists the possibility that an instance of malware being executed in it may detect that it is being instrumented and alter its behavior (e.g., halting its execution instead of generating hidden-code) in order to evade extraction of its unpacked code [22].

V. ALTERNATIVE APPROACH

As described earlier, all the covered techniques have their own limitations. Even, poly-unpack suffers from inefficiency as it involves single-step debugging. Hence an alternative approach to the above mentioned techniques is *OmniUnpack [21]*. Omni unpacking technique addresses the shortcomings of existing systems. This is a generic approach to handle any type of packer and any type of self-modifying code. It does not depend on virtual environment, emulation or debugging for unpacking. It monitors the program execution and tracks written and written-then-executed memory pages. *Written-then-executed pages are indicative of unpacking but not indicative of the end of unpacking, as there could be multiple unpacking stages.* The approach uses heuristics to approximate the end of unpacking. In order to improve the performance, page-level monitoring is done. When stub, which is embedded in the program, writes the unpacked code to memory, the destination page is marked as writable but not executable. At the end of the unpacking stage, when the program accesses the same page for execution, the lack of execution permission causes a protection exception. If the program then makes a potentially damaging system call, a malware detector is invoked on the written memory pages. If the detection result is negative (i.e., no malware found), execution is resumed. The resulting low overhead means that this technique can be used for continuous monitoring of a production system.

When the virtual-physical address mapping needs to be updated or when the memory protection is violated, the hardware signals to the OS through an exception and allows the OS to repair the memory state before continuing execution. Existing features are used to intercept the first moment when a page is written and the first moment when a page is about to be executed after a write.

A disadvantage of the technique is imprecision of page-level tracking**.** Page-level tracking decreases the granularity of monitoring although it significantly reduces the overhead of memory-access tracking. It is less precise, often resulting in incorrectly detecting unpacking stages. It would be unnecessarily expensive to invoke the malware detector every time a written memory page is executed, because such an event (written-then-executed) is frequent. Hence determining the end of unpacking is hard to decide and is only an approximation. The technique assumes that a packed program will generally be malicious. Therefore, it provides the facility to automatically call the malware detection engine if a dangerous system call is made. This introduces anther level of complexity in the algorithm as decision has to be made about the choice of dangerous system calls**.** A dangerous system call is a system call whose execution can leave the system in an unsafe state. To achieve its malicious goal, the malware has to interact with the system. As a simple solution, any system call that modifies OS state is considered dangerous in this technique.

Because of the possibility of multiple unpacking stages and of the approximation being using to detect them, it is insufficient to monitor and scan the program only once during an

execution. This technique implements a continuous monitoring approach, where the execution is observed in its entirety. This is a necessary departure from the traditional view of unpacking and scanning as separate, one-time stages of the malware detection process. Also, efficiency can further be increased if all memory-page accesses are not observed. It is sufficient to observe the first memory access in an uninterrupted sequence of accesses of the same type. For example, only the first write to a page is useful, subsequent writes to the same page do not impact the result of the algorithm and can be ignored. Thus, this unpacking technique aims to be very generic as it supports binaries packed with any arbitrary algorithms applied any number of times [21].

VI. CONCLUSION

Current approaches for automatic analysis suffer from a number of shortcomings. One problem is that malicious code is often equipped with detection routines that check for the presence of a virtual machine or a simulated OS environment. When such an environment is detected, the malware modifies its behavior and the analysis delivers incorrect results. Malware also checks for software (and even hardware) breakpoints to detect if the program is run in a debugger. This requires that the analysis environment is invisible to the malicious code [15]. As any other dynamic-analysis technique, emulation places a time limit on the execution of the packed program and is restricted by the reliability of the emulation environment. Extracting packed binaries and finding the original entry point using dynamic analysis is feasible but these approaches either rely on some heuristics or require disassembling the packed program. However, heuristics about packed code may not be reliable in all cases and can be easily evaded. In addition, correctly disassembling a binary program itself is challenging and error-prone. Hybrid approaches for packed code extraction perform a series of static and dynamic analysis which leads to performance overhead.

VII. REFERENCES

[1] SHON HARRIS, ALLEN HARPER, CHRIS EAGLE AND JONATHAN NESS, "GRAY HAT HACKING", 2ND ED., MCGRAW-HILL, CHAP 21.

[2] Peter Szor (2005, Feb 3). "Art of Computer Virus Research and Defence", Chap 11, [Online] Available: http://safari.oreilly.com/0321304543/ch15lev1sec4.

[3] Dr. Jose Nazairo, "Botnet Tracking: Tools Techniques and Lessons Learned", presented at Lockdown 2007 University of Wisconsin-Madison, page 12.

[4] Gaith Taha, **"**Counterattacking the packers", presented at AVAR 2007 Conference in Seoul, page 1.

[5] Danny Quist and Valsmith, "Covert Debugging: Circumventing Software Armoring Techniques", presented at Black Hat Briefings USA August 2007, page 1-2.

[6] Miroslav Vnuk and Pavol Navrat, "Decompression of run-time compressed PE-files.", presented at IIT.SRC 2006 – Student Research Conference, Slovak University of Technology, Faculty of Informatics and Information Technologies, page 2-4.

[7] Andrew Lee and Pierre-Marc Bureau, "The Evolution of Malware", presented at Virus Bulletin Conference November 2007, page 8-10.

[8] Proceedings of the 5th Australian Digital Forensics Conference 3rd December 2007, Edith Cowan University, Mount Lawley Campus. Page 67, [Online] Available: http://scissec.scis.ecu.edu.au/conference_proceedings/2007/forensics/00_Forensics2007_Complete_Proceedings.pdf

[9] Andreas Beetz, "File Analysis", US Patent US 2004/0236884 A1, Nov. 25, 2004.

[10] Mihai Christodorescu, Somesh Jha, Johannes Kinder, Stefan Katzenbeisser and Helmut Veith, "Software Transformations to Improve Malware Detection" In Journal in Computer Virology, vol. 3, (4): pp. 253–265, November 2007.

[11] Johannes Kinder , "Model Checking Malicious Code", Diplomarbeit, Technische Universität München, 2005**.**

[12] Mihai Christodorescu and Somesh Jha, "Malware Normalization", Technical Report #1539, Nov. 2005, page 7-9.

[13] Raymond J. Canzanese, Matthew Oyer, Spiros Mancoridis and Moshe Kam, "A Survey of Reverse Engineering Tools for the 32-Bit Microsoft Windows Environment", Jan. 2005, page 17-20.

[14] Mario Alberto López, "Unpacking, a Hybrid Approach", presented at 2nd International CARO Workshop 1st & 2nd May 2008, The Netherlands.

[15] Ulrich Bayer, Andreas Moser, Christopher Kruegel and Engin Kirda, "Dynamic Analysis of Malicious Code", in Journal in Computer Virology 2(1): 67-77 (2006), page 2-5.

[16] Tobias Graf, "Generic Unpacking How to handle modified or unknown PE Compression Engines" presented at Virus. Bulletin Conference 2005.

[17] Tom Brosch and Maik Morgenstern, "Runtime Packers: The Hidden Problem.", presented in Black Hat briefings USA 2006, page 3.

[18] Min Gyung Kang, Pongsin Poosankam, and Heng Yin, "Renovo: A Hidden Code Extractor for Packed Executables", presented at 5th ACM Workshop on Recurring Malcode (WORM 2007), page 1-4.

[19] Kurt Natvig, "Sandbox technology inside AV Scanners", presented at Virus Bulletin Conference, September 2001, page 2.

[20] Norman SandBox Whitepaper, [Online] Available: http://www.norman.com/Download/White_papers/en , page 14.

[21] Lorenzo Martignoni, Mihai Christodorescu and Somesh Jha. "OmniUnpack: Fast, Generic, and Safe Unpacking of Malware", presented at 23rd ACSAC (Annual Computer Security Applications Conference) in Miami Beach FL USA (2007), page 1-4.

[22] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee, "PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware", presented at 22nd ACSAC (2006), page 1-6.

[23] Adrian E. Stephan, "Defeating Polymorphism: Beyond Emulation" presented at Virus Bulletin Conference, October 2005, page 1-7

[24] VMware: server and desktop virtualization, 2006. http://www.vmware.com/

[25] Xiaodong Tan, "Anti-unpack Tricks in Malicious Code", Security Labs, Websense Inc. presented in AVAR 2007, Seoul. Page 5-29.

[26] Lutz Bohne, Diploma Thesis "Pandora's Bochs: Automatic Unpacking of Malware", 28th January 2008, page 23-44.

[27] Jarkko Turkulainen, F-Secure Corporation "Emulators and disassemblers", T-110.6220, page 21-31.

[28] The Bochs Development Team. Bochs - The Cross Platform IA-32 Emulator 2007. http://bochs.sourceforge.net/.

[29] ASPack Software. ASPack and ASProtect http://www.aspack.com/.

[30] Bitsum Technologies. PECompact2. http://www.bitsum.com/pec2.asp.

[31] Obsidium Software, http://www.obsidium.de/show.php?home

[32] Teggo. MoleBox Pro, http://www.molebox.com/download.shtml

[33] Silicon Realms Toolworks. Armadillo,