Unpacking Virtualization Obfuscators

Rolf Rolles
rolf.rolles@gmail.com

***Abstract*—Nearly every malware sample is sheathed in an executable protection which must be removed before static analyses can proceed. Existing research has studied automatically unpacking certain protections, but has not yet caught up with many modern techniques. Contrary to prior assumptions, protected programs do not always have the property that they are reverted to a fully unprotected state at some point during the course of their execution. This work provides a novel technique for circumventing one of the most problematic features of modern software protections, so-called virtualization obfuscation. The technique enables analysis of heretofore impenetrable malware.**

I. Introduction

Originating conceptually in the form of file-infecting viruses, and later evolving into digital rights management schemes, executable protectors have existed in various guises for decades. Most recently, they have taken a central role in the malware epidemic: criminals have strong motives to prevent their malware from being analyzed, which is a mandatory step in performing incident response. Consequently, much effort is poured into the creation of increasingly difficult protections, which has in turn spurred defensive research toward removing them automatically. The need for automated removal is especially inflated by today's enormous volumes of malware. The result of this arms race is a collection of new protection features that are exceedingly difficult to remove, either by humans or by computers, some of which have never been publicly circumvented. Thus, however unfortunate, the offensive side is presently winning.

The classical model of an executable protection is that of a wrapper around a single executable. At the time of creation, the protector will compress and/or encrypt the contents of the executable's sections. It will then append a new code section that is responsible for decompressing and/or decrypting the sections when executed, as well as for thwarting attempts at reverse engineering. The executable's entrypoint is redirected into this new code (termed the "unpacking stub"), and upon completion, execution is transferred back to the original entrypoint. The program will subsequently function identically to the original, unprotected executable.

Previous forays into automated unpacking have attacked this model of protection with a good degree of success. Though these tools vary as to their stated goals and internal workings, they share a common theme. Each assumes that the hidden code from the protected program will be completely unprotected in memory at some point during execution, and each uses various measures to guide execution until this point. Some tools additionally have the goal, beyond merely identifying hidden code and the original entrypoint, of producing working, unprotected executables.

Unfortunately, in comparison with the myriad of techniques in modern protections, this model is merely a ruse of simplicity. Some protections utilize multiple executables: some will unpack the executable into a new process, some operate with a two-process model in which one executable debugs a modified version of the original. Some protections drastically obscure the relationship of the protected executable with shared libraries on the system. Most saliently for the goals of this paper, some protections never restore portions of the protected code; instead, they translate the code into a different language and execute it at run-time inside of a custom, often obfuscated interpreter. Much of the previous literature in this area has left addressing these protection features to future work.

II. Virtual Machine Software Protections

"Virtual machines" (VMs) are a somewhat unfortunately-termed class of software protections, which are perhaps the most potent of present-generation techniques (some particular incarnations of which have heretofore resisted public scrutiny). These protections implement a virtual environment and interpreter within which bytecode programs are executed. The language accepted by the interpreter is chosen at random at the time of protection. The interpreter itself is generated in accordance with the choice of language, and is also typically heavily obfuscated.

Although their assembly languages are often simple, VMs pose a challenge because they severely dilute the value of existing tools. Standard dynamic analysis with a debugger is possible, but very tedious because of the low ratio of signal to noise: one traces the same VM parsing / dispatching code over and over again. White-box static analysis is very time consuming because of the need to reverse engineer the interpreter beforehand, which can be poly- or meta-morphic. Patching the VM program requires a familiarity with the instruction set that must be gained through analysis of the VM parser. Reverse engineering such a component is repetitious, and the high-level details are obscured by the flood of low-level details.

There are two varieties of virtual machine software protection, which are classified according to the type of code that is executed thereunder. Standard virtual machine protections, such as [7], execute portions of the protected program's unpacker stub in the interpreter in order to obscure its inner workings. Though this type of executable protection is very effective at thwarting manual analysis, it does not pose any challenges for automated unpackers per se, and furthermore this type is well-studied in the literature.

The second variety of virtual machine software protections is the one with which the rest of this paper shall be concerned. So-called "virtualization obfuscators" (known in [1] and [10] as instruction virtualizers) translate portions of the program's original x86 machine code into a custom language which is then interpreted at run-time. The program's code is never restored to its original form. As a consequence, without reconstruction of the original x86 code, existing white- and gray-box analyses are impossible.

A. Virtual Machine Architecture

As stated, a virtualization obfuscator contains an interpreter, which shares much in common with an ordinary interpreter. Virtualization interpreters tend to be written in assembly language and implemented as a large switch statement, which may either contain a single point of dispatch or the opcodes may be directly threaded. The interpreter operates upon the data available to a machine-language program, namely, the processor's registers and flags.

The language interpreted by a virtualization obfuscator tends to be RISC-like. One x86 CISC instruction will translate into multiple virtualized instructions. For example, for an instruction with a complicated memory-addressing mode such as `mov eax, [ebx+ecx*4+123456h]`, the address calculation will be translated into several virtual instructions: one to fetch `ecx`, one to multiply `ecx` by four, one to fetch `ebx`, one to add these quantities together, one to add `123456h` to the result, and one to dereference the formed address.

Upon entering into the virtualization obfuscator at the beginning of a bytecode program, the interpreter must save the host's registers and flags into a structure (termed the VM's context structure). The net effect of bytecode execution is a sequence of manipulations applied to memory and the context structure. Upon exiting the bytecode program, the interpreter is responsible for restoring the host's registers and flags, and for transferring control back to the native x86 portion.

B. Template Languages

Virtualization obfuscators at their core involve the conversion of the native x86 machine language into another custom language, chosen at the time of protection at random from a family, which at run-time is interpreted in an obfuscated interpreter customized for that particular language. The original x86 code is permanently destroyed.

Internally, each virtualization obfuscator offers one or more template languages from which the final language is derived. For instance, Themida allows the user to choose between four different template languages in its protection options: RISC-64, RISC-128, CISC and CISC-2. Each of these separate choices results in a different "basis" language being used.

Considering executables protected with the same template language, the differences between any two particular instances generally consist of different encodings for the same instruction (e.g. byte `0x12` might represent addition in one instance of the VM, and multiplication in another), extra obfuscation within certain instructions (e.g. for instructions which take constant operands that are embedded in the instruction stream, the constants themselves may be obfuscated), and x86-level obfuscation upon the VM harness and each of its handlers.

C. Method of Circumvention

To truly break a virtualization obfuscator, we must convert the protected code from the bytecode language back into x86 machine code that resembles the original pre-protected code, thereby removing the program's dependency upon the interpreter. Doing so allows us to apply standard program analysis and reverse engineering techniques, such as testing the sample against an anti-viral signatures database, or classifying the sample according to its family [20].

One key observation behind circumventing virtualization obfuscators is in remembering that they are interpreters. If we had in our possession a compiler with a front-end accepting the language of a particular sample and a back-end that produced x86, we could re-compile the virtualized instructions back into machine code. Another key observation is that, since the virtualization obfuscator derives the language for each individual sample from a template language, the languages utilized by two different protected binaries will have many similarities. Whereas deciding the syntax and semantics of a language accepted by an interpreter may be difficult in general, answering the same question knowing that the language is derived from some particular known language family is more easily tractible.

Combining these two observations, the plan for attacking these protections is to create a back-end compiler infrastructure which translates some representation of the template language into x86 code, and a mechanism for generating a front-end for the compiler that is specific to the language accepted by a protected sample.

1) Reverse engineer the virtual machine: This step must be performed only once per virtualization obfuscator. A skilled reverse engineer must examine the virtual machine to determine the operations of which it is capable, design an intermediate language (IR) that captures the semantics of the language, and construct a translator which maps the the VM bytecode operations into a sequence of intermediate language instructions. [19] discusses a system that automatically extracts certain information from an arbitrary instance of a virtual machine, although more analysis is still required in order to fully break the protection.

We shall use VMProtect [14] as our running example. VMProtect's language is stack-based and RISC. On entry to the VM, in the x86 portion, VMProtect pushes the registers and the flags onto the stack (in an order that is randomly determined per sample). Inside of the VM bytecode program, the first action is to pop all registers off of the stack and store them into a 16-dword scratch area. Computations take place upon the scratch area in lieu of upon the processor's registers. Upon exit of a VM bytecode program, the contents of the scratch section are copied onto the host stack and then restored into the processor registers in a manner that is randomly determined per sample.

Two examples of VMProtect instructions follow. The first is responsible for popping a word off of the stack (`[ebp+0]`), adding the word to another word on the stack (`[ebp+4]`), and pushing result and the flags onto the stack.

```
mov   ax, [ebp+0]
sub   ebp, 2
add   [ebp+4], ax
pushf
pop   dword ptr [ebp+0]
```

The second pops a dword and then a word off of the stack, shifts the dword to the right by the word, and then pushes the result and the flags onto the stack.

```
mov   eax, [ebp+0]
mov   cl, [ebp+4]
sub   ebp, 2
shr   eax, cl
mov   [ebp+4], eax
pushf
pop   dword ptr [ebp+0]
```

When the VMProtect translator encounters an x86 instruction dictating that two words be added together, it will generate bytecode that makes use of the aforementioned addition opcode. For example, `add ax, bx` will translate into a fetch of the `bx` register, a fetch of the `ax` register, the addition of words opcode described above, an instruction to pop the flags off of the stack and store them somewhere in the scratch area, and finally an instruction to store the resultant word from the stack into the `ax` register in the context structure.

VMProtect's instruction set lacks many features in comparison with the x86 assembly language that it represents. For example, it lacks bitwise exclusive-or, or, and, and not instructions, as well as add with carry, subtraction, subtraction with borrow, increment, decrement, unary negation, setting of registers/memory with condition codes (`setcc` instructions), conditional moves, and conditional jump instructions. All of these features are implemented in terms of more primitive operations.

For example, the NOR instruction takes two arguments from the stack (`[ebp+0]` and `[ebp+4]`), negates them and computes the bitwise-AND of the two.

```
mov     eax, [ebp+0]
mov     edx, [ebp+4]
not     eax
not     edx
and     eax, edx
mov     [ebp+4], eax
pushf
pop     dword ptr [ebp+0]
```

This opcode is used to implement logical operations. For example, `and eax, ebx` is represented as NOR(NOR(`eax`,`eax`) NOR(`ebx`,`ebx`)). Similarly, the `add` instruction is used to implement the arithmetic instructions previously mentioned.

Further, parts of the x86 instruction set such as the SIMD instructions are not virtualized by VMProtect. When such an instruction needs to be executed, VMProtect will exit the bytecode program (restoring all flags and registers), jump to a location containing the non-virtualized instruction, and then re-enter the bytecode program.

2) Detect the locations at which control flow enters the virtualization obfuscator: Detecting entrypoints into the VM is a hard problem to solve statically: even assuming perfect disassembly (in contradiction of [11]), the control transfers into the VM look like any other control transfer. In practice, however, it is easy for a reverse engineer to locate the VM entrypoints, as it is usually obvious which portion of the program is sensitive enough to require protection (e.g., encryption schemes). [19] shows that the transfers into the VM can be detected through dynamic means.

3) Develop a procedure for producing a disassembler, given a protected executable: Additionally to knowing the layout of one instance of the virtualization interpreter, the reverse engineer must also understand in which respects two different derivations from the template language are the same, and how they differ. VMProtect offers two flavors, both of which we have evaluated: the demonstration version provided on the author's website, and the full version available to registered customers. For the first, we were able to create samples using the package provided by the author. For the second, a collection of malware samples were obtained from an antivirus industry partnership.

The files protected with the demonstration version were all found to recognize almost entirely the same language. The x86 implementation of each VM opcode dispatcher was identical across samples. The only differences were the orders in which the registers were saved on VM entry and restored upon exit, and the ordering of the dispatchers within the switch table. Recognizing the variant of the language in this case is easy: the dispatchers can simply be fingerprinted to decide which switch case corresponds with which operation, and the register save/restore sequences can be sliced out of the entry/exit sequences.

The files protected with the registered version were found to be rather different. First, the x86 implementation of the VM and each of its opcode handlers were obfuscated differently from sample to sample. Meaningful instructions were interleaved with "junk" instructions that have no effect on the legitimate computations. An example follows.

```
; insert junk here
mov eax, [ebp+0]
; insert junk here
mov eax, fs:[eax]
; insert junk here
mov [ebp+0], eax
; insert junk here
```

Secondly, the VM opcodes which took constant parameters from the instruction stream (e.g. `push 0h`) had special obfuscation: the handlers would load the constants from the

instruction stream and apply a series of arithmetic transformations thereto before using the value (e.g. before pushing it onto the stack). These "constant obfuscations" were different per sample. The first six lines of the following snippet are an example of constant obfuscation.

```
neg     al
ror     al, 2
xor     al, 39h
dec     al
neg     al
not     al
movsx   edx, al
mov     edx, [ebp+0]
mov     [edi+eax], edx
```

There were certain other cosmetic obfuscations: some samples would read the instruction stream backwards (e.g. decrementing EIP after each instruction rather than incrementing it), some had further obfuscation in determining which byte values corresponded to which handlers, and some obfuscated the addresses of the VM instruction handlers.

To generate a disassembler, we must recognize the handlers despite their obfuscation, and further recognize the obfuscation in the handlers which take constant parameters, extracting the sequence of arithmetic operations responsible for deobfuscating the constants.

This task turned out to be readily amenable to ad hoc techniques. Since the obfuscation on the x86 representation of the VM harness and dispatchers consisted of "junking" only (insertion of random but harmless instructions between the legitimate instructions belonging to the handlers) as opposed to metamorphic obfuscation, every instruction which was present in the non-obfuscated handler is present in its literal form in the obfuscated handler. ([18] discusses a possible solution for metamorphic obfuscation in the handlers, namely deobfuscation of the x86 assembly representation via compiler optimizations.) Therefore, we can simply use regular expressions upon the disassembly of the obfuscated handlers to perform a comparison with the non-obfuscated handlers, and choose the largest match. This method can be used for both the evaluation version of VMProtect, which does not obfuscate handlers, as well as the commercial versions.

With the opcodes thus identified, we can then perform additional analysis for those handlers which take constants from the instruction stream, to slice out their constant deobfuscation routines. In the example above, we simply apply slicing techniques upon those instructions that modify the register `al` to obtain the constant deobfuscation routine.

The current implementation of the VMProtect disassembler generator is an IDA plugin, consisting of roughly 5KLoC of C++. It constructs OCaml source code for a disassembler that converts raw bytes into VMProtect bytecode.

Recent research in program analysis ([21] in particular) provides a more robust method for identifying handlers without constant obfuscations. By performing pure symbolic execution upon the VM opcode handlers, we obtain a representation of each handler as a mathematical function that is a map from its input space (the VM context structure, VM EIP and its surrounding bytes, and the program's memory) into the same space. By focusing on the transformation of one state to another, we can ignore irrelevant details such as those introduced through obfuscation (including metamorphic obfuscation). We can then apply a theorem prover to determine if the handler computes the same function as one of the handlers that are known a priori. For handlers with constant obfuscations, perhaps a TQBF solver could be used. We leave these investigations to future work.

Figure 1. Abstract syntax for the VMProtect IR

$\langle segment \rangle$ ::= CS | DS | ES | FS | GS | SS | Scratch
$\langle unop \rangle$ ::= −
$\langle binop \rangle$ ::= + | ≪ | ≫ | Nor | Rcl | Rcr | ÷ | Idiv | × | Imul
$\langle expr \rangle$ ::= Reg r | Temp t | Const c | Dereference $seg\ expr\ size$ | Binop $expr\ binop\ expr$ | Unop $unop\ expr$
$\langle ir \rangle$ ::= Assign $expr\ expr$ | Push $expr$ | Pop $expr$ | Jump $expr$ | x86Literal $littype$

4) Disassemble the bytecode and convert it into intermediate code: With a custom disassembler in hand, we can now disassemble the bytecode stream into a sequence of VMProtect bytecode instructions. However, we find that the disassembled code is hard to read due to the somewhat complicated semantics of each VMProtect instruction. Since VMProtect is a stack machine, all instructions either push or pop data from the stack, and it can be complicated to track the flow of data throughout a basic block (i.e. a value may be pushed onto the stack dozens of instructions before it is finally popped). Therefore, it is convenient to translate each VMProtect instruction into a series of instructions in a simpler language, so as to explicate all implicit operations (such as pushes and pops).

The OCaml source code that converts the VMProtect bytecode to intermediate representation is roughly 1000 lines of OCaml. The abstract syntax for the intermediate language is displayed in figure 1.

Here is one of the VMProtect bytecode handlers, and its corresponding translation into the intermediate representation.

```
mov eax, [ebp+0] ; pop dword from stack
mov eax, fs:[eax] ; read dword from address
mov [ebp+0], eax ; push dword onto stack
```

```
DeclareTemps([(0,D);(1,D)])
Pop (Temp 0)
Assign (Temp 1, Deref (FS, Temp 0, D))
Push (Temp 1)
```

We introduce temporary variables to ease the translation; these variables are not actually part of the VMProtect instruction set nor the produced x86 code. We shall eliminate them later with compiler optimizations.

5) Apply compiler optimizations to the IR: With the VMProtect bytecode suitably transformed, we can now apply

compiler optimizations locally to each basic block in order to transform the intermediate representation into something closer to x86. We begin with an unoptimized listing, immediately after translation into the intermediate language. The code is what one would expect from a stack machine: nearly every instruction is a push or a pop.

An unoptimized IR listing that we shall use as a running example follows.

```
push Dword(-88)
push esp
push Dword(4)
pop t3
pop t4
t5 = t3 + t4
push t5
push flags t5
pop Scratch:[52]
pop t6
pop t7
t8 = t6 + t7
push t8
push flags t8
pop Scratch:[12]
pop esp
```

We apply a simple analysis to rid the IR of its stack machine features. We scan the basic block from the beginning to the end, maintaining a stack. For each push instruction that we encounter, we make a note of the size of the push and the instruction generating it. For each pop, we simply inspect the bottom of the stack and note which instruction pushed the value onto the stack. From here, we can eliminate the push/pop pair and replace it with a direct assignment statement. The resulting program resembles a more conventional listing.

```
t3 = Dword(4)
t4 = esp
t5 = t3 + t4
Scratch:[52] = flags t5
t6 = t5
t7 = Dword(-88)
t8 = t6 + t7
Scratch:[12] = flags t8
esp = t8
```

Next, we apply the standard compiler optimizations copy propagation and constant propagation to eliminate the temporary variables; as a result, we have eliminated 80% of the original IR, and are left with a very readable listing.

```
Scratch:[52] = flags 4 + esp
esp = esp - 84
Scratch:[12] = flags esp - 84
```

We perform copy propagation to decide which scratch areas correspond to which registers. The result of optimizing the running example is the single statement `esp = esp - 84`.

With the IR suitably transformed, we can now invert the transformations concerning the bitwise and arithmetic instructions as described previously. We simplify expressions such as `nor(nor(eax,eax),nor(ebx,ebx))` into expressions such as `and eax, ebx`.

A representative example of a fully-optimized block of VMProtect IR follows. As can be readily seen, the listing is virtually identical to x86 assembly language, with the exception of the presence of assignment statements instead of mov instruction, and explicit representation of the instructions' effect on the flags.

```
push ebp
ebp = esp
push -1
push 4525664
push 4362952
eax = FS:[0]
push eax
FS:[0] = esp
eflags = flags esp - 84
esp = esp - 84
push ebx
push esi
push edi
SS:[ebp-24] = esp
call [4590300]
```

For comparison, the original code before it was protected by VMProtect is reproduced below.

```
push ebp
mov ebp, esp
push 0FFFFFFFFh
push 450E60h
push offset sub_4292C8
mov eax, large fs:0
push eax
mov large fs:0, esp
add esp, 0FFFFFFA8h
push ebx
push esi
push edi
mov [ebp-18h], esp
call ds:[0460ADCh]
```

6) Generate x86 code: Code generation in this particular application has several differences from the operations performed by a regular compiler: we seek to generate code that is as close to the original, pre-protected code as is possible. For example, rather than choosing an arbitrary register allocation, we are able to infer which registers should be used in each operation through the compiler optimizations discussed previously. Additionally, since some of the VMProtect instructions have no analogues in the x86 instruction set such as the `nor` instruction, we seek to recognize the constructs built thereupon and translate them directly into assembly language, as opposed to compiling sequences involving `nor` constructs. Also, we must generate instructions that are not typically considered by

standard compilers, such as the privileged instructions `in` and `out`.

III. Related Work

A. Automated unpacking in general

It is the simplistic model of protection discussed in the introduction that has been targeted by previous work in this area. Previous automated, generic unpackers assume that protections will adhere to a simple model: namely, that there will be some point in time where the executable is entirely unprotected in memory, and that dumping the process image to the disk will result in a functioning executable. The goal of these systems is to monitor execution until the packed program has finished executing its unpacking stub, at which point it is assumed to be unprotected.

Automated unpackers produced heretofore can be divided into two categories: those that take advantage of either processor features and/or the operating system, and those based on emulation. The archetypical example of an in-guest automated unpacker is OllyBonE [6]. OllyBonE uses a kernel driver to simulate non-executable paging in software by taking advantage of x86's split instruction/data TLB organization, and by coopting the pagefault handler. The general workflow is very reminiscent of PaX [13]. This facility provides a semi-automatedunpacker: the user manually selects the section in which the program's original code resides, and the driver will trigger on all attempts to execute within that section.

Present-day automated unpackers such as Renovo [1], Saffron [8], and Azure [9] vary as to their internal workings, but the unifying idea behind them relies upon the simple notion of protection given above. These tools automatically execute the program (respectively under emulation, instrumentation, and hypervision) until the program counter is resting at the original entrypoint, at which point its unprotected code section is written to the disk. The literature surrounding these three tools does not mention executable recreation nor import rebuilding.

Renovo, as described in [1], is an automated unpacking system based on dynamic taint analysis, using BitBlaze's TEMU [2] modification to QEMU [3]. The emulated x86 code is instrumented such that each memory write renders the destination 'dirty'. If the instruction pointer ever resides over dirty memory, the system considers the surrounding region to be hidden code and dumps it to disk, along with the position of the instruction pointer within the dumped region. It then resets its clean/dirty bit-maps and continues emulation. Pandora's Bochs [10] is an unpacker based on Bochs [12] which employs similar principles as Renovo, although it additionally attempts to reconstruct working executables, including rebuilding import information.

Azure [9] is a proof-of-concept tracing utility designed for stealth, based on Intel's VT hypervisor capabilities. Its literature describes its efficacy in tracing unpacking stubs through to the original entrypoint undetected, and mentions that future work will see an unpacking framework implemented around it. Though its literature also indicates that it is capable of unpacking Armadillo [4], it is unclear how a tracing framework could break a multi-process protection such as that.

Anti-virus engines also have the goal of generically unpacking theretofore unknown protections, for the purposes of removing file-infecting viruses and also to be able to apply signature-based or behavioral detections to the executables underneath. These components are commonly constructed as a combination x86 processor emulator and Windows operating system emulator. Though on first glance this seems similar to systems like Renovo or Pandora's Bochs, the crucial difference is that, whereas the latter provide a genuine copy of Windows to the sample in consideration, the former provide only a simplistic, emulated model of it. This design decision, while sensible in light of real-world pressures such as resource efficiency, renders antivirus emulators especially susceptible to detection and evasion.

B. Unpacking virtualization obfuscators

Virtualization obfuscators have been examined in-depth recently. Rolles first published techniques for removing simple virtualization obfuscators in [15]. Shortly thereafter, Lau published details about applying his mixed static and dynamic tracing framework, DSD-Tracer, to analyzing virtualization obfuscators in [16]. Rolles then published in [17] more comprehensive techniques for removing industrial-grade virtualization obfuscators. [18] later discussed the use of compiler optimizations applied directly to the assembly instructions comprising the opcode dispatchers for the purpose of disassembler generation. Lastly, [19] discusses automatically extracting certain information from unknown virtualization obfuscators.

References

[1] M. G. Kang, P. Poosankam, and H. Yin. Renovo: A Hidden Code Extractor for Packed Executables. In Proc. of the 5th ACM Workshop on Recurring Malcode, 2007.

[2] TEMU: The BitBlaze Dynamic Analysis Component. http://bitblaze.cs.berkeley.edu/temu.html.

[3] F. Bellard: QEMU, a Fast and Portable Dynamic Translator. In Proceedings of the 2005 USENIX Conference, 2005.

[4] Silicon Realms Toolworks. Armadillo. http://siliconrealms.com/index.shtml

[5] Themida. http://www.oreans.com/

[6] OllyBonE. http://www.joestewart.org/ollybone/

[7] R. E. Rolles: Defeating HyperUnpackMe2 with an IDA Processor Module. http://www.openrce.org/articles/full_view/28

[8] D. Quist and V. Smith: Covert Debugging: Circumventing Software Armoring Techniques. In Black Hat Briefings USA, August 2007.

[9] P. Royal: Alternative Medicine: The Malware Analyst's Blue Pill. In Black Hat Briefings USA, August 2008.

[10] L. Boehne: Pandora's Bochs: Automated Unpacking of Malware. Diploma thesis. January 2008.

[11] R. N. Horspool, N. Marovac: An Approach to the Problem of Detranslation of Computer Programs. In Comput. J. 23(3): pages 223-229, 1980.

[12] Bochs: The Open Source IA-32 Emulation Project. http://bochs.sourceforge.net/

[13] The PaX Team. Pax. http://pax.grsecurity.net/

[14] VMPSoft. VMProtect. http://www.vmprotect.ru/

[15] R. E. Rolles: Compiler 1, X86 Virtualizer 0. April 4th, 2008. http://www.openrce.org/blog/view/1110/

[16] Dealing with Virtualization Packer. In CARO Conference, Amsterdam, May 2nd, 2008.

[17] R. E. Rolles: Unpacking VMProtect. August 6th, 2008. http://www.openrce.org/blog/view/1238/

[18] _g_: Fighting Oreans' VM (code virtualizer flavour). August 19th, 2008. http://www.woodmann.com/forum/showthread.php?t=12015
[19] M. Sharif, A. Lanzi, J. Giffin, W. Lee. Automatic Reverse Engineering of Malware Emulators. In Proc. of the 30th IEEE Symposium on Security and Privacy, 2009.
[20] Zynamics GmbH. http://www.zynamics.com/vxclass.html
[21] D. Gao, M. K. Reiter and D. Song. BinHunt: Automatically Finding Semantic Differences in Binary Programs. In Proc. of the 4th International Conference on Information Systems Security, December 2008.