

Malware Dynamic Recompilation

Sébastien Josse

DGA

sebastien.josse@polytechnique.edu

Abstract

Malware are more and more difficult to analyze, using conventional static and dynamic analysis tools, because they use commercially off-the-shelf specialized tools to protect their code. We present in this paper the bases of a multi-targets, generic and automatic binary rewriting tool adapted to the analysis of protected and potentially hostile binary programs. It implements an emulator and several specialized analysis functions to firstly observe the target program and its execution environment, and next extract and simplify its representation. This simplification is done through the use of a new and generic method of information extraction and de-obfuscation.

1. Introduction

Malware are more and more difficult to analyze, using conventional static and dynamic analysis tools. Considering the compiled malware, and the many targeted operating systems and underlying CPU architectures, the analyst can observe a lack of multi-target analysis software, sufficiently powerful to deal with the current software protection mechanisms (self-modifying code, virtual machine based obfuscation transformations, etc.).

Malware authors target a wide range of operating systems and take advantage of the advances in software protection made available by commercial off-the shelf specialized products. Moreover, due to malware inherently hostile nature, analysts need a safe and controlled analysis environment.

Current static and dynamic analysis tools suffer from some limitations when dealing with malware.

There are nevertheless many interesting tools for various types of code analysis, including binary code analysis. Unfortunately, they often come with their own intermediate representation (IR), non exportable, sometimes proprietary, making difficult their integration: VEX for Valgrind [16], VEX/Vine for BitBlaze [20], IDA Pro IR for CodeSurfer [21,1], REIL for BinNavi [7].

Moreover, many of them are not suitable for analysis of hostile or protected code.

When they are adapted (tools such as TTAalyze [2], Argos [17] and Renovo [13], based on QEMU, come with features to analyze malware, in a controlled emulated environment), they do not provide binary rewriting features, which are nevertheless very useful when dealing with protected executables.

We present in this paper the bases of a binary rewriting tool designed for analysis of protected and potentially hostile binary programs. This tool is designed to extract dynamically an intermediate representation of a binary and all the necessary information to apply certain simplifications, making its inner working easier to understand for the analyst.

One of the main motivations behind the design and implementation choices of our tool is to circumvent current limitations of existing malware and binary programs analysis solutions. The goal is to get as much information as possible from a binary program that uses all available techniques and tools to protect this information. The idea is to instrument the virtual computer processing unit and the guest operating system in a non intrusive way to get dynamically information required to rebuild the program and simplify its representation.

This tool is based on the dynamic binary translator engine of QEMU and on the LLVM compilation chain¹.

LLVM (Low Level Virtual Machine, [14]) is a compilation chain which comes with a consequent set of optimizations, which can be applied across the entire lifetime of a program. LLVM uses a strongly typed RISC-like instruction set and a static single assignment (SSA) representation (using this representation, each temporary variable is assigned only once). LLVM comes with many binary back-ends (x86, x86-64, SPARC, PowerPC, ARM, MIPS, CellSPU, XCore, MSP430, MicroBlaze, PTX) and some source code back-ends (C, C++)².

¹ LLVM 3.1 with Clang 3.1 front-end for the C family of languages.

² It should be noticed that the LLVM source code back-ends are unlikely to be supported in the future.

The QEMU (Quick EMUlator, [3]) Dynamic Binary Translator (DBT) is used to dynamically translate the binary code from the guest CPU architecture to the host CPU architecture, through the use of an intermediate representation (IR) called TCG (Tiny Code Generator, [4]). This language consists of simple RISC-like instructions, called micro-operations. The binary translation consists of two stages: the guest binary code is first translated in sequences of TCG instructions, called translation blocks (DBT front-end). Then, the translation blocks are converted into code executable by the host CPU (DBT back-end). QEMU's DBT comes with many binary front-ends (x86, x86-64, ARM, ETRAX CRIS, MIPS, Micro Blaze, PowerPC, SH4, SPARC).

Our tool inherits from QEMU the many binary front-ends and from LLVM the many back-ends, providing at reasonable cost a complete binary rewriting framework. The rewriting functions are implemented as LLVM passes.

Its current design builds upon works already done to convert TCG IR to LLVM IR (LLVM-QEMU [19] and S2E [6]), as well as upon design algorithms presented in [10, 11] and [12]. This paper completes this documentation by a description of new features and main evolutions of this tool.

Our ambition is that this tool may be able to collaborate with the many software analysis tools based on the LLVM compilation chain, through an "exported" representation of the malware program. In particular, LLVM representation was besides the object of works providing formal tools to reason on transformations that operate on this intermediate representation. Vellvm (Verified LLVM [23]) could allow us eventually to extract formally verified implementations of de-obfuscation passes implemented in our tool.

These design choices lead us to the exploration of new methods of information extraction and programs dynamic analysis. Among the techniques which are not described (to our knowledge) in the literature, we find in particular:

- Dynamic extraction and reconstruction of the relocation information from a binary program, essential to the conversion of its representation to the SSA form of LLVM.
- All the techniques used to "project" the LLVM representation of translation blocks, dynamically generated by the LLVM back-end of TCG, towards the host CPU. In other words, the method used to "extract" the program intermediate representation from the virtual machine and "project" it on the host machine.

- Rewriting passes of the LLVM intermediate representation used to strip the program from its protection and simplify its representation.

On this last point, we notice that the joint application of the partial evaluation inferred by the dynamic translation of target code to LLVM representation and the application of generic static optimization transformations provided by this compilation chain are enough to clear the program of number of its obfuscations. This is in our opinion the main new result of this paper, resulting in a new generic hybrid dynamic / static method of automatic de-obfuscation. Moreover, the first results concerning effective normalization obtained with this method are encouraging, and may be exploited to automatically extract detection schemes to be stored in malware detection engines databases.

This result encourages us to pursue the way of study of generic methods of de-obfuscation, applying automatically and without making any hypothesis about (the) used protection(s) mechanism(s).

The rest of the paper is organized as follows: section 2 presents the design of our tool, and its two most important analysis modules (unpacking and normalization modules). Section 3 presents the strategy developed to validate the efficiency of our tool, preliminary results and ways of improvement of its design and implementation. Section 4 presents future works and concludes this paper.

2. Design

2.1. QEMU DBT extension

Before presenting the architecture of our tool, let us see the way we have modified the QEMU software CPU to systematically invoke our instrumentation function and translate the TCG intermediate representation to the LLVM representation.

QEMU is a PC emulator using dynamic binary translation: the code written for a CPU instruction set is translated on the fly to a code for another CPU instruction set. We obtain a quicker execution than with simple emulation by using a cache: the idea is to translate a chunk of code, to put it in a cache, and to reuse it if necessary. To accelerate again the virtual processor (VPU), these blocks are chained.

The main interest of a PC emulator based on dynamic binary translation is its execution speed.

For each processor emulated by QEMU, the following translation is done: the target instruction set is translated to an intermediate representation (TCG micro operations) which is itself translated to the host instruction set. In QEMU, this intermediate

representation is independent from the host instruction set. The dynamic binary translation engine is based on this representation. It is for this reason said portable.

We have seen that the QEMU DBT engine performs the dynamic translation of the binary code from the guest processor architecture to the host processor architecture by using the TCG intermediate representation.

Let us see on a simple example what this language looks like. Consider this instruction:

```
0x0040104c: push 0xa
```

This instruction is translated as follows in the QEMU TCG representation:

```
(i) movi_i32 tmp0,$0xa
(ii) mov_i32 tmp2,esp
(iii) movi_i32 tmp13,$0xffffffffc
(iv) add_i32 tmp2,tmp2,tmp13
(v) qemu_st32 tmp0,tmp2,$0x1
(vi) mov_i32 esp,tmp2
(vii) movi_i32 tmp4,$0x40104e
(viii) st_i32 tmp4,env,$0x30
(ix) exit_tb $0x0
```

This TCG instructions block emulates the execution of instruction `push` on the software CPU. The performed operations are the following:

The integer `0xa` is stored in the variable `tmp0` (i). This variable is then stored on the stack (ii-vi). The address of the instruction following the current instruction is stored in `tmp4` (vii) then stored in the QEMU VPU register `cc_op`. Instruction (ix) indicates the end of the TCG block.

Our tool modifies the DBT mechanism in such a way that the instrumentation function of the virtual CPU is systematically invoked before the execution of a translation block. To achieve this, we add an extra micro operation that takes as operand the address of the instrumentation function.

The resulting TCG code is as follows:

```
(i) op_callback @vpu_callback
(ii) movi_i32 tmp0,$0xa
(iii) mov_i32 tmp2,esp
(iv) movi_i32 tmp13,$0xffffffffc
(v) add_i32 tmp2,tmp2,tmp13
(vi) qemu_st32 tmp0,tmp2,$0x1
(vii) mov_i32 esp,tmp2
(viii) movi_i32 tmp4,$0x40104e
(ix) st_i32 tmp4,env,$0x30
(x) exit_tb $0x0
```

This mechanism enables us to execute our instrumentation code at each execution cycle of the virtual CPU. Having access to VPU registers and to

the virtual PC memory, we can acquire a process context and extract information about its interactions with the guest operating system.

By instrumenting also the load and store TCG instructions, we can extract information about the interactions of the target process with the memory of the guest system. Thanks to this information, we can recover the relocation information of the process.

Now that we have seen how to modify the QEMU virtual CPU to enable the systematic invocation of our instrumentation function, let us examine the translation of TCG intermediate representation to LLVM representation. The result of the translation of the above TCG block is as follows:

```
(1) %esp_v.i = load i32* @esp_ptr
(2) %tmp2_v.i = add i32 %esp_v.i, -4
(3) %4 = inttoptr i32 %tmp2_v.i to i32*
(4) store i32 10, i32* %4
(5) store i32 %tmp2_v.i, i32* @esp_ptr
(6) store i32 4198478, i32* %next.i
(7) store i32 0, i32* %ret.i
```

The integer `0xa` is stored at the address pointed by the variable `%4`, which is equivalent to store it on the stack (1-4). The address of the instruction following the current instruction is stored in the variable `%next.i` (6). The instruction (7) finishes the LLVM block.

After the normalization process, this LLVM block is compiled to the following assembly code:

```
401269 ! mov dword ptr [esp-14h], 0ah
```

Now that we have given an overview of main modifications applied to the QEMU emulator, presented in a schematic manner in the figure 1, let us see the general architecture of the tool.

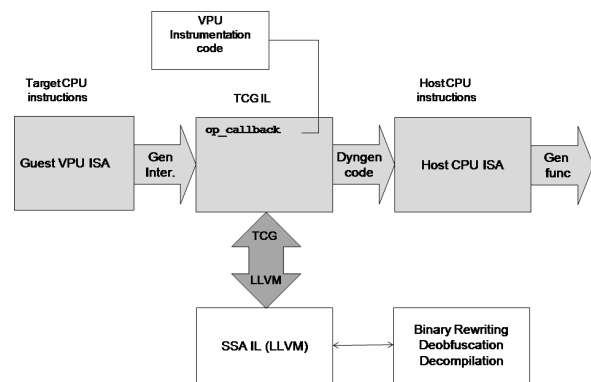


Figure 1: QEMU DBT extension

2.2. Architecture of the tool

Our tool implements an extended DBT engine and several specialized analysis functions (figure 2), to observe the target program and its execution environment.

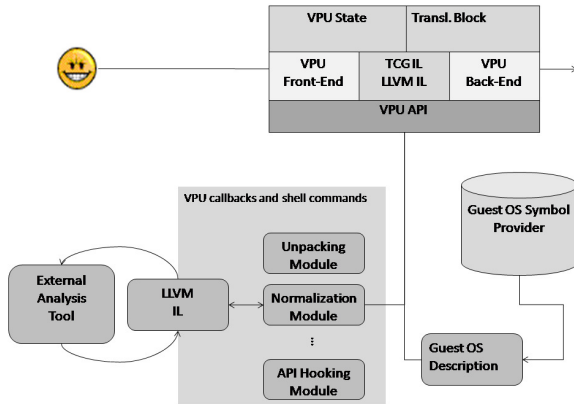


Figure 2: Architecture

A module manager handles activation and collaboration between these analysis functions, implemented as plug-ins.

These analysis functions extract semantic information from the target program. This information can be the trace of its interactions with APIs if the guest operating system, it can be the way it handles objects and structures of the guest operating system's executive or kernel, or more simply its machine code trace.

The extraction of this information rests on a description of the guest operating system³, which can be provided for example by a symbol server, as it is the case for the Windows operating systems family.

Among the module already implemented, we find notably:

- An *API hooking* module
- A *forensics analysis* module
- An *unpacking* module
- A *normalization* module

2.3. API hooking

The native and Windows API hooking module of our tool is based on forensic analysis of the guest operating system memory, without any interaction with the guest operating system.

³ The ntoskrnl.exe program database file (PDB) provides debug information used for example to locate and parse the linked list of executive process structures (EPROCESS). This symbol information can be downloaded from Microsoft Windows symbols server.

The recovery of imported libraries and functions is done by walking Windows executive structures used to represent a process [10].

2.4. Forensics / root-kit analysis

The forensics module [11] of our tool comes with additional features, to monitor and check the integrity of many locations within the guest platform where a hook can be installed. It walks through executive structures of the operating system in order to identify potential targets of a root-kit attack and monitor hardware components that could be corrupted by a root-kit. This information is crucial for the analyst to understand low-level viral attacks.

For the purposes of this paper, we can consider these features to be similar to those expected from a kernel debugger. We can attach a process, have a view of its CPU state and disassembled code, and trace the interaction of the target program with the operating system API. This inspection is done in a safe and controlled environment, without any intrusive interaction with the guest operating system.

Let us see in more details the working of its two most important analysis modules: the unpacking module and the normalization module.

2.5. Unpacking module

The unpacking module locates the original entry point (OEP) of the target executable, gets information relative to its interactions with the operating system API and extracts the relocation information.

The underlying idea of the unpacking algorithm [10] is a simple integrity check of the target program executable code: for each translation block of the program, a comparison between its value in virtual memory and its value on the host file system is made. As long as the values are identical, nothing is done. As soon as a difference is identified, the current translation block is written into the raw file in place of the old translation block. The first instruction of the newly generated translation block is identified as the OEP of the protected program. At the end of the analysis, data sections are written into the raw file in place of original data sections.

The same monitoring algorithm is applied for each translation block. The protection loader of the packed executable can have several deciphering layers.

As soon as the last deciphered translation block has been reached, the only thing to be done is to repair the target executable. In order to recover the

PE (Portable Executable [15]) structure of an unprotected executable, several tasks have to be carried out: set the original entry point, rebuild the imports and relocations tables and consistency check the PE header.

The method used by our unpacking engine in order to reconstruct the IAT and relocations is based on Win32 and native API hooking. During the unpacking process, all API calls are traced. A sorted table⁴ of API functions is initialized at load-time, by walking NT executive structures.

Next, after process execution has resumed, each API call is traced. This table is updated regularly during the target process execution, and is used to dynamically resolve API functions' names. Finally, after a dump of the target process memory space has been done, this table is used to fix the IAT in the PE executable.

Thanks to the load and store TCG instructions instrumentation, we can extract dynamically the relocation information of the program. This information can also be added in a new section of the executable.

As an example, here is the (useful) information extracted during the unpacking stage of a program that displays a dialog box (function `MessageBoxA`):

```
[INFO] eip=0x00401000
[RELOC] value=0x00403000 va=0x00401003
[RELOC] value=0x0040300f va=0x00401008
[RELOC] value=0x00402008 va=0x00401010
[APICALL] api_pc=0x77d8050b api_oep=0x77d8050b
dll_name=C:\WINDOWS\system32\user32.dll
func_name=MessageBoxA
value=0x00402008 va=0x00401010
```

The relocation information is made of pairs (va, value), giving respectively the virtual address and the value to relocate. We can observe that for this packer, the prologue of the function `MessageBoxA` is not emulated by the protection. Otherwise, the external address that is effectively called (`api_pc`) is different from the entry point of the API function (`api_oep`).

2.6. Normalization

⁴ This table, whose keys are virtual addresses, is sorted in order to bypass some protections which emulate first bytes of API functions and therefore do not jump to the original entry point of API functions. By maintaining a sorted table of API functions, API functions are not indexed by their entry point but by a memory range in memory. We are now able to trace API calls even if their first bytes are stolen (moved in the protection code area).

In most cases, after the unpacking stage, we are able to get (automatically) a binary stripped from its protection loader and without any rewritable code. Unfortunately, some obfuscation mechanisms (control flow flattening, virtual machine based obfuscation transformations, etc.) have now to be handled, in order to fully understand the inner working of a malware.

A first attempt to provide a solution to these problems has been implemented in our tool, through the use of the LLVM intermediate representation.

Better than trying to work on the binary after its memory image has been dumped, the idea is to work on its intermediate representation and to increase the amount of information (that has been dynamically collected) by embedding this information into the LLVM module. Such a representation is more suitable for further analysis.

The normalization module uses the output of previous analyses to generate the LLVM representation of translation blocks, on which several optimization transformations are applied. Let us examine this in more details.

During the execution of the target program, the LLVM backend of QEMU TCG outputs the LLVM representation of translated blocks. This LLVM code is linked with an initialization LLVM module (figure 3).

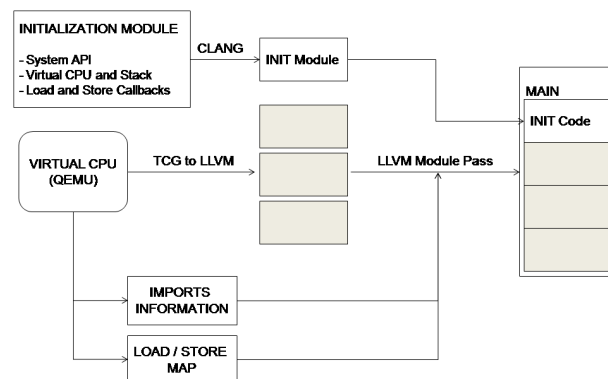


Figure 3: Normalization module

This initialization module implements load and store callbacks, declares system API prototypes and sets a virtual processor unit and its stack.

The normalization module uses the information dynamically collected during the target program execution to resolve imports, process relocations and retrieve data sections. Import table information is used to build LLVM API call instructions. The load / store memory map is used to apply relocations and

inject data from the target program into the LLVM module.

When the LLVM module is rebuilt, some additional optimization passes are applied to its representation.

The LLVM can next be compiled to the chosen architecture, by using one of available LLVM backends. It can also be translated to C or C++ code.

3. Discussion and evaluation

We present in this section the strategy developed to validate the efficiency of our tool, preliminary results and we discuss limits of the current implementation and ways of improvement of its design and implementation.

3.1. Security requirements

The main security requirement for a malware analysis tool is isolation (propagation containment). Another requirement for such a tool is that searched information can be obtained through analysis. In particular, the analysis must be stealth because if emulation is detected [18, 8], the target executable will no longer provide expected information about its inner working. The design of our tool is guided by these two security requirements.

To reach these goals, the implementation of the core emulation engine has been modified to make the hardware emulation more accurate, and thus more difficult to detect. It includes the behavior of several CPU instructions (CPU identification, time stamp counter reading, etc.) but also the way it handles successive faults for example. Any imprecision with regards to CPU architecture specifications can be used to detect an emulated execution environment and results in non accurate analysis of the target program.

The QEMU emulator has been adapted in order to implement the core emulation engine of our tool. In its first version [10], the Windows guest operating system embedded a kernel service which communicated through a virtual network interface with the monitor of our tool. This communication channel was used to upload targets binaries into the virtual machine, to start the execution of the main target program and to get information from the kernel which makes it possible to drive the execution of the guest process from the host system.

The current implementation does not use an embedded kernel service anymore. The target program is directly written to the virtual disk, and the

process localization information is retrieved by forensic analysis of the guest Windows operating system memory, by using information provided by the Microsoft symbol server. Such a method is probably already used by some JTAG client sides and some kernel debuggers. We can attach a target process and acquire its context without any interaction with the guest operating system. We can then drive its execution through the virtual CPU.

An embedded service is nevertheless required to start a target program. This action may be done through stealth code injection, by writing the loader invocation code in the TCG intermediate representation, and then using the DBT backend of QEMU to start (in suspended mode) and resume the execution of a target process.

3.2. Performance considerations

Our tool can be used in batch or interactive console modes, and optionally in graphic mode.

- The batch mode automatically uploads the target executable into the virtual machine, unpacks it and gets required information about its interactions with the guest operating system. The batch mode applies default analysis options, given by a parameters file.
- The interactive mode makes it possible for the malware analyst to dynamically drive the execution of the target executable and interact with the virtual machine, by controlling its states. It is mainly used when the target executable forks one of its components as a new process, in order to acquire the new context and trace the new process execution.
- The main program can also be used in graphics mode. This mode is useful when the analysis process requires interactions between the user and the target program through a graphical interface. The same options as in console modes (default batch or interactive) are available when using the graphical mode.

Performances of our tool are related to the type and number of analysis modules that are registered through callbacks. When dealing with malware protected by a secure loader implementing thousand of decryption layers, the cost induced by the dynamic TCG to LLVM IR translation becomes prohibitive.

This is the main reason for which unpacking and normalization modules have to be executed

separately. The unpacking module is executed with a minimum number of information extraction call-backs activated. The normalization module registering and LLVM code generation begin only after the supposed original entry point.

3.3. Tests

Tests are done on a standard PC (Intel Core 2 Duo T6600, 4GB of RAM), executing the Windows 7 operating system. The test environment comprises our software, a QEMU VM with the Windows XP SP2 operating system, and the LLVM 3.1 compilation chain.

The test methodology consists to validate the main functions of our tool, by providing a convenient scenario. The following table presents the strategy and the first obtained results. This tests set is designed to be applied to executable for which we know the entry point and the semantic of its interaction with the operating system API.

When the malicious program is already packed, we identify the packer with PEID and validate the semantic of its interaction with the OS, by manually analyzing the program resulting from the unpacking module execution and by comparing the analysis result with information made available by antivirus software editors.

The evaluation of the functionalities of our tool relies on several third party pieces of software, developed to validate its efficiency and make easier its development.

- A modular packer, based on y0da's Crypter [22], able to selectively apply basic protection mechanisms (including notably anti-VM techniques and multi-layer encryption) that can be found in the commercial off-the shelf packers. This tool shall be used to unitarily test some detection and protection mechanisms, which are described in the literature but not necessarily implemented in commercial packers.
- A fixing tool which (on the basis of the dynamically collected information) adds two additional data sections containing respectively the imports and relocations information. To validate the correction of the relocations information, it modifies the image base and then applies relocations. This same tool is used to display this information (imports and relocations) after reconstruction.
- An obfuscator that proposes several obfuscation transformations, implemented as passes using the LLVM compilation chain

pass manager. These transformations are currently mainly control flow obfuscation (useless jump insertion, junk code insertion, control flow flattening, control flow flattening strengthened by using a hash function [5]). This tool has been developed to validate the efficiency of the normalization module and make easier its development.

3.4. Unpacking module efficiency

Test	Description
TST_UPK00	OEP localization. The goal is to automatically find the OEP.
TST_UPK01	Dynamic extraction of Imports and relocations information.
TST_FIX00	Reconstruction efficiency. First modify image base then apply relocations.

Table 4: Unpacking module tests set

To verify that the unpacking module generates accurate results, several packers were used with the same target program. In each case, we were able to retrieve automatically the original entry point of the protected executable and the memory map of the protected program.

Previous benchmarks that were given in [12] have been improved by two settings:

- Limit to the minimum the number of information extraction call-backs activated.
- Activate some optimization techniques, such as the instrumentation of some translation block terminator instructions (for example the "REPeat string operation" assembler instructions [9]), which are often used in decryption layers. If such a block terminator instruction is encountered, all information extraction call-backs are inhibited until the next translation block.

With these settings and for the tested packers, it is possible to strip the protected program from its protection loader in less than sixty seconds on a standard personal computer (Table 5). Observe that packers used during tests are a little old. We intend to complete this evaluation by using more recent packers and by enriching the protection mechanisms available in our packer.

Packer	Version	Time (seconds)
Armadillo	4.05	30
ASPack	2.12	15
ASProtect	1.23 RC4	20
PECompact	1.56	7
PEtite	2.3	9
PolyCryptPE	2.1	8
Shrinker	3.4	10

UPX	1.24w	5
Vbox	4.3	8
WWPack32	1.20	6
yC	1.2	7

Table 5: Unpacking module efficiency

3.4. Normalization module efficiency

The goal of the following tests is to unitarily validate the good behavior of the normalization module with regards to different types of calls (usual and less usual conventions, such as using a `ret` instruction to invoke an API function).

The LLVM representation is compiled, by using the back-end `llvm-ld` (or `llc` and then `gcc`). The resulting binary code is edited using a disassembler.

Test	Description
TST_NRM00	C, X86_StdCall and X86_FastCall reconstruction.
TST_NRM01	Variable arguments (vaargs) functions calls.
TST_NRM02	ret call reconstruction.
TST_NRM10	Useless jumps stripping.
TST_NRM11	Junk code deletion.
TST_NRM12	Code unflattening (CFF).
TST_NRM13	Code unflattening (ECFF).
TST_NRM14	Basic code virtualization unflattening.

Table 6: Normalization module tests set (including calling conventions support)

The support for several calling conventions is currently implemented in the normalization module. LLVM supports currently ten of them, among which we find notably: C, X86_StdCall and X86_FastCall.

Test	P size (asm lines)	O(P) size (asm lines)	D(O(P)) size (asm lines)	D(O(P)) time (seconds)
TST_NRM10	152	2152	72	88
TST_NRM11	152	1747	72	70
TST_NRM12	152	2380	72	95
TST_NRM13	152	2635	72	105
TST_NRM14	152	2133	72	85

Table 7: Normalization module efficiency (to give an idea, $D(O(P))$ time may be optimized)

First results show that standard optimization used in conjunction with the partial evaluation induced by the dynamic translation of target code to its LLVM representation are sufficient to drastically reduce and simplify the code under analysis. In each test (table 6), the obfuscated program $O(P)$ is “normalized” or recompiled to the same executable, $D(O(P))$.

Let us illustrate this concept on a simple “toy” example: consider the following program (which displays: “y = 22”), after unpacking and reconstruction:

```

..... ! entrypoint:
..... !   push    ebp
401001 !   mov     ebp, esp
401003 !   sub     esp, 10h
401006 !   mov     dword ptr [ebp-4], 0
40100d !   mov     dword ptr [ebp-0ch], 2
401014 !   mov     dword ptr [ebp-8], 0ah
40101b !
..... ! loc_40101b:
..... !   cmp     dword ptr [ebp-0ch], 6
40101f !   jnl     loc_40108c
401021 !   mov     eax, [ebp-0ch]
401024 !   mov     [ebp-10h], eax
401027 !   mov     ecx, [ebp-10h]
40102a !   sub     ecx, 2
40102d !   mov     [ebp-10h], ecx
401030 !   cmp     dword ptr [ebp-10h], 3
401034 !   ja      loc_40108a
401036 !   mov     edx, [ebp-10h]
401039 !   jmp     dword ptr [edx*4+data_4010a4]
401040 !   mov     dword ptr [ebp-4], 2
401047 !   mov     dword ptr [ebp-0ch], 3
40104e !   jmp     loc_40108a
401050 !   cmp     dword ptr [ebp-8], 0
401054 !   jng     40105fh
401056 !   mov     dword ptr [ebp-0ch], 4
40105d !   jmp     401066h
40105f !   mov     dword ptr [ebp-0ch], 6
401066 !   jmp     loc_40108a
401068 !   mov     eax, [ebp-4]
40106b !   add     eax, 2
40106e !   mov     [ebp-4], eax
401071 !   mov     dword ptr [ebp-0ch], 5
401078 !   jmp     loc_40108a
40107a !   mov     ecx, [ebp-8]
40107d !   sub     ecx, 1
401080 !   mov     [ebp-8], ecx
401083 !   mov     dword ptr [ebp-0ch], 3
40108a !
..... ! loc_40108a:
..... !   jmp     loc_40101b
40108c !
..... ! loc_40108c:
..... !   mov     edx, [ebp-4]
40108f !   push    edx
401090 !   push    strz_yd_402008
401095 !   call    dword ptr [msvcrt.dll:printf]
40109b !   add     esp, 8
40109e !   xor     eax, eax
4010a0 !   mov     esp, ebp
4010a2 !   pop     ebp
4010a3 !   ret     return 0;

```

The control flow graph (CFG) of such a program has the property to be flattened.

The normalization module execution produces (when we do not apply all optimizations) the following code:

```

..... !   push    eax
4011f1 !   mov     dword ptr [esp-0ch], 0ah
4011f9 !   mov     dword ptr [esp-8], 4
401201 !   dec     dword ptr [esp-0ch]
401205 !   add     dword ptr [esp-8], 2
40120a !   dec     dword ptr [esp-0ch]
40120e !   add     dword ptr [esp-8], 2
401213 !   dec     dword ptr [esp-0ch]
401217 !   add     dword ptr [esp-8], 2
40121c !   dec     dword ptr [esp-0ch]
401220 !   add     dword ptr [esp-8], 2
401225 !   dec     dword ptr [esp-0ch]
401229 !   add     dword ptr [esp-8], 2
40122e !   dec     dword ptr [esp-0ch]
401232 !   add     dword ptr [esp-8], 2

```



```

401237 ! dec     dword ptr [esp-0ch]
40123b ! add     dword ptr [esp-8], 2
401240 ! dec     dword ptr [esp-0ch]
401244 ! add     dword ptr [esp-8], 2
401249 ! dec     dword ptr [esp-0ch]
40124d ! add     dword ptr [esp-8], 2
401252 ! dec     dword ptr [esp-0ch]
401256 ! mov     eax, [esp-8]
40125a ! mov     [esp-18h], eax
40125e ! mov     dword ptr [esp-1ch],
        strz_yd_402010
401266 ! mov     ebp, esp
401268 ! lea     eax, [esp-1ch]
40126c ! mov     esp, eax
40126e ! call    crtddll.dll:printf_4012d8
401273 ! mov     esp, ebp
401275 ! mov     ebp, esp
401277 ! lea     eax, [esp+8]
40127b ! mov     esp, eax
40127d ! mov     esp, ebp
40127f ! xor     eax, eax
401281 ! pop     edx
401282 ! ret

```

We can observe here that the dynamic generation of code operated by our tool naturally unflattens the flatten code. The application of standard optimization transformations results in a program stripped from this obfuscation:

```

..... ! push    eax
4011f1 ! mov     dword ptr [esp-18h], 16h
4011f9 ! mov     dword ptr [esp-1ch],
        strz_yd_402010
401201 ! mov     ebp, esp
401203 ! lea     eax, [esp-1ch]
401207 ! mov     esp, eax
401209 ! call    crtddll.dll:printf_401268
40120e ! mov     esp, ebp
401210 ! mov     ebp, esp
401212 ! lea     eax, [esp+8]
401216 ! mov     esp, eax
401218 ! mov     esp, ebp
40121a ! xor     eax, eax
40121c ! pop     edx
40121d ! ret

```

Observe however that obfuscation transformations studied from now on are quite basic. We intend to complete this evaluation by using code virtualization tools that are commercially available (and by completing the obfuscation passes currently available in our obfuscator).

Now that the program is de-obfuscated, we can expect to retrieve the high level code of our example program, thanks to a de-compiler (`printf("y=%d\n", 22);`). We have thus judged useful to test C and C++ back-ends of LLVM.

3.4. De-compilation efficiency

Test	Description
TST_DCOMP00	LLVM source back-ends testing.
TST_DCOMP01	Hexrays de-compiler.

LLVM C and C++ back-ends do not produce a code with a quality comparable to the one that can be obtained by using other de-compilation tools.

It may be observed that the decompiled code produced by state-of-art de-compilers, such as HexRays, is often erroneous (for example, the high level code of our example program provided by HexRays is `printf(0)`), because of unsupported calling conventions. We may take advantage of our normalization module to design a de-compiler adapted to malware (which in most case do not respect the usual calling conventions!).

4. Conclusion

Even if there is still work before obtaining a software that supports the set of software protection tools usable by malware authors, first results encourage us to pursue the study of generic methods of unpacking and normalization, with the goal to automate as much as possible the tasks conducted by an analyst.

More work has to be done to make virtual computer processing units (and the other components of a virtual computer) more resilient against the wide range of emulation detection techniques. The emulation must be as precise as possible, and such a consideration is not currently a priority for the developer community of virtual computers.

Concerning the unpacking module, even if a universal generic unpacking algorithm is not feasible, we can improve the main algorithm to fight the most recent protections. In particular, we can implement the following features:

- The dynamic adaptation of the memory area under monitoring, to capture the code moved in the protection or in a dynamically allocated memory area.
- The dynamic exploration of unused branches of the control flow, to deal with the environmental triggers and improve the code coverage.
- The dynamic acquisition of several processes contexts to deal with the “multi-fork” protection.

This tool provides a self sufficient piece of software for malware analysis. However, one of the future goals of this project is to develop the ability of the tool to interact with other analysis tools. By design, this tool may be able to collaborate with any software analysis tool based on the LLVM compilation chain. In addition, Vellvm (Verified LLVM) may be used to extract formally verified

implementations of de-obfuscation passes implemented by our tool.

We can at last imagine other uses of this tool, than malware threat analysis: detection scheme extraction, software protection solutions security evaluation, antivirus software robustness analysis.

10. References

- [1] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum, "Codesurfer/x86 : a platform for analyzing x86 executables", in *Compiler Construction*, Springer, pp. 139-139, 2005.
- [2] U. Bayer, A. Moser, C. Kruegel, and E. Kirda, "Dynamic Analysis of Malicious Code", in *Journal in Computer Virology*, vol. 2, no. 1, Springer, 2006, pp. 67-77.
- [3] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator", in *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41-46.
- [4] "Tiny code generator", <http://wiki.qemu.org/>.
- [5] J. Cappaert and B. Preneel, "A general model for hiding control flow", in *Proceedings of the tenth annual ACM workshop on Digital rights management*, 2010, pp. 35-42.
- [6] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems", *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 265-278, 2011.
- [7] T. Dullien and S. Porst, "Reil: A platform-independent intermediate representation of disassembled code for static code analysis", *CanSecWest*, 2009.
- [8] P. Ferrie, "Attacks on Virtual Machine Emulators", in *Proceedings of the 2006 AVAR Conference*, Auckland, New Zealand, December 3-5, 2006.
- [9] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual, Instruction Set Reference", 2012.
- [10] S. Josse, "Secure and advanced unpacking using computer emulation", in *Proceedings of the AVAR 2006 Conference*, Auckland, New Zealand, December 3-5, pages 174-190, 2006.
- [11] S. Josse, "Rootkit detection from outside the Matrix", in *Journal in Computer Virology*, vol. 3, pages 113-123. Springer, 2007.
- [12] S. Josse, "Dynamic analysis and detection of viral code in a cryptographic context". PhD Dissertation, Ecole polytechnique, 2009.
- [13] M. Kang, P. Poosankam, and H. Yin, "Renovo: a hidden code extractor for packed executables", *Proceedings of the 2007 ACM workshop on Recurring malware*, pp. 46-53, 2007.
- [14] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation", *International Symposium on Code Generation and Optimization*, pp. 75-86, 2004.
- [15] Microsoft, "Microsoft Portable Executable and Common Object File Format Specification, revision 8.0", <http://msdn.microsoft.com/>, 2013.
- [16] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation", *ACM Sigplan Notices*, vol. 42, no. 6, pp. 89-100, 2007.
- [17] G. Portokalidis, A. Slowinska, and H. Bos, "Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation", in *Proceedings of the 2006 EuroSys conference*, ACM Press New York, NY, USA, 2006, pp. 15-27.
- [18] J. Rutkowska, "Red Pill... or how to detect VMM using (almost) one CPU instruction", 2005.
- [19] T. Scheller, "Llvm-qemu, backend for qemu using llvm", 2007, google Summer of Code, <http://code.google.com/p/llvm-qemu/>.
- [20] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A new approach to computer security via binary analysis" in *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper*, Hyderabad, India, Dec. 2008.
- [21] T. Teitelbaum, "Codesurfer", *ACM SIGSOFT Software Engineering Notes*, vol. 25, no. 1, p. 99, 2000.
- [22] "Yoda's Protector", 2012, <http://yodap.sourceforge.net/>.
- [23] J. Zhao, S. Nagarakatte, M.M.K. Martin, S. Zdancewic, « Formalizing the LLVM intermediate representation for verified program transformations », in *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 427-440, 2012.