M.Sc. Thesis
Master of Science in Engineering

DTU Compute
Department of Applied Mathematics and Computer Science

Attacking virtualization-based obfuscation

Recovering lost instructions from embedded emulators of virtual machine architectures

Valdemar Carøe (s153859)

Kongens Lyngby 2021

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Matematiktorvet
Building 303B
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Abstract (English)

Increasing advancements in software protection products and their protection techniques are causing severe problems for modern anti-virus products that are now struggling to distinguish between ”clean” software and ”malicious” software for software that has been protected by means of packing, obfuscation and/or virtualization. This study aims to determine if it is possible to revert software virtualization and restore a protected procedure to its original machine code state. Specifically, it investigates the possibility of restoring protected procedures both manually (by means of dynamic analysis) and automatically (by means of static analysis). In this context, dynamic analysis refers to executing and monitoring the protected software while static analysis refers to reading and parsing the protected software.

To test whether it is possible to restore a protected procedure to its original machine code state, analyses were conducted for two distinct virtual machine architectures supported by a market leader in the software protection industry known as Themida. These analyses included in-depth dissections of the virtual machine architectures as well as manual attempts at restoring protected procedures and later automating that same process. The results of these analyses were working methodologies for manually and automatically restoring protected procedures to their original machine code states.

The resulting methodologies explicitly confirm that it is indeed possible to restore protected procedures to their original machine code states and furthermore shows that it is possible to do so without any loss of information. However, as the thesis work was carried out on the basis of Themida, all the herein documented results are not necessarily applicable to other software virtualization products.

Abstract (Danish)

Stigende fremskridt inden for softwarebeskyttelsesprodukter og deres beskyttelsesteknikker forårsager alvorlige problemer for moderne anti-virus produkter, der nu møder store udfordringer i at skelne mellem ”clean” software og ”malicious” software, for software der er blevet beskyttet ved hjælp af pakning, obfuskering og/eller virtualisering. Denne tese forsøger at afgøre hvorvidt det er muligt at gendanne en beskyttet procedure til dens oprindelige maskinkode tilstand. Specifikt undersøger den muligheden for at gendanne beskyttede procedurer både manuelt (ved hjælp af dynamisk analyse) og automatisk (ved hjælp af statisk analyse). I denne sammenhæng henviser dynamisk analyse til kørsel og overvågning af den beskyttede software, hvorimod statisk analyse henviser til læsning og bearbejdelse af den beskyttede software.

For at teste, om det er muligt at gendanne en beskyttet procedure til dens oprindelige maskinkode tilstand, blev der udført analyser for to forskellige virtuelle maskine arkitekturer som er understøttet af en markedsleder inden for softwarebeskyttelsesindustrien kendt som Themida. Disse analyser omfattede dybdegående dissektioner af de virtuelle maskine arkitekturer samt manuelle forsøg på at gendanne beskyttede procedurer og senere automatisere den selvsamme proces. Resultaterne af disse analyser var metoder til manuel og automatisk gendannelse af beskyttede procedurer.

De resulterende metoder bekræfter eksplicit, at det faktisk er muligt at gendanne beskyttede procedurer til deres oprindelige maskinkode tilstande og viser desuden, at det er muligt at gøre det uden tab af information. Da afhandlingens arbejde blev udført på basis af Themida, er alle de heri-dokumenterede resultater dog ikke nødvendigvis gældende for andre softwarevirtualiseringsprodukter.

Preface

This master thesis was prepared at the department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfillment of the requirements for acquiring a master's degree in Computer Science and Engineering.

Kongens Lyngby, June 7, 2021

Valdemar Carøe

Valdemar Carøe (s153859)

Contents

CHAPTER 1

Introduction

There are many legitimate reasons for wanting to protect a proprietary software product from reverse-engineering and analysis efforts, as the success of a business is often tightly correlated to the confidentiality of the inner workings of its software product(s).

For instance, licensed software requires a proprietary license validation algorithm, which, if successfully recovered by an analyst, can potentially be reverted and used to generate valid licenses for the software in question. This can prove fatal to the business model for the software product.

In the gaming industry, cheaters have invaded first-person shooter games, where they abuse in-game mechanics to see through walls or perform automatically-aimed fatal headshots on any opponent player in visible range. A number of legitimate players are being coerced into quitting these games out of spite for abusive players, which can lead to massive revenue losses for game product vendors. In order to reduce the amount of abusive players, most vendors deploy an anti-cheat solution, whose sole purpose is to detect modifications made to the game client. However, the effectiveness of an anti-cheat solution is directly correlated to the confidentiality of the detection routines, which, if successfully recovered by an analyst, can be bypassed to allow abusive players to continue their cheating efforts while remaining undetected by the game platform.

There are a multitude of alternative legitimate reasons for desiring to limit external access to the internal workings of a proprietary software product. For this particular reason, a variety of software protection techniques have been developed over the years including software packing, software obfuscation, and last but not least, software virtualization.

These technologies have all been researched en masse, resulting in the development and public distribution of various methods of reversal for all except the software virtualization technique. Software virtualization therefore remains the de facto standard for present day protection of software products.

Although initially conceived in good faith, these software protection techniques are also being used for illegitimate purposes. Due to the proven resilience against reverse-engineering, that software virtualization provides, this technique has found its way into the malware development community. While this might not seem alarming at a glance, the ability to protect malware with strong software protection is proving to be a vast problem for the anti-virus industry.

In a presentation[9] from 2010 by Microsoft Malware Protection Center employee, Zhenxiang Jim Wang, whose primary work at Microsoft involves analysis and research of virtual machine technology, he addresses a number of issues relating to the pervasive usage of software protection technologies in modern malware. The main points raised throughout the presentation, is that emulation-based analysis of binaries that have been protected by software virtualization often leads to exhaustion of resources and execution times far too long to tolerate, especially for on-access scans. Anti-virus products are therefore in need of static unpacking and devirtualization templates for individual software protection products, for which the development cycle requires a massive investment of time.

In another paper[13] from 2014 by Dr. Mafaz Mohsin Khalil Al-Anezi from Mosul University in Iraq, it is estimated that over 80% of modern malware samples are packed using an arbitrary software protection product. However, given that this is presented as a growing trend, it can be rightfully assumed that this percentage has further increased since then.

It is therefore of paramount importance that research be made in this field, such that anti-virus products can once again protect its end-users from the clutch of malware attacks.

1.1 Aim of the study

As will be explained later in section 2.3, software virtualization makes permanent modifications to an executable binary in an effort to hide the implementation details of a protected procedure. This thesis shall determine if it is possible to revert software virtualization and restore a protected procedure to its original machine code state.

Because the area of research is vast and unexplored, this thesis will focus solely on *Themida*, one of two market leaders in the software protection industry. Furthermore, to simplify the analyses and their generated results, only 32-bit architectures will be considered throughout this thesis. However, the findings presented in this thesis will also apply to equivalent architectures in the 64-bit domain.

Throughout the thesis, separate analyses will be presented for two distinct software virtualization architectures supported by Themida. These analyses will be carried out independently and will each attempt to answer the following research questions.

R1. *How can a virtualized procedure be recovered manually using dynamic methods?*

R2. *How can a virtualized procedure be recovered automatically using static methods?*

Both of these analyses will be carried out and documented in an indentical manner, such that similarities and differences can be trivially identified.

Finally, it shall be discussed if the discovered results are capable of bringing much needed value to the anti-virus industry or if further research and adjustments are necessary.

1.2 Report structure

This section presents a chapter-by-chapter summary of what to expect from this thesis.

Chapter 2, Software protection, iterates the foundational knowledge necessary to understand the contents of this thesis, including various techniques within the area of research such as packing, obfuscation and virtualization.

Chapter 3, Themida, introduces the virtual machine architectures supported by Themida and a variety of configurations applicable to these. The chapter also introduces the target of the analyses and details how to prepare for an analysis by unpacking an executable binary that has been protected by Themida.

Chapter 4, The CISC architecture, details an analysis of the CISC virtual machine architecture supported by Themida. This includes dissecting the virtual machine architecture, documenting how the interpreter works, and deriving answers to the research questions for the given virtual machine architecture.

Chapter 5, The FISH architecture, details an analysis of the FISH virtual machine architecture supported by Themida. This includes dissecting the virtual machine architecture, documenting how the interpreter works, and deriving answers to the research questions for the given virtual machine architecture.

Chapter 6, Results and evaluation, gathers the results from the previous chapters, 4 and 5 respectively, and evaluates these results according to various metrics such as efficiency, effectiveness and coverage.

Chapter 7, Conclusion, assesses the overall success of the thesis by detailing the strengths and weaknesses of the developed solutions. Finally, the chapter concludes the thesis by identifying subjects for future research and development.

CHAPTER 2

Software protection

This chapter contains an introduction to various techniques that are commonly found in commercial software protection products. This includes how software packers protect binaries from static analysis as well as from being debugged, modified and/or executed in virtualized environments. The chapter further introduces commonly used mechanisms for obfuscating native machine code to make reverse engineering efforts more tedious and more time consuming. Finally, the chapter is concluded with an introduction to the cutting-edge software protection technique known as code virtualization.

In order to understand how software protection helps protect binaries and how any such type of protection can be bypassed to recover an unprotected binary, it is important to understand how binaries work in the first place.

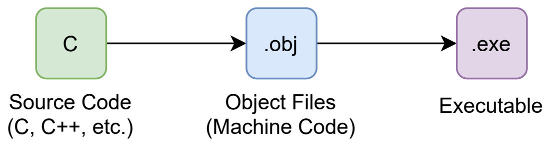


Figure 2.1: The native compilation process.

As depicted in figure 2.1 above, when compiling a binary in a low-level unmanaged language such as *C* or *C++*, all source code files (*.c* or *.cpp* respectively) are compiled into separate object (*.obj*) files. These object files contains the machine code generated as a result of the individual compilations. Once all object files have been generated from the source code, these are *linked* together to form the final executable binary.

It is not important to understand exactly how these concepts work in greater detail, but it is important to understand that the generated binary now contains the source code of the application in a machine code format formally known as *assembly*. Any application built for the Windows platform will have its source code transformed into Intel x86 assembly for 32-bit applications or into Intel x86-64 assembly for 64-bit applications respectively.

Due to the fact that the source code is still present in the compiled binary albeit in a machine code format rather than that of the original programming language, the internal workings of the binary is inherently deductible by analysing the machine code residing in the binary and reconstructing the original code flow. This is known as *reverse engineering*.

A compiled Windows executable follows the PE file format shown in figure 2.2 below.

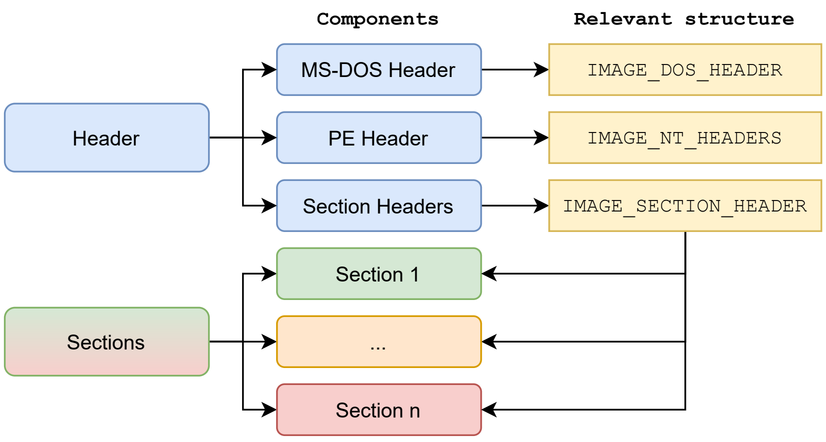


Figure 2.2: The PE file format.

The PE file format contains a header with multiple subcomponents and a variable amount of subsequent sections.

The first subcomponent of the header is the legacy MS-DOS header. The MS-DOS header is followed by an MS-DOS executable stub that will only ever be reached if attempts are made to execute the binary in DOS mode. Most Windows compilers, if not all, will write the same universally accepted MS-DOS executable stub that prints "*This program cannot be run in DOS mode*" to the terminal. At the end of the MS-DOS header is a field that contains the offset of the PE header.

The PE header contains two subcomponents, the *file header* and the *optional header*. The file header contains various information about the executable file, but most importantly it contains the intended file architecture (32-bit or 64-bit), the amount of sections in the file and the size of the optional header in bytes. The optional header contains a plethora of information, but most notably it contains the image base and size of the executable (i.e. where in memory the executable has been mapped), the address of the entry point of the executable (i.e. which procedure in the executable is the first to be run), and finally a list of *data directories*. These data directories points to various clusters of information such as which procedures the executable *imports* from other binaries or which procedures the executable *exports* to other binaries.

The PE header is followed by a linear array of section headers. These section headers contains information about individual sections in the executable such as the name of the section, the physical offset and size of the section data in the executable file on disk, the virtual address and size of the section when mapped into memory and various characteristics about the section such as section type, section alignment and section permissions.

Subsequent to all the header subcomponents, the executable file contains the data chunks for the various file sections. By convention, the primary code section that contains the executable code of the binary is the `.text` section which is usually listed as the first section in an executable file. Following the code section is an arbitrary number of data sections such as the `.bss` section which contains uninitialized global or static data, the `.data` section which contains initialized global or static data or the `.rsrc` section which contains resource data such as images or icons used throughout the executable.

Executable code sections in a compiled binary contain raw machine code in which each encoded instruction has a one-to-one correlation to a relevant assembly language mnemonic. This essentially makes raw machine code interpretable in a humanly readable format. By parsing through the code sections of a compiled executable binary, most notably the `.text` section, it is possible to read the machine code stored in the binary. This is usually done using a *disassembler* as shown in figure 2.3 below.

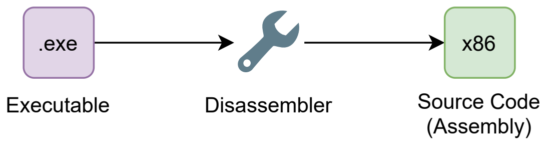


Figure 2.3: The native decompilation process.

A disassembler is a primitive file parser that is able to locate executable code sections in a binary file and translate the encoded raw machine code instructions to their corresponding assembly language mnemonic. This is essentially the enabler of reverse engineering of executable binaries, as it allows an analyst to read the underlying code stored in a given binary.

However, a *disassembler* is not to be confused with a *decompiler*, which is a similar file parser with a far more sophisticated capability of further translating the assembly code resulting from the disassembling of the executable binary to a higher-level language such as C or C++. In the majority of cases, this allows for much faster analysis of binaries as higher-level languages are more concise and easier to read.

2.1 Software packing

Due to the structure of the PE file format as discussed above and the fact that executable binaries contains raw machine code in their respective code sections that can be parsed and read using a disassembler, these binaries are neither efficient in terms of size nor resistant towards code analysis efforts. In an attempt to combat these known weaknesses, a series of software protection techniques have been developed over the years. The first such technique to be conceived was *software packing* which can be seen as the first line of defense against reverse engineering efforts.

Back when software packing was initially introduced, the capacity of storage devices were incomparable to current standards and transmission speeds were even worse. To this end, software vendors would begin to compress their executable binaries and turn them into archives that would self-extract during runtime. This greatly reduced the storage costs necessary to transfer these components and allowed installation medias such as CD-ROMs to carry packages that would normally exceed the maximum capacity of the storage media.

Interestingly, creating self-extracting compressed executable binaries also proved useful for less benevolent actors such as malware developers. Compressing all the code sections of an executable binary would make reverse engineering infeasible as the original raw machine code stored in these code sections would no longer be stored in its original format. Furthermore, this type of compression also masked literal strings stored in data sections as well as modify signatures of the raw machine code stored in code sections, allowing potentially malicious executable binaries to bypass detection from signature-based scanning performed by antivirus softwares in order to detect known malicious patterns.

A number of alternatives to this type of software packing exists, in which the executable binary is packed using encryption rather than compression or using both encryption and compression in combination. The packing process is the exact same for either variant of the technique and is depicted in figure 2.4 below.

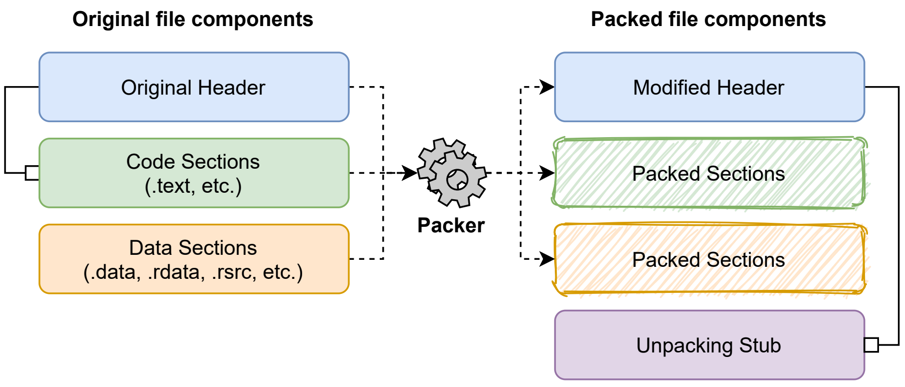


Figure 2.4: The general software packing process.

The software packing technique works by compressing and/or encrypting all sections of the executable binary and inserting an unpacking stub responsible for decompressing and/or decrypting the other sections of the executable binary during runtime. The entry point of the application is changed from the original entry point (OEP) located in the `.text` section to the unpacking stub located at an arbitrary location in the file. Once the file is executed, the new entry point of the executable binary will point to the unpacking stub which will restore the packed sections to their original states and transfer execution to the original entry point and thereby continue execution of the executable binary without any noticable difference to the end-user.

However, this software packing solution suffers a major vulnerability that makes it insufficient as a stand-alone protection technique. Due to the fact that the unpacking stub of a protected executable binary will restore all other sections to their original states during runtime, it is possible to simply dump the binary from memory after the self-extraction process and thereby procure an unprotected version of the executable binary. For this very reason, a range of protection techniques have been conceived to complement this generic software packing technique.

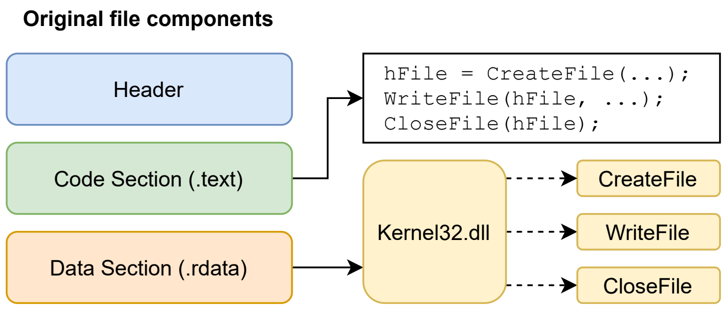
Anti-debugging techniques are implemented to detect or deter the use of debuggers during execution of a binary. For example, it is possible to detect a live debugger by invoking the `IsDebuggerPresent` Windows API or checking the `NtGlobalFlag` member of the Process Environment Block (PEB), both of which can be used to indicate if the current process is being controlled by a debugger. An alternative way of detecting that the current process is being debugged is by fetching timestamps at two distinct points in the execution flow of the application and comparing these to see if the difference between the timestamps exceeds the expected default execution time. This should only happen when an analyst is manually stepping through the binary code using a debugger as the process is then paused on each instruction until the analyst instructs the debugger to continue execution. There is a plethora of other methods for detecting or even breaking debugging efforts, but it would require a sizable amount of effort and space to describe them all in greater detail while adding little to no additional value towards the goal of this thesis and they have therefore been omitted.

Anti-virtualization techniques are implemented to detect the use of virtualized environments such as VMWare or VirtualBox and to change the runtime behaviour of the executable binary when running in any such sandboxed environment. It is no secret that malware analysts or reverse engineers in general use virtualized environments to run, unpack or analyse executable binaries. This is both due to the fact that the repercussions of running these potentially malicious binaries does not have an impact on the host system as well as the fact that these guest systems can be rolled back to a previous point in time once the analysis of a binary has been concluded. By forcing the flow of binaries in virtualized environments to either terminate the process prematurely or to execute an alternative branch of the application that is only reached from within guest systems, it is possible to deter reverse engineers of low to mediocre skill from analysing a given executable binary or to trick them into misinterpreting what the binary actually does. There are many different ways to determine if the current process is running in a virtualized environment such as checking for known services, processes, files, registry keys or even MAC addresses since hypervisors such as VMWare uses static MAC address prefixes.

Anti-tampering techniques are implemented to prevent an end-user from making modifications to either the raw file on disk or to the runtime process once mapped into memory. This type of software protection serves a variety of different purposes such as to stop an advanced analyst from patching out other software protection features present in the executable binary or to stop cracking groups from removing proprietary integrity checks in licensed software. This is achieved by computing a checksum for each code section in the binary during runtime and comparing the results to precomputed unmodified checksums.

Anti-dumping techniques are implemented to prevent the inherent flaw in software packers as described above, in which an analyst can dump a compressed and/or encrypted binary from memory after the self-extraction process has been carried out. A naive approach to achieving this type of protection is to clear the entire file header in memory by overwriting it with null-bytes, as the file header has already lived out its purpose once the file has been mapped into memory. However, while the file header may no longer exist in memory and while dumping this memory mapped file may no longer create runnable executable binaries, it is possible to repair these dumps by copying the header from the protected executable binary on disk and modifying the entry point field so that it points to the original entry point found in the unpacked `.text` code section.

However, an effective approach that most modern software packers appear to use is the *Import Address Table (IAT) hijacking* technique in which a proxy call stub is inserted into the protected executable binary. This proxy call stub is responsible for dynamically resolving and transfering execution to requested imported procedure calls. In unprotected executable binaries, imported procedures are invoked as shown in figure 2.5 below.



```
hFile = CreateFile(...);
WriteFile(hFile, ...);
CloseFile(hFile);
```

Figure 2.5: Imported procedures in an unprotected file.

Inside any executable binary that follows the PE file format there is an *Import Name Table (INT)* which contains the names of libraries referenced throughout the executable binary as well as all external procedures referenced from these libraries. These are known as *imports*. When an executable binary is being loaded into memory, the loader attempts to resolve all imports found in the INT by ensuring that the referenced libraries are also loaded into memory and by writing the location of all imported procedures from these loaded libraries into the *Import Address Table (IAT)*.

For example, in figure 2.5 above, the executable binary references three distinct procedures from the *kernel32.dll* library. Effectively, this means that the INT will contain the name of the library, *kernel32.dll*, as well as the names of the three imported procedures *CreateFile*, *WriteFile* and *CloseFile*. Once the loader attempts to map this executable binary into memory, the *kernel32.dll* library will be loaded aswell and the location of these three imported procedures will be written into the IAT. Now in order to invoke e.g. *CreateFile*, the executable binary will invoke an entry from the IAT corresponding to where the executable binary expects the imported *CreateFile* location to be stored.

In protected executable binaries, for which the *IAT hijacking* technique has been applied, imported procedures are invoked as shown in figure 2.6 below.

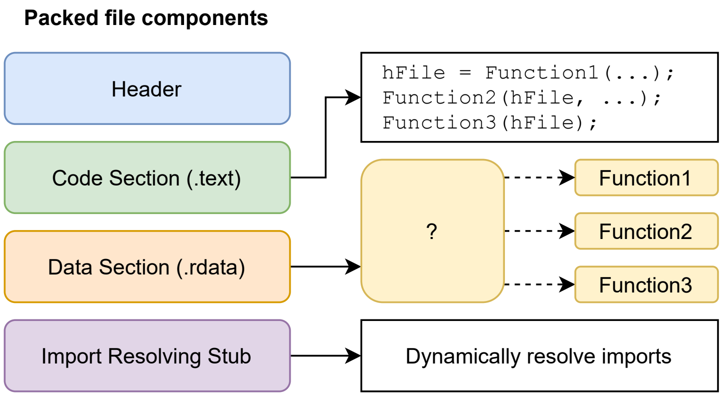


Figure 2.6: Imported procedures in a protected file.

Basically, IAT hijacking works by stripping the INT of the executable binary so that the loader is not made aware of any imports that needs to be resolved. As a result, the IAT is never populated by the loader, and the location of imported procedures used throughout the executable binary are therefore not known. Since the INT is no longer populated and since the IAT still needs resolving in order to make the executable binary functional, the software packer performs dynamic resolving of the imports during the unpacking process.

Any attempts at dumping the unpacked executable binary from memory will now generate faulty binaries as the imported procedures are never resolved by the loader. However, manual reconstruction of the INT is possible so long as the analyst is able to figure out which imported procedures the entries of the IAT refers to. For this reason, most modern software packers apply an additional layer of security to this technique by pointing the IAT to convoluted obfuscated code chunks that eventually flow into an imported procedure rather than storing the raw location of the imported procedure in the IAT.

It should be noted that for each of the software packing techniques discussed throughout this section thus far, there exists a direct counter technique. In other words, none of these software packing techniques are irreversible. With the correct approach and methodology it is possible to bypass every single one of these software packing techniques or to revert the modifications they may have imposed on the executable binary entirely. Realistically, this means that while software packing can make reverse engineering an executable binary very hard and tedious for the mediocre analyst, it should always be assumed that someone will eventually manage to successfully create a functional dump of the executable binary in its unpacked (and therefore unprotected) state.

Because of this, advanced modern software packers also use *obfuscation* and *virtualization* to make permanent modifications to the binary that does not self-revert during runtime.

2.2 Software obfuscation

When building an executable binary in C or C++, the compiler performs a series of optimizations on the source code before assembling it into machine code instructions. The product of this optimization is usually more concise and easier to read than the original version of the code. Software obfuscation makes use of this observation by intentionally reducing the optimization of the machine code stored in an executable binary. While the executable binary might suffer a loss of performance, the additional noise in the machine code instructions makes analysis efforts less productive and far more time consuming. Due to the fact that these obfuscation techniques are applied to the raw machine code stored in an executable binary, this section will focus solely on the obfuscation of Intel x86 assembly language constructs in 32-bit execution mode.

Obfuscation practically aims to turn an arbitrary group of instructions into a larger group of instructions that accomplishes the exact same functional goal. This can be done in a variety of ways. One such method is by emulating one or more instructions using different instructions as depicted in figure 2.7 below.

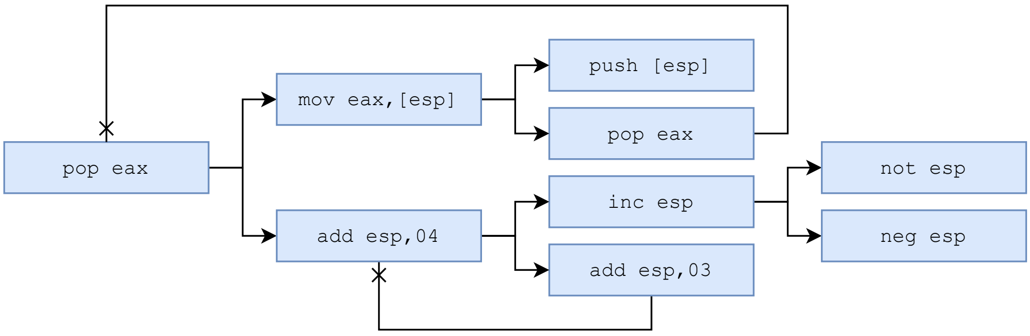


Figure 2.7: Instruction expansion obfuscation.

The `pop` instruction is responsible for moving the top element of the stack into a designated destination and removing it from the top of the stack. In this case, that destination is `eax`. There are many ways to emulate this behaviour such as by moving the top element of the stack (`[esp]`) into `eax` and then, since the stack grows backwards, incrementing the stack pointer (`esp`) by the size of one stack element which is 4 bytes in 32-bit mode. The `pop eax` instruction has now been split into two instructions that performs the same task.

This approach can then be further applied to the newly generated construct. For example, moving the top element of the stack into `eax` can be accomplished by using the `push` instruction to move it onto the top of the stack and then using the `pop` instruction to move it from the top of the stack into `eax`. Interestingly, the construct has now arrived at another `pop` instruction, which is also the core of this whole construct expansion. This is indicative of the fact that this expansion approach can potentially be performed an unlimited amount of times.

It should be mentioned that there are multiple different ways to emulate most constructs and that almost every single instruction in the Intel x86 instruction set can be emulated using other instructions from the same instruction set. The most effective way to counteract this type of obfuscation is to do it in reverse by collapsing instructions in larger instruction groups to form smaller instruction groups. This technique is known as *peephole optimization* and requires a large series of precompiled optimization patterns, as it is only possible to collapse patterns that are known beforehand. However, it is also possible to increase the size of instruction groups based on constant expansion rather than instruction expansion. One such method is depicted in figure 2.8 below.

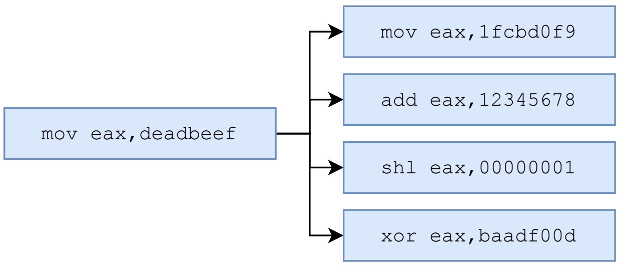


Figure 2.8: Constant expansion obfuscation.

Instead of assigning a desired constant value directly to a designated destination, the constant value is dynamically constructed through the usage of arithmetic and/or similar instructions. Since constant values can be constructed in an infinite amount of ways, there are no limitations to how substantially this obfuscation technique can be applied to expand an arbitrary instruction group. The most effective way to counteract this type of obfuscation is to compute the result of these operations and to replace the instruction sequence with the resulting constant value. This technique is known as *constant folding*. Another similar method of constant obfuscation is depicted in figure 2.9 below.

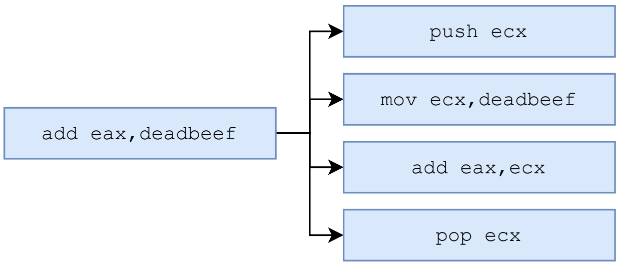


Figure 2.9: Constant duplication obfuscation.

Instead of using a desired constant value directly in an arithmetic operation, the constant value is proxied through a temporary carrier. The `push` and `pop` instructions are used to backup the original contents of the temporary carrier and to restore the contents after it has served its purpose. This example could be simplified in the C programming language as expanding [`int x = 0xdeadbeef;`] into [`int tmp = 0xdeadbeef; int x = tmp;`].

In most cases this technique is used in conjunction with the constant expansion obfuscation technique, as any combination of arithmetic, bitwise or similar instructions could be used to form the constant value that goes into the temporary carrier. The most effective way to counteract this type of obfuscation is to perform elimination of the temporary carrier by writing the resulting constant value directly into the arithmetic operation. This technique is known as *constant propagation.*

Next to expanding existing instruction groups, modern obfuscation engines also insert seemingly random code that does not have any effect on the state of the application and thus does not impact the overall task carried out by the instruction group. This type of obfuscation is known as *deadcode insertion* and is depicted in figure 2.10 below.

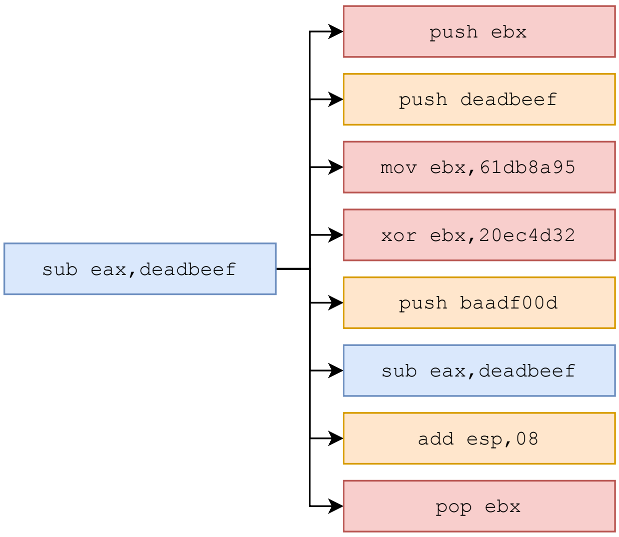


Figure 2.10: Deadcode insertion obfuscation.

In this regard, deadcode refers to code that does not attribute any meaningful changes to the instruction group. In other words, deadcode instructions are inserted for the sole purpose of adding mass to the instruction group and does not make lasting changes to the state of the application throughout its execution. When this type of obfuscation has been carried out, it can be difficult to locate the "real" instructions that make stateful changes to the application.

In the above figure, there are two subgroups of deadcode constructs each of which have been given a different color. In the orange group, the `add esp,08` instruction removes two 32-bit objects from the top of the stack, effectively making the previous two `push` instructions obsolete. In the red group, the `pop ebx` instruction removes a 32-bit object from the top of the stack and stores it in `ebx`, effectively overwriting any prior changes made to `ebx`. Furthermore, since the preceding `push` instruction operates on the same register as the `pop` instruction, the two instructions ends up cancelling eachother.

The most effective way to counteract this type of obfuscation is to perform elimination of any instruction in the instruction group that does not propagate any meaningful information to the state of the application. This technique is known as *deadcode elimination*.

However, not all code obfuscation techniques increases obscurity by increasing mass. Instead, some obfuscation techniques aims at making it harder to navigate the instruction group for analysis purposes or to visualize the entity as a whole. These techniques usually pertain to modifying or otherwise controlling the flow of the application. The most basic of these techniques is *control flow obfuscation* as depicted in figure 2.11 below.

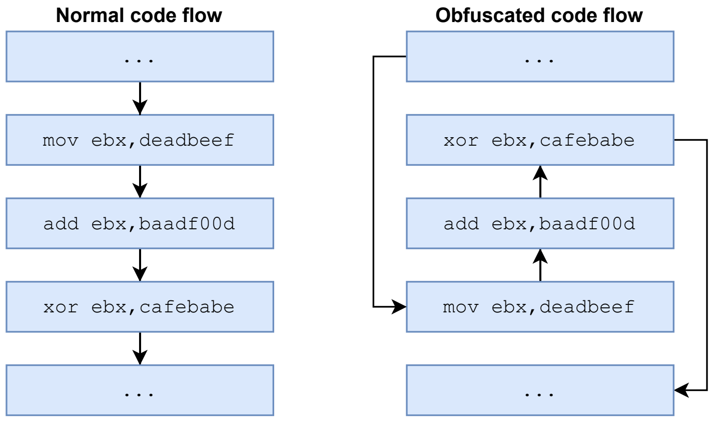


Figure 2.11: Control flow obfuscation.

In a compiler-generated version of an instruction group, the code flow is linear. In other words, the execution of the code flows in a single direction. The obvious exceptions to this claim are loops or similar constructs. When applying basic control flow obfuscation, the execution order of non-looping constructs is shuffled so that this basis no longer holds true.

This is usually done by splitting an instruction group into smaller subgroups and placing each subgroup in a random location in the executable binary. These subgroups are then connected by appending a `jmp` instruction that transfers program control to the next subgroup in the original order of execution. This makes no logical changes to the executed instruction group, but it does make manual analysis of the executable binary infeasible as the analyst can only hover over a single subgroup in a disassembler at a time and therefore has to remember or write down every single subgroup he or she passes through.

In terms of automatic analysis, however, this is very straight-forward to implement. The most effective way to counteract this type of obfuscation is to collect the original instruction group by tracing through each individual subgroup. This is done using the appended `jmp` instruction to locate the next entry in the subgroup chain. When the entire instruction group has been collected, the intermediate `jmp` instructions are then eliminated. This technique is known as *control flow optimization*.

There are also other variants of this type of obfuscation, one of which is *branch obfuscation* as depicted in figure 2.12 below.

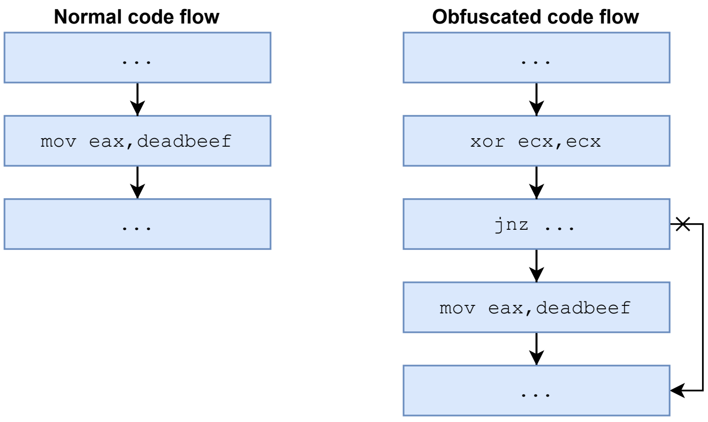


Figure 2.12: Branch obfuscation.

The idea behind this obfuscation technique is the same as for *control flow obfuscation*, but instead of using unconditional `jmp` instructions, this technique makes use of conditional `jcc` instructions. Due to the fact that each inserted `jcc` instruction has two potential destinations, one for true and one for false, it is less trivial to analyse, as it is no longer possible to simply follow the instruction destination.

Usually, when implementing this type of obfuscation, the conditional `jcc` instruction is used to emulate an unconditional `jmp` instruction by forcing a specific condition to hold true prior to reaching the `jcc` instruction. In this case, a `xor ecx,ecx` instruction resets the value of `ecx` to zero and thereby sets the `ZF` (zero-flag) field of the `EFLAGS` register to `true`. This results in the `jnz` (jump if not zero) instruction never transfering execution to its true-branch destination.

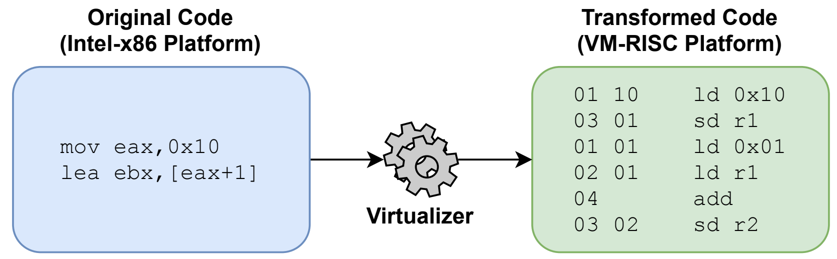
The most effective way to counteract this type of obfuscation is to verify if the conditional `jcc` instruction depends on a known or an unknown state. If the instruction depends on an unknown state, i.e. a state that is not specified by the instruction group itself, it can be assumed that the conditional `jcc` instruction originates from the original instruction group and is not a part of this obfuscation technique. However, if the instruction depends on a known state, the state can be predicted by computing the outcome of the instructions leading to the known state and it is thereby possible to decide whether or not to follow the conditional branch. This technique is known as *branch prediction*.

Conclusively, the obfuscation techniques shown in this section all suffer the same level of weakness against generic compiler theory constructs and optimization algorithms. This makes them less than ideal as primary drivers for modern software protection. Modern software protection thus require more specialized obfuscation techniques that are not defeated by generic optimization techniques.

2.3 Software virtualization

Software virtualization, also known as *code virtualization* or *VM-based code obfuscation*, is a sophisticated obfuscation technique based on instruction emulation rather than instruction group expansion. This technique in its many forms is the flagship product of leading solutions in the software protection market, including *Themida* by *Oreans Technologies* and *VMProtect* by *VMProtect Software*.

This obfuscation technique can be summarized as translating the native machine code of an executable binary into instructions from an arbitrary, potentially custom, architecture and emulating these instructions by embedding a virtual machine capable of interpreting these into the executable binary. An example follows in figure 2.13 below.



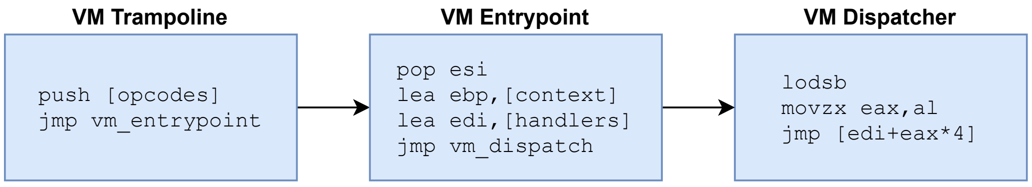
```
mov eax,0x10
lea ebx,[eax+1]
```

```
01 10     ld 0x10
03 01     sd r1
01 01     ld 0x01
02 01     ld r1
04        add
03 02     sd r2
```

Figure 2.13: Example of code transformation for a stack-based RISC VM.

The original x86 code depicted in figure 2.13 above mimics a simple assignment in C, which could have also been written as [`int x = 0x10; int y = x+1;`]. When protecting the executable binary with a software virtualization product, this native machine code is translated into an arbitrary custom architecture. In this particular case, that architecture is a made-up stack-based RISC architecture which draws heavy mnemonic insiration from the RISC-V instruction set architecture.

Once the original machine code has been translated into the custom architecture, a small virtual machine capable of interpreting and executing these custom instructions is embedded into the executable binary. Furthermore, the original machine code is removed from the binary and replaced with a trampoline gadget that loads the byte-code for the custom instructions to be executed and jumps into the embedded virtual machine. An example of this can be seen in figure 2.14 below.



```
push [opcodes]
jmp vm_entrypoint
```

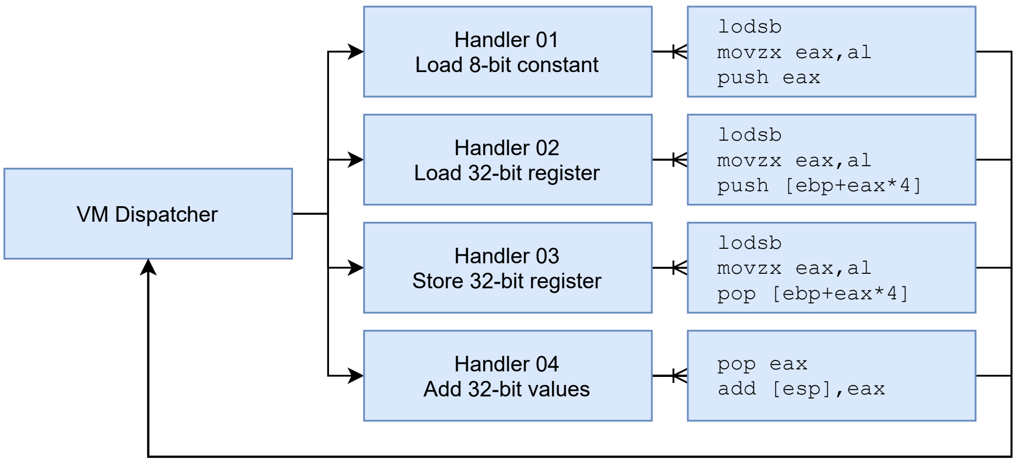
```
pop esi
lea ebp,[context]
lea edi,[handlers]
jmp vm_dispatch
```

```
lodsb
movzx eax,al
jmp [edi+eax*4]
```

Figure 2.14: Example of main stages in a stack-based RISC VM.

In the above figure, there are three undefined symbols, `opcodes`, `context` and `handlers`. The `opcodes` symbol refers to a buffer that contains the byte-code of the custom instructions to be executed by the virtual machine. The `context` symbol refers to a structure that contains all the variables necessary for the virtual machine to operate. This includes the general-purpose registers supported by the architecture as well as the base address of the executable binary in memory, which is used to align addresses for invoking procedures that are external to the virtual machine. In other implementations that are structured differently, the context may also contain other vital constructs such as an internal instruction pointer or a pointer to an emulated stack. The `handlers` symbol refers to a function table containing all the procedures responsible for emulating single instructions in the custom architecture. In other words, there is a designated single procedure for all instructions in the custom architecture instruction set, which is responsible for carrying out the expected behaviour of that single instruction.

As shown in figure 2.14 above, there are three main stages aside from the handler procedures. These are the `VM Trampoline`, the `VM Entrypoint` and the `VM Dispatcher`. Each protected function in the executable binary has its own `VM Trampoline` responsible for loading its particular byte-code buffer and flowing into the `VM Entrypoint`. However, all the individual `VM Trampoline` areas of protected functions flow into the same unique `VM Entrypoint`, which is responsible for loading all necessary constructs such as the `context` structure and the `handlers` table. Finally, the `VM Dispatcher` is responsible for invoking the individual instruction handlers as depicted in figure 2.15 below.



```
lodsb
movzx eax,al
push eax
```

```
lodsb
movzx eax,al
push [ebp+eax*4]
```

```
lodsb
movzx eax,al
pop [ebp+eax*4]
```

```
pop eax
add [esp],eax
```

Figure 2.15: Example of VM handlers for a stack-based RISC VM.

As seen in the above figure, each instruction handler is implemented as a micro procedure that performs a single trivial task, such as loading a 32-bit context register onto the stack or removing an element from the stack by storing it back into a 32-bit context register. After each handler procedure has been invoked, the execution flow returns to the `VM Dispatcher` for redirection into the next handler procedure in the byte-code sequence.

The above example showcases a simplified generic model for embedded virtual machines in software virtualization products. However, the model varies between different virtual machine architectures as will be discovered later in this thesis. This is especially true when accounting for virtual machine architectures supported by different software protection vendors. Some custom architectures mimics the Intel-x86 architecture closely by using a similar instruction set or by using similar general-purpose registers, while other architectures might have a completely different instruction set or support anything from 1 to 20 general-purpose registers. Some implementations even remove the single unique `VM Dispatcher` routine by appending a similar scheme to each of the handler procedures.

Whereas the obfuscation methods showcased in the previous section had simple counter techniques inspired by compiler-theory designs and algorithms, software virtualization is much more sophisticated and requires a far more tailored approach. Due to the varying nature of custom architectures, it is extremely infeasible, if not impossible, to create a catch-all algorithm for counteracting this type of obfuscation. Analysts must hence focus on a single specific architecture when devising a counter-agent.

However, the methodology of recovering instructions that have been protected by software virtualization can be generalized into the following high-level steps.

1. Deduce the functional responsibility of each individual handler procedure.
2. Map the byte-code sequence to handler procedures (recreate the code-flow).
3. Translate the custom architecture code-flow back to its original architecture.

Since most virtual machines operate with instruction handlers implemented as micro procedures, the first step is generally the easiest. These handlers are usually obfuscated, but using counter-techniques elaborated on in section 2.2 above, it is possible to reduce these handlers to nothing but their core instructions. Once reduced to its core, it is usually trivial to deduce the purpose of each individual handler procedure.

Mapping the byte-code sequence to its respective handler procedures in order to recreate the code-flow can range from extremely trivial to very tricky depending on how the virtual machine is implemented. Some embedded architectures operate on encrypted byte-code sequences that are decrypted during runtime, as will be shown later on in this thesis.

Lastly, translating the recreated code-flow of a custom architecture back to its original architecture, in this case Intel-x86, can prove extremely problematic. This is usually the hardest step in the process, depending on how closely the custom architecture mimics the original architecture. For example, it can prove very difficult to translate a custom architecture code snippet that concurrently utilizes 20 general-purpose registers into a code-snippet that carries out the same machine logic in the Intel-x86 architecture which has only 8 general-purpose registers, of which one is reserved for the stack.

CHAPTER 3

Themida

This chapter aims to introduce Themida by Oreans Technologies. The vendor of Themida, Oreans Technologies, offer two additional products, Code Virtualizer and WinLicense. Code Virtualizer is a stand-alone software virtualization product and the core of the rest of the product catalogue. Themida is an extension of Code Virtualizer that additionally features software packing, software obfuscation and more. Similarly, WinLicense is an extension of Themida that additionally features advanced licensing control.

At present, Oreans Technologies with their flagship product, Themida, is one of the leaders in the software protection market challenged only by VMProtect. Software protection products such as Enigma, ASProtect and Obsidium support software virtualization technologies aswell, albeit at an extremely immature level in comparison to that of Themida and VMProtect.

The software virtualization technology supported by Themida has been in constant evolution since the initial release of Themida in December 2004, with its most notable changes listed below.

- Themida [1.8.0.0] (05-Sep-2006)

  ```
  [+] Added new mutable RISC-128 processor (virtual machine)
  [+] Added new mutable CISC processor (virtual machine)
  ```

- Themida [1.8.2.0] (06-Oct-2006)

  ```
  [+] Added new processor (CISC-2) in virtual machine
  ```

- Themida [2.2.5.0] (03-Oct-2013)

  ```
  [+] New Virtual Machine added (TIGER architecture)
  [+] New Virtual Machine added (FISH architecture)
  ```

- Themida [2.2.8.0] (18-Mar-2014)

  ```
  [+] Added PUMA VM (White, Red, Black)
  [+] Added SHARK VM (White, Red, Black)
  ```

- Themida [2.3.5.0] (22-Jun-2015)

  ```
  [+] Added DOLPHIN virtual machine
  [+] Added EAGLE virtual machine
  ```

- Themida [2.3.9.0] (13-Nov-2015)

  ```
  [!] Removed support for old CISC/RISC VMs
  ```

In the above changelog list, adding a new virtual machine actually means adding support for a new type of custom architecture. In terms of Themida, a distinction can be made between the *old* architectures and the *new* architectures as shown in table 3.1 below.

The old architectures	**The new architectures**
CISC, CISC-2, RISC-64, RISC-128	FISH, TIGER, DOLPHIN, PUMA, SHARK, EAGLE

Table 3.1: All virtual machines supported by Themida.

Interestingly, all virtual machines supported by Themida are *mutable* in the sense that parts of the virtual machines are randomly generated so that each instance of a virtual machine is unique. Furthermore, Themida allows the client to choose between different settings for each of the virtual machine architectures. The old architectures supports a range of settings as shown in table 3.2 below.

Setting	**Options**
Multiprocessor (CISC and CISC-2 only)	1 CPU 2 CPUs 4 CPUs 8 CPUs
Opcode Type	Static opcodes Metamorphic - Level 1 Metamorphic - Level 2 Metamorphic - Level 3
Dynamic Opcode	Disabled 20% Dynamic 40% Dynamic 60% Dynamic 80% Dynamic

Table 3.2: The settings supported by the old architectures.

The names assigned to these settings are not very descriptive of their functional purpose, so a minor description is in order.

- The *Multiprocessor* setting allows multiple mutable instances of the same virtual machine architecture to be generated for a single binary. This adds greatly to the overall complexity of the executable binary.
- The *Opcode Type* setting determines the complexity of the virtual machine by deciding the level of obfuscation applied throughout the main areas as well as the handler procedures in the virtual machine.
- The *Dynamic Opcode* setting determines the amount of deadcode instructions (in the custom architecture) to be added to the byte-code buffer right before the original translated instructions.

The new architectures no longer support the same level of fine-grained settings as the old architectures, but instead support a three-level color setting, WHITE, RED and BLACK. In reality, the color settings are symbolic names for roughly the same functionality as the *Opcode Type* setting from the old architectures in that they determine the level of obfuscation applied throughout the main areas as well as the handler procedures in the virtual machine.

- The *WHITE* setting is the lowest level of obfuscation, and feature minor application of instruction expansion and deadcode insertion.
- The *RED* setting is the medium level of obfuscation, and feature moderate application of instruction expansion, constant expansion, deadcode insertion, and control flow obfuscation.
- The *BLACK* setting is the highest level of obfuscation, and feature excessive application of instruction expansion, constant expansion, deadcode insertion, control flow obfuscation and branch obfuscation.

In the new architectures, a further distinction can be made between core architectures and layered architectures. Core architectures are basic stand-alone architectures and includes FISH, TIGER and DOLPHIN. Layered architectures are architectures that consists of nested core architectures and includes PUMA, SHARK and EAGLE. An overview of the layers involved in these architectures can be found in table 3.3 below, where each row reads *"[Name] is actually [Layer 1] virtualized by [Layer 2]"*.

Name	**Layer 1**	**Layer 2**
PUMA	TIGER	FISH
SHARK	FISH	TIGER
EAGLE	FISH	DOLPHIN

Table 3.3: All layered virtual machines supported by Themida.

Despite the amount of information being illustrated in this thesis, the internal workings of Themida are largely undocumented in the public domain. This especially holds true for the new architectures. The only published works about the internal workings of the software virtualization technology by Themida, is found to be "Inside Code Virtualizer" from 2007 [3], which focuses on an early version of the CISC virtual machine architecture. Unfortunately, most section of the paper appears rushed and does not adequately explain the vital parts of the virtual machine architecture. However, the paper did inspire parts of the naming conventions and syntax used throughout this thesis for constructs in the custom architecture instruction sets.

3.1 Defining the target of analysis

The subsequent chapters, 4 and 5, aims to dissect and understand the old CISC architectures and the new FISH architecture. The chosen primary target for analysis throughout these chapters is a simple application written in C as shown in listing 3.1 below.

```
#include <stdio.h>
#include "ThemidaSDK.h"

__declspec(naked) int test()
{
  VM_START
  __asm
  {
    push ecx
    xor ecx, ecx
    mov ecx, 0x05
    add ecx, 0xbe
    sub ecx, 0x34
    shl ecx, 0x08
    mov eax, ecx
    pop ecx
    ret
  }
  VM_END
}

int main()
{
  int n = test();
  printf("%x", n);
  return 0;
}
```

Listing 3.1: The program to be analysed.

The preprocessor macros, `VM_START` and `VM_END` are obtained from the Themida SDK, and marks the area to be protected by software virtualization. These macros work for all 32-bit versions of Themida. The `test` function was written in native x86 assembly for precise control over the instructions that must be recovered later in this thesis. Once the application has been compiled, it is a good idea to open the resulting executable binary in a disassembler such that the general layout of the binary can be assessed. For example, the `main` function of the executable binary is found to be as shown in listing 3.2 below.

```
.text:00401040 ; int main(int argc, const char **argv, const char **envp)
.text:00401040 _main           proc near
.text:00401040
.text:00401040 argc            = dword ptr  0x04
.text:00401040 argv            = dword ptr  0x08
.text:00401040 envp            = dword ptr  0x0C
.text:00401040
.text:00401040                 call    sub_401000
.text:00401045                 push    eax
.text:00401046                 push    offset Format   ; "%x"
.text:0040104B                 call    ds:printf
.text:00401051                 add     esp, 0x08
.text:00401054                 xor     eax, eax
.text:00401056                 retn
.text:00401056
.text:00401056 _main           endp
```

Listing 3.2: The `main` function in the program after being compiled.

Here, the `main` function invokes another function located at address `401000`, which we can assume to be the `test` function, and uses the value returned from this function in a subsequent invocation of the `printf` function. The function that is assumed to be the `test` function is found to be as shown in listing 3.3 below.

```
.text:00401000 sub_401000      proc near
.text:00401000
.text:00401000 ; ---------------------------------------------------------------
.text:00401000                 db EB, 10, 57, 4C, 20, 20, 0C, 00, 00
.text:00401009                 db 00, 00, 00, 00, 00, 57, 4C, 20, 20
.text:00401012 ; ---------------------------------------------------------------
.text:00401012
.text:00401012                 push    ecx
.text:00401013                 xor     ecx, ecx
.text:00401015                 mov     ecx, 0x05
.text:0040101A                 add     ecx, 0xBE
.text:00401020                 sub     ecx, 0x34
.text:00401023                 shl     ecx, 0x08
.text:00401026                 mov     eax, ecx
.text:00401028                 pop     ecx
.text:00401029                 retn
.text:00401029
.text:00401029 ; ---------------------------------------------------------------
.text:0040102A                 db EB, 10, 57, 4C, 20, 20, 0D, 00, 00
.text:00401033                 db 00, 00, 00, 00, 00, 57, 4C, 20, 20
.text:00401033 ; ---------------------------------------------------------------
.text:00401033
.text:00401033 sub_401000      endp
```

Listing 3.3: The `test` function in the program after being compiled.

Notice how the `VM_START` and `VM_END` preprocessor macros have expanded into these irregular binary chunks that can be easily identified by Themida. Interestingly, they are both initiated by `EB 10` which is the binary representation for a jump instruction that skips over the rest of the bytes in the binary chunk.

Essentially, this means that the binary chunks allows the execution flow to run through them without causing disruption in the application. Obviously, this only occurs when a tagged binary is executed prior to being protected by Themida. This is a very important feature, as had the function not been declared `naked` (i.e. compiled without a prologue or epilogue), the `VM_START` macro would have appeared after the function prologue and the `VM_END` macro would have appeared before the function epilogue. Effectively, this means that the execution flow of the application would have run through both of them everytime the `test` function was invoked.

The binary is ready to have software virtualization applied to it by Themida. Due to the preprocessor macros being supported by all 32-bit versions of Themida, the binary need not be recompiled between the two subsequent chapters and can furthermore be protected over and over by the same protection template to determine variabilities in the analysed virtual machine architectures.

3.2 How to setup an unpacking environment

When Themida applies software virtualization to an executable binary, a range of other protections are also applied, including the software packing techniques previously detailed in section 2.1. Before the software virtualization technology can be analysed, the protected executable binary must therefore undergo a procedure known as *unpacking*, in which it is stripped of software packing protection.

When unpacking protected software, the goal is to reach the *Original Entry Point* (OEP), which is the address that was originally assigned as the entry point of the executable binary prior to having software packing applied to it. Since this implies performing a partial execution of the executable binary and since packed software is often malicious, this procedure should always be carried out in a guest virtual machine rather than on the host system. Throughout the remaining sections of this chapter, it shall be assumed that all actions are carried out in a VMWare guest system running a 32-bit version of Windows XP with Service Pack 3.

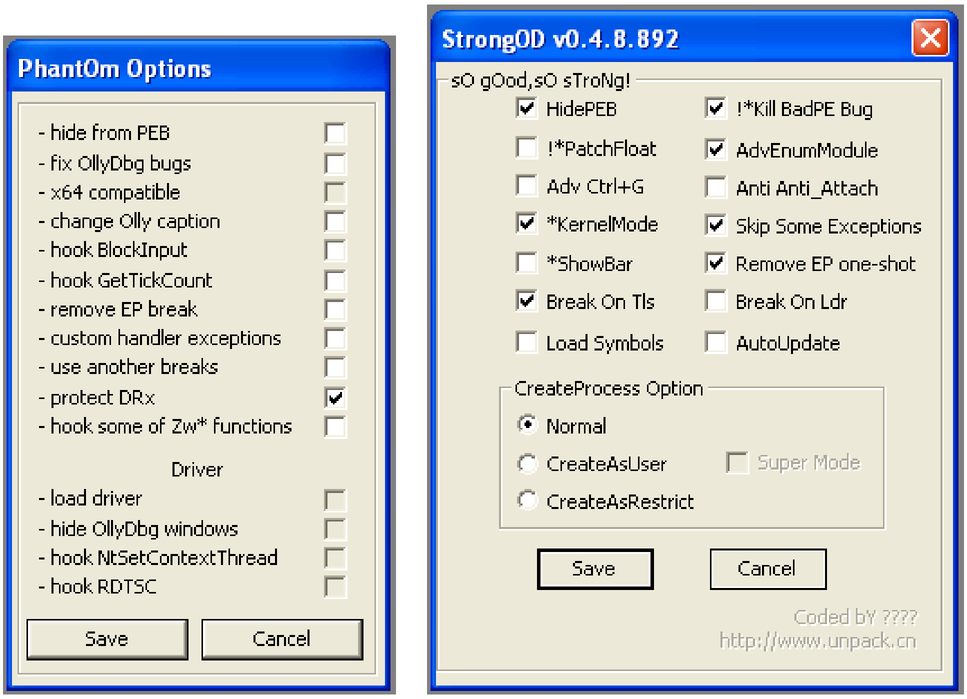
The most accomplished publicly available unpacking scripts for Themida are written for OllyDbg v1.10[1], which is therefore the chosen debugger throughout the upcoming sections. Unfortunately, Themida inserts a number of anti-debugging templates into protected executable binaries, so in order to run unpacking scripts under the OllyDbg debugger, all anti-debugging routines must be bypassed using a selection of anti-anti-debugging plugins. For a guest machine running 32-bit Windows XP with Service Pack 3, it is advised to use both the *PhantOm*[2] and the *StrongOD*[3] plugins.

These plugins are installed by placing their dynamic link library (.dll) files in the plugin folder for OllyDbg, which can be configured in the *Directories* tab of the appearance configuration window (Options Menu → Appearance) in OllyDbg. By default, this is configured as the root directory of the OllyDbg installation. Once installed, the following options must be enabled in the configuration (Plugins Menu → *plugin-name* → Options).

- PhantOm v1.85
  - Protect DRx
- StrongOD v0.4.8.892
  - Hide PEB
  - Kernel Mode
  - Break on TLS
  - Kill BadPE Bug
  - Skip Some Exceptions
  - Remove EP One-Shot

The correctly configured UIs are depicted in table 3.4 below.

[1]https://www.ollydbg.de/odbg110.zip
[2]https://forum.tuts4you.com/topic/13402-phantom
[3]https://forum.tuts4you.com/topic/19480-strongod



PhantOm Options

- hide from PEB ☐
- fix OllyDbg bugs ☐
- x64 compatible ☐
- change Olly caption ☐
- hook BlockInput ☐
- hook GetTickCount ☐
- remove EP break ☐
- custom handler exceptions ☐
- use another breaks ☐
- protect DRx ✓
- hook some of Zw* functions ☐

Driver

- load driver ☐
- hide OllyDbg windows ☐
- hook NtSetContextThread ☐
- hook RDTSC ☐

StrongOD v0.4.8.892

sO gOod,sO sTroNg!

✓ HidePEB	✓ !*Kill BadPE Bug
☐ !*PatchFloat	✓ AdvEnumModule
☐ Adv Ctrl+G	☐ Anti Anti_Attach
✓ *KernelMode	✓ Skip Some Exceptions
☐ *ShowBar	✓ Remove EP one-shot
✓ Break On Tls	☐ Break On Ldr
☐ Load Symbols	☐ AutoUpdate

CreateProcess Option

● Normal
○ CreateAsUser ☐ Super Mode
○ CreateAsRestrict

Coded bY ????
http://www.unpack.cn

Table 3.4: The plugin configurations for PhantOm and StrongOD.

With these configured, almost every anti-debugging technique inserted by Themida is bypassed. However, the StrongOD options dialog does not support modification of all configurable settings, including the *DriverName* setting which is configured as *"fengyue0"* by default. The OllyDbg configuration file (ollydbg.ini) located in the root directory of the OllyDbg installation, depicts the active setting in the StrongOD plugin group as shown in listing 3.4 below.

```
[Plugin StrongOD]
...
DriverName=fengyue0
...
```

Listing 3.4: The *DriverName* default configuration for StrongOD.

This value must undergo manual modification to represent a non-default value. For every unpacking procedures carried out in the subsequent chapters of this thesis, the setting was configured as *"something_unique"*. With the plugins configured to bypass all anti-debugging techniques currently introduced by Themida, unpacking a protected executable binary is finally possible. Lastly, in order to run the scripts presented in the subsequent sections of this chapter, the *ODbgScript*[4] plugin must also be installed.

[4]https://sourceforge.net/projects/odbgscript/

3.3 How to unpack Themida 2.1.8.0

A serial contributor of the *tuts4you* reverse-engineering community, named *LCF-AT*, has published an unpacking script[5] for Themida 2.1.8.0 and earlier versions. This script can be downloaded as a text file (.txt) and executed using the *Run Script* functionality of the ODbgScript plugin (Plugins Menu → ODbgScript → Run Script).

Throughout the execution of the script, the user is prompted to make a series of choices regarding actions to be carried out by the unpacking script. The majority of prompts suggests a default option to be attempted first and an alternative option to be attempted if the default option causes a malfunction. However, there are a few prompts that does not suggest a default choice and which must be answered as depicted in table 3.5 below.

Prompt	**Answer**
Find VM Ware pointer? » quosego «	Yes
Do you want to use the magic jumps as eax is an API place?	Yes
Fixing IAT with the » Fast IAT Patch Method way by LCF-AT «	Yes

Table 3.5: Answers to prompts that does not include suggestions.

A final message will signal the termination of the unpacking script, at which point the running process has reached the original entry point of the unpacked executable binary which can therefore be dumped from memory using the all-in-one tool named *Scylla*[6].

However, when dumped from memory, the unpacked executable binary suffers a broken import name table as a result of the anti-dump technique known as *IAT hijacking* which was previously described in section 2.1. In order to correct for this, the import name table must be restored, which, given the correct parameters, is also possible using Scylla.

In order to restore the import name table, the neccessary IAT parameters must be identified. Towards the end of the unpacking script, these are printed to the OllyDbg log window (View Menu → Log) as shown in listing 3.5 below.

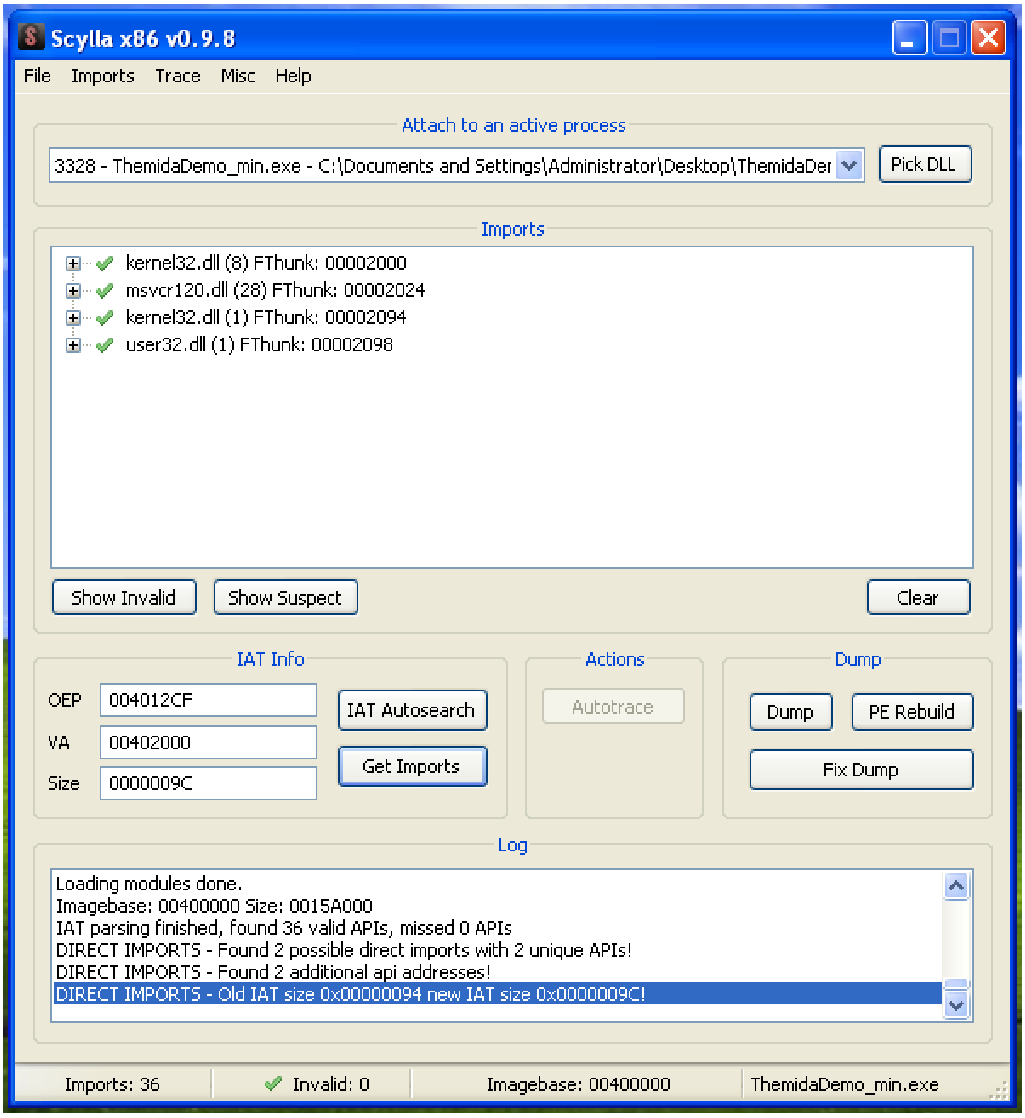
```
IAT RESULTS IN VA
------------------------------
oep: 004012CF

IAT_START: 00402000
IAT_END: 00402094
```

Listing 3.5: The import address table parameters output by the unpacking script.

By inserting these parameters and clicking ”Get Imports” after attaching to the running process, Scylla is able to reconstruct the list of imports as shown in figure 3.1 below.

[5]https://forum.tuts4you.com/topic/25554-themida-winlicense-1x-2x-multi-pro-edition-12
[6]https://github.com/NtQuery/Scylla



```
3328 - ThemidaDemo_min.exe - C:\Documents and Settings\Administrator\Desktop\ThemidaDer
```

```
kernel32.dll (8) FThunk: 00002000
msvcr120.dll (28) FThunk: 00002024
kernel32.dll (1) FThunk: 00002094
user32.dll (1) FThunk: 00002098
```

```
OEP   004012CF
VA    00402000
Size  0000009C
```

```
Loading modules done.
Imagebase: 00400000 Size: 0015A000
IAT parsing finished, found 36 valid APIs, missed 0 APIs
DIRECT IMPORTS - Found 2 possible direct imports with 2 unique APIs!
DIRECT IMPORTS - Found 2 additional api addresses!
DIRECT IMPORTS - Old IAT size 0x00000094 new IAT size 0x0000009C!
```

```
Imports: 36    Invalid: 0    Imagebase: 00400000    ThemidaDemo_min.exe
```

Figure 3.1: The Scylla window after import retrieval.

The *"Dump"* button dumps the unpacked executable binary from memory to disk but with a broken import name table. The *"Fix Dump"* button uses the reconstructed list of imports shown in the UI to restore the import name table of a previously generated dump.

Following all of the above steps restores an executable binary to its original state prior to have software packing applied to it by Themida. However, the software virtualization protection is still intact in the executable binary and therefore possible to analyse.

3.4 How to unpack Themida 2.2.5.0

A serial contributor of the *tuts4you* reverse-engineering community, named *LCF-AT*, has published an unpacking script[7] for Themida 2.2.5.0 and later versions. This script can be downloaded as a text file (.txt) and executed using the *Run Script* functionality of the ODbgScript plugin (Plugins Menu → ODbgScript → Run Script).

Throughout the execution of the script, the user is prompted to make a series of choices regarding actions to be carried out by the unpacking script. The prompts suggests a default option to be attempted first and an alternative option to be attempted if the default option causes a malfunction. A final message will signal the termination of the unpacking script, at which point the running process has been dumped to disk with its import name table fully restored. The dump is placed in the same location as the packed executable binary although with a *"_DP"* suffix.

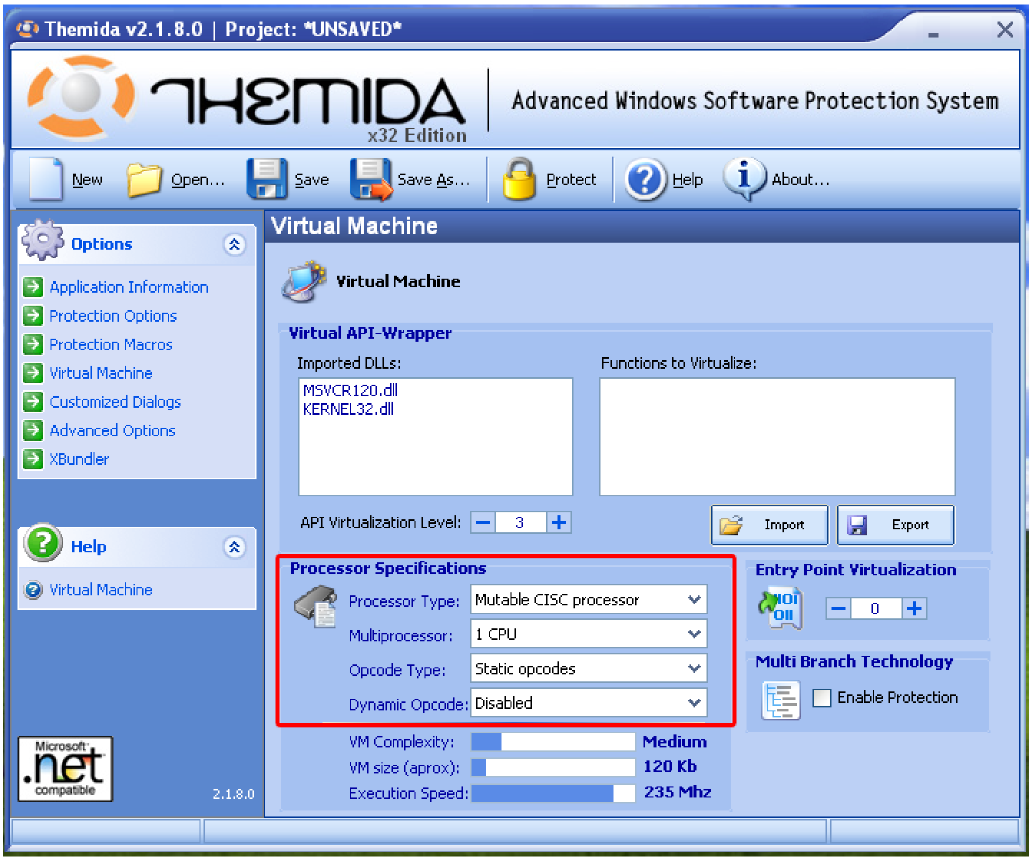
Following all of the above steps restores an executable binary to its original state prior to have software packing applied to it by Themida. However, the software virtualization protection is still intact in the executable binary and therefore possible to analyse.

[7]https://forum.tuts4you.com/topic/34085-themida-winlicense-ultra-unpacker-14

CHAPTER 4

The CISC architecture

This chapter will focus on Themida version 2.1.8.0 and the supported CISC and CISC-2 virtual machine architectures. The herein analysed binary is protected by Themida with the lowest levels of virtual machine protection settings as shown in figure 4.1 below.



Imported DLLs:
MSVCR120.dll
KERNEL32.dll

Functions to Virtualize:

API Virtualization Level: 3

Processor Specifications
Processor Type: Mutable CISC processor
Multiprocessor: 1 CPU
Opcode Type: Static opcodes
Dynamic Opcode: Disabled

Entry Point Virtualization: 0
Multi Branch Technology: Enable Protection ☐

VM Complexity: Medium
VM size (aprox): 120 Kb
Execution Speed: 235 Mhz

Figure 4.1: The virtual machine options chosen for protecting the program.

Before the protected binary and its virtualization protection can be analysed, it must first be unpacked. An unpacking environment setup guide and an unpacking walkthrough for Themida 2.1.8.0 can be found in sections 3.2 and 3.3 respectively.

4.1 Analysis of the virtual machine

The binary has now been protected by Themida 2.1.8.0, and a reanalysis of the protected `test` function depicted in listing 4.1 below should be carried out.

```
.text:00401000 sub_401000      proc near
.text:00401000
.text:00401000                 jmp     sub_4B9157
.text:00401000
.text:00401000 ; ---------------------------------------------------------------
.text:00401005                 db 18, C1, 17, 65, E8, 7E, B4, 8B, 66, B1, B5
.text:00401010                 db B2, 1B, 30, E8, E0, 49, 13, 8D, 00, C8, FF
.text:0040101B                 db 62, 21, 3F, 70, A0, E5, 0B, E0, 71, B0, D0
.text:00401026                 db B2, B0, BF, E8, EB, 10, 57, 4C, 20, 89, FF
.text:00401031                 db 89, C0, 89, C9, 89, D2, 89, DB, 89, C0, 90
.text:00401031 ; ---------------------------------------------------------------
.text:00401031
.text:00401031 sub_401000      endp
```

Listing 4.1: The `test` function in the program after being protected.

Here, it can be seen that the machine code of the `test` function has been replaced with a jump to the virtual machine segment, specifically to the `VM Trampoline` area referenced in section 2.3, and that the rest of the function has been overwritten by a random sequence of bytes. The contents of the `VM Trampoline` area is shown in listing 4.2 below.

```
.themida:004B9157 sub_4B9157      proc near
.themida:004B9157
.themida:004B9157                 push    0872300C ; opcode key
.themida:004B915C                 jmp     loc_40DA3E
.themida:004B915C
.themida:004B915C sub_4B9157      endp
```

Listing 4.2: The `VM Trampoline` area for the `test` function.

The `VM Trampoline` area stores the opcode key for the `test` function on the stack and then jumps to the `VM Entrypoint` area of the virtual machine. However, the `VM Entrypoint` area is obfuscated as briefly shown in listing 4.3 below.

```
.themida:0040DA3E loc_40DA3E:
.themida:0040DA3E
.themida:0040DA3E                 pushf
.themida:0040DA3F                 push    00007616
.themida:0040DA44                 mov     [esp], eax
.themida:0040DA47                 push    000040E5
.themida:0040DA4C                 mov     [esp], ecx
.themida:0040DA4F                 push    0000065C
.themida:0040DA54                 mov     [esp], edx
.themida:0040DA57                 push    000035BE
.themida:0040DA5C                 mov     [esp], ebx
.themida:0040DA69                 ...
```

Listing 4.3: The obfuscated entry point for the virtual machine.

The obfuscation produced by Themida follows a monotone and predictable pattern, allowing for rather trivial elimination using the counter-techniques described in section 2.2. For example, the above listing contains a `pushfd` instruction followed by the obfuscated first half of a `pushad` instruction. Upon reduction, the `VM Entrypoint` area should reveal its core instructions as shown in listing 4.4 below.

```
0040da3e  pushfd
0040da3f  pushad
0040db14  cld

; Initialize context (edi)
0040db7c  call $+5
0040db81  pop edi
0040dbeb  sub edi,0x08677a39
0040dc54  and edi,0xfffff000
0040dc9a  add edi,0x14
0040dcf5  mov eax,edi                    ; delta
0040dd44  add edi,0x086776ee             ; context

; Initialize handlers
0040ddbd  cmp eax,[edi+0x64]
0040ddc3  jz 0x0040dfa8
0040ddce  mov [edi+0x64],eax
0040de18  mov ecx,0xaa                   ; handler count = 0xaa (170)
0040dea2  jmp 0x0040dfa0
0040dea7  add [edi+ecx*4+0x90],eax       ; handler table
0040df2d  add ecx,0xffffffff
0040dfa0  or ecx,ecx
0040dfa2  jnz 0x0040dea7

; Initialize opcode buffer (esi) and opcode key (ebx)
0040dfa8  mov esi,[esp+0x24]
0040dff0  mov ebx,esi                    ; opcode key
0040e02b  add esi,eax                    ; opcode buffer

; Spinlock (only one instance permitted at a time)
0040e0a4  mov ecx,0x1
0040e10d  xor eax,eax
0040e10f  lock cmpxchg [edi+0x88],ecx
0040e117  jnz 0x0040e10d
0040e11d  ...                            ; VM Dispatcher
```

Listing 4.4: The deobfuscated entry point for the virtual machine.

The `VM Entrypoint` area consists of five minor sub-areas, each of which hold the following responsibilities.

- The first sub-area pushes the `EFLAGS` register and all general-purpose registers to the stack, so that the virtual machine can pop them into its own proprietary registers as will be showcased later in section 4.2.

- The second sub-area uses the return address from a relative call to calculate the address of the virtual machine context structure as well as a delta value that is used for initialization of certain data in the virtual machine.

- The third sub-area uses the delta value from the second sub-area to perform a one-time initialization of the handler procedure table.
- The fourth sub-area assigns the value from the `VM Trampoline` area to the opcode key and initializes the byte-code buffer by adding the delta value from the second sub-area to the opcode key. The opcode key is used throughout the virtual machine for decryption of opcodes from the byte-code buffer.
- The fifth sub-area performs a mutual exclusion spinlock procedure, in order to ensure that only a single byte-code buffer is being processed at a time. This is important because the virtual machine uses a shared global context structure across executions.

Once all sub-areas have been executed, the virtual machine is ready for processing and the execution flow is transferred to the `VM Dispatcher` area illustrated in listing 4.5 below.

```
0040e11d  lodsb
0040fdde  sub al,bl
00419fd5  xor al,0x4d
00417e3c  sub al,0x40
00414848  sub bl,al
004196f9  movzx eax,al
004196fc  jmp dword ptr [edi+eax*4]
```

Listing 4.5: The main dispatch handler procedure.

The `VM Dispatcher` is responsible for deducing which handler procedure to invoke next based on data from the byte-code buffer. It is important to note that since the byte-code buffer is located in `ESI`, the `lods` instruction can be used to read data from the byte-code buffer into `EAX` without having to manage buffer increments manually.

Furthermore, it can be seen that the data stored in the byte-code buffer is decrypted using a series of arithmetic operations involving the opcode key and two constant values. The opcode key is also updated after each decryption operation, indicating that opcodes stored in the byte-code buffer has been encrypted using different opcode keys.

Once the data from the byte-code buffer has been decrypted, execution flow is transferred to a handler procedure using the decrypted data from the byte-code buffer as an offset into the context structure which contains the handler procedure table as shown in listing 4.4 above. For example, the execution flow could transfer into the `load imm32` handler procedure shown in listing 4.6 below.

```
lodsd
xor eax,ebx
add eax,0x674cca9f
xor eax,0x070ef2f5
sub ebx,eax
push eax
jmp 0040e11d ; vm_dispatch
```

Listing 4.6: The virtual handler procedure for `load imm32`.

Interestingly, the sequence of registers in the decryption operations are identical to those previously seen in the `VM Dispatcher` area depicted in listing 4.5 above. However, the operation mnemonics as well as the constant values appear to be randomized between different handler procedures. This observation will prove useful for automated opcode recovery as shown in section 4.3.

After thorough analysis of all handler procedures in the virtual machine, it becomes evident that the virtual machine architecture follows a stack-based model, and that the majority of supported operations can be separated into three distinct groups. These are *load* operations, *store* operations and *basic* operations. Load operations are equivalent to the x86 `push` instruction and are responsible for loading source operands onto the stack. Store operations are equivalent to the x86 `pop` instruction and are responsible for storing data from the stack into destination operands. Basic operations are responsible for consuming loaded source operands to produce results for one or more destination operand(s).

The comprehensive list of supported load operations can be found in table 4.1 below.

Operation	**8-bit**	**16-bit**	**32-bit**	**Description**
load imm	x	x	x	Push integer to the stack
load [imm]	x	x	x	Push dereferenced offset to the stack
load reg			x	Push vm register to the stack
load [reg]	x	x		Push dereferenced vm register to the stack
load offset reg			x	Push address of vm register to the stack
load addr			x	Push EDX to the stack
load [addr]	x	x	x	Push [EDX] to the stack
load fs:[addr]	x	x	x	Push fs:[EDX] to the stack
load stack		x	x	Push the stack pointer to the stack
load param			x	Push `vm param` to the stack
load delta			x	Push `vm delta` to the stack
load delta align			x	Push data aligned by `vm delta` to the stack

Table 4.1: Load operations supported by the virtual machine.

The comprehensive list of supported store operations can be found in table 4.2 below.

Operation	**8-bit**	**16-bit**	**32-bit**	**Description**
store [imm]	x	x	x	Pop the top stack element into dereferenced offset
store reg	x	x	x	Pop the top stack element into register
store reg high	x			Pop the top stack element into register high byte
store addr			x	Pop the top stack element into EDX
store [addr]	x	x	x	Pop the top stack element into [EDX]
store fs:[addr]	x	x	x	Pop the top stack element into fs:[EDX]
store stack		x	x	Pop the top stack element into the stack pointer
store param			x	Pop the top stack element into `vm param`
store eflags			x	Pop the top stack element into `vm eflags`

Table 4.2: Store operations supported by the virtual machine.

The comprehensive list of supported basic operations can be found in table 4.3 below.

Operation	**I**	**O**	**Flags**	**8-bit**	**16-bit**	**32-bit**	**Description**
inc	1	1	x	x	x	x	Increment an integer
dec	1	1	x	x	x	x	Decrement an integer
neg	1	1	x	x	x	x	Negate an integer
not	1	1		x	x	x	Bitwise NOT an integer
add	2	1	x	x	x	x	Add two integers
addx	2	1				x	Add two integers (no flags)
adc	2	1	x	x	x	x	Add two integers with carry
sub	2	1	x	x	x	x	Subtract two integers
subx	2	1				x	Subtract two integers (no flags)
sbb	2	1	x	x	x	x	Subtract two integers with borrow
mul	2	2	x	x	x	x	Mulitply two unsigned integers
imul	2	2	x	x	x	x	Mulitply two signed integers
imulc	2	1	x	x	x	x	Mulitply two signed integers
div	2	2	x	x	x	x	Divide two unsigned integers
idiv	2	2	x	x	x	x	Divide two signed integers
and	2	1	x	x	x	x	Bitwise AND two integers
or	2	1	x	x	x	x	Bitwise OR two integers
xor	2	1	x	x	x	x	Bitwise XOR two integers
xorx	2	1				x	Bitwise XOR two integers (no flags)
shr	2	1	x	x	x	x	Bitwise shift an unsigned integer right
sar	2	1	x	x	x	x	Bitwise shift a signed integer right
shl/sal	2	1	x	x	x	x	Bitwise shift an integer left
shlx/salx	2	1				x	Bitwise shift an integer left (no flags)
ror	2	1	x	x	x	x	Bitwise rotate an integer right
rcr	2	1	x	x	x	x	Bitwise rotate an integer right through carry
rol	2	1	x	x	x	x	Bitwise rotate an integer left
rcl	2	1	x	x	x	x	Bitwise rotate an integer left through carry
cmp	2	0	x	x	x	x	Compare two integers (using subtraction)
test	2	0	x	x	x	x	Compare two integers (using bitwise AND)
bt	2	1	x		x	x	Bit-test string
btc	2	1	x		x		Bit-test and complement string
btr	2	1	x		x	x	Bit-test and reset string
bts	2	1	x		x	x	Bit-test and set string
bswap	1	1				x	Invert the byte-order of an integer
movsxb	2	1			x	x	Move byte with sign-extension
movsxw	2	1				x	Move word with sign-extension
movzxb	2	1			x	x	Move byte with zero-extension
movzxw	2	1				x	Move word with zero-extension
readptr	1	1				x	Read the value of a 32-bit pointer
writeptr	2	0				x	Assign a value to a 32-bit pointer

Table 4.3: Basic operations supported by the virtual machine.

For clarification, the `I` and O columns depict the amount of consumed parameters (inputs) and generated results (outputs) respectively. The `Flags` column depicts if the `EFLAGS` register is loaded onto the stack subsequent to the core operation being carried out.

Apart from the stack-based operations shown above, the virtual machine architecture also support a variety of miscellaneous operations pertaining to manipulation of the `EFLAGS` register, execution flow control, and edge-case necessities as shown in table 4.4 below.

Operation	**Description**
nop	No operation
clc	Clear carry flag
cmc	Complement carry flag
cld	Clear direction flag
stc	Set carry flag
std	Set direction flag
sti	Set interrupt flag
mov addr, imm32	Move an integer into EDX
mov addr, stack32	Move the stack pointer into EDX
add addr, imm32	Add an integer to EDX
add addr, reg32	Add a register to EDX
add addr, reloc	Add `vm reloc` to EDX
add [stack32], reloc	Add `vm reloc` to the top stack element
sub addr, imm32	Subtract an integer from EDX
xor addr, imm32	Bitwise XOR EDX with an integer
xchg [stack32], addr	Exchange the top stack element with EDX
set param	Move the second topmost stack element into `vm param`
set counter offset	Reset the ecx context offset for use in ecx-based `jcc` evaluation
set stack offset	Reset the esp offset for stack re-alignment in subsequent `ret` instruction
reset key	Reset opcode key
jcc evaluate	Evaluate condition for subsequent `jcc imm32` instruction
jcc imm32	Increment opcode pointer and reset opcode key if condition evaluated to true
jmp imm32	Increment opcode pointer and reset opcode key
ret	Reset the stack, registers, eflags and spinlock and return from the virtual machine

Table 4.4: Remaining operations supported by the virtual machine.

Using sequences of supported instructions from all of the above tables combined, it is possible to emulate almost any x86 usermode instruction in the virtual machine architecture.

However, the above tables depict only high-level descriptions of the supported operations and does not adequately outline their implementation details. For a exhaustive list of handler procedures and their core instructions respectively, please refer to appendix A.

Now that the design of the virtual machine has been mapped and analysed, it is possible to iterate the byte-code buffer in order to determine the sequence of invoked handler procedures and thereby reconstruct the custom architecture code flow.

4.2 Recovering virtual opcodes manually

By placing a software breakpoint in the `VM Dispatcher` area and stepping into every invoked handler procedure to recover the decrypted byte-code data, it is possible to reconstruct the custom architecture code flow as shown in listing 4.7 below.

```
load offset reg32(3)
store addr
store [addr]
load offset reg32(4)
store addr
store [addr]
load offset reg32(1)
store addr
store [addr]
load offset reg32(5)
store addr
store [addr]
set counter offset
load offset reg32(5)
store addr
store [addr]
load offset reg32(6)
store addr
store [addr]
load offset reg32(0)
store addr
store [addr]
load offset reg32(2)
store addr
store [addr]
load offset reg32(7)
store addr
store [addr]
mov addr, stack32
load addr
load dword(00000004)
add dword (no flags)
store stack32
load offset reg32(0)
store addr
load [addr]
load offset reg32(0)
store addr
load [addr]
load offset reg32(0)
store addr
load [addr]
xor dword
load addr
load addr
load dword(00000004)
add dword (no flags)
store addr
store addr
store reg32(7)
load offset reg32(0)
store addr
store [addr]
load dword(00000005)
load offset reg32(0)
store addr
store [addr]
load offset reg32(0)
store addr
load addr
load addr
load word(765A)
load word(90DB)
store addr
load dword(00000004)
add dword (no flags)
store addr
store addr
load [addr]
load dword(000000BE)
add dword
store reg32(7)
load offset reg32(0)
store addr
store [addr]
load addr
load addr
load dword(00000004)
add dword (no flags)
store addr
store addr
load offset reg32(0)
load offset reg32(6)
load offset reg32(0)
store addr
store addr
load offset reg32(5)
store addr
store addr
load [addr]
load dword(00000034)
sub dword
load addr
load word(8F74)
load word(E76B)
store addr
load offset reg32(1)
store addr
load [addr]
store addr
store addr
store reg32(7)
load addr
load word(09D8)
load word(D427)
store addr
store addr
load offset reg32(0)
store addr
store [addr]
load offset reg32(0)
store addr
load addr
load offset reg32(3)
load offset reg32(5)
store addr
store addr
load [addr]
store addr
store addr
load [addr]
load byte(08)
shl dword
load addr
load addr
load dword(00000004)
load addr
load dword(00000004)
add dword (no flags)
store addr
add dword (no flags)
store addr
store addr
store reg32(7)
load addr
load offset reg32(1)
store addr
store addr
load offset reg32(0)
store addr
store [addr]
load offset reg32(0)
load offset reg32(4)
store addr
load offset reg32(2)
store addr
load [addr]
store addr
store addr
load [addr]
load offset reg32(2)
load word(07BA)
load word(6E5D)
load offset reg32(6)
store addr
load [addr]
store addr
store addr
store addr
store [addr]
load offset reg32(0)
store addr
store [addr]
jmp imm32(0000000E)
load offset reg32(7)
store addr
load [addr]
load offset reg32(2)
store addr
load [addr]
load offset reg32(0)
store addr
load [addr]
load offset reg32(6)
store addr
load [addr]
load offset reg32(5)
store addr
load [addr]
load offset reg32(5)
store addr
load [addr]
load offset reg32(1)
store addr
load [addr]
load offset reg32(4)
store addr
load [addr]
load offset reg32(3)
store addr
load [addr]
ret
```

Listing 4.7: The raw extracted virtual opcodes for the `test` function.

Please note that the above process is extremely time consuming and very prone to errors.

The recovered custom architecture code flow contains minor obfuscation patterns that must be dealt with in order to reveal the core instructions of the code snippet. First of all, it is possible to remove instructions whose purpose is entirely bound to the internals of the virtual machine, such as the `set counter offset` instruction. Additionally, it can be noticed that there are a lot of deadcode instructions in the snippet, such as overlapping `store` instructions that repeatedly overwrites the same destination operand. Removing all of these instructions yields a cleaner result as shown in listing 4.8 below.

```
load offset reg32(3)
store addr
store [addr]
load offset reg32(4)
store addr
store [addr]
load offset reg32(1)
store addr
store [addr]
load offset reg32(5)
store addr
store [addr]
load offset reg32(5)
store addr
store [addr]
load offset reg32(6)
store addr
store [addr]
load offset reg32(0)
store addr
store [addr]
load offset reg32(2)
store addr
store [addr]
load offset reg32(7)
store addr
store [addr]
mov addr, stack32
load addr
load dword(00000004)
add dword (no flags)
store stack32
load offset reg32(0)
store addr
load [addr]
load offset reg32(0)
store addr
load [addr]
load offset reg32(0)
store addr
load [addr]
xor dword
store reg32(7)
load offset reg32(0)
store addr
store [addr]
load dword(00000005)
load offset reg32(0)
store addr
store [addr]
load offset reg32(0)
store addr
load [addr]
load dword(000000BE)
add dword
store reg32(7)
load offset reg32(0)
store addr
store [addr]
load offset reg32(0)
store addr
load [addr]
load dword(00000034)
sub dword
store reg32(7)
load offset reg32(0)
store addr
store [addr]
load offset reg32(0)
store addr
load [addr]
load byte(08)
shl dword
store reg32(7)
load offset reg32(0)
store addr
store [addr]
load offset reg32(0)
store addr
load [addr]
load offset reg32(2)
store addr
store [addr]
load offset reg32(0)
store addr
store [addr]
load offset reg32(7)
store addr
load [addr]
load offset reg32(2)
store addr
load [addr]
load offset reg32(0)
store addr
load [addr]
load offset reg32(6)
store addr
load [addr]
load offset reg32(5)
store addr
load [addr]
load offset reg32(5)
store addr
load [addr]
load offset reg32(1)
store addr
load [addr]
load offset reg32(4)
store addr
load [addr]
load offset reg32(3)
store addr
load [addr]
ret
```

Listing 4.8: The virtual opcodes after removal of redundant opcodes.

Despite having removed deadcode instructions from the custom architecture code flow, there are still instances of instruction expansion obfuscation patterns present in the code. In particular, there are two distinct expansion patterns that can be observed excessively, as detailed in table 4.5 below.

`load reg32(x)`	`store reg32(x)`
`load offset reg32(x)`	`load offset reg32(x)`
`store addr`	`store addr`
`load [addr]`	`store [addr]`

Table 4.5: Reoccurring patterns in the virtual machine opcodes.

By loading the effective address of an internal virtual machine context register, storing it in the temporary `addr` container and then loading the value pointed to by the `addr` container, the virtual machine is loading the value stored in the internal context register albeit in a convoluted manner. The same logic applies to the equivalent convoluted `store` instruction. By reduction of these redundant patterns, it is possible to clean the code even further as shown in listing 4.9 below.

```
store reg32(3)
store reg32(4)
store reg32(1)
store reg32(5)
store reg32(5)
store reg32(6)
store reg32(0)
store reg32(2)
store reg32(7)
mov addr, stack32
load addr
load dword(00000004)
add dword (no flags)
store stack32
load reg32(0)
load reg32(0)
load reg32(0)
xor dword
store reg32(7)
store reg32(0)
load dword(00000005)
store reg32(0)
load reg32(0)
load dword(000000BE)
add dword
store reg32(7)
store reg32(0)
load reg32(0)
load dword(00000034)
sub dword
store reg32(7)
store reg32(0)
load reg32(0)
load byte(08)
shl dword
store reg32(7)
store reg32(0)
load reg32(0)
store reg32(2)
store reg32(0)
load reg32(7)
load reg32(2)
load reg32(0)
load reg32(6)
load reg32(5)
load reg32(5)
load reg32(1)
load reg32(4)
load reg32(3)
ret
```

Listing 4.9: The virtual opcodes after opcode optimization.

Once the obfuscation has been removed, the stack-based architecture operation constructs can be translated to equivalent x86 architecture operation constructs. For example, an integer addition operation in the custom stack-based architecture is performed by loading two source operands onto the stack, performing a stack-based addition, and storing the result of the operation from the stack into a destination operand. However, in the x86 architecture, an addition operation is performed using a single instruction, [`add operand1,operand2`], that leverages the first operand as both source and destination.

Similarly, a move operation in the custom stack-based architecture is performed by loading a source operand onto the stack and storing it from the stack into a destination operand. However, in the x86 architecture, a move instruction is performed using a single instruction [`mov operand1,operand`], that stores the second operand in the first operand.

Considering these transformations, it is possible to translate the stack-based architecture constructs to x86 architecture constructs as shown in listing 4.10 below.

```
store reg32(3)
store reg32(4)
store reg32(1)
store reg32(5)
store reg32(5)
store reg32(6)
store reg32(0)
store reg32(2)
store reg32(7)
add stack32,00000004
load reg32(0)
xor reg32(0),reg32(0)
mov reg32(0),00000005
add reg32(0),000000BE
sub reg32(0),00000034
shl reg32(0),08
mov reg32(2),reg32(0)
store reg32(0)
load reg32(7)
load reg32(2)
load reg32(0)
load reg32(6)
load reg32(5)
load reg32(5)
load reg32(1)
load reg32(4)
load reg32(3)
ret
```

Listing 4.10: The virtual opcodes after opcode transformation.

The custom stack-based architecture code is almost fully restored to its original x86 architecture state. The last step is to convert remaining custom architecture constructs to x86 architecture equivalents. This includes translating custom architecture instructions, `load` and `store`, to their x86 architecture equivalents, `push` and `pop`, as well as translating the virtual machine context registers to their equivalent x86 general-purpose registers.

Interestingly, the `VM Entrypoint` procedure and the `ret` handler procedure makes this process trivial, as they indirectly define which virtual machine context registers corresponds to which x86 general-purpose registers as depicted in table 4.6 below.

VM Entrypoint		VM Return	
0040da3e	pushfd	00417ea0	...
0040da3f	pushad	00418d09	popad
0040db14	cld	00418d0a	popfd
0040db7c	...	00418d0b	retn

Table 4.6: Exhibits of the `VM Entrypoint` and the `ret` handler procedures.

At the start of the `VM Entrypoint` procedure, a `pushfd` instruction and a `pushad` instruction loads the following registers onto the stack in the specified order: `EFLAGS`, `EAX`, `ECX`, `EDX`, `EBX`, `ESP`, `EBP`, `ESI`, `EDI`. Subsequent to this, the prologue of the custom architecture code stores them in virtual machine context registers in the reverse order.

Similar behaviour can be spotted in the epilogue of the custom architecture code flow, where all virtual machine context registers are pushed onto the stack in an order identical to that of the `VM Entrypoint` procedure. This time, however, they are retrieved by the `ret` handler procedure, that uses a `popad` instruction and a `popfd` instruction to propagate these internal register states to the equivalent x86 general-purpose registers.

Accounting for these observations, and the fact that the `stack` register in the custom stack-based architecture corresponds to the `esp` register in the x86 architecture, the code can be transformed and segregated as shown in table 4.7 below.

Prologue	Core	Epilogue
pop edi		pushfd
pop esi	push ecx	push eax
pop ebp	xor ecx,ecx	push ecx
pop ebx	mov ecx,0x05	push edx
pop ebx	add ecx,0xbe	push ebx
pop edx	sub ecx,0x34	push ebx
pop ecx	shl ecx,0x08	push ebp
pop eax	mov eax,ecx	push esi
popfd	pop ecx	push edi
add esp, 0x04		ret

Table 4.7: The virtual opcodes after register substitution and segregation.

As mentioned previously for the `set counter offset` instruction, it is possible to remove instructions whose purpose is entirely bound to the internal state of the virtual machine.

The prologue is solely responsible for populating the internal context registers with the states of the native x86 registers at the time of entering the virtual machine, as well as removing the opcode key from the stack, which was loaded by the `VM Trampoline` area.

The epilogue is solely responsible for loading the internal context registers to the stack and returning from the virtual machine, so that the `ret` handler procedure can populate the native x86 registers with the states of the internal context registers at the time of exiting the virtual machine.

It is thus conclusible that the prologue and epilogue can both be removed from the custom architecture code snippet as they are entirely responsible for mutation of the internal state of the virtual machine, thus serving no purpose for understanding what actions are being emulated by the virtual machine. By removing both the prologue and the epilogue, only the core instructions of the protected function remain as shown in listing 4.11 below.

```
push ecx
xor ecx,ecx
mov ecx,0x05
add ecx,0xbe
sub ecx,0x34
shl ecx,0x08
mov eax,ecx
pop ecx
ret
```

Listing 4.11: The virtual opcodes after removing the prologue and epilogue.

In comparison to the original program source code shown in section 3.1, the recovered procedure code is an exact match. In other words, it has proven possible to fully recover the original machine code of a protected procedure, albeit just for a single instance of the mutable virtual machine architecture.

4.3 Recovering virtual opcodes automatically

In order to fully automate the process of recovering the original machine code of a protected procedure, all previously documented steps must be automated and the mutability of the virtual machine architecture must be accounted for. However, the methodology still follows the general approach outlined in section 2.3 as shown below.

1. Deduce the functional responsibility of each individual handler procedure.
2. Map the byte-code sequence to handler procedures (recreate the code-flow).
3. Translate the custom architecture code-flow back to its original architecture.

The first step was done in the previous section. However, the mutability of the virtual machine architecture results in new obfuscation being applied to every single area of the virtual machine every time a new instance of the virtual machine is generated.

Furthermore, the handler procedure table is shuffled before inserting it into the virtual machine context structure at a randomly chosen offset. In fact, all virtual machine context offsets are randomized for every new instance of the virtual machine being generated.

Before any of the above steps are even approachable, a reliable deobfuscation engine must be developed. Ideally, this is done by implementing all of the counter-techniques described in section 2.2. However, the obfuscation patterns used by Themida are very monotone and predictable, and can be targeted specifically without the need for generic approaches.

Once the deobfuscation engine has been developed to a reliable extent, it must be applied to the `VM Entrypoint` area so that certain data can be carved from there. Specifically, as marked in listing 4.12 below, this includes the `delta` value, the address of the virtual machine context structure, the amount of handler procedures in the virtual machine and the handler procedure table offset into the virtual machine context structure.

```
0040db14  ...
; Initialize context (edi)
0040db7c  call $+5
0040db81  pop edi
0040dbeb  sub edi,0x08677a39
0040dc54  and edi,0xfffff000
0040dc9a  add edi,0x14
0040dcf5  mov eax,edi                ; delta
0040dd44  add edi,0x086776ee         ; context
; Initialize handlers
0040ddbd  cmp eax,[edi+0x64]
0040ddc3  jz 0x0040dfa8
0040ddce  mov [edi+0x64],eax
0040de18  mov ecx,0xaa               ; handler count
0040dea2  jmp 0x0040dfa0
0040dea7  add [edi+ecx*4+0x90],eax   ; handler table
0040df2d  add ecx,0xffffffff
0040dfa0  or ecx,ecx
0040dfa2  jnz 0x0040dea7
0040dfa8  ...
```

Listing 4.12: Exhibit of the deobfuscated entry point for the virtual machine.

Once the handler procedure table has been carved from the `VM Entrypoint` area, the table can be iterated and the `delta` value can be added to each of the stored handler procedures in order to initialize the actual address of the individual procedures.

It is now possible to deobfuscate each of the individual handlers and deduce their functional responsibilities as required by the first step of the general methodology. Since all operations supported by the virtual machine architecture have already been identified in section 4.1 above, with their handler procedures available in appendix A, this is simply a matter of identifying the core instructions of each individual handler procedure and using these to map the handler procedures to the predetermined set of supported operations.

When all handler procedures have been mapped to their respective virtual machine operations, the code-flow can be reconstructed from the byte-code sequence in accordance with step two of the general methodology. However, in order to parse the byte-code and perform this step, the opcode key must first be retrieved from the `VM Trampoline` area, and the byte-code buffer be initialized by addition of the `delta` value. Furthermore, as previously shown in section 4.1, all handler procedures that reads data from the byte-code buffer also contain a decryption routine as shown in listings 4.13, 4.14 and 4.15 below.

```
lodsb
add/sub/xor al,bl
add/sub/xor al,0xff         ; random constant
add/sub/xor al,0xff         ; random constant
add/sub/xor bl,al
```

Listing 4.13: Loading 8-bit data from the byte-code buffer.

```
lodsw
add/sub/xor ax,bx
add/sub/xor ax,0xffff       ; random constant
add/sub/xor ax,0xffff       ; random constant
add/sub/xor bx,ax
```

Listing 4.14: Loading 16-bit data from the byte-code buffer.

```
lodsd
add/sub/xor eax,ebx
add/sub/xor eax,0xffffffff  ; random constant
add/sub/xor eax,0xffffffff  ; random constant
add/sub/xor ebx,eax
```

Listing 4.15: Loading 32-bit data from the byte-code buffer.

For each of these handler procedures, including the `VM Dispatcher` area, the decryption routine must be identified and emulated using the relevant instructions and constants in order to decrypt the custom architecture data stored in the byte-code buffer. It should be noted that the decryption routines support only a small subset of possible instructions, consisting of the `add`, the `sub` and the `xor` instructions. The custom architecture code can now be recovered by parsing all the opcode data stored in the encrypted byte-code buffer.

Once the custom architecture code-flow has been recovered from the byte-code buffer, it need only be translated back into its original x86 architecture equivalent. Most constructs follows an easily translatable [load data → perform operation → store data] pattern. However, complex instructions require complex constructs which may prove less trivial to identify and translate, as they do not follow streamlined patterns. One such construct is the emulation of the `lodsb` instruction depicted in table 4.8 below.

	`_forwards:`	`_backwards:`
`load byte [reg32(esi)]`	`load reg32(esi)`	`load reg32(esi)`
`store reg8(eax)`	`load dword(1)`	`load dword(1)`
`jcc evaluate (df=1)`	`addx dword`	`subx dword`
`jcc _backwards`	`store reg32(esi)`	`store reg32(esi)`
`jmp _forwards`	`jmp _exit`	`jmp _exit`

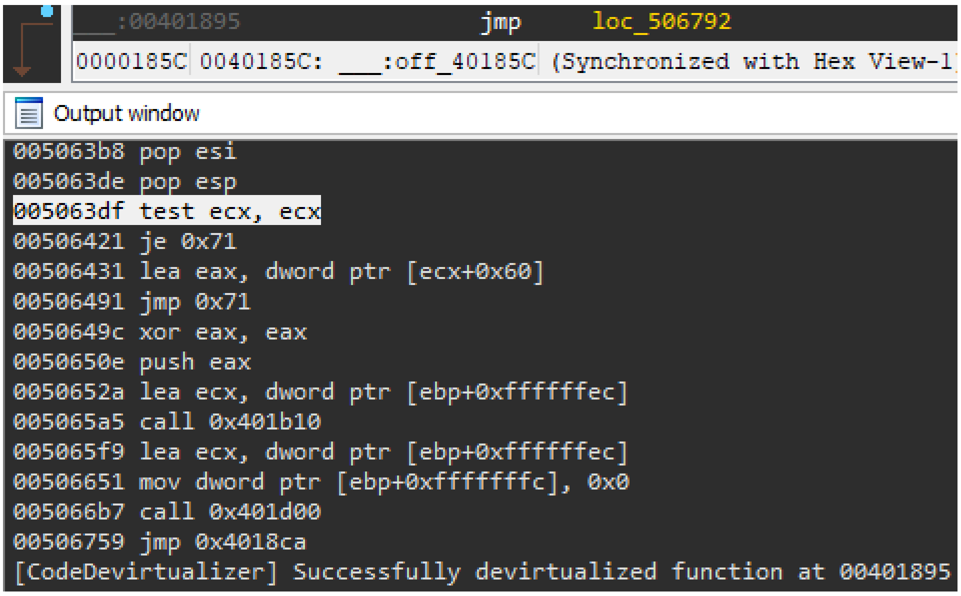
Table 4.8: Custom architecture construct of the `lodsb` instruction.

There are approximately 1000-1500 unique operation constructs covering a variety of implementations of distinct x86 architecture instructions. An approximation of 200-300 of these constructs are identified as complex instruction constructs, as they do not follow the easily translatable pattern described above. Thus, in order to reliably and successfully translate any protected function back to its original x86 architecture equivalent, all of these distinct constructs must be identified and accounted for in the translation process.

However, prior to removal of deadcode instructions in the custom architecture code-flow, as was demonstrated during the manual opcode recovery performed in section 4.2, a lot of constructs are ill-formed and therefore not translatable to equivalent x86 architecture constructs. It is thus of great importance that all deadcode instructions be stripped from the custom architecture code-flow before attempting to translate the custom architecture constructs to their x86 architecture equivalent.

While the chapters in this thesis focus primarily on the disection and analysis of Themida virtual machine architectures, the study further aims to verify that the solutions proposed in this thesis can be applied to Themida protected binaries in order to perform automated recovery of protected procedures. To this end, a devirtualization tool for Themida was developed in parallel to the writing of this thesis.

For demonstration purposes, a protected binary that has been virtualized with the CISC virtual machine architecture was downloaded from the internet[1] and unpacked using the guide available in section 3.4. The recovered procedure is shown in figure 4.2 below.



```
___:00401895                    jmp      loc_506792
0000185C 0040185C: ___:off_40185C (Synchronized with Hex View-1
```

```
005063b8 pop esi
005063de pop esp
005063df test ecx, ecx
00506421 je 0x71
00506431 lea eax, dword ptr [ecx+0x60]
00506491 jmp 0x71
0050649c xor eax, eax
0050650e push eax
0050652a lea ecx, dword ptr [ebp+0xffffffec]
005065a5 call 0x401b10
005065f9 lea ecx, dword ptr [ebp+0xffffffec]
00506651 mov dword ptr [ebp+0xfffffffc], 0x0
005066b7 call 0x401d00
00506759 jmp 0x4018ca
[CodeDevirtualizer] Successfully devirtualized function at 00401895
```

Figure 4.2: Example of code recovery for public UnpackMe binary.

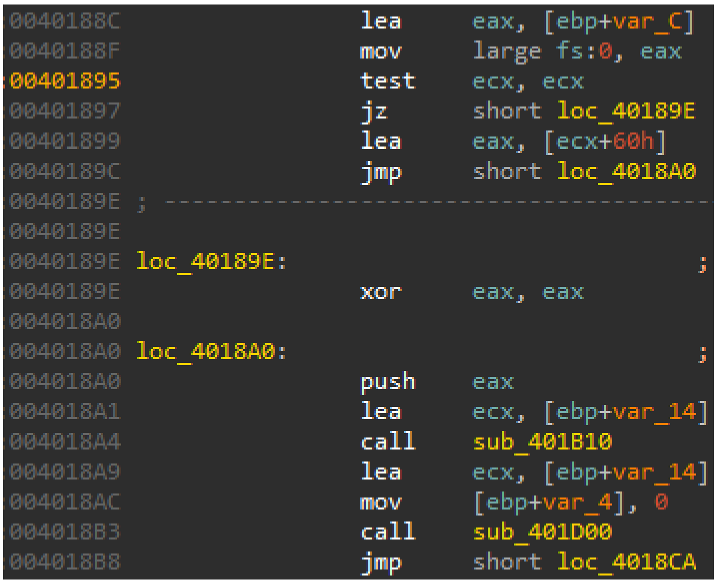
[1]`https://forum.tuts4you.com/topic/31743-unpackme-themida-2240/`

For clarification, the downloaded protected binary contains a virtualization-protected procedure which starts at address `00401895` with a `jmp` instruction leading into the `VM Trampoline` area. The code shown in the bottom of the figure is the custom architecture procedure after it has been recovered and retranslated to its x86 architecture equivalent.

Notice that the procedure has been protected with the *Dynamic Opcode* setting configured at *20% Dynamic*, leading to deadcode instructions being inserted into the custom architecture code-flow prior to the original procedure instructions. These instructions are usually easy to distinguish from the original procedure code, as the original procedure instructions are always located at the end of the custom architecture code flow and since the inserted deadcode instructions usually resolves to unsensible operations involving the stack. The highlighted instruction thus marks the start of the original procedure code in the custom architecture code-snippet.

For validation of the correctness and precision of the developed devirtualization tool for the CISC virtual machine architecture, the recovered code from the protected procedure is compared to the same code region in another restored binary that has been posted as a challenge solution by a known user in the community from which the binary was procured.

The same code region in the alternate binary, which has been accepted as a valid solution by the challenge author, can be seen in figure 4.3 below.



```
0040188C                  lea     eax, [ebp+var_C]
0040188F                  mov     large fs:0, eax
00401895                  test    ecx, ecx
00401897                  jz      short loc_40189E
00401899                  lea     eax, [ecx+60h]
0040189C                  jmp     short loc_4018A0
0040189E ; ---------------------------------------------
0040189E
0040189E loc_40189E:                                  ;
0040189E                  xor     eax, eax
004018A0
004018A0 loc_4018A0:                                  ;
004018A0                  push    eax
004018A1                  lea     ecx, [ebp+var_14]
004018A4                  call    sub_401B10
004018A9                  lea     ecx, [ebp+var_14]
004018AC                  mov     [ebp+var_4], 0
004018B3                  call    sub_401D00
004018B8                  jmp     short loc_4018CA
```

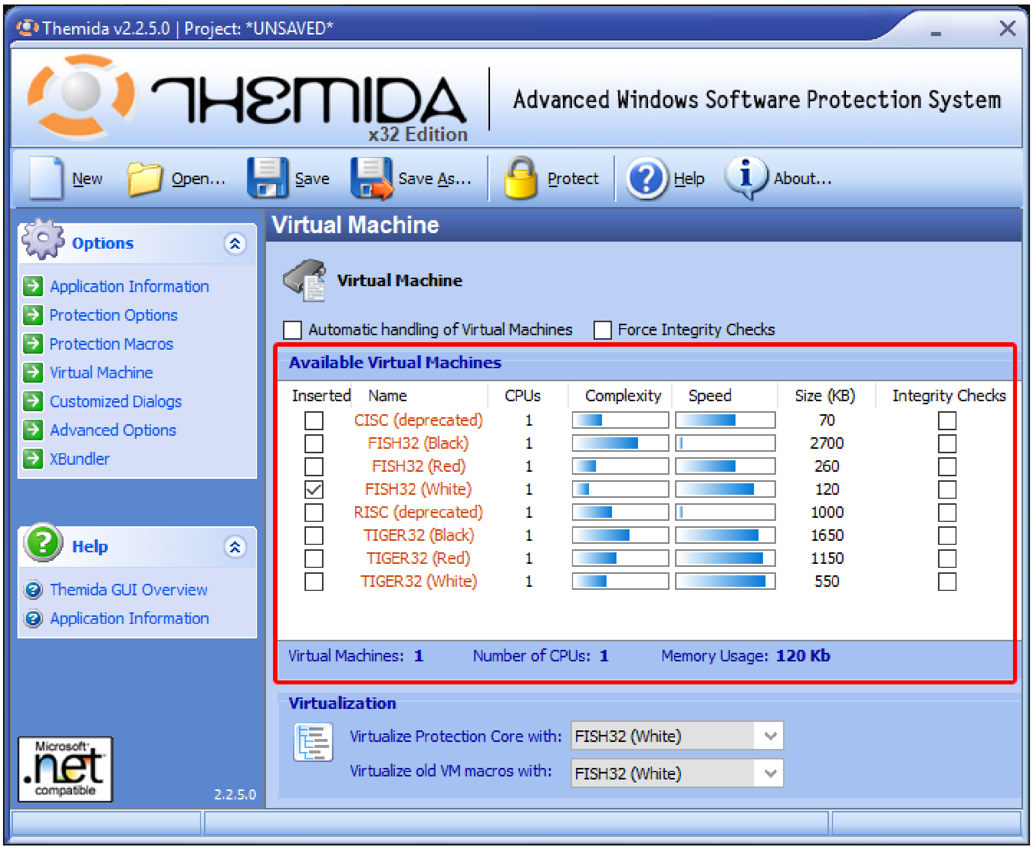
Figure 4.3: Solution of code recovery for public UnpackMe binary.

The two solutions are an exact match. It has thus proven possible to automate recovery of procedures that have been protected using the CISC virtual machine architecture and has furthermore been achieved with the herein developed devirtualization tool.

CHAPTER 5

The FISH architecture

This chapter will focus on Themida version 2.2.5.0 and the supported FISH virtual machine architecture. The herein analysed binary is protected by Themida with the lowest virtual machine protection setting, WHITE, as shown in figure 5.1 below.



☐ Automatic handling of Virtual Machines ☐ Force Integrity Checks

Available Virtual Machines

Inserted	Name	CPUs	Complexity	Speed	Size (KB)	Integrity Checks
☐	CISC (deprecated)	1			70	☐
☐	FISH32 (Black)	1			2700	☐
☐	FISH32 (Red)	1			260	☐
✓	FISH32 (White)	1			120	☐
☐	RISC (deprecated)	1			1000	☐
☐	TIGER32 (Black)	1			1650	☐
☐	TIGER32 (Red)	1			1150	☐
☐	TIGER32 (White)	1			550	☐

Virtual Machines: **1** Number of CPUs: **1** Memory Usage: **120 Kb**

Virtualization

Virtualize Protection Core with: FISH32 (White)

Virtualize old VM macros with: FISH32 (White)

Figure 5.1: The virtual machine options chosen for protecting the program.

Before the protected binary and its virtualization protection can be analysed, it must first be unpacked. An unpacking environment setup guide and an unpacking walkthrough for Themida 2.2.5.0 can be found in sections 3.2 and 3.4 respectively.

5.1 Analysis of the virtual machine

The binary has now been protected by Themida 2.2.5.0, and a reanalysis of the protected `test` function depicted in listing 5.1 below should be carried out.

```
.text:00401000 sub_401000      proc near
.text:00401000
.text:00401000                 jmp     sub_4D4CC5
.text:00401000
.text:00401000 ; ---------------------------------------------------------------
.text:00401005                 db 9E, C0, 96, 55, A4, A6, 5F, 8B, 8B, F7, B4
.text:00401010                 db C6, 9A, 81, 4C, 17, B1, 17, 42, 07, 84, 02
.text:0040101B                 db BF, 23, 85, 48, 9E, 60, ED, 26, 90, A8, 27
.text:00401026                 db 48, 81, AC, 7A, EB, 10, 57, 4C, 20, 89, FF
.text:00401031                 db 89, C0, 89, C9, 89, D2, 89, DB, 89, C0, 90
.text:00401031 ; ---------------------------------------------------------------
.text:00401031
.text:00401031 sub_401000      endp
```

Listing 5.1: The `test` function in the program after being protected.

Here, it can be seen that the machine code of the `test` function has been replaced with a jump to the virtual machine segment, specifically to the `VM Trampoline` area referenced in section 2.3, and that the rest of the function has been overwritten by a random sequence of bytes. The contents of the `VM Trampoline` area is shown in listing 5.2 below.

```
.themida:004D4CC5 sub_4D4CC5      proc near
.themida:004D4CC5
.themida:004D4CC5                 push    000D388B ; opcode key
.themida:004D4CCA                 push    3C ; first handler offset
.themida:004D4CCF                 jmp     loc_4348B9
.themida:004D4CCF
.themida:004D4CCF sub_4D4CC5      endp
```

Listing 5.2: The `VM Trampoline` area for the `test` function.

The `VM Trampoline` area stores the opcode key for the `test` function on the stack followed by the handler procedure table offset of the first handler procedure to be executed and then jumps to the `VM Entrypoint` area of the virtual machine. However, despite using the lowest obfuscation level, WHITE, the `VM Entrypoint` area remains obfuscated as briefly shown in listing 5.3 below.

```
.themida:004348EA                 ...
.themida:004348EB                 mov     ebx,75
.themida:004348F0                 mov     [ebp+ebx],ecx
.themida:004348F4                 mov     ebx,3F
.themida:004348F9                 mov     [ebp+ebx],400000
.themida:00434901                 ...
```

Listing 5.3: The unoptimized entry point for the virtual machine.

Using the counter-techniques described in section 2.2 on the `VM Entrypoint` area should reveal its core instructions as shown in listing 5.4 below.

```
004348b9   pushfd
004348ba   pushad

; Initialize context (ebp)
004348bb   call $+5
004348c0   pop ecx
004348c1   sub ecx,0x07
004348c4   sub ecx,0x000348b9           ; delta
004348ca   mov ebp,0x00016d4e
004348cf   add ebp,ecx                  ; context

; Spinlock (only one instance permitted at a time)
004348d1   push ecx
004348d2   mov ecx,0x00000001
004348dc   xor eax,eax
004348de   lock cmpxchg [ebp+0x36],ecx
004348e4   jz 0x004348ea
004348e6   pause
004348e8   jmp 0x004348dc
004348ea   pop ecx

; Initialize image base (delta) and opcode buffer
004348f0   mov [ebp+0x75],ecx           ; actual image base
004348f9   mov [ebp+0x3f],0x00400000    ; preferred image base
00434906   mov eax,[esp+0x28]
0043490a   add eax,ecx
0043490c   mov [ebp+0x08],eax           ; opcode buffer

; Initialize handlers
00434910   mov eax,0x000346a1
00434915   add eax,ecx
00434920   cmp [ebp+0x4f],eax           ; handler table
00434922   jz 0x00434940
00434925   mov ebx,0x00000218
0043492a   shr ebx,0x02                 ; handler count = 0x86 (134)
0043492d   push eax
0043492e   test ebx,ebx
00434930   jz 0x0043493a
00434932   add [eax],ecx
00434934   add eax,0x04
00434937   dec ebx
00434938   jmp 0x0043492e
0043493a   pop eax
0043493c   mov [ebp+0x4f],eax

; Initial handler dispatch
00434940   mov ebx,[esp+0x24]
00434944   shl ebx,0x02
00434947   add eax,ebx
00434949   jmp dword ptr [eax]
```

Listing 5.4: The optimized entry point for the virtual machine.

The VM Entrypoint area consists of six minor sub-areas, each of which hold the following responsibilities.

- The first sub-area pushes the EFLAGS register and all general-purpose registers to the stack, so that the virtual machine can pop them into its own proprietary registers as will be showcased later in section 5.2.
- The second sub-area uses the return address from a relative call to calculate the address of the virtual machine context structure as well as a delta value that is used for initialization of certain data in the virtual machine. In this instance, the delta value is the image base of the protected application.
- The third sub-area performs a mutual exclusion spinlock procedure, in order to ensure that only a single byte-code buffer is being processed at a time. This is important because the virtual machine uses a shared global context structure across executions.
- The fourth sub-area populates members of the virtual machine context structure, specifically the current image base, the preferred image base and the byte-code buffer which is initialized by adding the delta value from the second sub-area to the opcode key from the VM Trampoline area.
- The fifth sub-area uses the delta value from the second sub-area to perform a one-time initialization of the handler procedure table.
- The sixth sub-area uses the initial handler procedure offset from the VM Trampoline area to transfer execution flow to the handler procedure in question and start the virtual machine byte-code processing.

However, the last sub-area does not act as a central VM Dispatcher area, since it only transfers execution flow to a single fixed handler procedure. Instead, every handler in the custom architecture, that does not unconditionally exit the virtual machine, contains a proprietary dispatcher at the end of its procedure as depicted in listing 5.5 below.

```
mov [ebp+0x53],0x00000000
mov [ebp+0xa7],0x00000000
mov [ebp+0x89],0x00000000
mov [ebp+0x14],0x00000000
mov [ebp+0x8f],0x00000000
mov [ebp+0xa0],0x0000
mov [ebp+0x57],0x00
mov [ebp+0x68],0x00000000
; dispatcher area
mov eax,[ebp+0x4f] ; handler table
mov ebx,[ebp+0x08] ; byte-code buffer
movzx ebx,word ptr [ebx+0x00] ; byte-code buffer offset for next handler
sub ebx,0x2277293e ; random decrypt constant
and ebx,0xffff
add dword ptr [ebp+0x08],0x02 ; advance byte-code buffer
jmp dword ptr [eax+ebx*4]
```

Listing 5.5: The virtual handler procedure for reset.

As depicted in the above listing, opcode data in the custom virtual machine architecture is read using direct offsets into the byte-code buffer. Consequently, the byte-code buffer must be incremented by the amount of bytes consumed by the handler procedure, before transferring execution flow to the next handler procedure in the code-flow sequence. The dispatcher area itself consumes two bytes from the byte-code buffer, since it has to read the handler procedure table offset of the next handler procedure in the sequence. This results in the byte-code buffer being incremented by a minimum of two bytes for all handler procedures that contains a dispatcher area. Notice that the dispatcher area of individual handlers at random contains one or more decryption operations, such as the [`sub ebx,0x2277293e`] and the [`and ebx,0xffff`] instructions found in the listing above.

After thorough analysis of all handler procedures in the virtual machine, it is found that the supported operations can be separated into two distinct groups, *basic* operations and *complex* operations. An operation type is considered *basic*, if an explicit dedicated handler procedure exists for every single form of the operation in question, i.e. with one combination of operand types (register, memory, constant) and one operand size (8-bit, 16-bit, 32-bit). In contrast, an operation type is considered *complex*, if a single handler procedure covers every single form of the operation in question, i.e. with varying operand types (register, memory, constant) and varying operand sizes (8-bit, 16-bit, 32-bit).

The `reset` handler procedure depicted in the above listing is responsible for clearing a range of virtual machine context entries, henceforth referred to as the *context keys*, which are used to decrypt the mnemonic key in multi-complex handler procedures as shown in listing 5.6 below.

```
... ; Pre-mutate context keys
and [ebp+0xa7],0x198dd351
or  [ebp+0x89],0x4ad0b120
... ; Load mnemonic (subhandler)
mov ebx,[ebp+0x08] ; byte-code buffer
movzx ecx,byte ptr [ebx+0x00] ; encrypted mnemonic identifier
xor ecx,[ebp+0x53]
xor ecx,[ebp+0xa7]
add ecx,[ebp+0x68]
and [ebp+0xa7],0x6a381f86
sub ecx,[ebp+0x89]
xor [ebp+0x57],cl ; mnemonic key
... ; Post-mutate context keys
sub [ebp+0xa7],0x6d3caedd
or  [ebp+0x89],0x665b765e
xor [ebp+0x53],0x1446b272
or  [ebp+0x14],0x4ed21ba6
add [ebp+0x68],0x1981bd60
... ; Evaluate mnemonic identifier
mov bl,[ebp+0x57] ; mnemonic key
add bl,0x35
cmp bl,0xff ; mnemonic identifier
jnz _skip_mnemonic
...
```

Listing 5.6: Excerpts from the `binary operation` handler procedure

Multi-type handler procedures are operations that contains execution logic for multiple distinct mnemonics. In other words, multi-type handler procedures includes any handler procedure capable of emulating more than one instruction.

In multi-complex handler procedures, the virtual machine context keys are leveraged in a sequence of operations, which results in the mnemonic key for the handler procedure being decrypted and stored in the byte-sized context key. Notice that the context keys are being mutated before, during and after the decryption of the mnemonic key in the listing above.

After decryption, the mnemonic key is compared against a range of mnemonic identifiers to figure out which execution logic should be carried out by the handler procedure.

The list of operations supported by the multi-complex `unary operation` handler procedure can be found in table 5.1 below.

Operation	**Description**
inc	Increment an operand
dec	Decrement an operand
neg	Negate an operand
not	Bitwise NOT an operand

Table 5.1: Multi-complex unary operations supported by the virtual machine.

Similarly, the list of operations supported by the multi-complex `binary operation` handler procedure can be found in table 5.2 below.

Operation	**Description**
add	Add two operand
sub	Subtract two operand
imul	Mulitply two operand
and	Bitwise AND two operand
or	Bitwise OR two operand
xor	Bitwise XOR two operand
shr	Bitwise shift an operand right
shl	Bitwise shift an operand left
ror	Bitwise rotate an operand right
rcr	Bitwise rotate an operand right through carry
rol	Bitwise rotate an operand left
rcl	Bitwise rotate an operand left through carry
cmp	Compare two operand (using subtraction)
test	Compare two operand (using bitwise AND)
mov	Copy an operand into another operand
movsx	Copy an operand into another operand with sign-extension
movzx	Copy an operand into another operand with zero-extension

Table 5.2: Multi-complex binary operations supported by the virtual machine.

Lastly, the list of operations supported by the multi-complex `stack operation` handler procedure can be found in table 5.3 below.

Operation	**Description**
push	Push an operand to the stack
pop	Pop the top stack element into an operand

Table 5.3: Multi-complex stack operations supported by the virtual machine.

The three tables above details all the individual operations supported by each of the three multi-complex handler procedures that exist in the custom virtual machine architecture. Note that each of the above-mentioned multi-complex handler procedures are capable of carrying out each of the supported operations depicted in their respective tables, including in any form supported by the operation in question.

The virtual machine also support a range of complex operations shown in table 5.4 below.

Operation	**Description**
call	Return to an operand address and assign a relative constant return address
xchg	Exchange the contents of two operands

Table 5.4: Remaining complex operations supported by the virtual machine.

In contrast to the multi-complex operations listed above, these two complex operations are not implemented by the same handler procedure. Instead, they are each implemented by their own individual handler procedure. However, they are still considered complex operations, as their single individual handler procedure implements every supported form of their individual operations.

The virtual machine supports a single multi-basic operation. The list of operations supported by the multi-basic `EFLAGS` handler procedure can be found in table 5.5 below.

Operation	**Description**
`clc`	Clear the carry flag (`CF = 0`)
`cld`	Clear the direction flag (`DF = 0`)
`cli`	Clear the interrupt flag (`IF = 0`)
`cmc`	Complement the carry flag (`CF = NOT(CF)`)
`stc`	Set the carry flag (`CF = 1`)
`std`	Set the direction flag (`DF = 1`)
`sti`	Set the interrupt flag (`IF = 1`)

Table 5.5: Multi-basic `EFLAGS` operations supported by the virtual machine.

Similar to the multi-complex handler procedures, this is a single handler procedure that implements multiple distinct operations. However, the operations are considered basic as they do not support any type of operands at all.

The list of supported basic string operations can be found in table 5.6 below.

Operation	**8-bit**	**16-bit**	**32-bit**	**Description**
lods	x	x	x	Immitates the `lods` instruction albeit on arbitrary registers
stos	x	x	x	Immitates the `stos` instruction albeit on arbitrary registers
cmps	x	x	x	Immitates the `cmps` instruction albeit on arbitrary registers
subs	x	x	x	Same as `cmps`, but uses the `sub` instruction instead of `cmp`
movs	x	x	x	Immitates the `movs` instruction albeit on arbitrary registers

Table 5.6: Basic string operations supported by the virtual machine.

The basic string operations depicted in the above table differs from multi-type handler procedures in that they have a single individual handler procedure for each individual operation type. Moreover, they differ from complex handler procedures in that they have an individual handler procedure for each supported form of their individual operations. In other words, there are distinct handler procedures for every form of every operation depicted in the above table.

Lastly, the virtual machine support a variety of miscellaneous basic operations pertaining to execution flow control and edge-case necessities as shown in table 5.7 below.

Operation	**Description**
reset	Reset all internal context keys to zero
crypt	Populates a context register with the crypt seed value
load stack	Read the stack pointer into the context stack register
store stack	Write the context stack register to the stack pointer
add stack,imm8	Add an 8-bit constant to the stack pointer and context stack register
pushfd	Push the context `EFLAGS` register to the stack
popfd	Pop the top stack element into the context `EFLAGS` register
load align imm32	Read a constant value aligned by the image base into a context register
jcc internal imm32	Increment the byte-code buffer by a constant value if condition evaluates to true
jcc external imm32	Return to a relative constant address if condition evaluates to true
jmp internal imm32	Increment the byte-code buffer by a constant value
jmp external imm32	Return to a relative constant address
jmp external imm32 dll	Return to a relative constant address and realign up to two offsets
jmp external reg	Return to an address stored in a context register
jmp external [reg]	Return to an address stored in a dereferenced context register
ret	Reset the stack and return from the virtual machine

Table 5.7: Remaining operations supported by the virtual machine.

Using sequences of supported instructions from all of the above tables combined, it is possible to emulate almost any x86 usermode instruction in the virtual machine architecture.

However, the above tables depict only high-level descriptions of the supported operations and does not adequately outline their implementation details. For a exhaustive list of handler procedures and their core instructions respectively, please refer to appendix B.

Notice however, that the proprietary dispatcher areas and all references to mutation of the context keys have been omitted in these handler procedure listings for readability reasons, as they do not convery any information about the core responsibility of the individual handler procedures.

Furthermore, the majority of the complex handler procedures detailed in appendix B reuse identical subhandler procedure code to manage support for multi-type mnemonic handling and arbitrary operands. For this reason, these complex handler procedures have been shortened to include a prologue list of included subhandler procedures followed by the unique code of the particular complex handler procedure. For a list of subhandler procedures and their core instructions respectively, please refer to appendix C.

Now that the entire virtual machine has been mapped and analysed, it is possible to iterate the byte-code buffer in order to determine the sequence of invoked handler procedures and thereby reconstruct the custom architecture code flow.

5.2 Recovering virtual opcodes manually

By placing a software breakpoint in the initial handler dispatch of the `VM Entrypoint` area and stepping through each of the invoked handler procedures in the code-flow execution chain to recover the decrypted byte-code data, it is possible to reconstruct the custom architecture code flow as shown in listing 5.7 below.

```
reset
load stack
stack(0xb5) (1, 3, 0x85)
stack(0xb5) (1, 3, 0x3b)
stack(0xb5) (1, 3, 0x1a)
stack(0xb5) (1, 3, 0x6c)
stack(0xb5) (1, 3, 0x6c)
stack(0xb5) (1, 3, 0x2e)
stack(0xb5) (1, 3, 0x9c)
stack(0xb5) (1, 3, 0x22)
popfd
add stack,0x08
... ; deadcode instructions
stack(0x42) (1, 3, 0x9c)
binary(0x08) (1, 3, 0x9c),(1, 3, 0x9c)
binary(0xd7) (1, 3, 0x9c),(3, 3, 0x05)
binary(0xa2) (1, 3, 0x9c),(3, 3, 0xbe)
binary(0x10) (1, 3, 0x9c),(3, 3, 0x34)
binary(0x8d) (1, 3, 0x9c),(3, 3, 0x08)
binary(0xd7) (1, 3, 0x22),(1, 3, 0x9c)
stack(0xb5) (1, 3, 0x9c)
stack(0x42) (1, 3, 0x22)
pushfd
stack(0x42) (1, 3, 0x22)
stack(0x42) (1, 3, 0x9c)
stack(0x42) (1, 3, 0x2e)
stack(0x42) (1, 3, 0x6c)
stack(0x42) (1, 3, 0x93)
stack(0x42) (1, 3, 0x1a)
stack(0x42) (1, 3, 0x3b)
stack(0x42) (1, 3, 0x85)
ret
```

Listing 5.7: The raw extracted virtual opcodes for the `test` function.

Please note that the above process is extremely time consuming and very prone to errors.

The recovered custom architecture code flow contains minor obfuscation patterns that must be dealt with in order to reveal the core instructions of the code snippet. First of all, it is possible to remove instructions whose purpose is entirely bound to the internals of the virtual machine, such as the `reset` instruction and the `load stack` instruction. Additionally, it is possible to remove deadcode instructions, of which the recovered code snippet above contained approximately 300 instances. However, for readability purposes, these are represented by an ellipsis and related comment in the above code snippet.

Furthermore, the instruction operands are represented by a tuple structure of the format, `(type, size, data)`, of which the representative values can be found in table 5.8 below.

ID	Type	Size
1	Register	8-bit
2	Memory	16-bit
3	Immediate	32-bit

Table 5.8: Operand values for types and sizes.

The code snippet is thus reducible as shown in listing 5.8 below.

```
stack(0xb5) reg32(0x85)
stack(0xb5) reg32(0x3b)
stack(0xb5) reg32(0x1a)
stack(0xb5) reg32(0x6c)
stack(0xb5) reg32(0x6c)
stack(0xb5) reg32(0x2e)
stack(0xb5) reg32(0x9c)
stack(0xb5) reg32(0x22)
popfd
add stack,0x08
stack(0x42) reg32(0x9c)
binary(0x08) reg32(0x9c),reg32(0x9c)
binary(0xd7) reg32(0x9c),imm32(0x05)
binary(0xa2) reg32(0x9c),imm32(0xbe)
binary(0x10) reg32(0x9c),imm32(0x34)
binary(0x8d) reg32(0x9c),imm32(0x08)
binary(0xd7) reg32(0x22),reg32(0x9c)
stack(0xb5) reg32(0x9c)
stack(0x42) reg32(0x22)
pushfd
stack(0x42) reg32(0x22)
stack(0x42) reg32(0x9c)
stack(0x42) reg32(0x2e)
stack(0x42) reg32(0x6c)
stack(0x42) reg32(0x93)
stack(0x42) reg32(0x1a)
stack(0x42) reg32(0x3b)
stack(0x42) reg32(0x85)
ret
```

Listing 5.8: The virtual opcodes after optimization.

The reduced custom virtual machine architecture code flow contains a range of unresolve mnemonics. These are represented by the name of their respective multi-complex handler procedure as well as the mnemonic key that was decrypted during audit of the handler procedure. In order to resolve these mnemonic keys and map them to their corresponding mnemonic identifiers, each of the responsible handler procedures must be analysed. Two excerpts from the `stack operation` handler procedure can be seen in table 5.9 below.

push	pop

`...`
`cmp cl,0x42 ; push`
`jnz _skip_push`
`...`
	`...`
	`cmp cl,0xb5 ; pop`
	`jnz _skip_pop`
	`...`

Table 5.9: Excerpts from the `stack operation` handler procedure.

Inside the `stack operation` handler procedure, the mnemonic identifier is compared to `0x42` before carrying out the `push` instruction and similarly compared to `0xb5` before carrying out the `pop` instruction. It can thus be concluded that these mnemonic identifiers map to their respective instruction mnemonics. Performing a similar analysis for the unary operations handler procedure or the binary operations handler procedure reveals another set of identifier-to-mnemonic mappings as depicted in tables 5.10 and 5.11 below.

Mnemonic	ID	Mnemonic	ID
inc	0x41	neg	0x96
dec	0x72	not	0x39

Table 5.10: Mnemonic identifier mappings for unary operations.

Mnemonic	ID	Mnemonic	ID	Mnemonic	ID
add	0xa2	shr	0x1f	cmp	0xc1
sub	0x10	shl	0x8d	test	0x06
imul	0xc4	ror	0x15	mov	0xd7
and	0x49	rcr	0x4d	movsx	0xca
or	0x40	rol	0xe3	movzx	0x9c
xor	0x08	rcl	0x05		

Table 5.11: Mnemonic identifier mappings for binary operations.

By consideration of these identifier-to-mnemonic mappings, it is possible to reconstruct the mnemonics in the custom architecture code flow as shown in listing 5.9 below.

```
pop reg32(0x85)
pop reg32(0x3b)
pop reg32(0x1a)
pop reg32(0x6c)
pop reg32(0x6c)
pop reg32(0x2e)
pop reg32(0x9c)
pop reg32(0x22)
popfd
add stack,0x08
push reg32(0x9c)
xor reg32(0x9c),reg32(0x9c)
mov reg32(0x9c),imm32(0x05)
add reg32(0x9c),imm32(0xbe)
sub reg32(0x9c),imm32(0x34)
shl reg32(0x9c),imm32(0x08)
mov reg32(0x22),reg32(0x9c)
pop reg32(0x9c)
push reg32(0x22)
pushfd
push reg32(0x22)
push reg32(0x9c)
push reg32(0x2e)
push reg32(0x6c)
push reg32(0x93)
push reg32(0x1a)
push reg32(0x3b)
push reg32(0x85)
ret
```

Listing 5.9: The virtual opcodes after restoration of mnemonics.

The custom virtual machine architecture code flow has now almost been restored to its original x86 architecture construct, with the sole exception of the operation operands.

The last step is therefore to resolve the final operands by translating the virtual machine context registers to their equivalent x86 general-purpose registers.

Interestingly, the `VM Entrypoint` procedure and the `ret` handler procedure makes this process trivial, as they indirectly define which virtual machine context registers corresponds to which x86 general-purpose registers as depicted in table 5.12 below.

VM Entrypoint		VM Return	
004348b9	pushfd	00428cc4	...
004348ba	pushad	00428cca	popad
004348bb	...	00428cd1	popfd
004348c0	...	00428cd2	add esp,0x04
004348c1	...	00428cd8	ret

Table 5.12: Excerpts from the `VM Entrypoint` and the `ret` handler procedures.

At the start of the `VM Entrypoint` procedure, a `pushfd` instruction and a `pushad` instruction loads the following registers onto the stack in the specified order: `EFLAGS`, `EAX`, `ECX`, `EDX`, `EBX`, `ESP`, `EBP`, `ESI`, `EDI`. Subsequent to this, the prologue of the custom architecture code stores them in virtual machine context registers in the reverse order.

Similar behaviour can be spotted in the epilogue of the custom architecture code flow, where all virtual machine context registers are pushed onto the stack in an order identical to that of the `VM Entrypoint` procedure. This time, however, they are retrieved by the `ret` handler procedure, that uses a `popad` instruction and a `popfd` instruction to propagate these internal register states to the equivalent x86 general-purpose registers.

Accounting for these observations, and the fact that the `stack` register in the custom architecture corresponds to the `esp` register in the x86 architecture, the code can be transformed and segregated as shown in table 5.13 below.

Prologue	Core	Epilogue
pop edi	...	push eax
pop esi	push ecx	pushfd
pop ebp	xor ecx,ecx	push eax
pop ebx	mov ecx,0x05	push ecx
pop ebx	add ecx,0xbe	push edx
pop edx	sub ecx,0x34	push ebx
pop ecx	shl ecx,0x08	push esp
pop eax	mov eax,ecx	push ebp
popfd	pop ecx	push esi
add esp,0x08		push edi
		ret

Table 5.13: The virtual opcodes after register substitution and segregation.

As mentioned previously for the `reset` instruction and the `load stack` instruction, it is possible to remove instructions whose purpose is entirely bound to the internal state of the virtual machine.

The prologue is solely responsible for populating the internal context registers with the states of the native x86 registers at the time of entering the virtual machine, as well as removing the opcode key from the stack, which was loaded by the `VM Trampoline` area.

The epilogue is solely responsible for loading the internal context registers to the stack and returning from the virtual machine, so that the `ret` handler procedure can populate the native x86 registers with the states of the internal context registers at the time of exiting the virtual machine.

It is thus conclusible that the prologue and epilogue can both be removed from the custom architecture code snippet as they are entirely responsible for mutation of the internal state of the virtual machine, thus serving no purpose for understanding what actions are being emulated by the virtual machine. By removing both the prologue and the epilogue, only the core instructions of the protected function remain as shown in listing 5.10 below.

```
...
push ecx
xor ecx,ecx
mov ecx,0x05
add ecx,0xbe
sub ecx,0x34
shl ecx,0x08
mov eax,ecx
pop ecx
ret
```

Listing 5.10: The virtual opcodes after removing the prologue and epilogue.

In comparison to the original program source code shown in section 3.1, the recovered procedure code is an exact match. In other words, it has proven possible to fully recover the original machine code of a protected procedure, albeit just for a single instance of the mutable virtual machine architecture.

5.3 Recovering virtual opcodes automatically

In order to fully automate the process of recovering the original machine code of a protected procedure, all previously documented steps must be automated and the mutability of the virtual machine architecture must be accounted for. However, the methodology still follows the general approach outlined in section 2.3 as shown below.

1. Deduce the functional responsibility of each individual handler procedure.
2. Map the byte-code sequence to handler procedures (recreate the code-flow).
3. Translate the custom architecture code-flow back to its original architecture.

The first step was done in the previous section. However, the mutability of the virtual machine architecture results in new obfuscation being applied to every single area of the virtual machine every time a new instance of the virtual machine is generated. Furthermore, the handler procedure table is shuffled before inserting it into the virtual machine context structure at a randomly chosen offset. In fact, all virtual machine context offsets are randomized for every new instance of the virtual machine being generated.

Before any of the above steps are even approachable, a reliable deobfuscation engine must be developed. Ideally, this is done by implementing all of the counter-techniques described in section 2.2. However, the obfuscation patterns used by Themida are very monotone and predictable, and can be targeted specifically without the need for generic approaches.

Once the deobfuscation engine has been developed to a reliable extend, it must be applied to the `VM Entrypoint` area so that certain data can be carved from there. Specifically, as marked in listing 5.11 below, this includes the `delta` value, the virtual machine context structure, the image base offsets, the amount of handler procedures in the virtual machine and the handler procedure table offset into the virtual machine context structure.

```
004348ba  ...
; Initialize context (ebp)
004348bb  call $+5
004348c0  pop ecx
004348c1  sub ecx,0x07
004348c4  sub ecx,0x000348b9           ; delta
004348ca  mov ebp,0x00016d4e
004348cf  add ebp,ecx                  ; context
; Initialize image base (delta) and opcode buffer
004348f0  mov [ebp+0x75],ecx           ; actual image base
004348f9  mov [ebp+0x3f],0x00400000    ; preferred image base
00434906  mov eax,[esp+0x28]
0043490a  add eax,ecx
0043490c  mov [ebp+0x08],eax           ; opcode buffer
; Initialize handlers
00434910  mov eax,0x000346a1
00434915  add eax,ecx
00434920  cmp [ebp+0x4f],eax           ; handler table
00434922  jz 0x00434940
00434925  mov ebx,0x00000218
0043492a  shr ebx,0x02                 ; handler count = 0x86 (134)
0043492d  push eax
0043492e  test ebx,ebx
00434930  jz 0x0043493a
00434932  add [eax],ecx
00434934  add eax,0x04
00434937  dec ebx
00434938  jmp 0x0043492e
0043493a  pop eax
0043493c  mov [ebp+0x4f],eax
00434940  ...
```

Listing 5.11: Exhibit of the deobfuscated entry point for the virtual machine.

Once the handler procedure table has been carved from the `VM Entrypoint` area, the table can be iterated and the `delta` value can be added to each of the stored handler procedures in order to initialize the actual address of the individual procedures.

It is now possible to deobfuscate each of the individual handlers and deduce their functional responsibilities as required by the first step of the general methodology. Since all operations supported by the virtual machine architecture have already been identified in section 5.1 above, with their handler procedures available in appendix B, this is simply a matter of identifying the core instructions of each individual handler procedure and using these to map the handler procedures to the predetermined set of supported operations. It should however be noted, that the core instructions are prone to rearrangement and that the same holds true for the placement of subhandlers in the complex handler procedures.

When all handler procedures have been mapped to their respective virtual machine operations, the code-flow can be reconstructed from the byte-code sequence in accordance with step two of the general methodology. However, in order to parse the byte-code and perform this step, the opcode key and the initial handler procedure offset must first be retrieved from the `VM Trampoline` area, and the byte-code buffer be initialized by addition of the `delta` value. Furthermore, as previously shown in section 5.1, all operations that does not unconditionally exit the virtual machine, contains a proprietary dispatcher at the end of the handler procedure as shown in listing 5.12 below.

```
mov eax,[ebp+0x4f] ; handler table
movzx ebx,word ptr [ebx+0x02] ; byte-code buffer offset for next handler
add/sub/xor ebx,0xffffffff ; random constant
and ebx,0xffff
add dword ptr [ebp+0x08],0x04 ; advance byte-code buffer
jmp dword ptr [eax+ebx*4]
```

Listing 5.12: The general handler procedure dispatcher layout.

Note that the dispatcher area may contain one or more decryption operations as depicted by [`add/sub/xor ebx,0xffffffff`] and [`and ebx,0xffff`] in the above listing. For each of these handler procedures, including the dispatcher area of the `VM Entrypoint` area, the decryption routine must be identified and emulated using the relevant instructions and constants in order to locate the next handler procedure in the execution sequence. It should furthermore be noted that the first decryption operation support only a small subset of possible instructions, consisting of the `add`, the `sub` and the `xor` instructions. The custom architecture code flow can now be recovered by parsing and emulating the dispatcher area of every handler procedure encountered in the byte-code buffer.

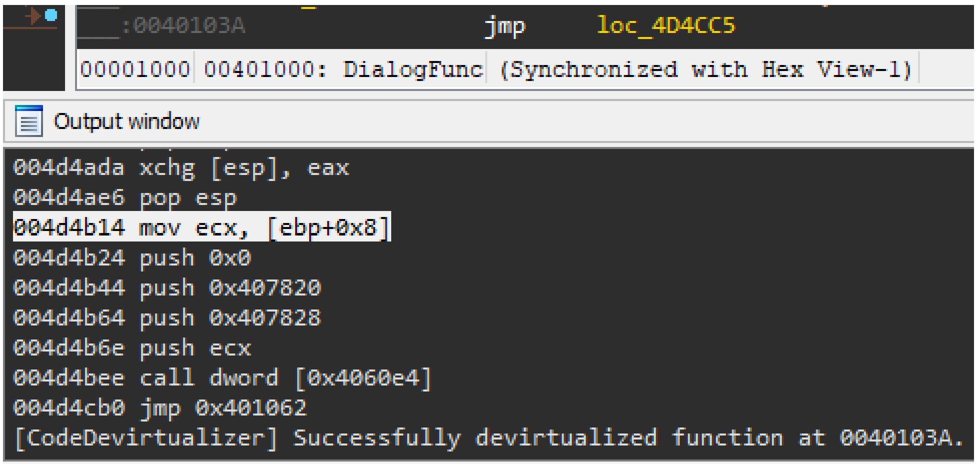
Once the custom architecture code flow has been recovered from the byte-code buffer, it must be translated back into its original x86 architecture equivalent. Most constructs are identical in both the custom architecture and the x86 architecture, and therefore does not require additional processing to translate between architectures. However, the exception to this observation are the multi-type handler procedures supported by the custom virtual machine architecture. Because these individual handler procedures are responsible for multiple corresponding operations in the x86 architecture, an additional step must be carried out to resolve the particular instruction being carried out by each particular invocation of a multi-complex handler procedure.

As previously explained in section 5.1, the context keys are responsible for resolving the mnemonic key for multi-complex handler procedures. For each handler procedure in the virtual machine architecture, context key operations must thus be identified and emulated using the relevant instructions and constants in order to decrypt the intended mnemonic identifiers for multi-complex handler procedures and translate them to their equivalent x86 architecture instruction. The context offsets to be identified as context keys can be parsed from the `reset` handler procedure.

The final aspect in which the custom virtual machine architecture deviates from the Intel x86 architecture, is that it contains two intermediary registers, herein referred to as `addr1` and `addr2`, which are meant for computing dynamic constants and relative addresses. Any construct that makes use of these intermediary registers can be translated to the x86 architecture by replacing the `addr` register with dynamically resolved value.

While the chapters in this thesis focus primarily on the disection and analysis of Themida virtual machine architectures, the study further aims to verify that the solutions proposed in this thesis can be applied to Themida protected binaries in order to perform automated recovery of protected procedures. To this end, a devirtualization tool for Themida was developed in parallel to the writing of this thesis.

For demonstration purposes, a protected binary that has been virtualized with the FISH virtual machine architecture was downloaded from the internet[1] and unpacked using the guide available in section 3.4. The recovered procedure is shown in figure 5.2 below.


```
:0040103A                    jmp      loc_4D4CC5
00001000 00401000: DialogFunc (Synchronized with Hex View-1)
```

```
004d4ada xchg [esp], eax
004d4ae6 pop esp
004d4b14 mov ecx, [ebp+0x8]
004d4b24 push 0x0
004d4b44 push 0x407820
004d4b64 push 0x407828
004d4b6e push ecx
004d4bee call dword [0x4060e4]
004d4cb0 jmp 0x401062
[CodeDevirtualizer] Successfully devirtualized function at 0040103A.
```

Figure 5.2: Example of code recovery for public UnpackMe binary.

For clarification, the downloaded protected binary contains a virtualization-protected procedure which starts at address `0040103a` with a `jmp` instruction leading into the `VM Trampoline` area. The code shown in the bottom of the figure is the custom architecture

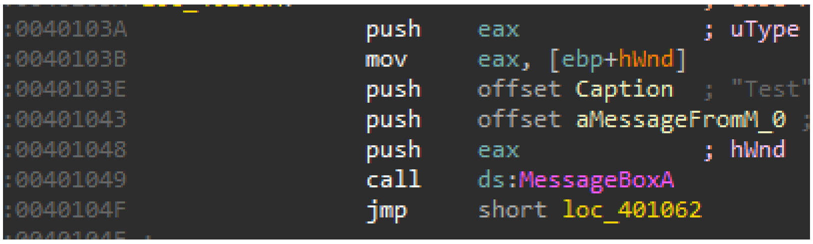
[1]`https://forum.tuts4you.com/topic/33562-unpackme-themida-2260/`

procedure after it has been recovered and retranslated to its x86 architecture equivalent.

Notice that the procedure being protected leads to deadcode instructions being inserted into the custom architecture code-flow prior to the original procedure instructions. These instructions are usually easy to distinguish from the original procedure code, as the original procedure instructions are always located at the end of the custom architecture code flow and since the inserted deadcode instructions usually resolves to unsensible operations involving the stack. The highlighted instruction thus marks the start of the original procedure code in the custom architecture code-snippet.

For validation of the correctness and precision of the developed devirtualization tool for the FISH virtual machine architecture, the recovered code from the protected procedure is compared to the same code region in another restored binary that has been posted as a challenge solution by a known user in the community from which the binary was procured.

The same code region in the alternate binary, which has been accepted as a valid solution by the challenge author, can be seen in figure 5.3 below.



```
:0040103A                push    eax                ; uType
:0040103B                mov     eax, [ebp+hWnd]
:0040103E                push    offset Caption     ; "Test"
:00401043                push    offset aMessageFromM_0 ;
:00401048                push    eax                ; hWnd
:00401049                call    ds:MessageBoxA
:0040104F                jmp     short loc_401062
```

Figure 5.3: Solution of code recovery for public UnpackMe binary.

The resolved address for the imported `MessageBoxA` function is referenced from address `004060e4`, and the addresses, `00407820` and `00407828`, contains the same string references as depicted in the alternate binary shown in the figure above. The two solutions are therefore an extremely close match, with the only difference being the general-purpose register used to carry the `hWnd` parameter for the invocation of the imported `MessageBoxA` function. It has thus proven possible to automate recovery of procedures that have been protected using the FISH virtual machine architecture and has furthermore been achieved with the herein developed devirtualization tool.

CHAPTER 6

Results and evaluation

This chapter aims to gather the results collected throughout this thesis up until this point and evaluate them according to industry needs. Thus far, two indepedent virtual machine architectures supported by the leading software protection product, Themida, have been dissected and analysed in great detail in chapters 4 and 5 respectively. For each of these virtual machine architectures, the control flow mechanisms have been identified and all supported operations have been mapped and thoroughly documented. It has also been documented how the virtual machine architectures parse and interpret virtual opcodes from the byte-code buffers and how the embedded virtual machine architecture delivers generated results to the respective procedures in the executable binary. Lastly, a manual and dynamic approach for restoring virtual machine opcodes to their original x86 machine code states have been presented for each of the virtual machine architectures and have furthermore been developed into an automated and static process capable of restoring procedures that have been protected by software virtualization from Themida.

In order to evaluate if the derived solutions are suitable for industrial needs, there are a range of parameters that must be considered. For example, as explained in the introduction of this thesis, anti-virus products are very concerned about efficiency because it interferes with the usability of their on-access scan feature. The approach used throughout this thesis, in which an entire virtual machine architecture must be analysed deeply and have a specific routine developed for that particular architecture, involves extremely long development processes albeit at the benefit of splendid runtime performance. However, in order to validate if a given runtime performance is qualified for a particular use-case, a tolerance threshold must be defined. For an on-access scan feature supplied by an anti-virus product, in which a noticeable delay in execution times should not be experienced by an end-user, a tolerable time-frame should not exceed 5 seconds. To compare the herein developed solutions against the established tolerance threshold, the execution times of the solutions have been captured over five consecutive runs against the UnpackMe binaries previously presented in sections 4.3 and 5.3 respectively. The results of these captures can be found in table 6.1 below.

VM	**Decode VM (seconds)**					**Decode Opcodes (seconds)**				
	#1	**#2**	**#3**	**#4**	**#5**	**#1**	**#2**	**#3**	**#4**	**#5**
CISC	0.229	0.205	0.196	0.213	0.226	0.277	0.294	0.301	0.397	0.418
FISH	0.495	0.479	0.489	0.479	0.483	0.279	0.250	0.251	0.247	0.246

Table 6.1: Performance tests for automated devirtualization of UnpackMe binaries.

The ”Decode VM” column depicts the time it takes for the developed solutions to parse and decode the `VM Entrypoint` area of the respective virtual machine architecture as well as to identify all of the individual handler procedures and map them to their corresponding virtual machine architecture operations. Note that this process need only be carried out once for a given instance of a virtual machine architecture.

The ”Decode Opcodes” column depicts the time it takes for the developed solutions to parse and decode virtual machine opcodes from a byte-code buffer and translate these operations from the virtual machine architecture back to its original x86 architecture equivalents. Note that this process must be carried out for all protected procedures that are to be restored to their original machine code state.

The average execution times are found as shown in table 6.2 below.

VM	Decode VM (seconds)	Decode Opcodes (seconds)
CISC	0.2138	0.3374
FISH	0.4850	0.2546

Table 6.2: Average performance for automated devirtualization of UnpackMe binaries.

The execution times in relation to eachother matches initial expectations. The CISC virtual machine architecture is much simpler than the FISH virtual machine architecture, and with significantly smaller handler procedures, leading to the vast difference in the ”Decode VM” execution time column. However, the CISC virtual machine architecture is also fundamentally different from the original x86 architecture and requires additional processing to convert stack-operations into flat x86 architecture operations. In constrast to that, the FISH virtual machine architecture is almost identical to the original x86 architecture and most operations can be translated directly between the two.

Granted that the virtual machine architecture need only be decoded once in order to allow the decoding of arbitrary protected procedures, the amount of protected procedures that can be recovered from the CISC virtual machine architecture before passing the tolerance threshold of 5 seconds can be calculated as follows.

$$(5 - 0.2138) \div 0.3374 = 14.1855$$

Similarly, the amount of protected procedures that can be recovered from the FISH virtual machine architecture before passing the tolerance threshold can be calculated as follows.

$$(5 - 0.4850) \div 0.2546 = 17.7337$$

However, the above calculations assume that an executable binary is not protected by software packing and that its protected procedures have been located in advance. Since this is usually not the case, the above numbers depict optimistic scenarios at best. Furthermore, the obfuscation levels used in the tested UnpackMe binaries varies a lot, as the executable binary protected by the CISC virtual machine architecture has medium-level obfuscation settings enabled, whereas the executable binary protected by the FISH virtual machine architecture is using the lowest obfuscation level, WHITE, yielding an overall unbalanced comparison. In a realistic scenario where the user might opt-in for the highest obfuscation level, BLACK, execution time captures for decoding the virtual machine would be much higher and the size of the opcode buffers would be vastly different. It should further be noted that no efforts have been made to optimize the routines implemented in the automated devirtualization process, so the execution times could be improved significantly given the necessary time investment.

In section 5.3, the automatically recovered code from the FISH virtual machine architecture matched the accepted solution very closely, albeit with the slight deviation of using a different general-purpose register when translated back into its original x86 machine code. By manual re-inspection of the binary, and by numerous tests and validations of other protected binaries, it has been derived that there are no errors in the way that the automated devirtualization software restores the FISH virtual machine opcodes to their original x86 equivalents. The slight mismatch in restored general-purpose register for the accepted solution is therefore accredited to the author of the solution.

While the efficiency and effectiveness parameters have turned out great, the same cannot be said for the coverage of the solutions. Currently, the solutions support only 32-bit architectures and the CISC virtual machine family reached end of life in 2015. While the herein performed analysis for the CISC virtual machine architecture is relevant for all existing versions of the architecture family, and can be used for older executable binaries that have been protected by Themida, the solution is not applicable for modern content. Furthermore, the performed analysis for the FISH virtual machine architecture covers only the inital version released in 2013, for which a great number of updates and security templates have since been introduced.

Despite the limited coverage of the derived solutions, the knowledge presented throughout this thesis is believed to bring insight and value to the reverse-engineering community, as the first academic work to document the inner workings of various virtual machine architectures supported by Themida.

The methodologies presented in the previous chapters are furthermore applicable to other virtual machine architectures than those supported by leading software protection products such as Themida or VMProtect. For example, a paper[15] from 2018 by ESET employee, Filip Kafka, documents a custom virtual machine architecture used by the infamous FinFisher malware, also known as FinSpy, which has been the center of attention for large human rights organizations in recent times as it has supposedly been used by government agencies to spy on their citizens[1]. While extremely simple in comparison to the virtual machine architectures supported by Themida, there is a clear resemblance between the CISC virtual machine architecture and the custom FinFisher virtual machine architecture documented in the paper. In fact, they are so similar that the methodologies outlined in this paper are directly applicable to the FinFisher virtual machine architecture in order to recover protected procedures to their original machine code state.

[1]https://en.wikipedia.org/wiki/FinFisher

CHAPTER 7

Conclusion

The purpose of this thesis was to determine if it is possible to revert software virtualization technologies and restore protected procedures to their original machine code states. For this reason, two distinct analyses were carried out with the intention of answering the following research questions.

R1. *How can a virtualized procedure be recovered manually using dynamic methods?*

R2. *How can a virtualized procedure be recovered automatically using static methods?*

The underlying idea is that the existence of adequate answers to these questions is also an implicit answer to the overall problem definition that the thesis attempts to solve. In other words, if any one of the research questions listed above can be answered in full for any one of the analyses carried out throughout this thesis, then it can be concluded that it is possible to revert software virtualization technology.

Based on the herein documented analyses, the research indicates that it is very much possible to revert software virtualization technology and restore protected procedures to their original machine code states, as it has proven possible to create answers to both of the research questions listed above for both of the analyses carried out throughout this thesis.

The two analyses were carried out in an identical manner and the methodologies resulting from these analyses were devised on the basis of the following pre-emptively conjured methodology, which is assumed to be applicable for all similar analysis attempts.

1. Deduce the functional responsibility of each individual handler procedure.
2. Map the byte-code sequence to handler procedures (recreate the code-flow).
3. Translate the custom architecture code-flow back to its original architecture.

The developed methodologies are both efficient and effective, as they both operate at low execution times despites having made no efforts to optimize the involved routines and restore protected procedures to a machine code state that is identical to that of the same procedure prior to having software virtualization applied to it. However, the particularity of the virtual machine architectures targeted by the developed methodologies constitutes a high restriction on coverage. Fortunately, all modern virtual machine architectures supported by Themida are built on the same core engine, allowing for feasible extension of the developed methodologies to target everything currently supported by Themida.

Future work and research should therefore be focused on the expansion of coverage for the developed methodologies. In particular, the following issues should be addressed.

- Optimizations
  - Optimizations should be considered where applicable; there are a multitude of places to create optimizations that can speed up the entire process significantly.
- Deobfuscation
  - The deobfuscation engine used throughout these developed methodologies relies on monotone and predictable patterns used by Themida, and should instead be implemented as a generalized construct based on compiler-theory algorithms as depicted in section 2.2.
- Bitness
  - Themida offers both 32-bit and 64-bit variants of all supported virtual machine architectures. However, the developed methodologies are built solely around 32-bit architectures, and efforts should therefore be made to include support for 64-bit variants of all supported virtual machine architectures.
- Version
  - The developed methodologies currently support only the initial version of the FISH virtual machine architecture. Themida versions since 2013 should therefore be analysed incrementally and have their changes supported by the developed methodologies until coverage includes the newest versions of the virtual machine architectures.
- Core architectures
  - The developed methodologies should be expanded to include support for the TIGER and DOLPHIN virtual machine architectures in order to cover all core virtual machine architectures currently supported by Themida.
- Layered architectures
  - The developed methodologies should be expanded to include support for the layered PUMA, SHARK and EAGLE virtual machine architectures which are based on nested compositions of the core virtual machine architectures.

The list comprises all of the modifications that must be implemented to fully support the restoration of any procedure that has been protected by Themida using modern virtual machine architectures. However, the list would be far longer if the aim of the study was to support additional software virtualization products, such as VMProtect or Obsidium.

APPENDIX A

Themida CISC handlers

dispatcher

```
lodsb
sub al,bl
xor al,0x4d
sub al,0x40
sub bl,al
movzx eax,al
jmp dword ptr [edi+eax*4]
```

load imm8

```
lodsb
sub al,bl
xor al,0x36
xor al,0xd9
add bl,al
movzx eax,al
push ax
```

load imm16

```
lodsw
xor ax,bx
sub ax,0xc487
xor ax,0x5c00
add bx,ax
movzx eax,ax
push ax
```

load imm32

```
lodsd
xor eax,ebx
add eax,0x674cca9f
xor eax,0x70ef2f5
sub ebx,eax
push eax
```

load byte [imm32]

```
lodsd
sub eax,ebx
add eax,0x5f5e9239
sub eax,0x33324026
add ebx,eax
movzx ax,byte ptr [eax]
push ax
```

load word [imm32]

```
lodsd
xor eax,ebx
sub eax,0x29493e59
xor eax,0x513cf2fd
add ebx,eax
push word ptr [eax]
```

load dword [imm32]

```
lodsd
add eax,ebx
add eax,0x3b673ccb
add eax,0x5bfee4ee
xor ebx,eax
push dword ptr [eax]
```

load reg32

```
lodsb
sub al,bl
sub al,0x2f
xor al,0x4e
xor bl,al
movzx eax,al
push dword ptr [edi+eax*4]
```

load byte [reg32]

```
lodsb
xor al,bl
sub al,0xf7
add al,0x84
xor bl,al
movzx eax,al
mov eax,dword ptr [edi+eax*4]
movzx ax,byte ptr [eax]
push ax
```

load word [reg32]

```
lodsb
xor al,bl
sub al,0xc4
xor al,0xa7
add bl,al
movzx eax,al
mov eax,dword ptr [edi+eax*4]
push word ptr [eax]
```

load offset reg32

```
lodsb
add al,bl
xor al,0x59
sub al,0x28
sub bl,al
movzx eax,al
lea eax,dword ptr [edi+eax*4]
push eax
```

load addr

```
push edx
```

load byte [addr]

```
movzx ax,byte ptr [edx]
push ax
```

load word [addr]

```
push word ptr [edx]
```

load dword [addr]

```
push dword ptr [edx]
```

load byte fs:[addr]

```
movzx ax,byte ptr fs:[edx]
push ax
```

load word fs:[addr]

```
mov ax,word ptr fs:[edx]
push ax
```

load dword fs:[addr]

```
push dword ptr fs:[edx]
```

load stack16

```
push sp
```

load stack32

```
push esp
```

load param

```
push dword ptr [edi+0x60]
```

load delta

```
push dword ptr [edi+0x64]
```

load align

```
lodsd
sub eax,ebx
xor eax,0x7ba62371
sub eax,0x1ca98778
add ebx,eax
add eax,dword ptr [edi+0x64]
push eax
```

store byte [imm32]

```
lodsd
add eax,ebx
sub eax,0x7be0d78f
add eax,0x44a0917e
sub ebx,eax
pop dx
mov byte ptr [eax],dl
```

store word [imm32]

```
lodsd
xor eax,ebx
add eax,0x3b1c7265
add eax,0x3972e026
sub ebx,eax
pop word ptr [eax]
```

store dword [imm32]

```
lodsd
sub eax,ebx
sub eax,0x5a66f329
xor eax,0x4e547aaf
add ebx,eax
pop dword ptr [eax]
```

store reg8

```
lodsb
add al,bl
sub al,0xf7
xor al,0x0
sub bl,al
movzx eax,al
pop dx
mov byte ptr [edi+eax*4],dl
```

store reg8 high

```
lodsb
xor al,bl
add al,0x89
sub al,0x9d
xor bl,al
movzx eax,al
pop dx
mov byte ptr [edi+eax*4+0x1],dl
```

store reg16

```
lodsb
sub al,bl
sub al,0x41
xor al,0x18
sub bl,al
movzx eax,al
pop word ptr [edi+eax*4]
```

store reg32

```
lodsb
add al,bl
xor al,0x66
xor al,0xaa
xor bl,al
movzx eax,al
pop dword ptr [edi+eax*4]
```

store addr

```
pop edx
```

store byte [addr]

```
pop ax
mov byte ptr [edx],al
```

store word [addr]

```
pop word ptr [edx]
```

store dword [addr]

```
pop dword ptr [edx]
```

store byte fs:[addr]

```
pop ax
mov byte ptr fs:[edx],al
```

store word fs:[addr]

```
pop ax
mov word ptr fs:[edx],ax
```

store dword fs:[addr]

```
pop dword ptr fs:[edx]
```

store stack16

```
pop sp
```

store stack32

```
pop esp
```

store param

```
pop dword ptr [edi+0x60]
```

store eflags

```
pop dword ptr [edi+0x1c]
```

inc byte

```
pop ax
inc byte ptr [esp]
pushfd
```

inc word

```
pop ax
inc word ptr [esp]
pushfd
```

inc dword

```
pop eax
inc dword ptr [esp]
pushfd
```

dec byte

```
pop ax
dec byte ptr [esp]
pushfd
```

dec word

```
pop ax
dec word ptr [esp]
pushfd
```

dec dword

```
pop eax
dec dword ptr [esp]
pushfd
```

neg byte

```
neg byte ptr [esp]
pushfd
```

neg word

```
neg word ptr [esp]
pushfd
```

neg dword

```
neg dword ptr [esp]
pushfd
```

not byte

```
not byte ptr [esp]
```

not word

```
not word ptr [esp]
```

not dword

```
not dword ptr [esp]
```

add byte

```
pop ax
add byte ptr [esp],al
pushfd
```

add word

```
pop ax
add word ptr [esp],ax
pushfd
```

add dword

```
pop eax
add dword ptr [esp],eax
pushfd
```

addx dword

```
pop eax
add dword ptr [esp],eax
```

adc byte

```
push dword ptr [edi+0x1c]
popfd
pop ax
adc byte ptr [esp],al
pushfd
```

adc word

```
push dword ptr [edi+0x1c]
popfd
pop ax
adc word ptr [esp],ax
pushfd
```

adc dword

```
push dword ptr [edi+0x1c]
popfd
pop eax
adc dword ptr [esp],eax
pushfd
```

sub byte

```
pop ax
sub byte ptr [esp],al
pushfd
```

sub word

```
pop ax
sub word ptr [esp],ax
pushfd
```

sub dword

```
pop eax
sub dword ptr [esp],eax
pushfd
```

subx dword

```
pop eax
sub dword ptr [esp],eax
```

sbb byte

```
push dword ptr [edi+0x1c]
popfd
pop ax
sbb byte ptr [esp],al
pushfd
```

sbb word

```
push dword ptr [edi+0x1c]
popfd
pop ax
sbb word ptr [esp],ax
pushfd
```

sbb dword

```
push dword ptr [edi+0x1c]
popfd
pop eax
sbb dword ptr [esp],eax
pushfd
```

mul byte

```
pop cx
pop ax
mul cl
movzx cx,ah
push cx
movzx cx,al
push cx
pushfd
```

mul word

```
pop cx
pop ax
mul cx
push dx
push ax
pushfd
```

mul dword

```
pop ecx
pop eax
mul ecx
push edx
push eax
pushfd
```

imul byte

```
pop cx
pop ax
imul cl
movzx cx,ah
push cx
movzx cx,al
push cx
pushfd
```

imul word

```
pop cx
pop ax
imul cx
push dx
push ax
pushfd
```

imul dword

```
pop ecx
pop eax
imul ecx
push edx
push eax
pushfd
```

imulc word

```
pop ax
pop cx
imul cx,ax
push cx
pushfd
```

imulc dword

```
pop eax
pop ecx
imul ecx,eax
push ecx
pushfd
```

div byte

```
pop cx
pop ax
div cl
movzx cx,ah
push cx
movzx cx,al
push cx
pushfd
```

div word

```
pop cx
pop ax
pop ax
div cx
push dx
push ax
pushfd
```

div dword

```
pop ecx
pop eax
pop edx
div ecx
push edx
push eax
pushfd
```

idiv byte

```
pop cx
pop ax
idiv cl
movzx cx,ah
push cx
movzx cx,al
push cx
pushfd
```

idiv word

```
pop cx
pop ax
pop dx
idiv cx
push dx
push ax
pushfd
```

idiv dword

```
pop ecx
pop eax
pop edx
idiv ecx
push edx
push eax
pushfd
```

and byte

```
pop ax
and byte ptr [esp],al
pushfd
```

and word

```
pop ax
and word ptr [esp],ax
pushfd
```

and dword

```
pop eax
and dword ptr [esp],eax
pushfd
```

or byte

```
pop ax
or byte ptr [esp],al
pushfd
```

or word

```
pop ax
or word ptr [esp],ax
pushfd
```

or dword

```
pop eax
or dword ptr [esp],eax
pushfd
```

xor byte

```
pop ax
xor byte ptr [esp],al
pushfd
```

xor word

```
pop ax
xor word ptr [esp],ax
pushfd
```

xor dword

```
pop eax
xor dword ptr [esp],eax
pushfd
```

xorx dword

```
pop eax
xor dword ptr [esp],eax
```

shr byte

```
pop cx
shr byte ptr [esp],cl
pushfd
```

shr word

```
pop cx
shr word ptr [esp],cl
pushfd
```

shr dword

```
pop cx
shr dword ptr [esp],cl
pushfd
```

sar byte

```
pop cx
sar byte ptr [esp],cl
pushfd
```

sar word

```
pop cx
sar word ptr [esp],cl
pushfd
```

sar dword

```
pop cx
sar dword ptr [esp],cl
pushfd
```

shl/sal byte

```
pop cx
shl byte ptr [esp],cl
pushfd
```

shl/sal word

```
pop cx
shl word ptr [esp],cl
pushfd
```

shl/sal dword

```
pop cx
shl dword ptr [esp],cl
pushfd
```

shlx dword

```
pop ecx
shl dword ptr [esp],cl
```

ror byte

```
pop cx
ror byte ptr [esp],cl
pushfd
```

ror word

```
pop cx
ror word ptr [esp],cl
pushfd
```

ror dword

```
pop cx
ror dword ptr [esp],cl
pushfd
```

rcr byte

```
push dword ptr [edi+0x1c]
popfd
pop cx
rcr byte ptr [esp],cl
pushfd
```

rcr word

```
push dword ptr [edi+0x1c]
popfd
pop cx
rcr word ptr [esp],cl
pushfd
```

rcr dword

```
push dword ptr [edi+0x1c]
popfd
pop cx
rcr dword ptr [esp],cl
pushfd
```

rol byte

```
pop cx
rol byte ptr [esp],cl
pushfd
```

rol word

```
pop cx
rol word ptr [esp],cl
pushfd
```

rol dword

```
pop cx
rol dword ptr [esp],cl
pushfd
```

rcl byte

```
push dword ptr [edi+0x1c]
popfd
pop cx
rcl byte ptr [esp],cl
pushfd
```

rcl word

```
push dword ptr [edi+0x1c]
popfd
pop cx
rcl word ptr [esp],cl
pushfd
```

rcl dword

```
push dword ptr [edi+0x1c]
popfd
pop cx
rcl dword ptr [esp],cl
pushfd
```

cmp byte

```
pop ax
pop cx
cmp cl,al
pushfd
```

cmp word

```
pop ax
pop cx
cmp cx,ax
pushfd
```

cmp dword

```
pop eax
pop ecx
cmp ecx,eax
pushfd
```

test byte

```
pop ax
pop cx
test al,cl
pushfd
```

test word

```
pop ax
pop cx
test ax,cx
pushfd
```

test dword

```
pop eax
pop ecx
test eax,ecx
pushfd
```

bt word

```
pop ax
bt word ptr [esp],ax
pushfd
```

bt dword

```
pop eax
bt dword ptr [esp],eax
pushfd
```

btc word

```
pop ax
btc word ptr [esp],ax
pushfd
```

btr word

```
pop ax
btr word ptr [esp],ax
pushfd
```

btr dword

```
pop eax
btr dword ptr [esp],eax
pushfd
```

bts word

```
pop ax
bts word ptr [esp],ax
pushfd
```

bts dword

```
pop eax
bts dword ptr [esp],eax
pushfd
```

bswap dword

```
pop eax
bswap eax
push eax
```

movsxb word

```
pop cx
pop ax
movsx cx,al
push cx
```

movsxb dword

```
pop ecx
pop eax
movsx ecx,al
push ecx
```

movsxw dword

```
pop ecx
pop eax
movsx ecx,ax
push ecx
```

movzxb word

```
pop cx
pop ax
movzx cx,al
push cx
```

movzxb dword

```
pop ecx
pop eax
movzx ecx,al
push ecx
```

movzxw dword

```
pop ecx
pop eax
movzx ecx,ax
push ecx
```

readptr dword

```
pop eax
push dword ptr [eax]
```

writeptr dword

```
pop eax
pop ecx
mov dword ptr [eax],ecx
```

clc

```
and dword ptr [edi+0x1c],0xfffffffe
```

cmc

```
mov eax,dword ptr [edi+0x1c]
and eax,0x1
or eax,eax
jz _skip
and dword ptr [edi+0x1c],0xfffffffe
; _skip:
mov ebx,ebx
```

cld

```
mov dword ptr [edi+0x74],0x0
and dword ptr [edi+0x1c],0xfffffbff
```

stc

```
or dword ptr [edi+0x1c],0x1
```

std

```
mov dword ptr [edi+0x74],0x1
or dword ptr [edi+0x1c],0x400
```

sti

```
or dword ptr [edi+0x1c],0x200
```

move addr, imm32

```
lodsd
sub eax,ebx
xor eax,0x6c2c03b6
xor eax,0x25ab2676
add ebx,eax
mov edx,eax
```

move addr, stack32

```
mov edx,esp
```

add addr, imm32

```
lodsd
xor eax,ebx
xor eax,0x10e18bd7
sub eax,0x651e460e
add ebx,eax
add edx,eax
```

add addr, reg32

```
lodsb
sub al,bl
xor al,0xaf
xor al,0x7a
sub bl,al
movzx eax,al
cmp eax,0x7
jz _stack
mov eax,dword ptr [edi+eax*4]
jmp _continue
; _stack:
mov eax,esp
; _continue:
add edx,eax
```

add addr, reloc

```
mov eax,dword ptr [edi+0x7c]
add edx,eax
```

add [stack32], reloc

```
mov eax,dword ptr [edi+0x7c]
add dword ptr [esp],eax
```

sub addr, imm32

```
lodsd
add eax,ebx
sub eax,0x711a9801
add eax,0x596ec402
sub ebx,eax
sub edx,eax
```

xor addr, imm32

```
lodsd
add eax,ebx
add eax,0xc13d6ff
xor eax,0x4873a200
xor ebx,eax
xor edx,eax
```

xchg [stack32], addr

```
pop eax
push edx
mov edx,eax
```

set param

```
mov eax,dword ptr [esp+0x4]
mov dword ptr [edi+0x60],eax
pop eax
add esp,0x4
push eax
```

set counter offset

```
lodsb
sub al,bl
xor al,0x37
add al,0xf6
xor bl,al
mov byte ptr [edi+0x68],al
```

set stack offset

```
lodsb
sub al,bl
xor al,0xdd
add al,0xb1
xor bl,al
mov byte ptr [edi+0x70],al
```

reset key

```
mov ebx,0x0
```

ret

```
mov ecx,dword ptr [edi+0x70]
mov edx,edi
or ecx,ecx
jz _skip_stack
lea esi,[esp+0x24]
lea edi,[esi+ecx]
std
mov ecx,0xa
rep movsd
add esp,dword ptr [edx+0x70]
mov dword ptr [edx+0x70],0x0
; _skip_stack:
cmp dword ptr [edx+0x74],0x0
jz _skip_df
or dword ptr [esp+0x20],0x400
mov dword ptr [edx+0x74],0x0
; _skip_df:
mov dword ptr [edx+0x88],0x0
popad
popfd
retn
```

jmp imm32

```
lodsd
add esi,eax
mov ebx,0x0
```

jcc imm32

```
lodsd
cmp dword ptr [edi+0x8c],0x0
jz _skip
add esi,eax
mov ebx,0x0
; _skip:
mov eax,eax
```

jcc evaluate

```
lodsb
add eax,ebx
sub eax,0x67db6747
xor eax,0x74136396
add ebx,eax
and al,0x7f
push ebx
mov ebx,eax
mov dword ptr [edi+0x8c],0x1
mov dword ptr [edi+0x78],0x0
xor edx,edx
mov eax,ebx
and eax,0x200
mov ecx,dword ptr [edi+0x1c]
and ecx,0x1 ; cf
shr ecx,0x0
or eax,eax
jz _skip_cf
mov eax,ebx
and eax,0x100
shr eax,0x8
xor eax,ecx
not eax
and eax,0x1
or edx,eax
shl edx,0x1
add dword ptr [edi+0x78],0x1
; _skip_cf:
mov eax,ebx
and eax,0x800
or eax,eax
jz _skip_zf
mov ecx,dword ptr [edi+0x1c]
and ecx,0x40 ; zf
shr ecx,0x6
mov eax,ebx
and eax,0x400
shr eax,0xa
xor eax,ecx
not eax
and eax,0x1
or edx,eax
shl edx,0x1
add dword ptr [edi+0x78],0x1
; _skip_zf:
mov eax,ebx
and eax,0x2000000
or eax,eax
jz _skip_df
mov ecx,dword ptr [edi+0x74]
mov eax,ebx
and eax,0x1000000
shr eax,0x18
xor eax,ecx
not eax
and eax,0x1
or edx,eax
shl edx,0x1
add dword ptr [edi+0x78],0x1
; _skip_df:
mov eax,ebx
and eax,0x2000
or eax,eax
jz _skip_sf
mov ecx,dword ptr [edi+0x1c]
and ecx,0x80 ; sf
shr ecx,0x7
mov eax,ebx
and eax,0x1000
shr eax,0xc
xor eax,ecx
not eax
and eax,0x1
or edx,eax
shl edx,0x1
add dword ptr [edi+0x78],0x1
; _skip_sf:
mov eax,ebx
and eax,0x8000
or eax,eax
jz _skip_of
mov ecx,dword ptr [edi+0x1c]
and ecx,0x800 ; of
shr ecx,0xb
mov eax,ebx
and eax,0x4000
shr eax,0xe
xor eax,ecx
not eax
and eax,0x1
or edx,eax
shl edx,0x1
add dword ptr [edi+0x78],0x1
; _skip_of:
mov eax,ebx
and eax,0x20000
or eax,eax
jz _skip_pf
mov ecx,dword ptr [edi+0x1c]
and ecx,0x4 ; pf
shr ecx,0x2
mov eax,ebx
and eax,0x10000
shr eax,0x10
xor eax,ecx
not eax
and eax,0x1
or edx,eax
shl edx,0x1
add dword ptr [edi+0x78],0x1
```

```
; _skip_pf:
mov eax,ebx
and eax,0x80000
or eax,eax
jz _skip_sf_of
mov ecx,dword ptr [edi+0x1c]
and ecx,0x80 ; sf
shr ecx,0x7
mov eax,dword ptr [edi+0x1c]
and eax,0x800 ; of
shr eax,0xb
xor ecx,eax
mov eax,ebx
and eax,0x40000
shr eax,0x12
xor eax,ecx
not eax
and eax,0x1
or edx,eax
shl edx,0x1
add dword ptr [edi+0x78],0x1
; _skip_sf_of:
mov eax,ebx
and eax,0x200000
or eax,eax
jz _skip_cx
mov eax,dword ptr [edi+0x68]
mov eax,dword ptr [edi+eax*4]
and eax,0xffff
or eax,eax
jnz _skip_cx
mov edx,0x1
; _skip_cx:
mov eax,ebx
and eax,0x800000
or eax,eax
jz _skip_ecx
mov eax,dword ptr [edi+0x68]
mov eax,dword ptr [edi+eax*4]
or eax,eax
jnz _skip_ecx
mov edx,0x1
; _skip_ecx:
mov ecx,dword ptr [edi+0x78]
mov eax,0x1
shl eax,cl
add eax,0xffffffff
and ebx,0x10
or ebx,ebx
jnz _evaluate
mov dword ptr [edi+0x8c],edx
jmp _continue
; _evaluate:
shr edx,0x1
cmp eax,edx
jz _continue
mov dword ptr [edi+0x8c],0x0
; _continue:
pop ebx
```

APPENDIX B

Themida FISH handlers

reset

```
mov [ebp+0x53],0x00000000
mov [ebp+0xa7],0x00000000
mov [ebp+0x89],0x00000000
mov [ebp+0x14],0x00000000
mov [ebp+0x8f],0x00000000
mov [ebp+0xa0],0x0000
mov [ebp+0x57],0x00
mov [ebp+0x68],0x00000000
```

load stack

```
mov ebx,[ebp+0x08]
movzx eax,word ptr [ebx+0x00]
mov [ebp+eax],esp
```

store stack

```
mov ebx,[ebp+0x08]
movzx eax,word ptr [ebx+0x00]
mov esp,[ebp+eax]
```

add stack,imm8

```
mov ebx,[ebp+0x08]
movzx eax,byte ptr [ebx+0x00]
add esp,eax
movzx edx,word ptr [ebx+0x01]
add [ebp+edx],eax
```

pushfd

```
mov ebx,[ebp+0x08]
movzx eax,word ptr [ebx+0x00]
sub dword ptr [ebp+eax],0x04
movzx eax,word ptr [ebx+0x02]
push dword ptr [ebp+eax]
```

popfd

```
mov ebx,[ebp+0x08]
movzx eax,word ptr [ebx+0x00]
add dword ptr [ebp+eax],0x04
movzx eax,word ptr [ebx+0x02]
pop dword ptr [ebp+eax]
```

crypt

```
mov ebx,[ebp+0x08]
movzx eax,word ptr [ebx+0x00]
mov edx,[ebp+0x2a]
mov [ebp+eax],edx
```

eflags operation

```
mov ebx,[ebp+0x08]
movzx esi,word ptr [ebx+0x00]
mov dl,[ebx+0x02]
cmp dl,0xbb ; clc
jnz _skip_clc
and [ebp+esi],0xfffffffe ; NOT(0x01)
; _skip_clc:
cmp dl,0xc8 ; cld
jnz _skip_cld
and [ebp+esi],0xfffffbff ; NOT(0x400)
; _skip_cld:
cmp dl,0x33 ; cli
jnz _skip_cli
and [ebp+esi],0xfffffdff ; NOT(0x200)
; _skip_cli:
cmp dl,0x95 ; cmc
jnz _skip_cmc
test [ebp+esi],0x01
jz _set_cmc
and [ebp+esi],0xfffffffe ; NOT(0x01)
jmp _skip_cmc
; _set_cmc:
or [ebp+esi],0x01
; _skip_cmc:
cmp dl,0x11 ; stc
jnz _skip_stc
or [ebp+esi],0x01
; _skip_stc:
cmp dl,0x7b ; std
jnz _skip_std
or [ebp+esi],0x400
; _skip_std:
cmp dl,0x1c ; sti
jnz _skip_sti
or [ebp+esi],0x200
; _skip_sti:
```

lods byte

```
mov ebx,[ebp+0x08]
movzx ecx,word ptr [ebx+0x00]
movzx edx,word ptr [ebx+0x02]
mov esi,[ebp+ecx]
mov al,[esi]
mov [ebp+edx],al
movzx eax,word ptr [ebx+0x04]
test [ebp+eax],0x400
jz _forwards
sub dword ptr [ebp+ecx],0x01
jmp _finish
; _forwards:
add dword ptr [ebp+ecx],0x01
; _finish:
```

stos byte

```
mov ebx,[ebp+0x08]
movzx ecx,word ptr [ebx+0x00]
movzx edx,word ptr [ebx+0x02]
mov edi,[ebp+ecx]
mov eax,[ebp+edx]
mov [edi],al
movzx eax,word ptr [ebx+0x04]
test [ebp+eax],0x400
jz _forwards
sub dword ptr [ebp+ecx],0x01
jmp _finish
; _forwards:
add dword ptr [ebp+ecx],0x01
; _finish:
```

lods word

```
mov ebx,[ebp+0x08]
movzx ecx,word ptr [ebx+0x00]
movzx edx,word ptr [ebx+0x02]
mov esi,[ebp+ecx]
mov ax,[esi]
mov [ebp+edx],ax
movzx eax,word ptr [ebx+0x04]
test [ebp+eax],0x400
jz _forwards
sub dword ptr [ebp+ecx],0x02
jmp _finish
; _forwards:
add dword ptr [ebp+ecx],0x02
; _finish:
```

stos word

```
mov ebx,[ebp+0x08]
movzx ecx,word ptr [ebx+0x00]
movzx edx,word ptr [ebx+0x02]
mov edi,[ebp+ecx]
mov eax,[ebp+edx]
mov [edi],ax
movzx eax,word ptr [ebx+0x04]
test [ebp+eax],0x400
jz _forwards
sub dword ptr [ebp+ecx],0x02
jmp _finish
; _forwards:
add dword ptr [ebp+ecx],0x02
; _finish:
```

lods dword

```
mov ebx,[ebp+0x08]
movzx ecx,word ptr [ebx+0x00]
movzx edx,word ptr [ebx+0x02]
mov esi,[ebp+ecx]
mov eax,[esi]
mov [ebp+edx],eax
movzx eax,word ptr [ebx+0x04]
test [ebp+eax],0x400
jz _forwards
sub dword ptr [ebp+ecx],0x04
jmp _finish
; _forwards:
add dword ptr [ebp+ecx],0x04
; _finish:
```

stos dword

```
mov ebx,[ebp+0x08]
movzx ecx,word ptr [ebx+0x00]
movzx edx,word ptr [ebx+0x02]
mov edi,[ebp+ecx]
mov eax,[ebp+edx]
mov [edi],eax
movzx eax,word ptr [ebx+0x04]
test [ebp+eax],0x400
jz _forwards
sub dword ptr [ebp+ecx],0x04
jmp _finish
; _forwards:
add dword ptr [ebp+ecx],0x04
; _finish:
```

cmps byte

```
mov ebx,[ebp+0x08]
movzx ecx,word ptr [ebx+0x00]
movzx edx,word ptr [ebx+0x02]
mov esi,[ebp+ecx]
mov edi,[ebp+edx]
mov al,[esi]
cmp [edi],al
pushfd
movzx eax,word ptr [ebx+0x04]
test [ebp+eax],0x400
jz _forwards
sub dword ptr [ebp+ecx],0x01
sub dword ptr [ebp+edx],0x01
jmp _finish
; _forwards:
add dword ptr [ebp+ecx],0x01
add dword ptr [ebp+edx],0x01
; _finish:
pop esi
cmp byte ptr [ebx+0x06],0x00
jz _skip_flags
movzx eax,word ptr [ebx+0x07]
mov [ebp+eax],esi
; _skip_flags:
```

cmps dword

```
mov ebx,[ebp+0x08]
movzx ecx,word ptr [ebx+0x00]
movzx edx,word ptr [ebx+0x02]
mov esi,[ebp+ecx]
mov edi,[ebp+edx]
mov eax,[esi]
cmp [edi],eax
pushfd
movzx eax,word ptr [ebx+0x04]
test [ebp+eax],0x400
jz _forwards
sub dword ptr [ebp+ecx],0x04
sub dword ptr [ebp+edx],0x04
jmp _finish
; _forwards:
add dword ptr [ebp+ecx],0x04
add dword ptr [ebp+edx],0x04
; _finish:
pop esi
cmp byte ptr [ebx+0x06],0x00
jz _skip_flags
movzx eax,word ptr [ebx+0x07]
mov [ebp+eax],esi
; _skip_flags:
```

cmps word

```
mov ebx,[ebp+0x08]
movzx ecx,word ptr [ebx+0x00]
movzx edx,word ptr [ebx+0x02]
mov esi,[ebp+ecx]
mov edi,[ebp+edx]
mov ax,[esi]
cmp [edi],ax
pushfd
movzx eax,word ptr [ebx+0x04]
test [ebp+eax],0x400
jz _forwards
sub dword ptr [ebp+ecx],0x02
sub dword ptr [ebp+edx],0x02
jmp _finish
; _forwards:
add dword ptr [ebp+ecx],0x02
add dword ptr [ebp+edx],0x02
; _finish:
pop esi
cmp byte ptr [ebx+0x06],0x00
jz _skip_flags
movzx eax,word ptr [ebx+0x07]
mov [ebp+eax],esi
; _skip_flags:
```

subs byte

```
mov ebx,[ebp+0x08]
movzx ecx,word ptr [ebx+0x00]
movzx edx,word ptr [ebx+0x02]
mov esi,[ebp+ecx]
mov edi,[ebp+edx]
mov al,[esi]
sub [edi],al
pushfd
movzx eax,word ptr [ebx+0x04]
test [ebp+eax],0x400
jz _forwards
sub dword ptr [ebp+ecx],0x01
sub dword ptr [ebp+edx],0x01
jmp _finish
; _forwards:
add dword ptr [ebp+ecx],0x01
add dword ptr [ebp+edx],0x01
; _finish:
pop esi
cmp byte ptr [ebx+0x06],0x00
jz _skip_flags
movzx eax,word ptr [ebx+0x07]
mov [ebp+eax],esi
; _skip_flags:
```

```
subs word
```

```
mov ebx,[ebp+0x08]
movzx ecx,word ptr [ebx+0x00]
movzx edx,word ptr [ebx+0x02]
mov esi,[ebp+ecx]
mov edi,[ebp+edx]
mov ax,[esi]
sub [edi],ax
pushfd
movzx eax,word ptr [ebx+0x04]
test [ebp+eax],0x400
jz _forwards
sub dword ptr [ebp+ecx],0x02
sub dword ptr [ebp+edx],0x02
jmp _finish
; _forwards:
add dword ptr [ebp+ecx],0x02
add dword ptr [ebp+edx],0x02
; _finish:
pop esi
cmp byte ptr [ebx+0x06],0x00
jz _skip_flags
movzx eax,word ptr [ebx+0x07]
mov [ebp+eax],esi
; _skip_flags:
```

```
subs dword
```

```
mov ebx,[ebp+0x08]
movzx ecx,word ptr [ebx+0x00]
movzx edx,word ptr [ebx+0x02]
mov esi,[ebp+ecx]
mov edi,[ebp+edx]
mov eax,[esi]
sub [edi],eax
pushfd
movzx eax,word ptr [ebx+0x04]
test [ebp+eax],0x400
jz _forwards
sub dword ptr [ebp+ecx],0x04
sub dword ptr [ebp+edx],0x04
jmp _finish
; _forwards:
add dword ptr [ebp+ecx],0x04
add dword ptr [ebp+edx],0x04
; _finish:
pop esi
cmp byte ptr [ebx+0x06],0x00
jz _skip_flags
movzx eax,word ptr [ebx+0x07]
mov [ebp+eax],esi
; _skip_flags:
```

```
movs byte
```

```
mov ebx,[ebp+0x08]
movzx ecx,word ptr [ebx+0x00]
movzx edx,word ptr [ebx+0x02]
mov esi,[ebp+ecx]
mov edi,[ebp+edx]
mov al,[esi]
mov [edi],al
movzx eax,word ptr [ebx+0x04]
test [ebp+eax],0x400
jz _forwards
sub dword ptr [ebp+ecx],0x01
sub dword ptr [ebp+edx],0x01
jmp _finish
; _forwards:
add dword ptr [ebp+ecx],0x01
add dword ptr [ebp+edx],0x01
; _finish:
```

```
movs word
```

```
mov ebx,[ebp+0x08]
movzx ecx,word ptr [ebx+0x00]
movzx edx,word ptr [ebx+0x02]
mov esi,[ebp+ecx]
mov edi,[ebp+edx]
mov ax,[esi]
mov [edi],ax
movzx eax,word ptr [ebx+0x04]
test [ebp+eax],0x400
jz _forwards
sub dword ptr [ebp+ecx],0x02
sub dword ptr [ebp+edx],0x02
jmp _finish
; _forwards:
add dword ptr [ebp+ecx],0x02
add dword ptr [ebp+edx],0x02
; _finish:
```

```
movs dword
```

```
mov ebx,[ebp+0x08]
movzx ecx,[ebx+0x00]
movzx edx,[ebx+0x02]
mov esi,[ebp+ecx]
mov edi,[ebp+edx]
mov eax,[esi]
mov [edi],eax
movzx eax,word ptr [ebx+0x04]
test [ebp+eax],0x400
jz _forwards
sub dword ptr [ebp+ecx],0x04
sub dword ptr [ebp+edx],0x04
jmp _finish
; _forwards:
add dword ptr [ebp+ecx],0x04
add dword ptr [ebp+edx],0x04
; _finish:
```

jmp internal imm32

```
mov ebx,[ebp+0x08]
mov eax,[ebx+0x00]
test eax,0x80000000
jz _forwards
and eax,0x7fffffff
sub [ebp+0x08],eax
jmp _finish
; _forwards:
add [ebp+0x08],eax
; _finish:
```

jmp external imm32

```
mov ebx,[ebp+0x08]
mov ecx,[ebx+0x00]
add ecx,[ebp+0x75] ; image base
movzx eax,word ptr [ebx+0x04]
mov [esp+eax],ecx
mov [ebp+0x36],0x00000000
popad
popfd
ret
```

jmp external reg

```
mov ebx,[ebp+0x08]
movzx eax,word ptr [ebx+0x00]
mov ecx,[ebp+eax]
movzx eax,word ptr [ebx+0x02]
mov [esp+eax],ecx
mov [ebp+0x36],0x00000000
popad
popfd
ret
```

jmp external [reg]

```
mov ebx,[ebp+0x08]
movzx eax,word ptr [ebx+0x00]
mov ecx,[ebp+eax]
mov ecx,[ecx]
movzx eax,word ptr [ebx+0x02]
mov [esp+eax],ecx
mov [ebp+0x36],0x00000000
popad
popfd
ret
```

call

```
mov ebx,[ebp+0x08]
cmp byte ptr [ebx+0x00],0x11 ; imm32
jnz _skip_immediate
mov edx,[ebx+0x01]
add edx,[ebp+0x75] ; image base
movzx ecx,word ptr [ebx+0x05]
mov [esp+ecx],edx
; _skip_immediate:
cmp byte ptr [ebx+0x00],0x22 ; reg
jz _resolve_register
cmp byte ptr [ebx+0x00],0x33 ; mem
jnz _skip_register
; _resolve_register:
movzx edx,word ptr [ebx+0x01]
mov edx,[ebp+edx]
cmp byte ptr [ebx+0x00],0x33 ; mem
jnz _skip_memory
mov edx,[edx]
; _skip_memory:
movzx ecx,word ptr [ebx+0x03]
mov [esp+ecx],edx
; _skip_register
mov edx,[ebx+0x07]
add edx,[ebp+0x75] ; image base
mov [esp+ecx+0x04],edx
mov [ebp+0x36],0x00000000
popad
popfd
ret
```

ret

```
mov ebx,[ebp+0x08]
mov edx,[ebx+0x00]
lea esi,[esp+0x28]
lea edi,[esi+edx]
std
mov ecx,edx
test ecx,ecx
jz _skip_default
mov ecx,0x0a
; _skip_default:
rep movsd
add esp,edx
mov [ebp+0x36],0x00000000
popad
popfd
add esp,0x04
ret
```

jmp external imm32 dll

```
mov ebx,[ebp+0x08]
mov eax,[ebx+0x00]
add eax,[ebp+0x75] ; image base
cmp byte ptr [ebx+0x04],0x01
jz _realign
cmp byte ptr [ebx+0x04],0x02
jz _realign
jmp _skip
; _realign:
mov ecx,[ebx+0x05]
add ecx,[ebp+0x75]
movzx edx,byte ptr [ebx+0x09]
cmp [eax+edx],ecx
jz _skip
mov esi,[ebp+0x3f]
sub [eax+edx],esi
mov esi,[ebp+0x75]
add [eax+edx],esi
cmp byte ptr [ebx+0x04],0x02
jnz _skip
movzx edx,byte ptr [ebx+0x0a]
mov esi,[ebp+0x3f]
sub [eax+edx],esi
mov esi,[ebp+0x75]
add [eax+edx],esi
; _skip:
movzx edx,word ptr [ebx+0x0b]
mov [esp+edx],eax
mov [ebp+0x36],0x00000000
popad
popfd
ret
```

jcc internal imm32

```
mov ebx,[ebp+0x08]
movzx eax,word ptr [ebx+0x08]
mov eax,[ebp+eax]
mov byte ptr [ebp+0x97],0x00
mov ecx,eax
and ecx,0x80 ; sf
shr ecx,0x07
mov edx,eax
and edx,0x800 ; of
shr edx,0x0b
cmp byte ptr [ebx+0x0a],0x00 ; jz
jnz _skip_jz
test eax,0x40 ; zf = 1
jz _skip_jz
mov byte ptr [ebp+0x97],0x01
; _skip_jz:
cmp byte ptr [ebx+0x0a],0x11 ; jnz
jnz _skip_jnz
test eax,0x40 ; zf = 0
jnz _skip_jnz
mov byte ptr [ebp+0x97],0x01
; _skip_jnz:
cmp byte ptr [ebx+0x0a],0x22 ; ja
jnz _skip_ja
test eax,0x40 ; zf = 0
jnz _skip_ja
test eax,0x01 ; cf = 0
jnz _skip_ja
mov byte ptr [ebp+0x97],0x01
; _skip_ja:
cmp byte ptr [ebx+0x0a],0x33 ; jae
jnz _skip_jae
test eax,0x01 ; cf = 0
jnz _skip_jae
mov byte ptr [ebp+0x97],0x01
; _skip_jae:
cmp byte ptr [ebx+0x0a],0x44 ; jb
jnz _skip_jb
test eax,0x01 ; cf = 1
jz _skip_jb
mov byte ptr [ebp+0x97],0x01
; _skip_jb:
cmp byte ptr [ebx+0x0a],0x55 ; jbe
jnz _skip_jbe
test eax,0x40 ; zf = 1
jz _skip_jbe_zf
mov byte ptr [ebp+0x97],0x01
; _skip_jbe_zf:
test eax,0x01 ; cf = 1
jz _skip_jbe
mov byte ptr [ebp+0x97],0x01
; _skip_jbe:
cmp byte ptr [ebx+0x0a],0x66 ; jg
jnz _skip_jg
test eax,0x40 ; zf = 0
jnz _skip_jg
cmp ecx,edx ; sf == of
jnz _skip_jg
mov byte ptr [ebp+0x97],0x01
; _skip_jg:
cmp byte ptr [ebx+0x0a],0x77 ; jge
jnz _skip_jge
cmp ecx,edx ; sf == of
jnz _skip_jge
mov byte ptr [ebp+0x97],0x01
; _skip_jge:
cmp byte ptr [ebx+0x0a],0x88 ; jl
jnz _skip_jl
cmp ecx,edx ; sf != of
jz _skip_jl
mov byte ptr [ebp+0x97],0x01
; _skip_jl:
cmp byte ptr [ebx+0x0a],0x99 ; jle
jnz _skip_jle
test eax,0x40 ; zf = 1
jz _skip_jle_zf
mov byte ptr [ebp+0x97],0x01
; _skip_jle_zf:
cmp ecx,edx ; sf != of
jz _skip_jle
mov byte ptr [ebp+0x97],0x01
; _skip_jle:
cmp byte ptr [ebx+0x0a],0xaa ; jno
jnz _skip_jno
test eax,0x800 ; of = 0
jnz _skip_jno
mov byte ptr [ebp+0x97],0x01
; _skip_jno:
cmp byte ptr [ebx+0x0a],0xbb ; jnp
jnz _skip_jnp
test eax,0x04 ; pf = 0
jnz _skip_jnp
mov byte ptr [ebp+0x97],0x01
; _skip_jnp:
cmp byte ptr [ebx+0x0a],0xcc ; jns
jnz _skip_jns
test eax,0x80 ; sf = 0
jnz _skip_jns
mov byte ptr [ebp+0x97],0x01
; _skip_jns:
cmp byte ptr [ebx+0x0a],0xdd ; jo
jnz _skip_jo
test eax,0x800 ; of = 1
jz _skip_jo
mov byte ptr [ebp+0x97],0x01
; _skip_jo:
cmp byte ptr [ebx+0x0a],0xee ; jp
jnz _skip_jp
test eax,0x04 ; pf = 1
jz _skip_jp
```

```
mov byte ptr [ebp+0x97],0x01
; _skip_jp:
cmp byte ptr [ebx+0x0a],0xff ; js
jnz _skip_js
test eax,0x80 ; sf = 1
jz _skip_js
mov byte ptr [ebp+0x97],0x01
; _skip_js:
cmp byte ptr [ebp+0x97],0x00
jz _false_branch
; _true_branch:
mov eax,[ebx+0x04]
test eax,0x80000000
jz _forwards
and eax,0x7fffffff
sub [ebp+0x08],eax
jmp _finish
; _forwards:
add [ebp+0x08],eax
; _finish:
mov edi,[ebp+0x4f]
movzx eax,word ptr [ebx+0x00]
jmp dword ptr [edi+eax*4]
; _false_branch:
```

jcc external imm32

```
mov ebx,[ebp+0x08]
movzx eax,word ptr [ebx+0x06]
mov eax,[ebp+eax]
mov byte ptr [ebp+0x97],0x00
mov ecx,eax
and ecx,0x80 ; sf
shr ecx,0x07
mov edx,eax
and edx,0x800 ; of
shr edx,0x0b
cmp byte ptr [ebx+0x08],0x00 ; jz
jnz _skip_jz
test eax,0x40 ; zf = 1
jz _skip_jz
mov byte ptr [ebp+0x97],0x01
; _skip_jz:
cmp byte ptr [ebx+0x08],0x11 ; jnz
jnz _skip_jnz
test eax,0x40 ; zf = 0
jnz _skip_jnz
mov byte ptr [ebp+0x97],0x01
; _skip_jnz:
cmp byte ptr [ebx+0x08],0x22 ; ja
jnz _skip_ja
test eax,0x40 ; zf = 0
jnz _skip_ja
test eax,0x01 ; cf = 0
jnz _skip_ja
mov byte ptr [ebp+0x97],0x01
; _skip_ja:
cmp byte ptr [ebx+0x08],0x33 ; jae
jnz _skip_jae
test eax,0x01 ; cf = 0
jnz _skip_jae
mov byte ptr [ebp+0x97],0x01
; _skip_jae:
cmp byte ptr [ebx+0x08],0x44 ; jb
jnz _skip_jb
test eax,0x01 ; cf = 1
jz _skip_jb
mov byte ptr [ebp+0x97],0x01
; _skip_jb:
cmp byte ptr [ebx+0x08],0x55 ; jbe
jnz _skip_jbe
test eax,0x40 ; zf = 1
jz _skip_jbe_zf
mov byte ptr [ebp+0x97],0x01
; _skip_jbe_zf:
test eax,0x01 ; cf = 1
jz _skip_jbe
mov byte ptr [ebp+0x97],0x01
; _skip_jbe:
cmp byte ptr [ebx+0x08],0x66 ; jg
jnz _skip_jg
test eax,0x40 ; zf = 0
jnz _skip_jg
cmp ecx,edx ; sf == of
jnz _skip_jg
mov byte ptr [ebp+0x97],0x01
; _skip_jg:
cmp byte ptr [ebx+0x08],0x77 ; jge
jnz _skip_jge
cmp ecx,edx ; sf == of
jnz _skip_jge
mov byte ptr [ebp+0x97],0x01
; _skip_jge:
cmp byte ptr [ebx+0x08],0x88 ; jl
jnz _skip_jl
cmp ecx,edx ; sf != of
jz _skip_jl
mov byte ptr [ebp+0x97],0x01
; _skip_jl:
cmp byte ptr [ebx+0x08],0x99 ; jle
jnz _skip_jle
test eax,0x40 ; zf = 1
jz _skip_jle_zf
mov byte ptr [ebp+0x97],0x01
; _skip_jle_zf:
cmp ecx,edx ; sf != of
jz _skip_jle
mov byte ptr [ebp+0x97],0x01
; _skip_jle:
cmp byte ptr [ebx+0x08],0xaa ; jno
jnz _skip_jno
test eax,0x800 ; of = 0
jnz _skip_jno
mov byte ptr [ebp+0x97],0x01
; _skip_jno:
cmp byte ptr [ebx+0x08],0xbb ; jnp
jnz _skip_jnp
test eax,0x04 ; pf = 0
jnz _skip_jnp
mov byte ptr [ebp+0x97],0x01
; _skip_jnp:
cmp byte ptr [ebx+0x08],0xcc ; jns
jnz _skip_jns
test eax,0x80 ; sf = 0
jnz _skip_jns
mov byte ptr [ebp+0x97],0x01
; _skip_jns:
cmp byte ptr [ebx+0x08],0xdd ; jo
jnz _skip_jo
test eax,0x800 ; of = 1
jz _skip_jo
mov byte ptr [ebp+0x97],0x01
; _skip_jo:
cmp byte ptr [ebx+0x08],0xee ; jp
jnz _skip_jp
test eax,0x04 ; pf = 1
jz _skip_jp
```

```
mov byte ptr [ebp+0x97],0x01
; _skip_jp:
cmp byte ptr [ebx+0x08],0xff ; js
jnz _skip_js
test eax,0x80 ; sf = 1
jz _skip_js
mov byte ptr [ebp+0x97],0x01
; _skip_js:
cmp byte ptr [ebp+0x97],0x00
jz _false_branch
; _true_branch:
mov edx,[ebx+0x00]
add edx,[ebp+0x75] ; image base
movzx ecx,word ptr [ebx+0x04]
mov [esp+ecx],edx
mov [ebp+0x36],0x00000000
popad
popfd
ret
; _false_branch:
add esp,0x24
movzx ecx,word ptr [ebx+0x09]
add dword ptr [ebp+ecx],0x24
```

align

```
[load operand data (operand 1)]
```

```
mov edi,[ebp+0x26] ; value 1
sub edi,[ebp+0x60]
sub edi,0x1f7e5efd
and edi,0xffff
mov esi,[ebp+0x75] ; image base
add [ebp+edi],esi
```

xchg

```
[load operand info (operand 1)]
[load operand data (operand 1)]
[resolve register (operand 1)]
[resolve memory (operand 1)]
[load operand info (operand 2)]
[load operand data (operand 2)]
[resolve register (operand 2)]
[resolve memory (operand 2)]
```

```
mov al,[ebp+0x70] ; size 1
add al,0x3f

; _populate_operand_1:
mov ebx,[ebp+0x26] ; value 1
add ebx,[ebp+0x60]
xor ebx,0x02a59e86
mov edx,[ebp+0x1e] ; address 2
add edx,0x3897e232
cmp al,0x01
jnz _skip_byte_3
mov [edx],bl
; _skip_byte_3:
cmp al,0x02
jnz _skip_word_3
mov [edx],bx
; _skip_word_3:
cmp al,0x03
jnz _populate_operand_2
mov [edx],ebx

; _populate_operand_2:
mov ebx,[ebp+0x32] ; value 2
xor ebx,[ebp+0x60]
add ebx,0x1d937350
mov edi,[ebp+0x04] ; address 1
add edi,0x3897e232
cmp al,0x01
jnz _skip_byte_4
mov [edi],bl
; _skip_byte_4:
cmp al,0x02
jnz _skip_word_4
mov [edi],bx
; _skip_word_4:
cmp al,0x03
jnz _skip_dword_4
mov [edi],ebx
; _skip_dword_4:
```

stack operation

```
[load mnemonic]
[load operand info (operand 1)]
[load operand data (operand 1)]
[resolve register (operand 1)]
[resolve memory (operand 1)]
```

```
; _resolve_mnemonic:
mov edx,[ebp+0x26] ; value 1
sub edx,[ebp+0x60]
add edx,0x0378a6a3
mov cl,[ebp+0x57] ; mnemonic
add cl,0xa3
mov al,[ebp+0x70] ; size 1
sub al,0x95
and al,0x0f

; _resolve_push:
cmp cl,0x42 ; push
jnz _resolve_pop
cmp al,0x02
jnz _resolve_push32
push dx
jmp _resolve_pop
; _resolve_push32:
push edx

; _resolve_pop:
cmp cl,0xb5 ; pop
jnz _resolve_stack
mov edx,[ebp+0x04] ; address 1
xor edx,0x7e3d72cd
cmp al,0x02
jnz _resolve_pop32
pop word ptr [edx]
jmp _resolve_stack
; _resolve_pop32:
pop dword ptr [edx]

; _resolve_stack:
mov edx,[ebp+0x08]
movzx edx,word ptr [edx+0x06]
add edx,ebp
mov cl,[ebp+0x70] ; size
sub cl,0x95
and cl,0x0f
mov al,[ebp+0x57] ; mnemonic
add al,0xa3

; _resolve_stack_push:
cmp al,0x42 ; push
jnz _resolve_stack_pop
cmp cl,0x02
jnz _resolve_stack_pop32
sub dword ptr [edx],0x02
jmp _resolve_stack_pop
; _resolve_stack_pop32:
sub dword ptr [edx],0x04

; _resolve_stack_pop:
cmp al,0xb5 ; pop
jnz _skip_stack_pop
mov eax,[ebp+0x04] ; address 1
xor eax,0x7e3d72cd
cmp eax,edx
jz _skip_stack_pop
add dword ptr [edx],0x04
; _skip_stack_pop:
```

unary operation

```
[load mnemonic]
[load operand info (operand 1)]
[load operand data (operand 1)]
[resolve register (operand 1)]
[resolve memory (operand 1)]
```

```
; _resolve_neg:
mov bl,[ebp+0x57] ; mnemonic
sub bl,0x6f
cmp bl,0x96 ; neg
jnz _resolve_inc
mov esi,[ebp+0x26] ; value 1
sub esi,[ebp+0x60]
xor esi,0x7551026f
neg esi
pushfd
add esi,0x3b264be3
mov [ebp+0x7f],esi ; result

; _resolve_inc:
mov al,[ebp+0x57] ; mnemonic
sub al,0x6f
cmp al,0x41 ; inc
jnz _resolve_not
mov ecx,[ebp+0x26] ; value 1
sub ecx,[ebp+0x60]
xor ecx,0x7551026f
inc ecx
pushfd
add ecx,0x3b264be3
mov [ebp+0x7f],ecx ; result

; _resolve_not:
mov al,[ebp+0x57] ; mnemonic
sub al,0x6f
cmp al,0x39 ; not
jnz _resolve_dec
mov eax,[ebp+0x26] ; value 1
sub eax,[ebp+0x60]
xor eax,0x7551026f
not eax
cmp esi,edx
pushfd
add eax,0x3b264be3
mov [ebp+0x7f],eax ; result

; _resolve_dec:
mov bl,[ebp+0x57] ; mnemonic
sub bl,0x6f
cmp bl,0x72 ; dec
jnz _store_eflags
mov eax,[ebp+0x26] ; value 1
sub eax,[ebp+0x60]
xor eax,0x7551026f
dec eax
pushfd
add eax,0x3b264be3
mov [ebp+0x7f],eax ; result

; _store_eflags:
pop eax
mov ebx,[ebp+0x08]
movzx ebx,byte ptr [ebx+0x06]
cmp ebx,0x00
jz _store_result
mov ebx,[ebp+0x08]
movzx ebx,word ptr [ebx+0x07]
mov [ebp+ebx],eax

; _store_result:
mov dl,[ebp+0x57] ; mnemonic
sub dl,0x6f
cmp dl,0xc1 ; cmp
jz _skip_result
cmp dl,0x06 ; test
jz _skip_result
mov edi,[ebp+0x04] ; address 1
xor edi,0x465bb8a6
mov edx,[ebp+0x7f] ; result
sub edx,0x3b264be3
mov cl,[ebp+0x70] ; size 1
xor cl,0x5a
cmp cl,0x01
jnz _skip_byte
mov [edi],dl
; _skip_byte:
cmp cl,0x02
jnz _skip_word
mov [edi],dx
; _skip_word:
cmp cl,0x03
jnz _skip_result
mov [edi],edx
; _skip_result:
```

```
binary operation
```

```
[load mnemonic]
[load operand info (operand 1)]
[load operand data (operand 1)]
[resolve register (operand 1)]
[resolve memory (operand 1)]
[load operand info (operand 2)]
[load operand data (operand 2)]
[resolve register (operand 2)]
[resolve memory (operand 2)]
```

```
; _resolve_movsx:
mov bl,[ebp+0x57]
add bl,0x35
cmp bl,0xca ; movsx
jnz _resolve_imul
mov edx,[ebp+0x32] ; value 2
sub edx,[ebp+0x60]
xor edx,0x4953dd03
mov al,[ebp+0x3a] ; size 2
xor al,0xc7
cmp al,0x01
jnz _skip_movsx_byte:
movsx ecx,dl
; _skip_movsx_byte:
cmp al,0x02
jnz _skip_movsx_word
movsx ecx,dx
; _skip_movsx_word:
add edx,ecx
pushfd
add ecx,0x5b6fd93a
mov [ebp+0x7f],ecx ; result

; _resolve_imul:
mov al,[ebp+0x57]
add al,0x35
cmp al,0xc4 ; imul
jnz _resolve_add
mov ecx,[ebp+0x26] ; value 1
sub ecx,[ebp+0x60]
sub ecx,0x7f25f135
mov edi,[ebp+0x32] ; value 2
sub edi,[ebp+0x60]
xor edi,0x4953dd03
imul ecx,edi
pushfd
add ecx,0x5b6fd93a
mov [ebp+0x7f],ecx ; result

; _resolve_mov:
mov dl,[ebp+0x57]
add dl,0x35
cmp dl,0xd7 ; mov
jnz _resolve_rcr
mov ecx,[ebp+0x32] ; value 2
sub ecx,[ebp+0x60]
xor ecx,0x4953dd03
mov eax,ecx
add ecx,eax
pushfd
add eax,0x5b6fd93a
mov [ebp+0x7f],eax ; result

; _resolve_rcr:
mov al,[ebp+0x57]
add al,0x35
cmp al,0x4d ; rcr
jnz _resolve_or
mov edx,[ebp+0x26] ; value 1
sub edx,[ebp+0x60]
sub edx,0x7f25f135
mov edi,[ebp+0x32] ; value 2
sub edi,[ebp+0x60]
xor edi,0x4953dd03
mov ecx,edi
mov al,[ebp+0x70] ; size 1
xor al,0x3f
cmp al,0x01
jnz _skip_rcr_byte
rcr dl,cl
pushfd
; _skip_rcr_byte:
cmp al,0x02
jnz _skip_rcr_word
rcr dx,cl
pushfd
; _skip_rcr_word:
cmp al,0x03
jnz _skip_rcr_dword
rcr edx,cl
pushfd
; _skip_rcr_dword:
add edx,0x5b6fd93a
mov [ebp+0x7f],edx ; result

; _resolve_or:
```

```
mov al,[ebp+0x57]
add al,0x35
cmp al,0x40 ; or
jnz _resolve_shl
mov eax,[ebp+0x26] ; value 1
sub eax,[ebp+0x60]
sub eax,0x7f25f135
mov ecx,[ebp+0x32] ; value 2
sub ecx,[ebp+0x60]
xor ecx,0x4953dd03
or eax,ecx
pushfd
add eax,0x5b6fd93a
mov [ebp+0x7f],eax ; result

; _resolve_shl:
mov al,[ebp+0x57]
add al,0x35
cmp al,0x8d ; shl
jnz _resolve_cmp
mov ebx,[ebp+0x26] ; value 1
sub ebx,[ebp+0x60]
sub ebx,0x7f25f135
mov edx,[ebp+0x32] ; value 2
sub edx,[ebp+0x60]
xor edx,0x4953dd03
mov ecx,edx
mov al,[ebp+0x70] ; size 1
xor al,0x3f
cmp al,0x01
jnz _skip_shl_byte
shl bl,cl
pushfd
; _skip_shl_byte:
cmp al,0x02
jnz _skip_shl_word
shl bx,cl
pushfd
; _skip_shl_word:
cmp al,0x03
jnz _skip_shl_dword
shl ebx,cl
pushfd
; _skip_shl_dword:
add ebx,0x5b6fd93a
mov [ebp+0x7f],ebx ; result

; _resolve_cmp:
mov cl,[ebp+0x57]
add cl,0x35
cmp cl,0xc1 ; cmp
jnz _resolve_sub
mov edx,[ebp+0x26] ; value 1
sub edx,[ebp+0x60]
sub edx,0x7f25f135
mov ebx,[ebp+0x32] ; value 2
sub ebx,[ebp+0x60]
xor ebx,0x4953dd03
mov cl,[ebp+0x70] ; size 1
xor cl,0x3f
cmp cl,0x01
jnz _skip_cmp_byte
cmp dl,bl
pushfd
; _skip_cmp_byte:
cmp cl,0x02
jnz _skip_cmp_word
cmp dx,bx
pushfd
; _skip_cmp_word:
cmp cl,0x03
jnz _skip_cmp_dword
cmp edx,ebx
pushfd
; _skip_cmp_dword:
add edx,0x5b6fd93a
mov [ebp+0x7f],edx ; result

; _resolve_sub:
mov al,[ebp+0x57]
add al,0x35
cmp al,0x10 ; sub
jnz _resolve_rcl
mov ebx,[ebp+0x26] ; value 1
sub ebx,[ebp+0x60]
sub ebx,0x7f25f135
mov eax,[ebp+0x32] ; value 2
sub eax,[ebp+0x60]
xor eax,0x4953dd03
sub ebx,eax
pushfd
add ebx,0x5b6fd93a
mov [ebp+0x7f],ebx ; result

; _resolve_rcl:
mov bl,[ebp+0x57]
add bl,0x35
cmp bl,0x05 ; rcl
jnz _resolve_add
mov eax,[ebp+0x26] ; value 1
sub eax,[ebp+0x60]
sub eax,0x7f25f135
mov edx,[ebp+0x32] ; value 2
sub edx,[ebp+0x60]
xor edx,0x4953dd03
mov ecx,edx
mov bl,[ebp+0x70] ; size 1
xor bl,0x3f
cmp bl,0x01
jnz _skip_rcl_byte
rcl al,cl
pushfd
; _skip_rcl_byte:
cmp bl,0x02
```

```
jnz _skip_rcl_word
rcl ax,cl
pushfd
; _skip_rcl_word:
cmp bl,0x03
jnz _skip_rcl_dword
rcl eax,cl
pushfd
; _skip_rcl_dword:
add eax,0x5b6fd93a
mov [ebp+0x7f],eax ; result

; _resolve_add:
mov al,[ebp+0x57]
add al,0x35
cmp al,0xa2 ; add
jnz _resolve_rol
mov ebx,[ebp+0x26] ; value 1
sub ebx,[ebp+0x60]
sub ebx,0x7f25f135
mov edx,[ebp+0x32] ; value 2
sub edx,[ebp+0x60]
xor edx,0x4953dd03
add ebx,edx
pushfd
add ebx,0x5b6fd93a
mov [ebp+0x7f],ebx ; result

; _resolve_rol:
mov bl,[ebp+0x57]
add bl,0x35
cmp bl,0xe3 ; rol
jnz _resolve_test
mov edx,[ebp+0x26] ; value 1
sub edx,[ebp+0x60]
sub edx,0x7f25f135
mov eax,[ebp+0x32] ; value 2
sub eax,[ebp+0x60]
xor eax,0x4953dd03
mov ecx,eax
mov bl,[ebp+0x70] ; size 1
xor bl,0x3f
cmp bl,0x01
jnz _skip_rol_byte
rol dl,cl
pushfd
; _skip_rol_byte:
cmp bl,0x02
jnz _skip_rol_word
rol dx,cl
pushfd
; _skip_rol_word:
cmp bl,0x03
jnz _skip_rol_dword
rol edx,cl
pushfd
; _skip_rol_dword:
add edx,0x5b6fd93a
mov [ebp+0x7f],edx ; result

; _resolve_test:
mov cl,[ebp+0x57]
add cl,0x35
cmp cl,0x06 ; test
jnz _resolve_movzx
mov ebx,[ebp+0x26] ; value 1
sub ebx,[ebp+0x60]
sub ebx,0x7f25f135
mov edx,[ebp+0x32] ; value 2
sub edx,[ebp+0x60]
xor edx,0x4953dd03
mov cl,[ebp+0x70] ; size 1
xor cl,0x3f
cmp cl,0x01
jnz _skip_test_byte
test bl,dl
pushfd
; _skip_test_byte:
cmp cl,0x02
jnz _skip_test_word
test bx,dx
pushfd
; _skip_test_word:
cmp cl,0x03
jnz _skip_test_dword
test ebx,edx
pushfd
; _skip_test_dword:
add ebx,0x5b6fd93a
mov [ebp+0x7f],ebx ; result

; _resolve_movzx:
mov al,[ebp+0x57]
add al,0x35
cmp al,0x9c ; movzx
jnz _resolve_shr
mov ecx,[ebp+0x32] ; value 2
sub ecx,[ebp+0x60]
xor ecx,0x4953dd03
mov al,[ebp+0x3a] ; size 2
xor al,0xc7
cmp al,0x01
jnz _skip_movzx_byte
movzx esi,cl
; _skip_movzx_byte:
cmp al,0x02
jnz _skip_movzx_word
movzx esi,cx
; _skip_movzx_word:
add ecx,esi
pushfd
add esi,0x5b6fd93a
mov [ebp+0x7f],esi ; result

```

```
; _resolve_shr:
mov bl,[ebp+0x57]
add bl,0x35
cmp bl,0x1f ; shr
jnz _resolve_ror
mov edx,[ebp+0x26] ; value 1
sub edx,[ebp+0x60]
sub edx,0x7f25f135
mov eax,[ebp+0x32] ; value 2
sub eax,[ebp+0x60]
xor eax,0x4953dd03
mov ecx,eax
mov bl,[ebp+0x70] ; size 1
xor bl,0x3f
cmp bl,0x01
jnz _skip_shr_byte
shr dl,cl
pushfd
; _skip_shr_byte:
cmp bl,0x02
jnz _skip_shr_word
shr dx,cl
pushfd
; _skip_shr_word:
cmp bl,0x03
jnz _skip_shr_dword
shr edx,cl
pushfd
; _skip_shr_dword:
add edx,0x5b6fd93a
mov [ebp+0x7f],edx ; result

; _resolve_ror:
mov dl,[ebp+0x57]
add dl,0x35
cmp dl,0x15 ; ror
jnz _resolve_xor
mov eax,[ebp+0x26] ; value 1
sub eax,[ebp+0x60]
sub eax,0x7f25f135
mov esi,[ebp+0x32] ; value 2
sub esi,[ebp+0x60]
xor esi,0x4953dd03
mov ecx,esi
mov dl,[ebp+0x70] ; size 1
xor dl,0x3f
cmp dl,0x01
jnz _skip_ror_byte
ror al,cl
pushfd
; _skip_ror_byte:
cmp dl,0x02
jnz _skip_ror_word
ror ax,cl
pushfd
; _skip_ror_word:
cmp dl,0x03
jnz _skip_ror_dword
ror eax,cl
pushfd
; _skip_ror_dword:
add eax,0x5b6fd93a
mov [ebp+0x7f],eax ; result

; _resolve_xor:
mov cl,[ebp+0x57]
add cl,0x35
cmp cl,0x08 ; xor
jnz _resolve_and
mov ebx,[ebp+0x26] ; value 1
sub ebx,[ebp+0x60]
sub ebx,0x7f25f135
mov ecx,[ebp+0x32] ; value 2
sub ecx,[ebp+0x60]
xor ecx,0x4953dd03
xor ebx,ecx
pushfd
add ebx,0x5b6fd93a
mov [ebp+0x7f],ebx

; _resolve_and:
mov dl,[ebp+0x57]
add dl,0x35
cmp dl,0x49 ; and
jnz _store_eflags
mov esi,[ebp+0x26] ; value 1
sub esi,[ebp+0x60]
sub esi,0x7f25f135
mov ecx,[ebp+0x32] ; value 2
sub ecx,[ebp+0x60]
xor ecx,0x4953dd03
and esi,ecx
pushfd
add esi,0x5b6fd93a
mov [ebp+0x7f],esi ; result

; _store_eflags:
pop eax
mov ebx,[ebp+0x08]
movzx ebx,byte ptr [ebx+0x0b]
cmp ebx,0x00
jz _store_result
mov ebx,[ebp+0x08]
movzx ebx,word ptr [ebx+0x0c]
mov [ebp+ebx],eax

; _store_result:
mov bl,[ebp+0x57] ; mnemonic
add bl,0x35
cmp bl,0xc1 ; cmp
jz _skip_result
cmp bl,0x06 ; test
jz _skip_result
mov esi,[ebp+0x04] ; address 1
```

```
add esi,0x0435f803
mov ebx,[ebp+0x7f] ; result
sub ebx,0x5b6fd93a
mov dl,[ebp+0x70] ; size 1
xor dl,0x3f
cmp dl,0x01
jnz skip_byte
mov [esi],bl
; skip_byte:
cmp dl,0x02
jnz skip_word
mov [esi],bx
; skip_word:
cmp dl,0x03
jnz _skip_result
mov [esi],ebx
; _skip_result:
```

APPENDIX C Themida FISH subhandlers

load mnemonic

```
mov ebx,[ebp+0x08]
movzx eax,byte ptr [ebx+0x00]
add eax,[ebp+0x53]
add eax,[ebp+0xa7]
add eax,[ebp+0x68]
xor eax,[ebp+0x89]
xor [ebp+0x57],al ; mnemonic
```

load operand info (operand 1)

```
mov ebx,[ebp+0x08]
movzx eax,byte ptr [ebx+0x00]
mov bl,al
and bl,0x0f
sub bl,0x3f
and al,0xf0
shr al,0x04
xor al,0x0f
mov [ebp+0x4c],al ; type 1
mov [ebp+0x70],bl ; size 1
```

load operand info (operand 2)

```
mov ebx,[ebp+0x08]
movzx eax,byte ptr [ebx+0x00]
mov bl,al
and bl,0x0f
sub bl,0x82
and al,0xf0
shr al,0x04
sub al,0x9b
mov [ebp+0xa2],al ; type 2
mov [ebp+0x3a],bl ; size 2
```

load operand data (operand 1)

```
mov ebx,[ebp+0x08]
mov eax,[ebx+0x00]
sub eax,0x0378a6a3
add eax,[ebp+0x60]
mov [ebp+0x26],eax ; value 1
```

load operand data (operand 2)

```
mov ebx,[ebp+0x08]
mov eax,[ebx+0x00]
sub eax,0x1d937350
xor eax,[ebp+0x60]
mov [ebp+0x32],eax ; value 2
```

resolve register (operand 1)

```
mov al,[ebp+0x4c] ; type 1
xor al,0x0f
cmp al,0x01
jnz _skip
mov eax,[ebp+0x26] ; value 1
add eax,[ebp+0x60]
xor eax,0x02a59e86
and eax,0xffff
add eax,ebp
mov ebx,eax
sub eax,0x3897e232
mov [ebp+0x04],eax ; address 1
mov eax,[ebx]
xor eax,0x2a59e86
sub eax,[ebp+0x60]
mov [ebp+0x26],eax ; value 1
; _skip:
```

resolve register (operand 2)

```
mov al,[ebp+0xa2] ; type 2
add al,0x9b
cmp al,0x01
jnz _skip
mov eax,[ebp+0x32] ; value 2
xor eax,[ebp+0x60]
add eax,0x1d937350
and eax,0xffff
add eax,ebp
mov ebx,eax
sub eax,0x3897e232
mov [ebp+0x1e],eax ; address 2
mov eax,[ebx]
sub eax,0x1d937350
xor eax,[ebp+0x60]
mov [ebp+0x32],eax ; value 2
; _skip:
```

resolve memory (operand 1)

```
mov al,[ebp+0x4c] ; type 1
xor al,0x0f
cmp al,0x02
jnz _skip
mov eax,[ebp+0x26] ; value 1
add eax,[ebp+0x60]
xor eax,0x02a59e86
and eax,0xffff
add eax,ebp
mov eax,[eax]
mov ebx,eax
sub ebx,0x3897e232
mov [ebp+0x04],ebx ; address 1
mov bl,[ebp+0x70] ; size 1
add bl,0x3f
cmp bl,0x01
jnz _skip_byte
mov al,[eax]
; _skip_byte:
cmp bl,0x02
jnz _skip_word
mov ax,[eax]
; _skip_word:
cmp bl,0x03
jnz _skip_dword
mov eax,[eax]
; _skip_dword:
xor eax,0x02a59e86
sub eax,[ebp+0x60]
mov [ebp+0x26],eax ; value 1
; _skip:
```

resolve memory (operand 2)

```
mov al,[ebp+0xa2] ; type 2
add al,0x9b
cmp al,0x02
jnz _skip
mov eax,[ebp+0x32] ; value 2
xor eax,[ebp+0x60]
add eax,0x1d937350
and eax,0xffff
add eax,ebp
mov eax,[eax]
mov ebx,eax
sub ebx,0x3897e232
mov [ebp+0x1e],ebx ; address 2
mov bl,[ebp+0x3a] ; size 2
add bl,0x82
cmp bl,0x01
jnz _skip_byte
mov al,[eax]
; _skip_byte:
cmp bl,0x02
jnz _skip_word
mov ax,[eax]
; _skip_word:
cmp bl,0x03
jnz _skip_dword
mov eax,[eax]
; _skip_dword:
sub eax,0x1d937350
xor eax,[ebp+0x60]
mov [ebp+0x32],eax ; value 2
; _skip:
```

Bibliography

[1] Rolf Rolles. *Unpacking Virtualization Obfuscators.* URL: `https://static.usenix.org/event/woot09/tech/full_papers/rolles.pdf` (visited on May 20, 2021).

[2] Tom Brosch. *Runtime Packers: The Hidden Problem?* URL: `https://www.researchgate.net/profile/Tom-Brosch/publication/268030543_Runtime_Packers_The_Hidden_Problem/links/55f1c23a08ae199d47c47583/Runtime-Packers-The-Hidden-Problem.pdf` (visited on May 20, 2021).

[3] Scherzo. "Inside Code Virtualizer". In: (February 2007). URL: `http://index-of.es/Reverse-Engineering/Inside%20Code%20Virtualizer.pdf` (visited on May 20, 2021).

[4] Rolf Rolles. *VMProtect, Part 0: Basics.* August 2008. URL: `https://www.msreverseengineering.com/blog/2014/6/23/vmprotect-part-0-basics` (visited on May 20, 2021).

[5] Rolf Rolles. *VMProtect, Part 1: Bytecode and IR.* August 2008. URL: `https://www.msreverseengineering.com/blog/2014/6/23/1v20av0uhf5kygyyaprvj2i6u5ze2a` (visited on May 20, 2021).

[6] Rolf Rolles. *VMProtect, Part 2: Primer on Optimization.* August 2008. URL: `https://www.msreverseengineering.com/blog/2014/6/23/vmprotect-part-2-primer-on-optimization` (visited on May 20, 2021).

[7] Rolf Rolles. *VMProtect, Part 3: Optimizng and Compiling.* August 2008. URL: `https://www.msreverseengineering.com/blog/2014/6/23/vmprotect-part-3-optimization-and-code-generation` (visited on May 20, 2021).

[8] *Automatic Reverse Engineering of Malware Emulators.* May 2009. URL: `http://index-of.es/EBooks/lee-re-malware-emulators.pdf` (visited on May 20, 2021).

[9] Zhenxiang Jim Wang. *Virtual Machine Protection Technology and AV industry.* 2010. URL: `https://archive.f-secure.com/weblog/archives/Zhenxian_Wang-Virtual_machine_protection.pdf` (visited on May 20, 2021).

[10] Rolf Rolles. *Control flow deobfuscation via abstract interpretation.* July 2011. URL: `https://www.msreverseengineering.com/blog/2014/6/23/control-flow-deobfuscation-via-abstract-interpretation` (visited on May 20, 2021).

[11] Kevin Coogan, Gen Lu, Saumya Debray. *Deobfuscation of Virtualization-Obfuscated Software.* October 2011. URL: `https://www2.cs.arizona.edu/~debray/Publications/ccs-unvirtualize.pdf` (visited on May 20, 2021).

[12] Tora. *Devirtualizing FinSpy.* 2012. URL: `https://web.archive.org/web/20180712010754/http://linuxch.org/poc2012/Tora,%20Devirtualizing%20FinSpy.pdf` (visited on May 20, 2021).

[13] Dr. Mafaz Mohsin Khalil Al-Anezi. *Generic Packing Detection using Several Complexity Analysis for Accurate Malware Detection.* 2014. URL: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.428.9726&rep=rep1&type=pdf` (visited on May 20, 2021).

[14] Artem Baranov. *FinFisher rootkit analysis.* January 2017. URL: `https://artemonsecurity.blogspot.com/2017/01/finfisher-rootkit-analysis.html` (visited on May 20, 2021).

[15] Filip Kafka. "Eset's guide to deobfuscating and devirtualizing FinFisher". In: (January 2018). URL: `https://www.welivesecurity.com/wp-content/uploads/2018/01/WP-FinFisher.pdf` (visited on May 20, 2021).

[16] Rolf Rolles. *A walk-through tutorial, with code, on statically unpacking the FinSpy VM: Part One, x86 deobfuscation.* January 2018. URL: `https://www.msreverseengineering.com/blog/2018/1/23/a-walk-through-tutorial-with-code-on-statically-unpacking-the-finspy-vm-part-one-x86-deobfuscation` (visited on May 20, 2021).

[17] Rolf Rolles. *FinSpy VM Part 2: VM analysis and bytecode disassembly.* February 2018. URL: `https://www.msreverseengineering.com/blog/2018/1/31/finspy-vm-part-2-vm-analysis-and-bytecode-disassembly` (visited on May 20, 2021).

[18] Rolf Rolles. *FinSpy VM Unpacking Tutorial Part 3: Devirtualization.* February 2018. URL: `https://www.msreverseengineering.com/blog/2018/2/21/finspy-vm-unpacking-tutorial-part-3-devirtualization` (visited on May 20, 2021).

[19] Andrea Allievi, Elia Florio. *FinFisher exposed: A researcher's tale of defeating traps, tricks, and complex virtual machines.* March 2018. URL: `https://www.microsoft.com/security/blog/2018/03/01/finfisher-exposed-a-researchers-tale-of-defeating-traps-tricks-and-complex-virtual-machines/` (visited on May 20, 2021).