Software obfuscation based on virtualization

Bartosz Wodziński

August 17, 2021

Contents

Abstract

Software obfuscation is a process of transforming code of the program into a different, semantically equivalent form in order to hide implementation details from unauthorized access. The most promising and sophisticated obfuscation technique known is *virtualization*, which relies on transforming code of a program into a bytecode, which is then interpreted and executed by a virtual machine. Despite it being well known and effective, almost all implementations of this technique are closed source and designed for Windows executables. This Master Thesis presents the *VMObf* tool implementing this obfuscation technique, working on Linux programs as well. *VMObf* obfuscates programs by creating a different, unique virtual machine for each function present there and is designed to obfuscate programs written in high level languages that have compilers based on the *llvm* framework.

Obfuskacja kodu to proces polegający na transformacji kodu programu do innej, semantycznie równoważnej postaci w celu ukrycia szczegółów implementacyjnych przed nieautoryzowanym dostępem. Najbardziej obiecującą i wyrafinowaną techniką obfuskacji spośród wszystkich znanych jest *wirtualizacja*, która polega na transformacji kodu programu do kodu bajtowego, który jest następnie interpretowany i wykonywany przez maszynę wirtualną. Pomimo bycia dobrze znaną i efektywną techniką obfuskacji, prawie wszystkie jej implementacje są typu "closed source" i są dedykowane dla plików wykonywalnych na systemach Windows. W tej pracy magisterskiej zaprezentowane zostanie narzędzie *VMObf*, implementujące tę technikę obfuskacji, działające również dla programów dedykowanych dla systemów Linux. Narzędzie *VMObf* obfuskuje programy poprzez tworzenie unikalnych maszyn wirtualnych, innych dla każdej funkcji obecnej w programie i jest zaprojektowane do obfuskacji programów napisanych w językach wysokiego poziomu, dla których istnieją kompilatory bazujące na projekcie *llvm*.

Chapter 1

Introduction

Digital revolution completely changed the contemporary world. It opened many possibilities to learn, exchange information, and to earn money. However, it also opened new possibilities for theft and piracy. Unauthorized software usage leads to heavy financial loses. According to BSA Global Software Survey in 2016 [1], 39% of software installed on PCs around the world in 2015 was not properly licensed and led to losses estimated at over 52 billion $.

Along with rapid progression of software development, adequate piracy prevention measures have to be taken in order to stop criminals from stealing intellectual property, cracking software, or cheating in online games. One way of preventing software piracy is to use *software obfuscation* in order to hide vulnerable code parts from the potential adversary. Although there are many obfuscation techniques known, a fairly new and novel technique called *virtualization* is perceived to be the most sophisticated and hardest to break.

In this Master Thesis, I will present the *VMObf* tool, employing a virtualization technique in order to obfuscate programs compiled using the *llvm* framework. In the first chapter, a context of this work will be given along with its motivation and existing programs performing software obfuscation. Then, I will present an overview of how *VMObf* works. In the next chapter I will provide a more detailed description of this tool and the problems it has to solve in order to work properly. The last chapter will contain an evaluation of the tool and a description of what can be done to improve it.

Chapter 2

Motivation and existing tools

This section will briefly cover the most important definitions and present the most popular obfuscation techniques used in contemporary software, with special emphasis on virtualization. It will also describe some attacks against this technique and elaborate on existing tools that use virtualization in order to protect executables.

2.1 Obfuscation and ways to achieve it

Software obfuscation is a process of transforming the code of a program into a different, semantically equivalent, form in order to hide implementation details from unauthorized access.

Throughout this work, by *obfuscation*, I will refer to obfuscation performed on executable files (thus, not to scripts in particular), that is the files produced during the compilation process. I will additionally assume that the end user of such program has access only to the resulting executable, having no information about the source code written in a high level language.

In order to describe existing obfuscation techniques, we will need two definitions.

Definition 2.1. ***Basic block*** *(aka BB) is a sequence of instructions with no branches into it, except its first instruction, and no branches out, except its last instruction.*

Definition 2.2. ***Control flow graph*** *of a function (aka CFG) is a graph representing all possible paths that may be taken during function execution. It is defined on the basic blocks of a particular function.*

2.1.1 Obfuscation techniques

There are many obfuscation techniques known. The following list enumerates some of them, each with a short description.

- *string encryption* - used in order to conceal strings used in a program by transforming them into randomly looking byte sequences. Very often, the bitwise *xor* operation is used for this purpose.
- *instruction transformation* - works by substituting simple instructions with complex sequences performing the same task.
- *bogus control flow insertion* - relies on inserting new fake branches to the control flow of the function, thus making the CFG more complex.
- *opaque predicates* - it is done by inserting complicated branch instructions that always evaluate to one particular value (e.g. `true`). Usually used along with *bogus control flow insertion*.
- *control flow flattening* - hides control flow of the function by inserting one extra BB (*master block*), responsible for deciding which BB should be executed next. After execution of each BB, control is passed back to the *master block*. This transforms the CFG to a form where edges are only between the *master block* and the remaining blocks; all other edges are erased.
- *function merging* - when types of arguments and return values for two functions are identical, it is possible to merge them into one larger function by adding one extra argument to that larger function, informing which of the internal functions should be executed.
- *virtualization* - an obfuscation technique which uses a Virtual Machine (aka VM) in order to encode machine code instructions into a custom instruction set which only that particular VM can understand. This way, an adversary, in order to reverse engineer the program, has to perform an additional step: translating instructions of the VM back into the native assembly code, which may be (and hopefully is) a time consuming task. If each compiled program has a completely new VM with a unique instruction set, an attacker has to perform the same analysis for each obfuscated program, having no easy way to automate this task. This is one of the purposes that *VMObf* is trying to achieve. It is the most sophisticated and promising technique known, used in the most advanced commercial products.

2.1.2 Virtualization - known attacks

Since virtualization is considered to be the technique most resistant to reverse engineering, this section will list only anti-virtualization attacks.

A fairly simple and surprisingly very efficient anti-virtualization attack relies on opcode frequency analysis. It exploits the observation that some assembly instructions appear with certain frequency in non-obfuscated programs. For example, based on the analysis of several 64-bit Linux executables, Cheng, Lin, Gao and Jia [2] realised that several instructions appear much more frequently than other ones, for instance, `x86 mov` instruction covers approximately 30% of all instructions present in binaries. According to that observation they computed the average frequency of each instruction and were able to successfully match VM opcodes with `x86` assembly instructions with high accuracy, based solely on how often they were used in the target binary.

Another approach, proposed by Rolles in [3], depends on the fact that since VM code already has its interpreter embedded into the executable, it is possible to write a compiler that will work on the same opcodes and convert them back to native assembly. Assuming there is a small number of different (up to possibly opcode permutations) VMs across the executables produced by a certain obfuscator, and that they show some resemblance to each other, one has to examine just one of them in order to write such a compiler.

The third approach is the most sophisticated and general one. It was introduced in [4] in the tool called *Synthia* and aims to break VM protection fully automatically by utilizing *reinforcement learning*. It achieves it by analysing an *instruction trace* of a program and uses Monte Carlo Tree Search algorithm in order to recognise certain patterns appearing in arithmetic operations. It was tested against two state-of-art obfuscators that use virtualization and it recognised on average 95% of arithmetical and logical handlers present in the code, leaving only several for manual analysis.

2.2 Existing products

This section introduces several existing products incorporating virtualization as one of obfuscation techniques.

- *VMProtect* [5] is one of the two most advanced and popular virtualization-based obfuscators available. It is a commercial product that protects `x86` Windows binaries (including EXEs, DLLs, SYS files) from cracking and unauthorized use. In order to achieve better obfuscation, it uses *code mutation* technique along with virtualization. It also offers many other features including license management. Despite being one of the best protectors known, there exist attacks against it. One of them is *NoVmp* [6] - a static devirtualizer working with versions 3.0 -

3.5 of VMProtect. It utilizes *Virtual-machine Translation Intermediate Language* project (aka VTIL) [7] in order to translate VM opcodes back to native assembly code.

- The second of the two most remarkable software protectors using virtualization is *Themida* [8]. Similarly to VMProtect, it targets Windows binaries and uses many other obfuscation techniques. It works at the binary level, so it does not need access to the source code of a given program in order to obfuscate it.

- *DynOpVm* [2] is a virtualizer that aims to be resistant both to the frequency attack and the attack proposed by Rolles in [3]. It achieves that goal by implementing a VM with *dynamic opcode mapping*, where each VM opcode is hidden until the BB containing it is executed. This way, each opcode may be mapped to multiple handlers. The decision which particular handler from this set is to be executed depends only on the block that was executed just before. This tool is not publicly available.

- The only virtualizer based on llvm was written by Lukáš Turčan in [9]. It creates a VM for each BB in the code, thus operates on *basic block* level. It additionally provides string encryption and function merging, but has a time penalty estimated at 12, meaning that the obfuscated program runs 12 times slower than the original one.

- The last virtualizer presented in this list is *rewolf-x86-virtualizer* [10]. Like most aforementioned tools, it works on Windows PE files only. It is not so advanced, but it is the only open source implementation of virtualization I have found.

2.3 Summary and motivation

The main motivation of this thesis is to write a tool incorporating the virtualization technique, which has so far only one open source implementation.

All other tools are closed source, or commercial. The only virtualizer based on llvm performs the obfuscation at the *basic block* level, which means that the control flow of the function control flow is not hidden very well. Implementation of VM operating at *function level* provides better obfuscation and is more challenging.

Since nobody has done it before, I have decided to design *VMObf* so that it deals with that problem and operates at function level in llvm. Moreover, the majority of obfuscators are designed to work on Windows PE files only. *VMObf* is designed to operate on Linux executables too.

Chapter 3

VMObf - overview

3.1 llvm and SSA

llvm is a compiler framework used by *clang* compiler [11] and can be used to compile many high level languages. During the compilation process, the high level code is first transformed into the *intermediate representation* (aka IR). The IR has to obey the *Single Static Assignment* (SSA) form. *Single Static Assignment* requires that each variable be assigned exactly once, and every variable be defined before it is used.

In order to transform the code into the SSA form, the *PHI* instruction is used. It assigns a value to a register variable, depending on the last basic block executed. The following figure from llvm docs [12] shows a simple basic block that uses the *PHI* instruction:

```
Loop:       ; Infinite loop that counts from 0 on up...
  %indvar = phi i32 [ 0, %LoopHeader ], [ %nextindvar, %Loop ]
  %nextindvar = add i32 %indvar, 1
  br label %Loop
```

Figure 3.1: *PHI* instruction writing 0 to *indvar* if the last executed instruction belonged to the *LoopHeader* basic block, and *nextindvar* value when it belonged to the current (*Loop*) BB.

In order to illustrate how the code violating the SSA form looks like and how it can be transformed so that it obeys this form, consider the following simple function consisting of 4 basic blocks, written in pseudocode:

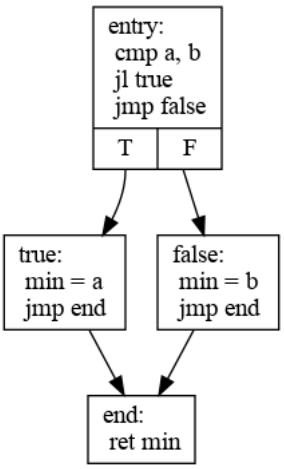


Figure 3.2: Code violating the SSA form

In this example, the `min` *register* variable is defined twice: both in `true` and in `false`. It violates the SSA form since each register variable has to be defined precisely once. In order to fix this, we can transform the code into the following, equivalent form:

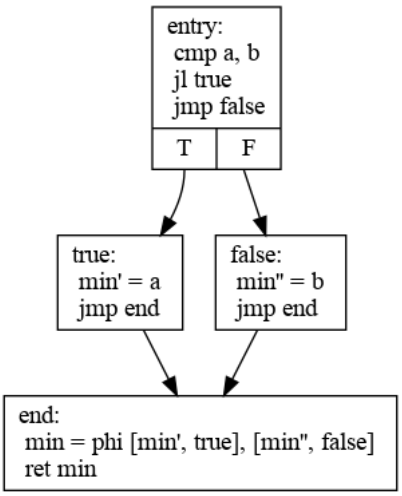


Figure 3.3: Proper SSA form

There exist several algorithms for transforming code into the SSA form. One of such algorithms (described in detail in [13]), works as follows.

At the beginning, each basic block is traversed top-down and two different events are handled: variable definition and variable read. When the algorithm encounters a definition of variable v, it renames v in that definition to some other, unique identifier, say v' (v'' in the next definition, etc.)

and records v' as a *current definition* of v. When an instruction that reads v is encountered, v is renamed in that instruction according to its *current definition*. When some variable v is read before it is assigned in a certain basic block B, its definitions are collected from all direct predecessors of B and a PHI instruction joining these definitions is created. The newly created PHI instruction is then recorded as the current definition of v.

If B has only one predecessor, current definition of v is searched for recursively. In order to avoid infinite recursion, for each visited basic block, an empty PHI instruction defining v is added before recursing. It is filled with the correct value when returning from the recursion.

Then, for each variable v, for each instruction that reads v, if the definition recorded for that read comes from a PHI instruction ϕ_v inserted by the algorithm, v is renamed to ϕ_v. At this point, for each basic block B, each variable that is read in B is defined in B before it is read. Moreover, each variable is defined only once.

As an optional last step, when each BB has been processed, redundant PHI instructions are removed from the code.

A slightly simpler approach is taken in *VMObf* and will be described in detail in the next chapter.

More information regarding llvm can be found in Appendix A.

3.2 VMObf - a high level overview

VMObf is designed to operate on the IR of llvm. Alternatively, it could have been implemented at the assembly level, which would give it more flexibility as some high level constructs required in IR are not needed when writing assembly code.

However, IR of llvm provides a lot of information about the program, in particular, shows the entire possible control flow and function arguments, which is in general impossible to extract from assembly code. Even worse, it is even impossible to distinguish code from data in all cases. Moreover, the knowledge about the program, such as variable types, provided by IR, that is normally erased at compilation time, may be used to obtain better obfuscation: for example, two variables of the same type can sometimes be merged together and share the same memory location. Thus, the information carried by the IR enables better obfuscation which would not be possible when working at the assembly level. That was the main reason for choosing

IR to work with.

3.3 Virtual machine model

VMObf obfuscates each function independently by *virtualizing* it, meaning that it creates a separate VM responsible for execution of each function.

Each VM is register-based; it contains a separate register for each of the local variables of the function (with some exceptions, described more thoroughly in the next chapter). Additionally, it has extra registers used to set proper control flow. VMs do not have a special exception handling mechanism; they rely on the way of dealing with exceptions provided by llvm (see Appendix A.1.2 for more information).

3.4 Obfuscation algorithm - a high level overview

In order to change the IR, *VMObf* has to implement a *pass* (passes in llvm are covered in depth in Appendix A.2.1). During this pass, it iterates through all functions in the module and obfuscates them, creating a unique VM for each. After that, all references to the original functions are replaced with their obfuscated counterparts and the original functions are removed from the module in the second pass.

Let F be the function to obfuscate. The obfuscation algorithm works in several steps. In the first step, the algorithm splits each BB into several smaller BBs forming a path. Then, it partitions the control flow graph of F into *code components* (aka CCs), which are basically connected components (not necessarily maximal) of the CFG. After that, a new function F' is created and CCs are inserted into F' without edges between them. Then, variable-updating code is inserted at the beginning of each CC and SSA form is restored. As the last step, the F' control flow is set so that F' has the same semantics as F. The resulting VM instructions strongly correspond to CCs extracted from the original function F. They additionally contain variable-updating code and instructions determining control flow, inserted in the last step of the obfuscation algorithm.

After the algorithm finishes its job, F' is the obfuscated (virtualized) version of F. However, there are several difficulties that have to be overcome in order to maintain original semantics of F and the required IR form. These are mentioned below:

- CCs that F has been partitioned into may have many edges from / to different CCs. This means that each of them can have several entry

and exit points. Thus, depending on the point when we enter such a weakly connected component C_1, some variables may or may not be defined. The information selecting the entry block has to be passed to each instruction of the virtual machine. This situation is illustrated on Figure 3.4.

- Some variables may be used in different CCs, so their values have to be updated at each entry BB where they are required and at each exit block after which they can be used.
- CCs of F are separated from each other when they are inserted into F'. This can disturb the SSA form required by llvm IR. This case is depicted in Figure 3.5.
- Some edges are required to belong to the same CC. Moreover, some extra care has to be taken when working on the exception handling code.

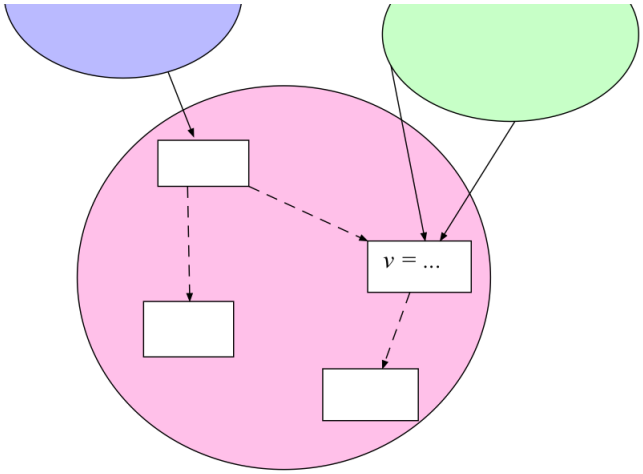


Figure 3.4: The pink CC has two entry points. The variable v is defined in one of them, but the second one has no information regarding v

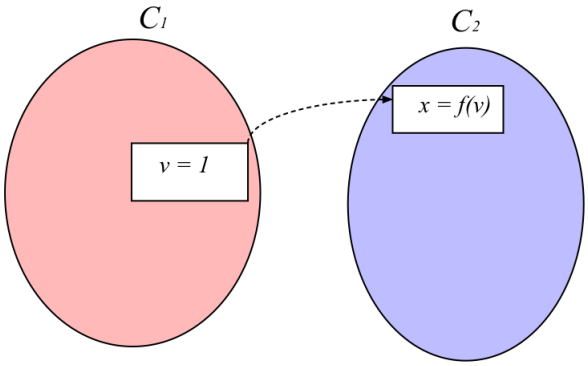


Figure 3.5: v is defined in basic block of C_1, but is used in the next BB, which belongs to C_2. There exists a path from the entry point of the F' to that BB, that does not go via C_1, which means v may be potentially used before it is defined, and such a situation is not allowed in IR. If we defined v in C_2 as well, it would violate the SSA form.

Chapter 4

VMObf - description

4.1 Obfuscation algorithm and VM architecture

4.1.1 VM architecture

Since functions in the module are obfuscated using virtualization, there has to be a dedicated VM for each function in order to interpret and handle its virtualized instructions. In order to describe each VM, we have to define its registers and instruction set.

In llvm IR, local variables of a function can be either of *register* or *stack* type. *register* type means that after the compilation, the variable of this type will be most likely placed in a hardware register, while *stack* variables will be placed on the stack. Only *register* variables are required to be in the SSA form. More information regarding local variables of a function can be found in Appendix A.1.1.

Let $S = \{v_1, v_2, ..., v_k\}$ denote the set of local *register* variables accessed within the function F. Additionally, by VM_F we denote the virtual machine created for the function F. Then, we define the register set R of VM_F as $R = \{r_{v_1}, r_{v_2}..., r_{v_k}, reg_C, reg_{EB}\}$. Hence, VM_F contains a dedicated register for each register variable of F, and two more registers: reg_C and reg_{EB}, which are used to determine the control flow of the obfuscated version of F (abbreviated by F' from now on).

Note. *The "C" in the first extra register name stands for the (code)* ***C****omponent of F, which should be executed next, while EB in the second one holds the index of the* ***E****ntry* ***B****lock in that component which should be executed at the beginning.*

The instruction set of VM_F is completely different for each function F. It depends on the decomposition of CFG of F into weakly connected components, which is performed at the beginning of the obfuscation algorithm and

will be described in the next section. For now, we assume, that CFG of F has already been partitioned into the set of components $C = \{C_1, C_2, ..., C_l\}$.

The instruction set of VM_F strongly corresponds to the aforementioned decomposition: there is a different instruction for each C_i. However, apart from the code contained in C_i, there has to be an additional piece of code that handles updating the values of variables that are used in C_i - they could potentially be changed in other CCs, so this step is necessary in order to retain semantics of F. The code in C_i does not operate directly on VM_F registers; it uses them only to update the values of its internal variables. For this reason, when exiting C_i, registers of VM_F have to be updated in order to contain up to date values of variables of F they are linked to.

Note. *Notice that stack variables of F do not have dedicated VM_F registers. The reason is that it could possibly change semantics of F in some cases. For example, the stack variable o may be pointed to by some variable t. It means that its value may be modified by accessing t, without referencing o directly. If we tried to overwrite the value of o based on VM_F register afterwards, it would overwrite the changes made to o via t. However, such a situation is rather rare (and did not occur in any of the several hundreds functions tested). For this reason, there is an implementation variant in VMObf that has dedicated registers for all variables, including stack ones - it is disabled by default, but can be enabled to provide better obfuscation with a small risk of changing the semantics of F.*

4.1.2 CC partition algorithm

As mentioned in the previous section, there is a strong connection between the CFG decomposition into CCs and VM_F instruction set. The decision restricting the partitioning to consist of connected components only is completely arbitrary - the CFG could be partitioned into non-connected components as well, and even each BB could land in a separate component (as in the *control flow flattening* technique). However, the current choice has a performance-based justification: there are fewer jumps (and thus less register updating code) when working on CCs, and the CFG of F is obfuscated equally well (assuming the number of CCs is big enough).

The current implementation partitions the CFG of F by eliminating (cutting) some of its edges at random, comparing the resulting CC count to a predefined threshold, and, if big enough, returning the list of CCs to be processed by subsequent steps. This algorithm is elaborated on in Section 4.6.

4.2 Variable and register updating code

When some partition is returned by the CC partitioning algorithm, the new function F' is created in order to replace F. Each C_i in the connected components set is inserted into F', but since each CC will belong to a separate instruction of VM_F, there are no edges between C_i and C_j in $\mathrm{CFG}_{F'}$. At this point, the register updating code has to be inserted, along with instructions handling the control flow of F'. This section will elaborate on the former.

After CCs insertion into F', we have to create VM_F registers for local variables accessed in F. Since each VM_F register will be read / written from possibly many code parts, the natural choice is to store each register value in a separate *stack* variable. These are declared in the entry block of F'.

Since each C_i has different variables needed / modified, we have to know precisely which registers of VM_F should be accessed before / after the execution of the code in C_i. For this purpose, *liveness analysis* is performed on the code of F (the standard *liveness analysis* algorithm using the worklist has been implemented - its pseudocode can be found in the last section of this chapter).

After the *liveness analysis* has been performed, we know which variables are required and which are modified in each CC. It is crucial to note, at this point, that each CC can possibly have many entry points and many exit points, and the set of variables needed may depend on where we enter it.

Current *VMObf* implementation updates variable values for each entry block `bb` for each C_i, based on the set of variables that are live at the beginning of `bb` (`in[bb]` in pseudocode). Though it is a fairly good approximation, it may be slightly inaccurate, meaning that some variables that were live at `bb` in F are not needed in C_i at all, but each variable that is needed in C_i has to be in the `in` set for some of its entry blocks. So, there may possibly be some redundant variable updates at the beginning of each CC. That does not affect program performance too much and has a side effect of obfuscating the code even more by adding fake instructions, so I decided not to remove them from the code.

At each exit block of each C_i, we have to update the registers linked to the variables that were modified. These registers are determined based on `out` (variables that are live at the end of BB) and `def` (variables that are defined in BB) sets - there is no point in updating variables that are not in `out` set of the exit block and we have to update only these variables that could have potentially been changed - precisely these that were in `def[bb]` for any `bb` in C_i.

4.3 Register vs stack variables

While the aforementioned variable updating algorithm works perfectly for *stack* variables (when *stack* variable obfuscation option is enabled; see section 4.1.1), it cannot be directly applied to *register* ones. Unlike *stack* variables, *register* variables cannot be updated by assigning a value at the beginning of the CCs, since they already have a definition inside one of them. Additionally, just as *stack* variables, they have to be defined before any use, since llvm requires that form. It does not matter which BBs will be executed first during program execution - in IR, every BB containing a variable use either has to define it first, or to be *dominated* [1] by the set of BBs defining that variable (which is a singleton in case of *register* variables, due to the SSA form).

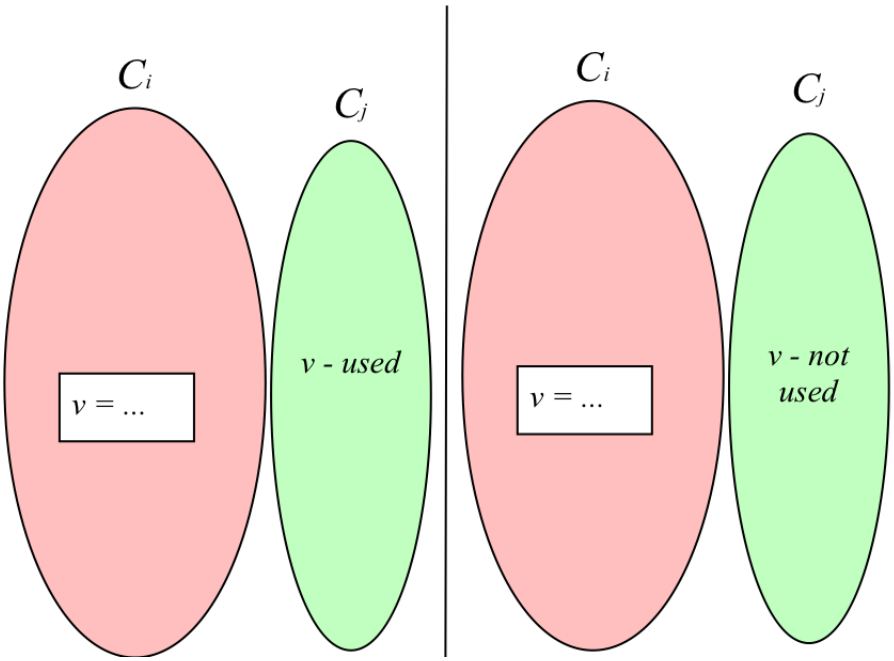
The easiest way of dealing with this problem, would be to use an existing *reg2mem* pass implemented in llvm. It "demotes all registers to memory references" [14]. This path, however, was not chosen, since it would slow down the obfuscated program, effectively stripping the program of many optimizations already performed during the IR code generation.

There are two pairs of cases we have to take care in order to solve the problem. In both cases, we assume that we are dealing with *register* variable v, which is defined in the *code component* C_i.

The first pair concerns uses of v in different code components C_j:

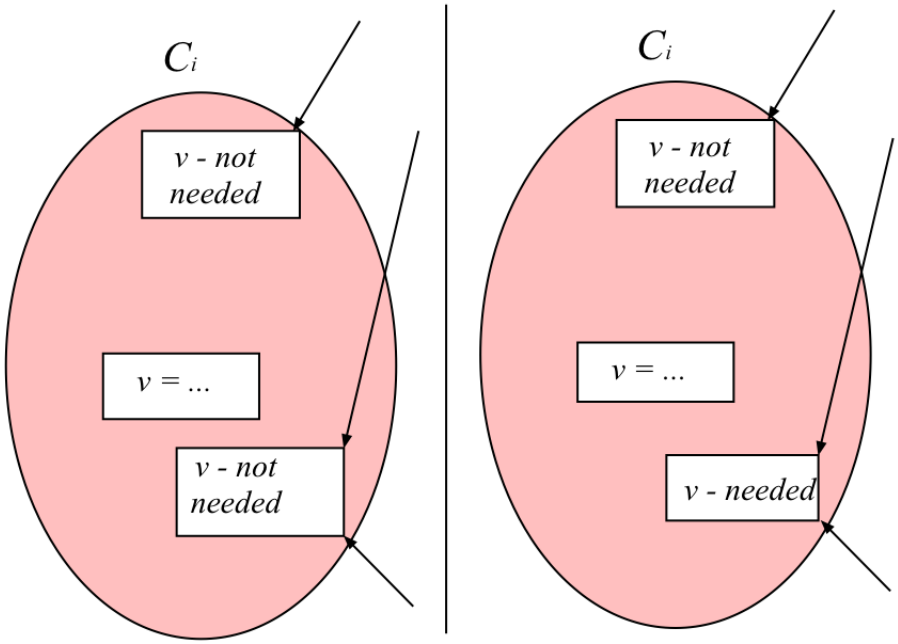
- v is used in some *code component* $C_{j \neq i}$.
- v is not used in any $C_{j \neq i}$.

[1] basic block `bb` is *dominated* by a set S of BBs in F, if each path starting at the entry block of F leading to `bb` has to cross at least one BB from S



The second pair covers uses of v in C_i.

- v is not live at the beginning of entry blocks of C_i.
- v is live at the beginning of at least one entry block of C_i.



Note that v may be needed in the same CC where it was defined because each CC may have many entry blocks, which can be virtually any of BBs in that CC as long as there is an edge to them from some other CC in CFG_F. In particular, the entry BB can even have predecessors in the CC where it belongs.

The first case is easy - it suffices to rename v to v^j in C_j and define it using the value stored at the register r_v of VM_F before entering C_j (in a separate BB executed before the entire C_j). This solution has an extra advantage of obfuscating the original variable set of F - now we have many variables in place of one variable v, so the value of v may be stored in different hardware registers when the IR code is compiled to machine code.

The second case is trivial - nothing has to be done in $C_{j\neq i}$ since v is not referenced there.

The third case is easy as well - we only have to notice that if v is defined in C_i and is not needed in any of the entry blocks of C_i, it means that all paths starting at these entry blocks either visit the BB where v is defined before BBs where it is used, or avoid all BBs where v is needed. Hence, we do not have to do anything here; we can just leave the code regarding v as it is; even if we tried to update the value of v at the beginning of C_i, it would be overwritten before any use of v.

The last and the hardest case is the one where v is both defined and needed in the same connected component C_i. We cannot rename v to v^i since that would not change anything, nor can we define v in C_i for the second time. For this reason, a completely different approach has to be taken here. It is similar to the algorithm for restoring the SSA form presented in [13], but does not use recursion and creates a new variable for each BB in the C_i it operates on.

First, for each basic block `bb` in C_i, a new, unique variable v_{bb} is defined using a *PHI* instruction depending only on preceding basic blocks of `bb` in C_i. The *PHI* instruction specifies only the predecessors of `bb`, with empty places left for variables defined there, since the CFG for C_i may contain cycles and hence there may be a cyclic dependency between the newly created *PHI* instructions.

At this point, each BB already knows all newly created variables for its predecessors that were defined instead of v, so we can fix the *PHI* instructions created in the previous step to contain these values. The entry blocks of C_i, however, have additional predecessors outside C_i: the BBs responsible for updating local variables of F' using values contained in VM_F registers.

For each such preceding BB, an additional variable v_{tmp} and instruction initialising its value based on r_v is inserted. These v_{tmp}s are taken into account in *PHI* instructions for entry blocks of C_i.

There is another small detail that has to be dealt with - if v is defined using a value returned by some function that may throw exceptions, the succeeding basic block U handling these exceptions (the so called *landingpad* block - see Appendix A.1.2 for a description llvm exception handling machanism) cannot access v. The reason for this is that if execution reaches U, it means that the function whose return value was used to define v, has not returned at all - it threw an exception and hence v is not visible in U. In this specific case, a dummy *register* variable v_{dummy} is created at the beginning of the block defining v (it is initialised using the value stored in r_v, just to ensure it has the same type as v). Then, v is changed to v_{dummy} in the *PHI* instruction in U.

As may be easily seen, the solution presented above maintains the SSA form and permits to update the value of v just before each entry BB of C_i. Nonetheless, it introduces noticeable complexity: the aforementioned algorithm has to traverse C_i for each variable accessed there. This gives a pessimistic complexity of $\theta(NV)$, where N is the number of basic blocks in F and V is the number of local *register* variables of F. It may be especially costly when using *VMObf* iteratively, that is, when obfuscating the program already obfuscated by *VMObf*, since *VMObf* adds new basic blocks and variables each time the obfuscation takes place.

4.4 Setting up the control flow of F'

As the last obfuscation step, correct control flow has to be set in F' to match the one determined by CFG_F. There are two VM_F registers delegated to this purpose: reg_C and reg_{EB}. reg_C holds the value that tells which instruction of VM_F has to be executed next. Since each instruction is linked to some connected component C_i of CFG_F, we have to additionally store another value, telling which entry block of C_i has to be executed - this parameter is stored in reg_{EB}. These register values have to be updated at each exit block of C_i. In fact, they are updated slightly later, just after the update of other registers of VM_F. The correct control flow is known, because we still have access to CFG_F at this point. Based on that knowledge, we just have to analyse all edges starting from each exit block, leading to BBs from other *code components*.

What is still left to do, is to create relevant branches in the code, that use values stored in reg_C and reg_{EB} in order to determine the BB to be executed

next, that is, to implement virtual machine *interpreter* code. When this is done, the VM implementation is over.

Note. *At this point, the obfuscation performed by VMObf looks somewhat similar to the control flow flattening technique. In particular, their common characteristic is that the VM instruction (or BB in control flow flattening) performed uniquely determines the next one, so one cannot write the program operating on instructions of VM in a different order than in the original program. In order to deal with this small issue, another approach was implemented: each instruction returned a value which was in turn interpreted by VM interpreter and the relevant branch was taken - this way it was possible to write an arbitrary program operating on VM instructions in any order. However, this approach was abandoned, since it introduced huge overhead (even a factor of 7) to the execution time of programs, and had just a slight obfuscation quality advantage over the solution presented before. Hence, in some sense, the VMs produced by VMObf may be viewed as optimized virtual machines, with some interpreter code moved into instructions themselves.*

4.5 Final actions

In order to remove all unobfuscated functions, all references to them are replaced with references to their obfuscated counterparts and then an additional pass is run that removes all unobfuscated functions from the module, leaving only the obfuscated ones.

The figure below represents a simple function before and after obfuscation performed by *VMObf*.

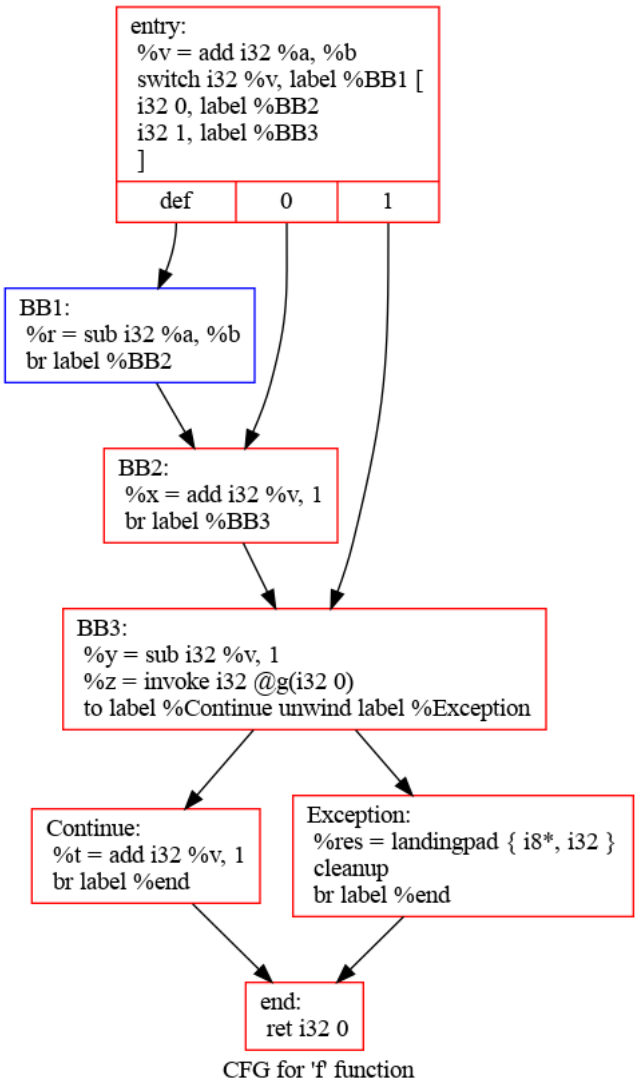


Figure 4.1: Simple function before obfuscation. Nodes are colored according to the CC they were partitioned into. There are two groups: one containing a single node (blue), and the rest (red). Note that BB2 is both an entry block of the red component and has an incoming edge from the same component it belongs to.

The obfuscated function is depicted in the following figure:

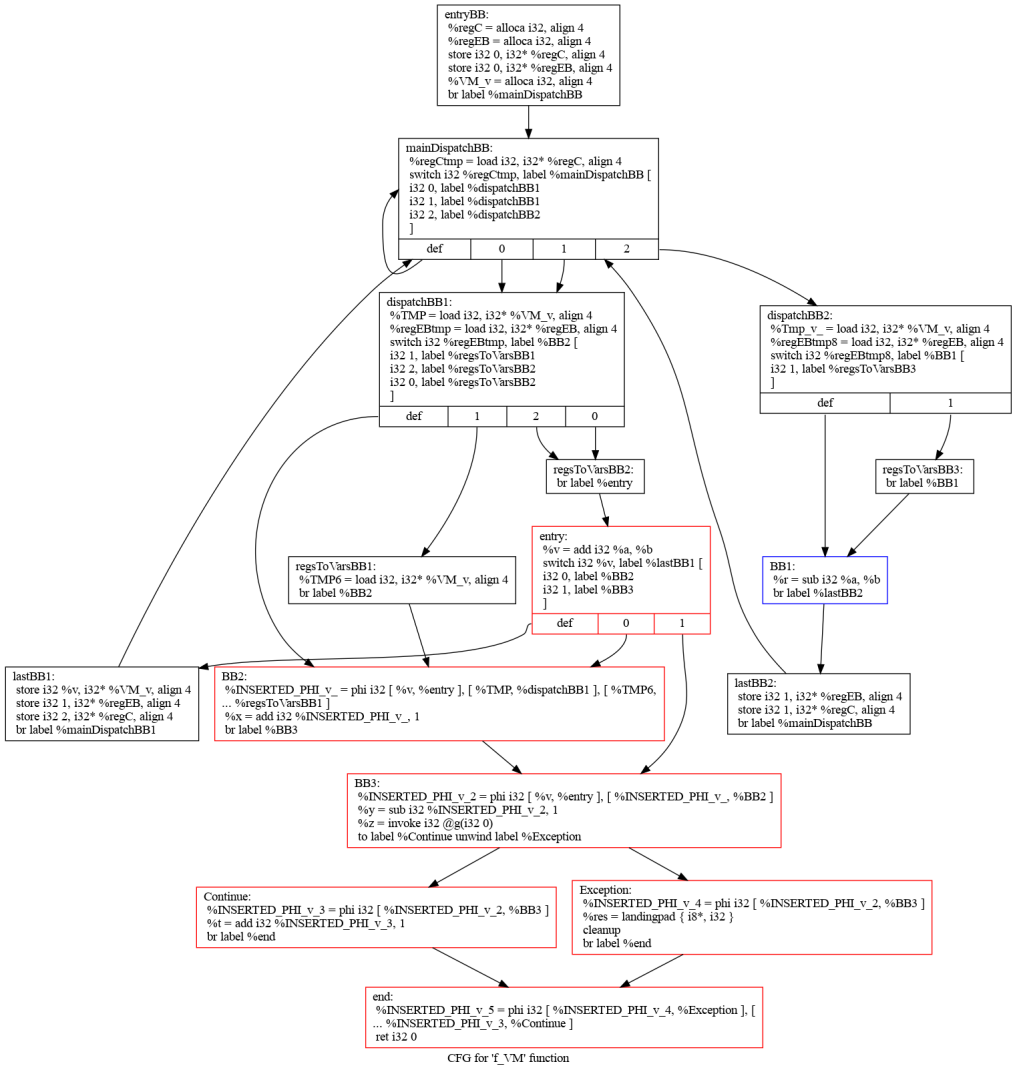


Figure 4.2: Obfuscated function with original CCs colored

There are several things that should be noticed here. Apart from the original basic blocks, there are a couple more added by the obfuscator:

- `entryBB` containing reg_C and reg_{EB} initialization and variables declaration (`VM_v` is VM register for the variable `v`).
- `mainDispatchBB` is a basic block deciding which CC should be executed next.
- `dispatchBB`s redirect the control flow according to reg_{EB}. They additionally create new *register* variables when needed. Here, the variable `v` is live in `BB1`, so `dispatchBB2` has to create an extra *register* variable `Tmp_v` that holds the value of `v`. Since `v` is not modified in `BB1` (and

hence, the entire CC), it does not have to be updated after execution of `BB1`.

- `regsToVarsBB`s are responsible for variable update with respect to VM registers. `regsToVarsBB1` creates an additional *register* variable `TMP6` holding the value of `v`, because `BB2` is an entry block to the red CC (it can be executed after `BB1` and `v` is live in `BB2`).

- `lastBB`s, on the other hand, are responsible for VM registers update. For instance, `lastBB1` has to write `v` into `VM_v` register, since `v` was modified in the `entry` basic block. They also fill reg_C and reg_{EB} with correct values. In `lastBB1`, they are initialised with 2 and 1 respectively, meaning that the second CC (blue) should be entered next and that its first entry block should be executed.

There are also several new instructions added to the original original basic blocks of the function. These have the "INSERTED_PHI" prefix. They correspond to the fourth case related to fixing SSA form, mentioned in Section 4.3.

4.6 CC decomposition algorithm - description

The CC decomposition algorithm, when expressed in pseudocode, looks as follows:

```
partitionCFG()
{
   int numOfEdges = 1;
   // connectedComponents - list of connected components
   // target - target number of CCs
   // nonCutEdges - set of edges that cannot be cut
   // edgesCut - set of edges that have been cut
   do
   {
       // cut numOfEdges edges in CFG
       // do not cut edges from nonCutEdges
       edgesCut = cutEdges(nonCutEdges, numOfEdges);
       connectedComponents = getConnectedComponents(edgesCut);
       bringBackAllCFGEdges();
       numOfEdges *= 2;
   }
   while (connectedComponents.size() < target);

   return connectedComponents;
}
```

The above `do-while` loop will run at most $O(\log(\texttt{target}))$ times, each time eliminating twice as many edges as in the previous iteration. Each iteration starts with the full CFG with all the edges. The `cutEdges` function removes some edges from CFG, but avoiding these in the `nonCutEdges` set. In case when `target` is not achieved, the last `connectedComponents` list is returned.

There are two cases when a (directed) edge (A, B) in the CFG of F is added to the `nonCutEdges` set:

- B contains a *PHI* instruction related to A; we will call such edge a *PHI edge*
- B contains the *landingpad* instruction for A (see Appendix A.1.2).

Since *PHI* instructions are very common in IR, especially in optimized code, we cannot just add all of the *PHI edges* to `nonCutEdges`, since otherwise we could only get some trivial CC decompositions consisting of only a few components. We cannot cut them either, so the following solution is implemented.

First, for each *PHI edge* (A, B), an empty BB (X) is inserted on this edge. So, instead of the edge (A, B) in CFG, we get two edges: (A, X) and (X, B). Then, the *PHI* instruction at B addresses X instead of A. This means that we can add (X, B) to `nonCutEdges` and we can freely cut (A, X). So, instead of cutting (A, B) from the original CFG, we cut (A, X) in the modified CFG.

In the second case, when B contains the *landingpad* instruction for A, we cannot use the same trick - llvm demands that the BB containing the *landingpad* instruction has to be preceded by the block with an *invoke* instruction. Although an extra BB containing that *invoke* instruction could be potentially inserted between A and B as before (and the *invoke* instruction removed from A), exception handling code is much less common than code utilizing *PHI* instructions, so we can just add (A, B) to `nonCutEdges` without worrying about the number of CCs returned by `partitionCFG` getting too small.

4.7 Liveness analysis - pseudocode

Liveness analysis algorithm that is implemented in *VMObf* utilises four important variable sets:

- `in[bb]` : these are the variables live at the beginning of the basic block `bb`.
- `out[bb]` contains the variables which are live at the end of `bb`.

- `use[bb]` is the set of variables used in `bb`. By “used", we mean ”accessed before defined".
- `def[bb]` contains the variables that are defined in `bb`. By “defined" we mean ”possibly assigned a value".

The algorithm could be expressed in pseudocode as follows:

```
livenessAnalysis(Function& F)
{
    // worklist - a list with all bbs for which out has changed
    // since we processed them last time
    foreach (bb : F.getBasicBlockList())
    {
        computeUseAndDef(bb);
        worklist.insert(bb);
    }
    while (!worklist.empty())
    {
        bb = worklist.pop();
        newOut = empty_list();
        for (succ : successors(bb))
            newOut += in[succ];
        out[bb] = newOut;

        newIn = (out[bb] - def[bb]) + use[bb];
        if (in[bb] != newIn)
        {
            for (pred : predecessors(bb))
                worklist.insert(pred);
        }
        in[bb] = newIn;
    }
    return (in, out, def, use);
}
```

Chapter 5

Evaluation

This chapter contains an evaluation of *VMObf* in several areas, including obfuscation quality and time and memory usage compared to unobfuscated executables. The last section contains final remarks, along with information on how *VMObf* can be extended in order to further improve obfuscation quality.

5.1 Resistance to known attacks

This section revisits the three known attacks targeting virtualization mentioned in the second chapter and describes whether they are effective against *VMObf*.

The *frequency* attack is completely ineffective when dealing with the code obfuscated with *VMObf*. It is due to the fact that the instructions used inside virtual machines created by *VMObf* correspond to many assembly instructions, often with nontrivial control flow inside. The adversary just does not have access to the set of instructions that VM instructions are mapped into, and the instruction set is unique per VM - the only thing that can be inferred from frequency analysis is how often each CC is jumped into, but that does not help much.

Rolles's approach could work, but it would require modifications in order to obtain the approximation of the original function control flow. In order to perform this attack, the adversary would have to learn the *VMObf* architecture, in particular, how the obfuscated function control flow is determined. Then, he could obtain the mapping between VM registers and original local variables, even though some register variables can be stored in different hardware registers when used in different CCs.

The attack performed by *Synthia* tool is designed for virtual machines

that encode relatively simple arithmetic operations and, as such, would not be able to classify the instructions of VMs created by *VMObf*, unless they are very small basic blocks, which is the case only for some trivial functions.

5.2 Requirements and how to run *VMObf*

In order to work correctly, *VMObf* needs the IR generated from the program source code. It depends on two tools: `llvm-11` and `clang++-11` (for C++), so they need to be installed before *VMObf* can be used.

For C++ programs, the IR representation may be generated using the following command:

```
clang++-11 -S -emit-llvm source.cpp -o source.ll
```

Then, it is possible to modify `source.ll` using the obfuscation pass by running:

```
opt -load-pass-plugin ./libVMObf.so -passes=MdlPass,MdlPass2
    -S -o obf.ll,
```

where `opt` is the llvm optimization tool [15] commonly used for running passes in llvm. `MdlPass` and `MdlPass2` are the two obfuscation passes described in this chapter. Finally, in order to obtain an executable from the obfuscated code from `obf.ll`, one can run:

```
llc-11 -filetype=obj obf.ll,
```

and then

```
clang++-11 -Wl,-s obf.o -o obf.x
```

There are two utility bash scripts provided, that run *VMObf*:

- `runVMObf.sh`
- `runVMObf_optimized.sh`

Both expect a file with `.cpp` extension as their only argument.
`runVMObf_optimized.sh` runs `clang` with `-O3` optimization option.

5.3 Evaluation programs

In order to measure both time and memory complexity of resulting obfuscated executables, three different `C++` programs were written:

- square *matrix multiplication* using classical $O(n^3)$ algorithm,
- a program computing *preorder* order in (sparse) graph using DFS,
- *stl sort*, sorting integers.

The first one tests how much virtualization done by *VMObf* slows down the target program. During the obfuscation process, the original CFG is decomposed into many connected components, so many additional jumps and variable reads / writes are added. Especially the latter may be costly when many operations (and thus many VM instruction switches) are performed.

The program computing *preorder* order uses recursion and thus tests the stack usage of the obfuscated programs - each virtualized function has more variables than the original one and this overhead is especially visible when using recursion.

The third program tests how well obfuscated programs behave when operating on huge data. Inputs vary from 10 million up to 100 million elements to sort.

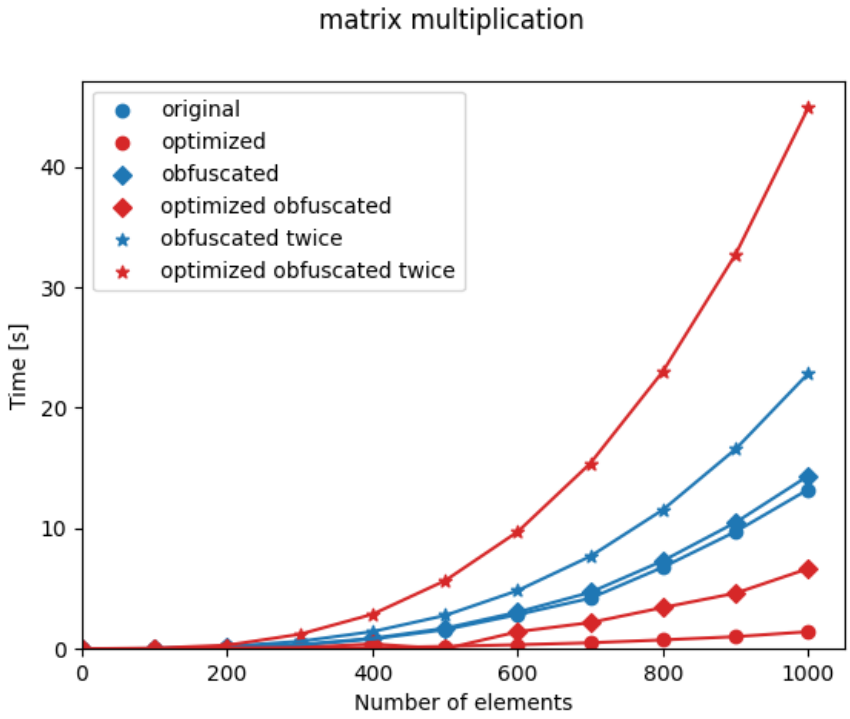
All programs are compiled both with `-O3` `clang` optimization option and without any optimizations.

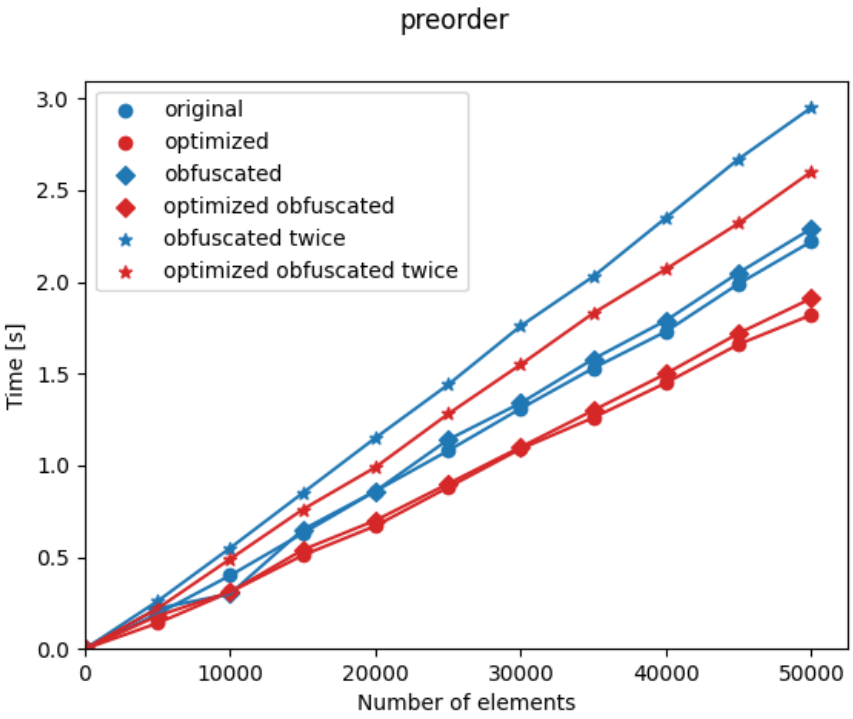
The obfuscation takes place on *all* functions present in the module, including `C++ stl` functions, such as, for example, `std::vector::push_back`.

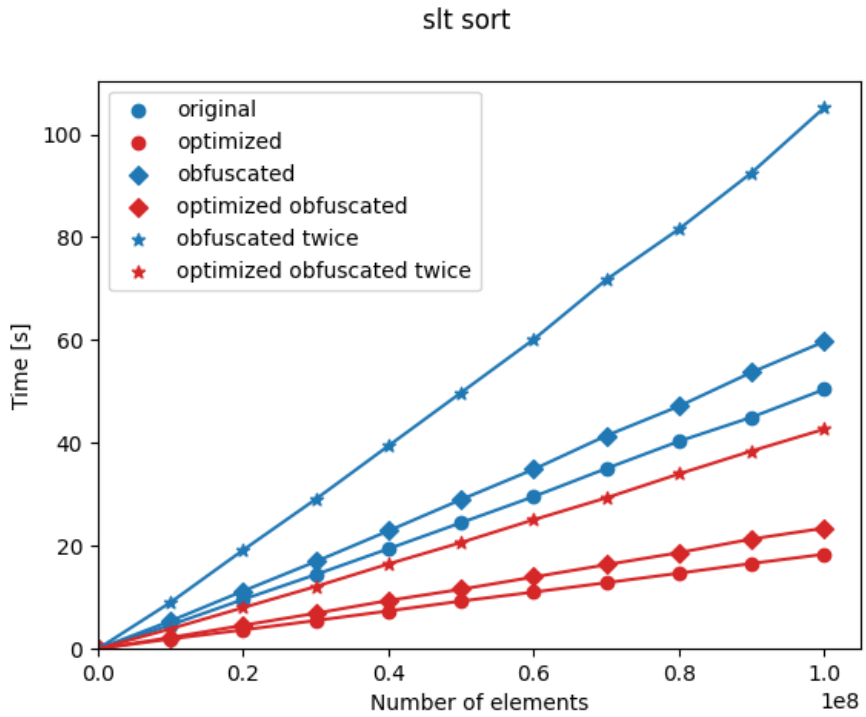
All these programs are obfuscated once and twice. The second approach gives better obfuscation (VM virtualized using another VM), but incurs higher time and memory usage.

5.4 Time penalty

In this section, the details regarding the time overhead in programs obfuscated by *VMObf* are given. The following plots illustrate how much time was consumed by each program.


matrix multiplication
original
optimized
obfuscated
optimized obfuscated
obfuscated twice
optimized obfuscated twice
Time [s]
Number of elements
0
10
20
30
40
0
200
400
600
800
1000


preorder
original
optimized
obfuscated
optimized obfuscated
obfuscated twice
optimized obfuscated twice
Time [s]
Number of elements
0.0
0.5
1.0
1.5
2.0
2.5
3.0
0
10000
20000
30000
40000
50000



As may be seen, in the *matrix multiplication* program, the difference between obfuscated and unobfuscated program is negligible. The difference is more noticeable, however, when the obfuscation is performed twice; on the largest input data, the program runs almost 1.5 times slower than the unobfuscated variant.

The optimized obfuscated version works 4-5 times slower than the original optimized version, but still 2 times faster than the original unoptimized program. The doubly obfuscated version however is a lot slower; for the largest data, the slowdown is huge - 32 times. The reason for that is that optimized code generated by `clang` tends to have many *register* variables. The obfuscation algorithm creates many of them as well, in order to maintain the SSA form, which was described in the previous chapter. Each obfuscation stage, however generates an additional *stack* variable for each of them and then the code updating these variables is added to the executable. Hence, there are many new memory read / write operations inserted (more than in case of unoptimized code) and the execution time increases proportionally.

In the *preorder* case, the slowest program is the one resulting from double obfuscation performed on the unoptimized version. The optimized version obfuscated once is again faster than the original program without optimizations.

In the *stl sort* case, all optimized obfuscation versions run faster than the unoptimized ones. The biggest overhead may be observed in doubly obfuscated original program, and for the largest input it equals 2.5.

In all three programs presented above, there was a very small time difference between the original programs and these programs obfuscated once. The same holds for their optimized versions. This means that the obfuscation performed by *VMObf* has small time overhead, which is barely noticeable and can be performed almost for free. The only issue may be related to memory consumption, which is discussed in the next section.

5.5 Memory penalty

As emphasized in the previous chapter, there is a memory overhead related to *VMObf*. Nevertheless, it is only visible when some of the obfuscated functions use recursion a lot.

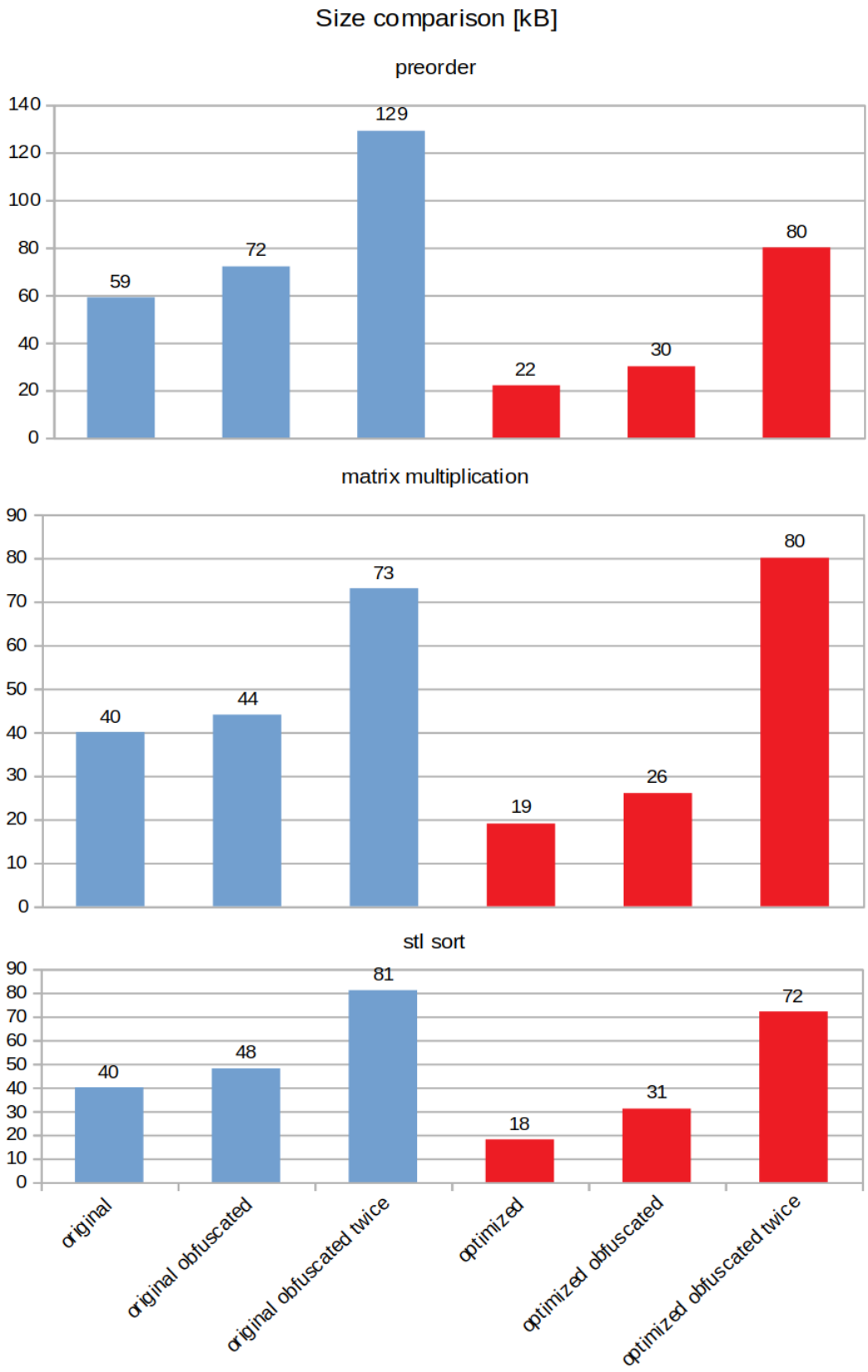
It is reflected in the *preorder* program: when the data provided is too large, or the process stack limit is decreased, obfuscated programs may fail due to stack overflow. The memory used by a function depends on local variable and basic block count, since in order to maintain the SSA form, additional variables are created. The final memory overhead is determined by a CC partition and by how many of *register* variables will be placed on the stack by the llvm optimizer.

That penalty applies to stack memory only - there is no change in the memory allocated on the heap. Hence, when the target program does not use deep recursion, *VMObf* obfuscation may be performed without any memory issues.

5.6 Executable sizes and basic block counts

5.6.1 Size comparison

The following chart illustrate how much the obfuscation done by *VMObf* increases executable sizes.


Size comparison [kB]
preorder
140
120
100
80
60
40
20
0
59
72
129
22
30
80
matrix multiplication
90
80
70
60
50
40
30
20
10
0
40
44
73
19
26
80
stl sort
90
80
70
60
50
40
30
20
10
0
40
48
81
18
31
72
original
original obfuscated
original obfuscated twice
optimized
optimized obfuscated
optimized obfuscated twice

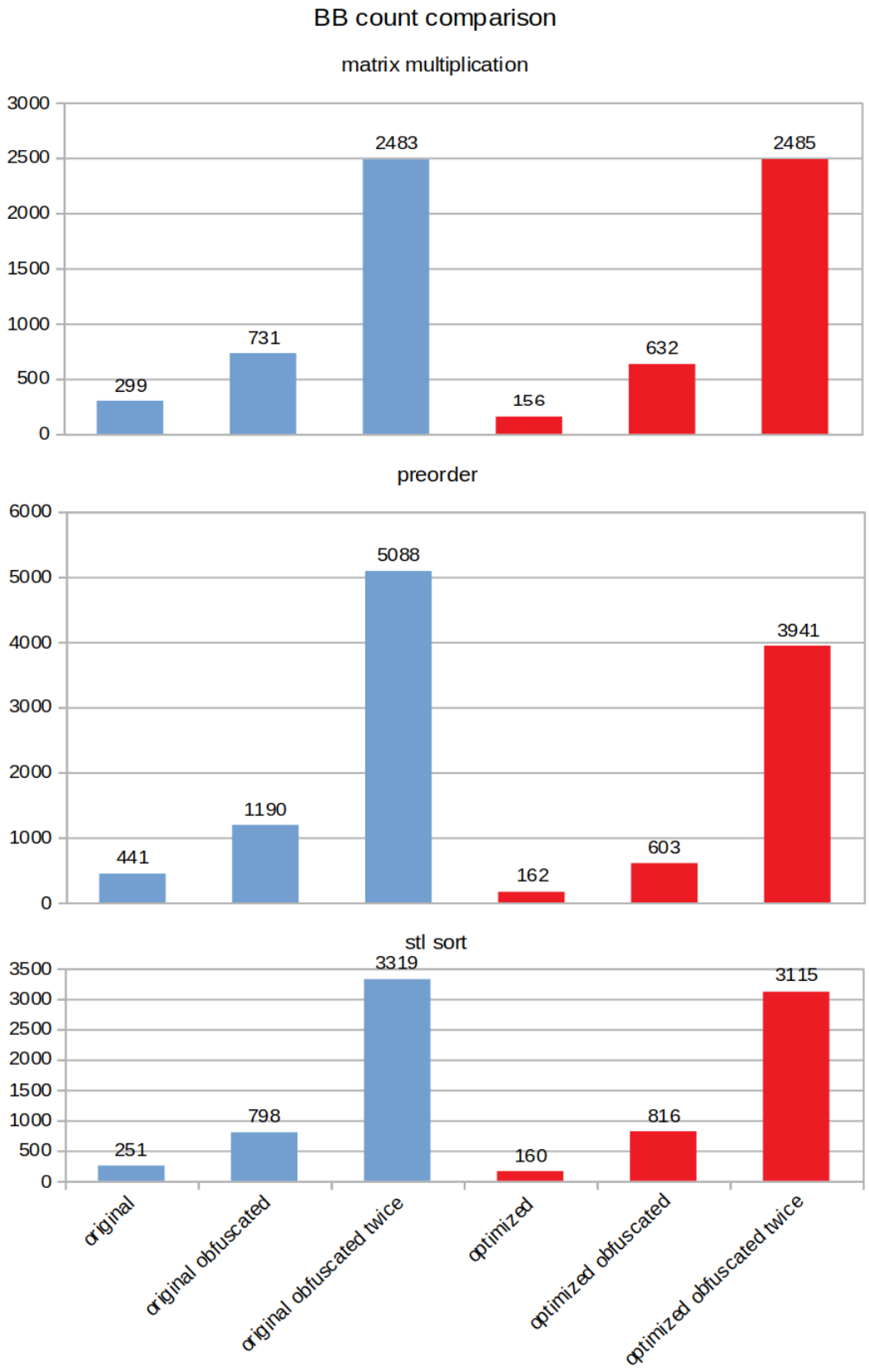
As in the case of time complexity, the programs obfuscated twice occupy considerably more space on disk than the original ones. Again, the reason is that the code for updating function variables takes significant amount of space. It makes the code much less readable, but it increases executable size as well, up to a factor of 4, as in the case of *stl sort*.

On the other hand, when the obfuscation is performed only once, the size increases by a factor of 1.33 on average: by 1.17 in case of unoptimized and by 1.48 in case of optimized programs.

In all presented programs, obfuscation is applied to all functions in the module. When performed only on the most vulnerable functions, it would not increase the executable size that much.

5.7 Basic block count comparison

In this section, data regarding the number of basic blocks in the programs is provided. The number of BBs present in the resulting IR is provided - it is easier to compute, and yet strongly corresponds to the number of BBs in the resulting executable. The chart below list the entire number of BBs present in each version of the three programs being evaluated.


BB count comparison
matrix multiplication
3000
2500
2000
1500
1000
500
0
299
731
2483
156
632
2485
preorder
6000
5000
4000
3000
2000
1000
0
441
1190
5088
162
603
3941
stl sort
3500
3000
2500
2000
1500
1000
500
0
251
798
3319
160
816
3115
original
original obfuscated
original obfuscated twice
optimized
optimized obfuscated
optimized obfuscated twice

As may have been expected, the number of basic blocks increases with the number of obfuscations performed. When the program is obfuscated twice, the number of BBs is on average 15.56 times higher than in the original program, meaning that the extra BBs added by the obfuscator contain 93.5% of all basic blocks present in the program. When the obfuscation is performed only once, this ratio drops to 71.7%.

The number of connected components into which the target function is partitioned before obfuscation closely corresponds to the number of BBs in the resulting IR. It can be set to any arbitrary value, but in the current implementation it is set to 1/3 of the total BB count, giving 3 basic blocks on average in each CC. Another parameter that has an influence on total BB count is the *basic block split ratio*, that is the maximal number of non-PHI instructions in a basic block - it is used at the beginning of obfuscation algorithm in order to spilt original basic blocks into smaller ones. This ratio is currently set to 5.

5.8 CFG comparison

This section shows differences between CFGs for the function multiplying two square matrices used in *matrix multiplication* program. Six CFGs are shown: one for each program version. The IDA disassembler [16] was used to produce their graphical representation.

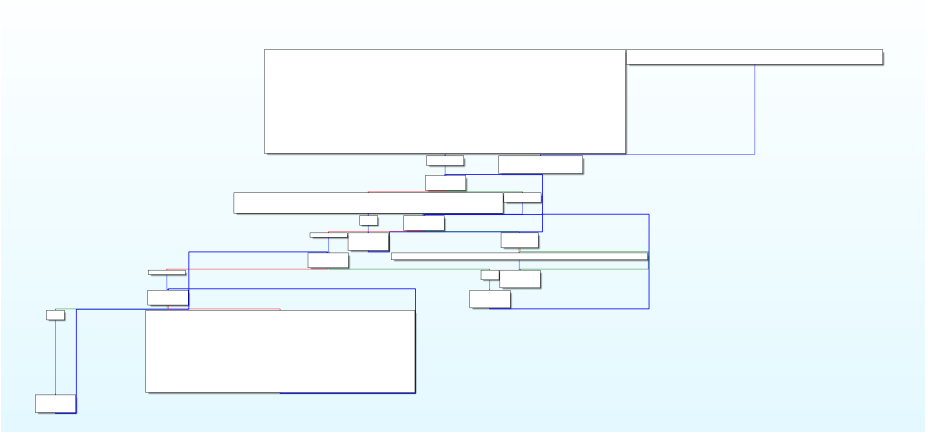


Figure 5.1: CFG of the original matrix multiplication function

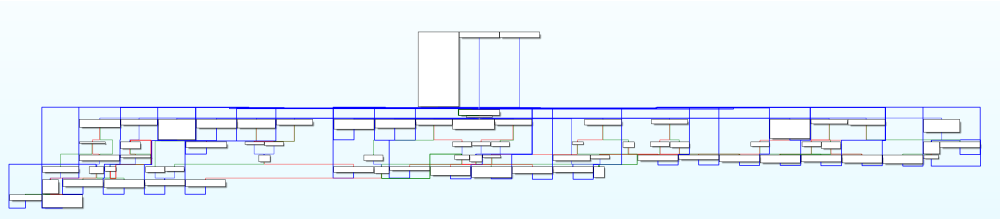


Figure 5.2: CFG of matrix multiplication function obfuscated once

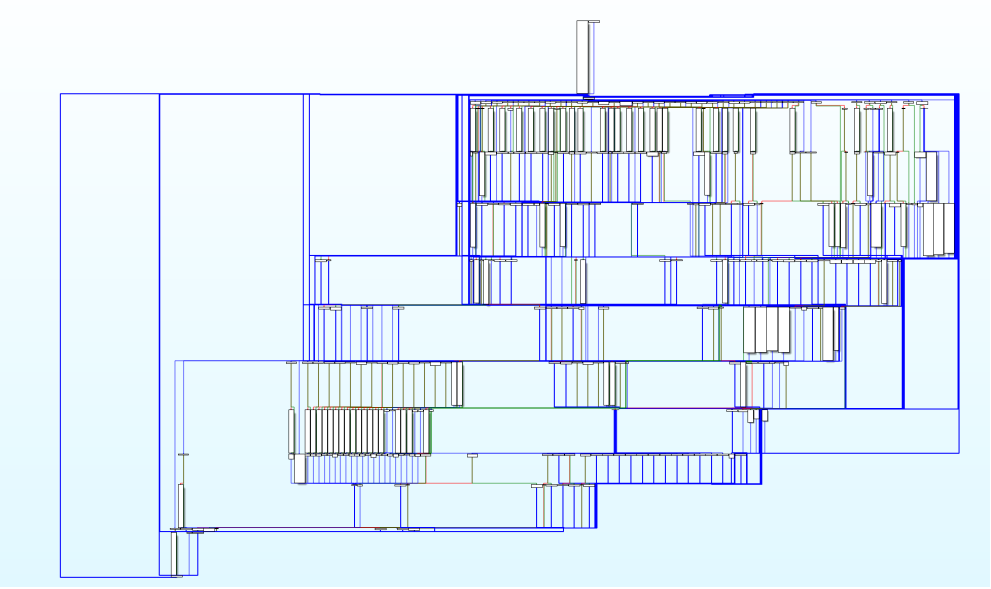


Figure 5.3: CFG of matrix multiplication function obfuscated twice

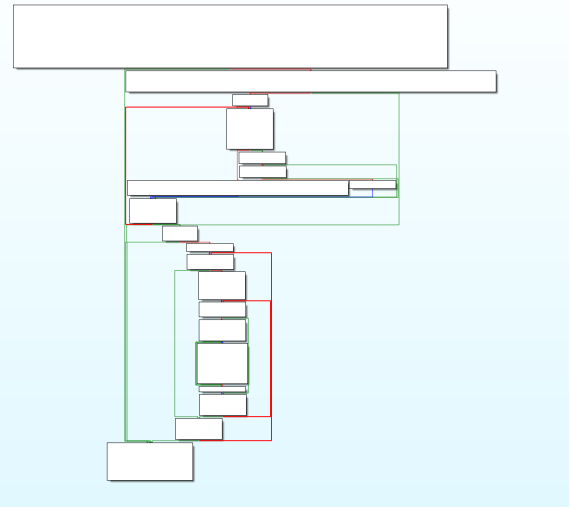


Figure 5.4: CFG of the original matrix multiplication function optimized

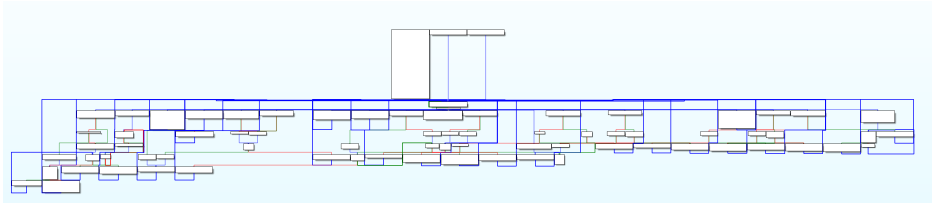


Figure 5.5: CFG of matrix multiplication function optimized and obfuscated once

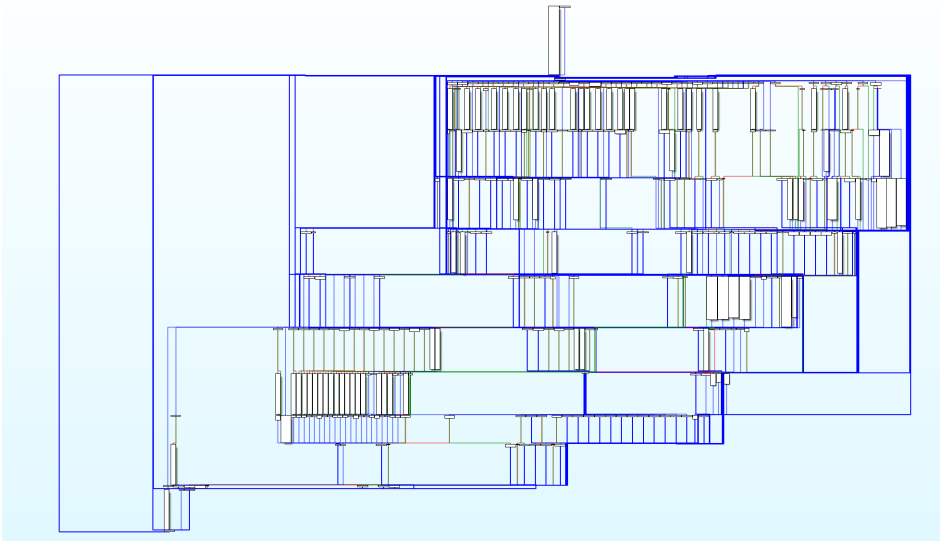


Figure 5.6: CFG of matrix multiplication function optimized and obfuscated twice

As one can observe, the obfuscation performed once already hides the original control flow and makes the resulting control flow a lot more complex. When obfuscation is performed twice, it seems impossible to manually analyse even a simple function.

Simpler functions may be obfuscated even more times. The figure below shows the CFG of a “Hello world" program obfuscated 5 times (a sequence of 5 nested VMs).

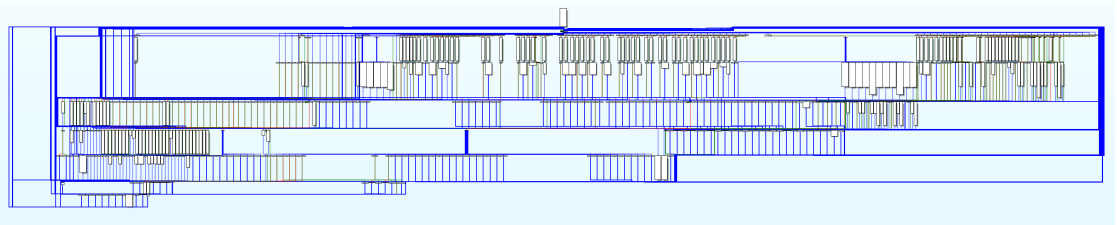


Figure 5.7: CFG of a "Hello world" program obfuscated 5 times

5.9 Possible extensions and enhancements

Although *VMObf* already provides decent obfuscation, it is still possible to extend its functionality and add more complex obfuscation.

The first thing that can be done is to obfuscate the virtual machines themselves. The current *VMObf* implementation keeps VMs as simple as possible, thus, for instance does not try to hide the values stored in reg_C and reg_{EB} registers. So, it is still possible to resolve the right control flow statically in some automated way. One thing that can be done in order to prevent it, is to encode the values stored in these registers in some less obvious way (for instance, put randomly looking numbers in reg_C and reg_{EB} and use some other function that will decode them when they are needed).

The next possible extension could merge *stack* variables together when possible, that is when there are no conflicts between them detected by *liveness analysis*. That could reduce the stack memory required by the program.

Another interesting idea would be to generate many versions of each VM instruction (or parts of it) and to decide at runtime which version should be executed, at random. This would generate exponentially many different execution traces, but also increase the executable size several times.

Finally, creating the VM operating at *module level* instead of the *function level* would give the best possible obfuscation. This would require much more work, however, since such a VM would have to implement its own way of dealing with function calls, return values and exception handling.

5.10 Final remarks

VMObf is an obfuscation tool using the *virtualization* obfuscation technique, working at the *function level* of llvm IR. For each function, it creates a VM with different, unique instruction set. It may be used to obfuscate programs written in several different programming languages on different operating

systems. The only requirement is to have `llvm` and a compiler producing llvm IR (for instance `clang` when working with C++ programs) installed.

It introduces only a small time overhead to the program being obfuscated. Despite giving a decent obfuscation itself, it is possible to combine it with other obfuscators working with llvm with almost no time penalty and only slight size and memory overhead.

Appendices

Appendix A

llvm - descirption

"The LLVM Project is a collection of modular and reusable compiler and toolchain technologies" [17]. It is a compiler framework that can be used to compile code written in many high level languages such as Ada, C, C++, Delphi, Objective-C and more [18].

During code compilation, llvm produces an *intermediate code representation* (aka *IR*). IR can be seen as a platform-independent assembly language, that can be used for optimizations, while not bothering with the original language the program was written in anymore. Thus, any change made at IR level can be done in isolation from the program source code. *VMObf* operates precisely at this level, making modifications only to the IR returned by llvm.

A.1 llvm IR

llvm Intermediate Representation is designed to be low level, but at the same time contains some features of high level languages. These include types, exception handling and access modifiers. Hence, operating at the IR level has a crucial advantage of being aware of variable types, function arguments and, in particular, all possible code paths that can be taken during program execution. However, that "high-level" nature of llvm IR has also a drawback - special care has to be taken during the obfuscation of this code to preserve this high level form. The next few paragraphs elaborate more on the most important IR features that one has to be aware of when writing an obfuscator for IR.

It is strongly typed, RISC (reduced instruction set computing) instruction set, and its virtual registers have to obey the static single assignment (SSA) form. It consists of many instructions resembling their `x86` counterparts, but usually contain more information, such as variable types, function

arguments, etc.

A.1.1 Local variables

Local variables of a function can be declared in two different ways: using `alloca` keyword or by assigning a value to a virtual register, which name is preceded by % symbol. Variables declared in the former way are allocated on the stack and are referred to as *stack* variables, while these represented by virtual registers are called *register* variables . *Register* variables are more efficient to use, since they may be stored in hardware registers during runtime, and are usually present in optimized code. Since llvm IR is a SSA language, each of the *register* variables has to be assigned exactly once, before any of its uses. The necessity of obeying the SSA form brings some difficulties while operating on virtual registers, especially when changing the CFG of a function.

A.1.2 Exception handling

llvm has its own mechanism for exception handling. There are several instructions responsible for that. These include:

- *landingpad*, which is responsible for catching, classifying the exception caught and branching to the relevant handler. It consists of three subinstructions: *catch*, *filter* and *cleanup*, which define the exceptions that should be handled inside it. The *landingpad* instruction has to be the first non-PHI instruction in the BB containing it.
- *invoke*, which is used instead of *call* when the function being called may throw an exception. It has to provide two BBs as possible continuation points: one when exception was not thrown, and one, containing a *landingpad* instruction, when an exception was thrown.

There are certain restrictions related to these instructions: the BB *landingpad* instruction has to be preceded by a BB containing the *invoke* instruction and there can be only one *landingpad* instruction per BB.

In order to support many different ways of exception handling present in high level languages, there is a mechanism for supplying *personality functions* which are responsible for checking whether the current *catch* catches the exceptions of the exception type thrown.

A.2 llvm API

A.2.1 llvm passes

In order to modify the llvm IR, one has to write a *pass* operating on it. Depending on its type, it can operate at different scopes, from small code fragments up to the entire module. There are multiple classes representing each of the available passes type, each inheriting from the `Pass` class [19]. Most interesting ones are listed below:

- `LoopPass` executes on each loop in the program. It processes loops from the most inner to the most outer one.
- `RegionPass` operates on each program *region*, that is, "a connected subgraph of a control flow graph that has exactly two connections to the remaining graph" [20]. Similarly to the `LoopPass`, the regions are processed in the nested order, starting from the most inner one.
- `FunctionPass` executes on each function of a program separately. It does not need to process functions in any particular order since it can only modify the function it currently processes.
- `ModulePass` is the most general pass available. It is capable of adding, modifying and removing functions to the program. Due to its flexibility, this is the pass implemented by *VMObf*.

Since the IR has to be changed during a *pass*, llvm has to provide an API in order to operate on it. The most important classes of this API are enumerated below.

- `Module` class represents the entire module and is used as an argument passed to the `ModulePass` handling function. It keeps track of all functions defined in the program and exposes them via the `Module::getFunctionList()` method. Additionally, it contains the list of global variables and libraries used by the module [21].
- `Value` class is the most important class in the llvm API. "It represents a typed value that may be used (among other things) as an operand to an instruction" [22]. It is a base class for many other important classes, such as `Constant`, `Argument`, `Instruction` and `Function`. It provides many functions to get / change operand names, check their types, or get their use list. The ones most frequently used in *VMObf* are:
  - `Value::setValueName(ValueName*)`
  - `Value::replaceAllUsesWith(Value*)`
  - `Value::getType()`

- `Instruction` class is the common base class for all instructions in the IR [22]. It is used in *VMObf* while iterating through function instruction list and when *liveness analysis* is performed.
- `Function` represents a single function. It keeps track of all BBs inside it, the list of arguments, and a symbol table [22].
- `BasicBlock` class represents a single BB. It maintains the list of all instructions that form it. The most important methods of this class include:
  - `BasicBlock::getInstList()`
  - `BasicBlock::getFirstNonPHI()`
  - `BasicBlock::insertInto(Function*)`

Additionally, the API contains several utility functions that are often used in *VMObf*:

- `isa<X>(val)`, that returns true if the parameter `val` is an instance of the template parameter `X`. By using this function, it is possible to check, for example, a type of some instruction (whether it is a PHI instruction) or whether some `Value` is a function argument.
- `predecessors(BasicBlock*)` / `successors(BasicBlock*)` return iterable collections of predecessors (or successors, respectively) of the BB.

Bibliography

[1] BSA Global Software Survey *https://globalstudy.bsa.org/2016/*, from 16.06.2021

[2] Cheng, X., Lin, Y., Gao, D., Jia, C.: *DynOpVm: VM-based Software Obfuscation with Dynamic Opcode Mapping*, 2006

[3] Rolles, R.: *Unpacking Virtualization Obfuscators*, 2009

[4] Blazytko, T., Contag, M. et al.: *Syntia: Breaking State-of-the-Art Binary Code Obfuscation via Program Synthesis*, 2018

[5] VMProtect website, available at *https://vmpsoft.com/*, accessed 17.06.2021

[6] NoVMP github repository, available at *https://github.com/can1357/NoVmp*, accessed 17.06.2021

[7] VTIL github repository, available at *https://github.com/vtil-project/VTIL-Core*, accessed 17.06.2021

[8] Themida website, available at *https://www.oreans.com/Themida.php*, accessed 17.06.2021

[9] Turčan. L,: *LLVM Obfuscator Based on Virtual Machines with Custom Opcodes and String Encryption*, 2019

[10] rewolf github repository, available at *https://github.com/rwfpl/rewolf-x86-virtualizer*, accessed 17.06.2021

[11] clang compiler website, available at *https://clang.llvm.org/*

[12] llvm docs, available at *https://llvm.org/docs/LangRef.html*, accessed 3.07.2021

[13] Braun M., Buchwald S., Hack S., Leißa R., Mallon C, Zwinkau A.: *Simple and Efficient Construction of Static Single Assignment Form*

[14] llvm *reg2mem* pass implementation, available at *https://llvm.org/doxygen/Reg2Mem_8cpp_source.html*

[15] llvm optimizer, available at *https://llvm.org/docs/CommandGuide/opt.html*, accessed 20.06.2021

[16] IDA official site, available at *https://hex-rays.com/ida-free/*, accessed 24.06.2021

[17] official llvm website, available at *https://llvm.org/*, accessed 18.06.2021

[18] llvm wikipedia page, *https://en.wikipedia.org/wiki/LLVM*, accessed 18.06.2021

[19] official llvm docs, available at *https://llvm.org/docs/WritingAnLLVMPass.html*, accessed 18.06.2021

[20] official llvm Region description, available at *https://llvm.org/doxygen/classllvm_1_1RegionBase.html*, accessed 18.06.2021

[21] official llvm Module description, available at *https://llvm.org/doxygen/classllvm_1_1Module.html*, accessed 18.06.2021

[22] official llvm Programmer's Manual, available at *https://llvm.org/docs/ProgrammersManual.html*, accessed 18.06.2021