Obfuscation of steel: meet my \textit{Kryptonite}

Axel "0vercl0k" Souchet

July 6, 2013

\textbf{Abstract}

For several months, I came across a lot of papers that use the LLVM framework to develop really cool tools like:

- decompilation framework (Dagger),
- universal deobfuscation (Opticode),
- bug-finding with static binary instrumentation (AddressSanitizer),
- fast C compiler (Clang),
- automatic test cases generator (Klee),
- etc.

In other words, LLVM is everywhere, and it's only the beginning.

In this paper, I will try in a first part, to give you an overview of the framework: basically what you can do with it and what you cannot. Then, I will introduce a PoC called \textit{Kryptonite}: a small obfuscater based on LLVM. We will talk about how you can build such tools and how they can be improved. I'm currently playing with the version 3.3 of LLVM (the latest when I'm writing this paper), so the code may changed a bit for the upcoming version (don't hesitate to shoot me an email if this is the case).

Keep in mind that no CPUs were harmed during this piece of research, trust me.

\footnote{Ironic, of course}
## Contents

1  LLVM’s overview 4  
1.1  Introduction 4  
1.2  The pipeline 5  
   1.2.1  Frontend 5  
   1.2.1.1  Emitting LLVM-IR via the C API 6  
   1.2.2  Transformation passes 9  
   1.2.3  Backend 11  
   1.2.4  Conclusion and going further 13  

2  Kryptonite 14  
2.1  Introduction 14  
2.2  Writing an optimization pass 14  
2.3  LLVM-IR obfuscation 15  
   2.3.1  Obfuscate add instructions 15  
   2.3.1.1  Theory: home made 32 bits adder 16  
   2.3.1.2  Practice: Emit the adder with the LLVM frontend API 17  
   2.3.2  Mess with other instructions 19  
   2.3.3  Inserting x86 assembly 20  
   2.3.4  Showcase: Kryptonite crackme 22
Chapter 1

LLVM’s overview

1.1 Introduction

The Low Level Virtual Machine project originally started years ago at the University of Illinois under the supervision of Vikram Adve and Chris Lattner (the maintainer). The purpose of the project was to build a framework to ease compiler and code generator writing. This infrastructure is written in C++ and is open-source: see the website llvm.org. But over the years, LLVM has really been improved by the community and Apple (mainly because they hired Chris and formed a team to work on LLVM): a lot of frontends are now available (C, C++, Objective-C, Ada, Haskell, etc.) and same thing for the backends (x86, x86_64, ARM, MBlaze, MIPS, PPC32, PPC64, Sparc, etc.). LLVM is also:

- Near from 2000 files (header and code files);
- Around 2300 classes;
- Around 770 000 lines.

Yes, it is quite a huge code base and a complex piece of software, and that is exactly the reason I wanted to write something about it. You have to spend hours to read the source code, to read tutorials and to debug your buggy code; I hope to give you enough materials to play safely with LLVM without reading tons of code :-). 

As I said, the purpose of this part is to go through some fundamental concepts and tools to understand how LLVM finally works. I will also try to give you some codes, some examples because that’s what matters. Keep in mind I am not an LLVM expert at all, I may misuse this wonderful tool; if this is the case don’t hesitate to shoot me an email, I will be glad to update the paper with the good way of how things should be done!

By the way, if you don’t want to compile yourself either the LLVM or the clang code, they have kindly uploaded already-compiled binaries here: Pre-built binaries, go grab one! I guess we are done for the introduction, make yourself comfortable, let’s go!
1.2 The pipeline

One of the LLVM's strengths is the modularity. It is made of essentially three very important parts: the frontend, the optimization passes, and the backend. Each of them has a very particular role in the compilation process, I will describe their roles in the following sections.

You can see each part as a black box that takes an input and produces an output, and usually that output is also the input of another black box: you can see that as a chain.

![Compilation process](image.png)

Figure 1.1: Compilation process

Note this design in three parts is not really new, the GNU Compiler already used this architecture.

1.2.1 Frontend

The frontend is the part you are interested in if you want to write a compiler, or if you want to tweak an existing compiler. This part takes in input a file that will be parsed by your frontend module, and your this module is responsible to generate the equivalent code using the LLVM Intermediate representation. The LLVM-IR is a really important thing: basically it is a language that will be used between the output of the frontend until reaching the input of the backend. This language aims to provide several important characteristics like:

- SSA-form based,
- type safety,
- low-level operations,
- simplicity,
- the capability of representing high-level languages.

To emit this LLVM-IR, you can use a dedicated API that will allow you to create instructions: if you want to see the type of instructions available the LLVM-IR read the LLVM Language
Reference Manual. If you want to see a real frontend, you can check Clang’s sources: this is maybe the most famous LLVM based frontend. Its role is to rewrite the C code into the LLVM-IR using the LLVM’s dedicated API I talked you about. As far as I know the frontend API is also available in C, in OCaml and in Python (check those slides: llvm-py, PyCon India, 2010) via bindings.

If you never seen the classical hello-world in LLVM-IR, here it is:

```c
@.str = private unnamed_addr constant [13 x i8] c"Hello world\0A\00", align 1
define i32 @main() {
  %1 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([13 x i8]* @.str, i32 /zero.noslash, i32 /zero.noslash))
  ret i32 /zero.noslash
}
declare i32 @printf(i8*, ...)
```

You can use your preferred language to write the hello-world, and then ask your frontend to output the LLVM-IR. To do that with clang you just have to run it like this:

```bash
$ clang -S -emit--lvm hello.c -o hello.il
```

Once you are able to generate this LLVM-IR you can use the rest of LLVM’s pipeline without modifications: building an ELF binary for SPARC is not a problem for example (because the SPARC backend already exists).

1.2.1.1 Emitting LLVM-IR via the C API

Before playing with the frontend API, you have to understand a bit how the API works. First, you have to know the core of LLVM is written in C++ (you can read it in the directory include/llvm) but they made also a C API built on the top of it (in the directory include/llvm-c).

Another important detail is when you are playing with LLVM you have to manipulate several type of containers, let me describe the main ones:

1. A module is a container of function and global variables, it is the equivalent of a .c file for example. This is really the top-level container used to store all the information of all other LLVM-IR objects. If you want to look at the declaration of the llvm::Module class see include/llvm/IR/Module.h,

2. A function is a container of basic-blocks: see the declaration of llvm::Function in include/llvm/IR/Function.h,

\(^1\)Of course you can do exactly the same with the C++ API, but in my opinion the C API is easier to understand :-).
3. A basic block is a container of instructions: the declaration of `llvm::BasicBlock` and `llvm::Instruction` are there: `include/llvm/IR/BasicBlock.h` and `include/llvm/IR/Instruction.h`.

As we said previously, the top-level container is the `llvm::Module` class, so let’s create one via `LLVMModuleCreateWithName` like that (and don’t forget to clean the memory with `LLVMDisposeModule` as said in the comments):

```c
LLVMModuleRef Module = LLVMModuleCreateWithName("module-c");
/// Do things with Module
LLVMDisposeModule(Module);
```

Once we have our module, we need to create a function via `LLVMAddFunction`; but if you look at its declaration you see that we need first to create the type of our function. The type of a function is the number and the type of its arguments, and the type of its return value. Let’s define the type of our main function with `LLVMFunctionType`, and add it to our module:

```c
/// void main(void)
LLVMTypeRef MainFunctionTy = LLVMFunctionType(
    LLVMVoidType(),
    NULL,
    0,
    false
);
LLVMValueRef MainFunction = LLVMAddFunction(Module, "main", MainFunctionTy);
```

Before going further, we still need to import somehow, the `printf` function:

```c
/// extern int printf(char*, ...)
LLVMTypeRef PrintfArgsTyList[] = { LLVMPointerType(LLVMInt8Type(), 0) };
LLVMTypeRef PrintfTy = LLVMFunctionType(
    LLVMInt32Type(),
    PrintfArgsTyList,
    0,
    true
);
LLVMValueRef PrintfFunction = LLVMAddFunction(Module, "printf", PrintfTy);
```

Now, we can instantiate a basic block via `LLVMAppendBasicBlock`, and create a builder via `LLVMCreateBuilder`. A builder is an object that helps you to create LLVM-IR instructions: you specify in which basic block you want to add an instruction, and you ask the builder to create one: convenient for us.

```c
// An instruction builder represents a point within a basic block and is
// the exclusive means of building instructions using the C interface.
```
Perfect, we are now ready to insert real instructions. For the classic hello-world we just need to add a global variable that will hold our string, to build a call-like instruction, and a ret-like instruction (all basic blocks must be terminated by a branch instruction). Again, the C API is very simple to use:

```c
LLVMValueRef Format = LLVMBuildGlobalStringPtr(
    Builder,
    "Hello, %s.\n",
    "format"
), World = LLVMBuildGlobalStringPtr(
    Builder,
    "World",
    "world"
);

/// printf("Hello, %s!", world);
LLVMValueRef PrintfArgs[] = { Format, World };

LLVMBuildCall(
    Builder,
    PrintfFunction,
    PrintfArgs, 2,
    "printf"
);

/// return;
LLVMBuildRetVoid(Builder);
```

Now, we need to compile our hello-world frontend with clang++ like this:

```
$ clang++ -xc llvm-c-frontend-hello.c 'llvm-config --cxxflags --ldflags --libs' -o ./llvm-c-hello
$ ./llvm-c-hello
@format = private unnamed_addr constant [12 x i8] c"Hello, %s.\D\A\00"
@world = private unnamed_addr constant [6 x i8] c"World\00"
declare i32 @printf(...) define void @main() {
    entrypoint:
    %printf = call i32 (...)* @printf(
        i8* getelementptr inbounds ([12 x i8]* @format, i32 0, i32 0),
        i8* getelementptr inbounds ([6 x i8]* @world, i32 0, i32 0)
    )
    ret void
```
You can even use the tool *lli* (we will talk more about this tool in the backend part) to really execute the LLVM-IR code we just emitted:

```
$ ./llvm-c-hello 2>&1 | lli
Hello, World.
```

OK so now you know a bit more about how a LLVM frontend looks like. If you want another example, I have made the `strlen` function to see how to build if/else branches: `llvm-c-frontend-playing-with-ir.c`. Also writing a frontend for a toy language like those ones is a cool exercise: whitespace, piet, shakespeare, etc.

### 1.2.2 Transformation passes

Basically, transformation passes can be of two types: either it really transforms the program (a transform pass), or either it’s only reading and collecting information about your code (an analysis pass). For example, it exists a pass called "dot-cfg-only" that generates the CFG of each function you have in your LLVM-IR file; this is an analysis pass:

![CFG-only of main](image)

Figure 1.2: CFG-only of main

But at the opposite, you can also have passes that will do real optimization or transformation of your code: for example the "Dead Code Elimination" pass. It will go through your LLVM-IR
code to find unreachable piece of code to remove them in order to simplify the program. If you want to look at the source code of the different passes, you can check the lib/Analysis directory for analysis passes, and the lib/Transforms for the transform passes.

When you are playing with this part of the pipeline, the important tool to know is opt: the LLVM optimizer. It is the tool that will apply the different passes you want, and will give you the optimized LLVM-IR code. Of course, it is also possible to extend its functionalities by writing new passes; the tool is able to load dynamically your pass and to execute it to apply some analysis or transformation operations. You can enumerate all the available passes by calling this command:

```
$ opt --help
OVERVIEW: llvm .bc -> .bc modular optimizer and analysis printer
[...]
Optimizations available:
- aa-eval - Exhaustive Alias Analysis Precision Evaluator
- adce - Aggressive Dead Code Elimination
- alloca-hoisting - Hoisting alloca instructions in non-entry blocks to the entry block
[...]
```

On my machine I can count exactly 157 different passes. As an example, we can try to optimize the generated LLVM-IR code for the `strlen` function I gave you in the previous part (llvm-c-frontend-playing-with-ir.c). Here is the code generated by our frontend:

```
$ cat strlen.ll
define i32 @strlen(i8* %s) {
init:
  %i = alloca i32
  store i32 /zero.noslash, i32* %i
  br label %check

check: ; preds = %body, %init
  %/zero.noslash = load i32* %i
  %1 = getelementptr i8* %s, i32 %/zero.noslash
  %2 = load i8* %1
  %3 = icmp ne i8 /zero.noslash, %2
  br i1 %3, label %body, label %end

body: ; preds = %check
  %4 = load i32* %i
  %5 = add i32 %4, 1
  store i32 %5, i32* %i
  br label %check

end: ; preds = %check
  %6 = load i32* %i
  ret i32 %6
}
```

The function is really simple: it loops until it finds a null byte and meanwhile it increments a counter to have the len of the string. Now let’s launch opt to optimize the previous code:
$ opt -S -p -O3 strlen.ll
; ModuleID = 'strlen.ll'

; Function Attrs: nounwind readonly
define i32 @strlen(i8* nocapture %s) {
  init:
    br label %check
  check: ; preds = %check, %init
    %storemerge = phi i32 [ /zero.noslash, %init ], [ %3, %check ]
    %0 = getelementptr i8* %s, i32 %storemerge
    %1 = load i8* %/zero.noslash
    %2 = icmp eq i8 %1, /zero.noslash
    %3 = add i32 %storemerge, 1
    br i1 %2, label %end, label %check
  end: ; preds = %check
    ret i32 %storemerge
}

We can clearly see the code has been quite optimized by the utility using the "Phi nodes". In this specific case you can understand the instruction as "if the execution flow comes from the basic block init, the value zero is moved in the variable %storemerge ; if it comes from the basic block %check, the variable %3 is moved in %storemerge.

In the second part of the paper, we will talk more in details about how you can write your own pass.

1.2.3 Backend

The last part of the pipeline is the backend: it is basically the software component that will traduce the LLVM-IR into the machine code for a specific CPU. We can have a list of the stable and already existing backend available in LLVM by using the tool llc (the LLVM compiler):

$ llc --version
LLVM (http://llvm.org/):
LLVM version 3.3
Optimized build.
Default target: i386-pc-linux-gnu
Host CPU: corei7

Registered Targets:
aarch64 - AArch64
arm - ARM
cpp - C++ backend
hexagon - Hexagon
mblaze - MBlaze
mips - Mips
mips64 - Mips64 [experimental]
mips64el - Mips64el [experimental]
mipsel - Mipsel
msp430 - MSP430 [experimental]
This tool is very handy: you give it an LLVM-IR module for example and it is capable of generating the assembly according to the target you have chosen. As an example, we can try to compile our `hello-world` LLVM-IR program into x86 and MIPS:

```
$ llc hello.ll -march=mips -o hello.mips.s
$ llc hello.ll -march=x86 -o hello.x86.s
$ cat hello.x86.s

main:       # @main
    .cfi_startproc
    # BB#/zero.noslash: # %entrypoint
    subl $12, %esp
    .Ltmp1:
    .cfi_def_cfa_offset 16
    movl $.Lworld, 4(%esp)
    movl $.Lformat, (%esp)
    call printf
    addl $12, %esp
    ret

$ cat hello.mips.s

main:     # [...]
# BB#/zero.noslash: # %entrypoint
    lui $2, %hi(_gp_disp)
    addiu $2, $2, %lo(_gp_disp)
    addiu $sp, $sp, -24
    $tmp2:
    .cfi_def_cfa_offset 24
    sw $ra, 20($sp)  # 4-byte Folded Spill

# BB#0:     # %entrypoint
    lw $1, %got($format)($gp)
    addiu $4, $1, %lo($format)
    lw $1, %got($world)($gp)
    lw $25, %call16(printf)($gp)
    jalr $25
    addiu $5, $1, %lo($world)
    lw $ra, 20($sp)  # 4-byte Folded Reload
    jr $ra
    addiu $sp, $sp, 24
```

Of course, once you got those assembly files you can just use whatever compiler you like to generate an executable binary. Here is an example with `clang`:

```
$ clang hello.x86.s -o hello
$ file hello
hello: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared libs),
for GNU/Linux 2.6.26, not stripped
$ ./hello
Hello, World.
```

If you have to create your own CPU target, this is surely the hardest part: it exists some really good tutorials but creating its own backend (even for a toy-cpu) is clearly tough.

Another other interesting part, is the JIT compiler engine that you can use directly via the `lli` tool. This tool allows you to take an LLVM-IR file, to JIT compile the code and to directly execute it. Basically, if you are on an x86 host computer, the `lli` program will JIT compile the code using the x86 backend and will run it. We can try to execute our `hello-world` program:

```
$ lli hello.ll
Hello, World.
```

### 1.2.4 Conclusion and going further

As you can see previously, LLVM is a really cool set of libraries to implement compiler or JIT compiler. What’s really nice is to be able to only implement only the part you need and once it is done you can use what already exists: frontends, optimization passes or backends. Though some parts may be a bit obscure and not really trivial to play with, that’s why I did a little list of interesting links you should read if you definitely want to go further (yeah, it was only a small introduction):

- **Kaleidoscope: Implementing a Language with LLVM**: if you want to write a frontend, read this, it’s perfect
- **Writing an LLVM Pass**: it gives you the basics to write your own analysis/transform pass
- **Creating an LLVM Backend for the Cpu0 Architecture**: this one focus the backend part; it’s very tough but it is a really good tutorial
Chapter 2

Kryptonite

2.1 Introduction

The first time I saw the LLVM’s pipeline picture, I was really interested in the LLVM-IR and by the passes parts. It is clearly here you want to play if you are interested in obfuscation, because you deal with the LLVM-IR and not the target CPU assembly. Basically it means your obfuscation can be reused by all the backends supported by LLVM. You can simply write your code in C, then you ask clang to generate the LLVM-IR code and from there you can really transform the LLVM-IR the way it pleases you. Once you are done: you just have to compile it for the CPU you target. Usually, when we see obfuscaters either the authors are modifying the source (and it is usable only for one language), or either it does the obfuscation at the assembly level: in this case it is CPU specific (you have also lot of problems with the instruction side-effects). In our case, you can use the language you want, among the available LLVM frontends of course, and your obfuscation ideas can be reused for others targets. You will see in that part we will not need to hack clang’s sources, or to mess with the code’s AST to generate heavy obfuscated binaries.

The purpose of this part is just to show you the small PoC I have written for the fun. You will, of course, find the sources of the project on my github account. By the way, I have prepared a little crackme that has been obfuscated with my tool Kryptonite to illustrate what type of binaries it can produce.

2.2 Writing an optimization pass

Before talking about the obfuscation part, we need to know how you are supposed to build an optimization pass. In a nutshell, it is a simple shared dynamic libraries that will be loaded by opt, the LLVM optimizer. If you read carefully, the Writing an LLVM Pass tutorial on the LLVM’s wiki, you see that a pass can be involved at several levels. By levels, I mean that you can create a pass to optimize basic blocks, to transform functions, or to optimize a whole module. To do so, you have to subclass the according LLVM class and to implement some specific routines:
Let’s try to make a hello-world pass that will display the name of each function. As I said, earlier we need to subclass llvm::FunctionPass and to implement the pure virtual llvm::FunctionPass::runOnFunction method.

```cpp
struct Hello : public llvm::FunctionPass {
  static char ID;

  Hello() : FunctionPass(ID) {}

  bool runOnFunction(llvm::Function &F) {
    printf("Function being handled: %s\n", F.getName().data());
    return false;
  }
};
```

Then you can compile it, and run it through the LLVM optimizer via those commands:

```
$ clang++ hello.cpp `llvm-config --cxxflags --ldflags --libs core` -shared -o hello.so
$ opt -load ./hello.so -help | grep hello
```

OK, now you know how to build a really basic pass. The other part is to play with the different containers I presented a bit earlier: you add instructions, you split basic blocks, you remove instructions, you insert new basic blocks; it’s simple, you just have to find the right API. Don’t hesitate to check my sources, I have examples of how you can change the CFG of a function, how to insert/split basic blocks etc.

### 2.3 LLVM-IR obfuscation

The purpose of this section is to focus on the obfuscation, to discuss what I have implemented, and to see how we could improve the PoC.

#### 2.3.1 Obfuscate add instructions

My idea was quite simple, I wanted to recode the equivalent of an add instruction but without addition. The add instruction is really important because it is used in most of all programs, and
if you think about it you can even transform some instructions to use \textit{add} instructions instead; we will see those cases a bit later.

2.3.1.1 Theory: home made 32 bits adder

I am pretty sure, almost all of you have already studied this younger: how to make a full 32 bits adder with only logic operators. But to do that, we have first to implement a full 1 bit adder. As you can see in the picture 2.1, a 1 bit adder system has 3 inputs:

- \( A \): the first bit you want to add
- \( B \): the second bit you want to add
- \( C_{\text{in}} \): the input carry (useful when chaining several 1 bit adders)

\[ \begin{array}{c}
\text{Figure 2.1: Full 1 bit adder (source: wikipedia.org)} \\
\end{array} \]

And it has 2 outputs:

- \( S \): the solution of the addition \((A + B + C_{\text{in}})\)
- \( C_{\text{out}} \): the output carry (this one will be introduced in the input carry of another adder)

Writing the truth table of this system gives the following:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Cin</th>
<th>S</th>
<th>Cout</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Now if you extract both the equations of $S$ and $C_{out}$, you get those ones:

- $S = A\overline{B}C_{in} \lor A\overline{B}C_{in} \lor \overline{A}BC_{in} \lor ABC_{in}$
- $C_{out} = A\overline{B}C_{in} \lor ABC_{in} \lor \overline{A}BC_{in} \lor ABC_{in}$

The watchful readers will see that the second equation is not simplified, and you can ask yourself why? That’s simple, we want to produce the most awful code possible, so we really don’t want to simplify it. Now we have those equations, we are able easily to write a system capable of adding two bits, that’s cool.

The plan now is to make a chain of 1 bit adder in order to have a real 32 bits adder like in the picture 2.2 (but with 32 blocks instead of 4).

![Figure 2.2: Full 4 bits adder (source: wikipedia.org)](image)

So this was the theory part, because we have to implement the 32 bits adder using the frontend API of LLVM to emit it in LLVM-IR as we did for the hello-world example in the first part.

### 2.3.1.2 Practice: Emit the adder with the LLVM frontend API

Let’s describe how to write a 1 bit adder in LLVM-IR. We need the two outputs described earlier, but we will focus on the $S$ one ($C_{out}$ is pretty much the same). Don’t forget a little thing though: you have to extract the bit you want in the original two operands of the add instructions. So if you have i32 operands $A$ and $B$, the first 1 bit adder will focus on the bit $n^0$ of both $A$ and $B$; and to do that we will have to do some bit manipulations (with right-shifts and and masks). Besides this detail, the LLVM-IR has all the binary operators we need, and we just have to follow the equations. We start by creating the $A$, $B$, $\overline{A}$ and $\overline{B}$ (we don’t need the $C_{in}$ because we add the two LSB):

```c
// LO_RShifted0 = A >> 0
llvm::Instruction *LO_RShifted0 = llvm::BinaryOperator::CreateLShr(              
    A, llvm::ConstantInt::get(Int32Ty, 0), 
    "", bbl
)```
// LO_RShiftedAnded/zero.noslash = (A >> /zero.noslash) & 1 = bit/zero.noslash of A
llvm::Instruction *LO_RShiftedAnded/zero.noslash = llvm::BinaryOperator::CreateAnd(
    LO_RShifted/zero.noslash, llvm::ConstantInt::get(Int32Ty, 1),
    "", bbl);

// LO_RShiftedAndedNoted/zero.noslash = ~(A >> /zero.noslash) & 1 = ~(bit/zero.noslash of A)
llvm::Instruction *LO_RShiftedAndedNoted/zero.noslash = llvm::BinaryOperator::CreateXor(
    LO_RShiftedAnded/zero.noslash, llvm::ConstantInt::get(Int32Ty, 1),
    "", bbl);

// Same thing for B
llvm::Instruction *RO_RShifted/zero.noslash = llvm::BinaryOperator::CreateL Shr(
    B, llvm::ConstantInt::get(Int32Ty, /zero.noslash),
    "", bbl);
llvm::Instruction *RO_RShiftedAnded/zero.noslash = llvm::BinaryOperator::CreateAnd(
    RO_RShifted/zero.noslash, llvm::ConstantInt::get(Int32Ty, 1),
    "", bbl);
llvm::Instruction *RO_RShiftedAndedNoted/zero.noslash = llvm::BinaryOperator::CreateXor(
    RO_RShiftedAnded/zero.noslash, llvm::ConstantInt::get(Int32Ty, 1),
    "", bbl);

Once we have our input variables ready, we can follow the equation of $S$ we saw earlier:

// Now we follow the equation and we build the successive AND
llvm::Instruction *R_And/zero.noslash1/zero.noslash = llvm::BinaryOperator::CreateAnd(LO_RShiftedAndedNoted/zero.noslash, RO_RShiftedAnded/zero.noslash, "", bbl);
llvm::Instruction *R_And/zero.noslash2/zero.noslash = llvm::BinaryOperator::CreateAnd(R_And/zero.noslash1/zero.noslash, llvm::ConstantInt::get(Int32Ty, 1), "", bbl);
llvm::Instruction *R_And11/zero.noslash = llvm::BinaryOperator::CreateAnd(LO_RShiftedAnded/zero.noslash, RO_RShiftedAndedNoted/zero.noslash, "", bbl);
llvm::Instruction *R_And12/zero.noslash = llvm::BinaryOperator::CreateAnd(R_And11/zero.noslash, llvm::ConstantInt::get(Int32Ty, /zero.noslash), "", bbl);
llvm::Instruction *R_And21/zero.noslash = llvm::BinaryOperator::CreateAnd(LO_RShiftedAndedNoted/zero.noslash, RO_RShiftedAndedNoted/zero.noslash, "", bbl);
llvm::Instruction *R_And22/zero.noslash = llvm::BinaryOperator::CreateAnd(R_And21/zero.noslash, llvm::ConstantInt::get(Int32Ty, /zero.noslash), "", bbl);
llvm::Instruction *R_And31/zero.noslash = llvm::BinaryOperator::CreateAnd(LO_RShiftedAnded/zero.noslash, RO_RShiftedAnded/zero.noslash, "", bbl);
llvm::Instruction *R_And32/zero.noslash = llvm::BinaryOperator::CreateAnd(R_And31/zero.noslash, llvm::ConstantInt::get(Int32Ty, /zero.noslash), "", bbl);

// ORing them
llvm::Instruction *R_Or/zero.noslash/zero.noslash = llvm::BinaryOperator::CreateOr(R_And/zero.noslash2/zero.noslash, R_And12/zero.noslash, "", bbl);
llvm::Instruction *R_Or1/zero.noslash = llvm::BinaryOperator::CreateOr(R_And22/zero.noslash, R_And32/zero.noslash, "", bbl);

// Gotcha, we have the result in R/zero.noslash
llvm::Instruction *R/zero.noslash = llvm::BinaryOperator::CreateOr(R_Or/zero.noslash/zero.noslash, R_Or1/zero.noslash, "", bbl);

In the previous example, the variable $R0$ will hold the result of the addition between the bit $n^0$ of both $A$ and $B$. Now you repeat those operations 32 times to have a complete adder!

I have written a Python script that generates the 32 bits adder, you can find the script here: `generate_homemade_32bits_adder_llvm_ir.py`. I also made a little program to emit the LLVM-IR code able to do the addition, to see how painful and how big the final assembly code is: `llvm-cpp-frontend-home-made-32bits-adder.cpp`. You can try it out yourself with those commands:

```
$ wget 'https://raw.githubusercontent.com/overcl0k/stuff/master/llvm-funz/llvm-cpp-frontend-home-made-32bits-adder.cpp'
$ clang++ llvm-cpp-frontend-home-made-32bits-adder.cpp 'llvm-config --cxxflags --ldflags --libs core' -o emit_adder
```

18
$ ./emit_adder 2> adder.ll # Now we can emit the LLVM-IR for the home made 32 bits adder
$ wc -l adder.ll
1016 adder.ll # That's only for one add instruction..:))
$ llc -O/zero.noslash adder.ll -o adder.s # We can also generate the x86 assembly
$ wc -l adder.s
1956 adder.s # Instead of one add instruction :P
$ clang adder.s -o adder
$ ./adder 137 1000 # And we can run it
Result: 1137
$ ./adder 4294967295 1338
Result: 1337

We are now able to implement a home made 32 bits adder, and I hope you saw that it generates a ton of x86 assembly line ; perfect for us. But now we want to modify the content of all basic blocks with our LLVM pass:

1. match all the add instructions in all the basic blocks of each function. To do so, you can iterate through each basic block, then through each instruction. Finally you have just to match what type of instruction it is.
2. replace them all with our adder. You insert your different instructions just before the add instruction you want to replace. Then, the important thing is to replace the old add instruction with the new using the function llvm::ReplaceInstWithInst.

Another dumb thing I have implemented is to decompose one add instruction into hundred others as you can see on figure 2.3. I did that to introduce more add in the program, this way if I run a second time my obfuscation pass I would be able to obfuscate heavily those ones with the home made 32 bits adder.

Figure 2.3: Tons of add that could be heavily-obfuscated with a home made adder.

2.3.2 Mess with other instructions

The idea here is the same: you want to write an instruction in a different way but you want to keep the same result ; because you don’t want to crash your program. Easy targets are
the binary operators like `mul`, `sub`, `xor`, etc. For example, you can recode the `xor` instruction only with `not`, `or`, and `and` instructions. You can also unroll a `mul` instruction into several `add` instructions. This is the part where you have to be creative, and where you have to express all the anger you have for the world. This is also the not-so-fun part: that’s why you will find in my PoC only two or three instructions obfuscated (it’s enough for the demo crackme :-)).

Note that you can also use `Z3py` to be sure your transformations are equivalent, or not.

```python
In [1]: from z3 import *
In [2]: a = BitVec('a', 32)
In [3]: b = BitVec('b', 32)
In [4]: prove(a^b == (a&(~b)|(~a)&b))
proved
```

### 2.3.3 Inserting x86 assembly

Another interesting thing was to be able to add directly assembly code for a specific CPU target. There is a dedicated class in the LLVM code base to do exactly that: `llvm::InlineAsm`. Then you just have to build a `call` instruction to trigger the execution of your assembly code.

```c
define void @main() {
   call void asm sideeffect "int3", "~{dirflag},~{fpsr},~{flags}"() #1, !srcloc !/zero.noslash
   ret void
}
```

To add a bit of fun in the demo-crackme, I decided to implement a simple ptrace-based anti-debug. I’m not really a linux guy, but I already spent some days to debug stuff in GDB and it’s really not fun when you have `fork` and `ptrace` stuff everywhere ; so I wanted to do something with those two. In the gnu debugger, you can either follow the child process, or the parent process (the default option) via the `follow-fork-mode` option. Here was my simple idea:

1. The process will fork to create another process
2. The son process will try to attach itself to the parent process using `ptrace`
   
   (a) If it works, that’s OK ; we will continue the flow of execution in the son (because the user will step in the parent, and we are nasty)
   
   (b) If it doesn’t work, we end the game: we kill both the parent and ourself
3. The father will wait. He will be killed anyway by the son, to let the son execute itself

That worked quite great in my head, but when I did try to test that on my GDB it just didn’t work. After some hours of debugging, I finally noticed my `.gdbinit` file were telling to the debugger to follow the child process instead of the parent. That means when I will try to `ptrace` the parent, GDB won’t be attached to the parent anymore but it will be attached to the son ; that’s why it didn’t work in GDB but did work with `strace`.  

20
void main()
{
    unsigned int pid, ppid;
    printf("Anti follow-fork-parent!
");

    pid = fork();
    if(pid == 0)
    {
        printf("[Son] Hi!
");
pid = getppid();
    if(ptrace(PTRACE_ATTACH, ppid, 0, 0) < 0)
    {
        printf("[Son] Father is debugged, let’s kill him!");
        kill(ppid, SIGKILL);
        exit();
    }
    else
    {
        waitpid(ppid, NULL, 0);
        printf("[Son] Continue the son, detaching from the father & killing him
");
        ptrace(PTRACE_DETACH, ppid, 0, 0);
        kill(ppid, SIGKILL);
    }
    }
    else
    {
        printf("[Father] Hi!, waiting my son attach
");
        waitpid(pid, NULL, 0);
    }
}

// do stuff 
printf("Done!
");

So I added to my test file the exact same steps, but the way around: the father will try to attach itself on the son to detect the follow-child mode of gdb. Finally, I ended up concatenating the two in order to detect both follow-child and follow-parent behavior. Here is the second part:

void main()
{
    unsigned int pid;
    printf("Anti follow-fork-child
");
    pid = fork();
    if(pid == 0)
    {
        printf("[Son] Hi, waiting my father..
");
        waitpid(getppid(), NULL, 0);
    }
    else
    {
        if(ptrace(PTRACE_ATTACH, pid, 0, 0) < 0)
        {
            printf("[Father] Son is debugged, kill him & kill myself!");
            kill(pid, SIGKILL);
            exit(0);
        }
    }
}
else
{
    waitpid(pid, NULL, 0);
    printf("[Father] Continue the father, detaching from the son & killing him\n");
    ptrace(PTRACE_DETACH, pid, 0, 0);
    kill(pid, SIGKILL);
}

printf("Continuing now.\n");
/* do stuff */
printf("Done!\n");
}

To sum up, it means you can also mess with the assembly and write really specific things for specific targets. Just make sure about the side effects of your assembly instructions, because again you don’t want to break your program. Of course my previous examples are a bit dumb, you can just nop the whole things very easily, but whatever.

2.3.4 Showcase: Kryptonite crackme

Yes, what was the best thing to do to test this little obfuscater? Try it out on a little challenge for sure!

The original one is coded is 60 lines of plain C and it is not using system specific stuff. The purpose is simple: find the password that gives the 'Good boy' message; this is not a patchme challenge. You will find:

- A Linux x86 binary with the little anti-debugs explained earlier (tested on a Debian 6.0 x86)
- A Linux x64 binary without anti-debugs (tested on a Debian 6.0 x64)
- A Windows x64 binary without anti-debugs (tested on a Windows 7 x64)

As an example, the Linux binary has been generated with the following commands:

```
$ cp kryptonite-crackme.original.ll kryptonite-crackme.ll
$ opt -S -load ./llvm-functionpass-kryptonite-obfuscater.so -kryptonite kryptonite-crackme.ll -o kryptonite-crackme.opti.ll
$ mv kryptonite-crackme.opti.ll kryptonite-crackme.ll
$ opt -S -load ./llvm-functionpass-kryptonite-obfuscater.so -kryptonite -heavy-add-obfu -enable-anti-dbg 66 kryptonite-crackme.ll -o kryptonite-crackme.opti.ll
$ mv kryptonite-crackme.opti.ll kryptonite-crackme.ll
$ llc -O/zero.noslash -filetype=obj -march=x86 kryptonite-crackme.ll -o kryptonite-crackme.o
$ clang -static kryptonite-crackme.o -o kryptonite-crackme
$ strip --strip-all ./kryptonite-crackme
```
All binaries are quite **fat** and **awful** to look at. Remember that was the purpose of our obfuscater :-P.

After one or two weeks, I will publish the original source of the crackme on my github account. If someone breaks it, I will be happy to offer him/her a beer somewhere in sometime!
2.4 Final words

Anyway, I hope I really gave you nasty ideas, and you want now to play with LLVM. It is a really powerful/cool tool, so feel free to hack it; but don’t forget to publish your sources :-). There are also a ton of ideas I wanted to try, if you have the courage to implement them go ahead:

- play with the floating arithmetic, hopefully the compiler will generate ugly SSE instructions; maybe we can even reuse what some of the work skier_t already did
- obfuscate even the standard functions and not only our functions
- try to generate a kernel module, or a Windows driver executable; would be awesome
- doing some complicated things like CFG flattening, hide the end of the loops, code encryption, etc
- obfuscate C++ code, I guess it will be even scarier and bigger
- string encryption
- re-implement manually other instructions the same way we did with the add instruction
- add integrity checks several watch-dog threads, to prevent the user to patch/debug the binary
- etc.

This is the end now guys, I hope you enjoy the read, and if you have any remarks, advises: shoot me an email or DM me on twitter.

By the way, all the binaries have been uploaded here, and the source of kryptonite is here; have fun! I would be really happy to see solutions to defeat that massive-heavy obfuscations!

Special thanks to those guys: @elvanderb, @gentilkiwi, @__x86 and @agixid.