HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Reconfigurable & embedded
Digital Systems

Just-in-Time Recompilation and Optimization of Compiled Binaries

Student: Lucas Elisei

Supervisor: Alberto Dassatti

July 2018

Abstract

Over the past decades, computer science fields grew up exponentially. More and more computational power is needed to solve modern problems and this implies a lot of energy consumption.

A way to reduce the energy consumption of data centers and their applications could be to optimize the applications so that they require less computational power.

This document explores an approach to optimize applications by recompiling them without sources. Recompiling entire binaries would take a lot of time so we focus only on parts of the applications which require a lot of computational power.

Our method is based on finding a function that takes a lot of time to execute, recompiling it into more efficient native code to finally patch the binary so it executes the newly optimized function.

Early testing showed that our solution can enhance a binary performance up to 25%. Unfortunately, the decompilers that exist nowadays are still in an early stage of development, thus limiting the capacities of our method. In a near future, the decompilers technology will evolve to allow even more performance enhancement.

Abstract

Ces dernières années, le domaine de l'informatique a connu un grand essor. De plus en plus de puissance de calcul est requise pour résoudre des problèmes modernes et cela implique une grande consommation d'énergie.

Une façon de palier notamment à la consommation d'énergie des data centers et de leurs applications serait d'optimiser ces dernières afin qu'elles demandent moins de puissance de calcul.

Ce document explore une approche pour optimiser n'importe quelle application en la recompilant sans avoir accès au code source. Recompiler un programme dans son intégralité demanderait beaucoup de temps de ce fait nous nous concentrons uniquement sur les parties critiques de l'application.

Notre méthode se base sur trouver une fonction critique, la recompiler en du code natif plus efficace pour enfin modifier le programme afin qu'il exécute la fonction optimisée.

Des tests ont permis de montrer que notre solution améliore de la performance d'une application jusqu'à 25%. Malheureusement, les décompilateurs existant de nos jours sont encore en phase de développement prématuré, limitant les capacités de notre méthode. Dans un futur proche, la technologie de décompilation va évoluer pour proposer des améliorations encore plus marquées.

Table of contents

1 Introduction

Over the past decades, computer science fields grew up exponentially. More and more computational power is needed to solve actual problems (e.g. machine learning or artificial intelligence).

In 1965, Gordon Moore, Intel's co-founder, stated that the number of transistors[1] that can be placed on a integrated circuit doubles roughly every two years [7]. It is known as Moore's Law. Due to physics and economic limitations, this law is unfortunately coming to an end and that might be critical for technological progress.

A way to negate the end of Moore's Law could be to optimize existing applications so that they require less computational power. Most programs are proprietary software, which means we don't have access to the source code, leaving us this compiled binaries[2].

This document explores an approach to optimize applications by recompiling them without sources. Just-in-Time recompilation can make use of runtime to dynamically recompile parts of the executed applications to generate a more efficient native code.
Hence, as a consequence of this, executable programs need less computational power, negating the end of Moore's Law. Furthermore, it could allow any applications to run on any architecture, thus reducing development time. Indeed, nowadays, developers are forced to develop specific code for each architecture to produce high-performance applications.

1.1 Project aim and objectives

The aim of this project is to optimize any application binary by finding the potential bottlenecks at run-time. Those bottlenecks are then decompiled to an *Intermediate Representation* (IR), optimized and finally recompiled so that the running application uses the new version.

To achieve this aim, the following objectives have been identified:

1. Review several decompilation tools and find one that suits our needs.
2. Find a way to successfully recompile a given function from a simple program.
3. Develop a program to automatize the process.

A long-term vision could be to extend the Just-in-Time recompilation and optimization to heterogeneous systems[3] by off-loading code to an FPGA or a GPU [39]. Doing so would significantly lower the CPU load and reduce application execution time.

1.2 Disposition

This document details every stage of the project. It follows a logical structure and outlines the major stages in chronological order. A brief summary of each section is presented in the list below.

1. **Introduction**
   Presents the project, its aim and objectives and introduces some technical terms.
2. **Literature review**
   Discusses papers and work that other teams have done and/or are doing with decompilers.
3. **State of the art tools**
   Reviews several decompilation tools and evaluates the best one to use for this project.
4. **Successful recompilation**
   Details how to achieve the second objective, the recompilation step.
5. **Automatization tool**
   Details the implementation of the third objective, the automatization tool.

[1] In a computer, a transistor is a component that represents the binary 0's and 1's (bits).
[2] Executable program.
[3] An heterogeneous system uses more than one kind of processor (e.g., CPU, GPU or FPGA) — Wikipedia

1.3 Requirements

The tools required to run and use the project are detailed in the appendix B.

1.4 Theoretical overview

Before diving into the core of the subject, it could be useful to review some technical terms that are necessary for the good understanding of this document.

1.4.1 Executable and Linkable Format

The Executable and Linkable Format (ELF) is common standard file format for executable files, object code and shared libraries. Since ELF is by design flexible, extensible and cross-platform, it has been adopted by a plethora of operating systems on many different platforms [8].

Each ELF file is made up of one ELF header. It starts with the four bytes magic number `0x7F`, `'E'`, `'L'`, `'F'`. The ELF file header contains general information about the executable, such as the addresses length (32- or 64-bit), the endianness (*big-endian* or *little-endian*), the object file type (executable, relocatable or shared object), the assembly architecture (e.g., `x86` or `ARM`), the virtual address of its entry point (which indicates the starting point of the program execution) and the offsets to the program and sections headers.

The program header is meaningful to executables and shared objects only. It contains a description for each segment (which contains one ore more sections) and other information the system needs to prepare the program for execution.

The section headers contain a description for each section like address location and access rights (i.e. read, write and execution).

The figure 1 shows an overview of how an ELF executable is structured.

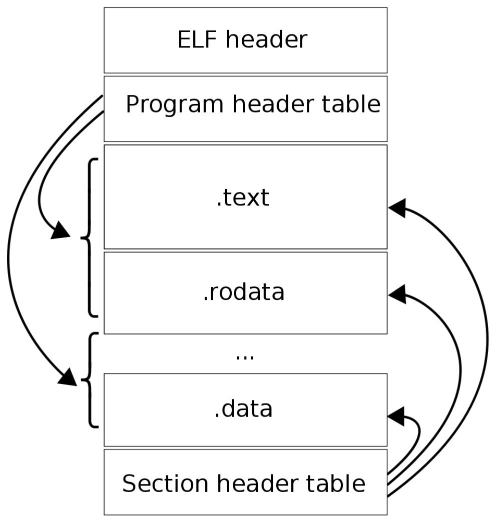

Figure 1: Structure of an ELF executable [8]

1.4.2 Decompilation

A compiler is a software that translates human-readable programming languages (e.g. C or Java) into a machine-code language (e.g. assembly) to produce an executable program. A decompiler does the exact opposite: it lifts code from a low-level into a higher-level representation.
The figure 2 shows the difference between a simple C code and its translation into machine code.

```c
int a = 5;
int b = 4;
int c = a + b;
```

```
movl    $5, %eax
movl    $4, %ebx
addl    %ebx, %eax
ret
```

Figure 2: On the left, a simple C program that adds two integers a and b into an integer c. On the right, the resulting assembly code after compilation.

Compared to decompilers, compilers have a more privileged position: the input language is strictly defined and plenty of information is available for functions, variables, types, etc. If the compiler cannot generate a valid code because the input does not conform to the language standards, it is allowed to simply print an error message and stop. Decompilers do not benefit from anything similar. Just the contrary [6]:

- Compiled architecture instructions generally use variable length encoding ;
- The input binary is often obfuscated ;
- Many decompilation problems are unsolved or proven to be unsolvable in generic cases ;
- The output is examined in details by a human being and any sub-optimality is noticed.

In conclusion, robust machine code decompilation is impossible. A decompiler will always have some imperfections and eventually generate wrong output. Our best hope is to diminish the undesired effects as much as possible. To achieve this, here are some basic ideas [16]:

- Make some configurable assumptions about the input (e.g. calling conventions). The user will be able to control the decompiler by specifying the missing information. In simple cases, the decompiler will deduce or guess it.
- Use solid theoretical approach to solve problems (e.g. instruction simplification).
- Use heuristics for unsolvable problems (indirect jumps, call arguments).
- Prefer to generate ugly but correct output rather than nice but incorrect. Let the user embellish if he wants to do so.
- Let the user guide the decompilation in difficult cases (e.g. function prototypes).

Decompilation can be broken down into several phases:

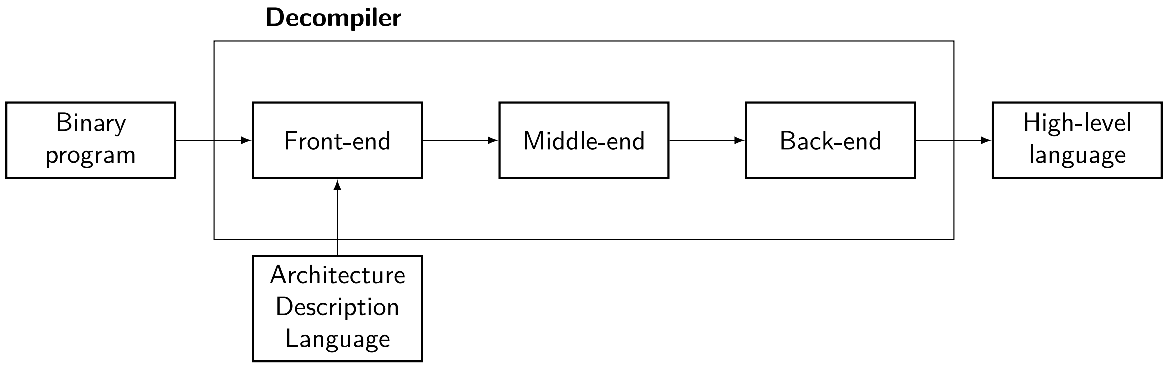

Figure 3: Decompiler structure.

1. **Front-end**
   Parses a binary program (ELF) and translates an architecture-specific machine code into a sequence of low-level IR. To do so, the front-end uses an architecture description language (ADL). The ADL contains information such as the processor resources (i.e. available registers and memory) and instruction set (i.e. assembler language syntax, binary encoding and behaviour of each instruction). Each instruction of the ELF binary is then decoded into an intermediate representation, which describes the program behaviour in a platform-independent way.

2. **Middle-end**
   Improves the properties of the previously generated low-level IR code and prepares it for the back-end phase. Improvements includes [5]:

   - Search for idioms and other types of program analysis such as constant or expression propagation.
   - Retrieval of high-level constructs, such as `if..else` statements or loops.
   - Code optimization.

   To do so, the decompiler generates a control-flow graph (CFG) and simplifies it to exclude useless instructions.

3. **Back-end**
   Finally, the back-end converts the optimized IR into the target high-level language (e.g. C or Python). During this conversion, loops and conditional statements are identified and reconstructed into a human-readable way. An other optimization is done and the binary is emitted in the form of the target high-level language.

2 Literature review

As stated previously, a robust decompilation is impossible. But in most cases, we just need it to work. A lot of papers and books talk about the decompilation process and how to resolve undesired problems.

A list of documents related to decompilation may include:

- 13th chapter of the book *Reversing: Secrets of Reverse Engineering*, written by Eldad Eilam [6] ;
- Conference paper about *Reconstruction of instructions idioms in a retargetable decompiler*, by Jakub Křoustek and Fridolín Pokorný [18] ;
- All publications available on the RetDec website [34].

Also, some interesting projects have been achieved thanks to decompilation:

- **Mac 68k emulator**
  Dynamic translating emulator for M68K code for Apple Macintoshes based on PowerPC [21].
- **StarCraft port to an ARM platform**
  The Pandora console's community [26] generated an ARM version of the video game StartCraft thanks to static recompilation [29].
- **Dolphin emulator**
  Dolphin emulates the GameCube and Wii consoles on PC thanks to Just-in-Time recompilation of PowerPC code to `x86` and `AArch64` [4].
- **x86 Intel CPUs**
  Since their Pentium Pro CPU, CISC instructions are translated to more RISC-like internal micro-operations [28].

3 State of the art tools

This section reviews an intermediate representation named LLVM IR [20] and several decompilation tools and evaluates the best candidate to use for this project.

3.1 Intermediate representations

An intermediate representation is similar to a coding language. An IR is designed to be capable of representing the source code without loss of information and independent of any particular source and target. A compiler often translates a high-level code to an IR before compiling it into machine instructions.

Nowadays, there are two widely-used compilers: GCC [10] and LLVM [19]. Both of them offer an intermediate representation. Since most of the available decompilers – if not all – use the LLVM IR, we will not discuss GIMPLE [11], the GCC's IR.

The LLVM IR aims to be light-weight and low-level while being readable, typed and extensible. You can see it as a human-readable assembly language representation [20].

Let's pick up the same C and assembly codes shown in figure 2. The same program is translated as follow in LLVM IR:

```
%a = add i32 5, 0
%b = add i32 4, 0
%c = add i32 %a, %b
ret i32 %c
```

The first two lines store the values into two variables and the third line performs their sum. The last line returns the result. As you can see, the LLVM IR is easier to read as a human than the assembly code. Furthermore, it keeps track of the type of the variables, which is really harder in the assembly form.

3.2 Existing tools

The D-Neliac decompiler [13], built in the 1960s, was the first decompiler to prove that decompilation is feasible. Since this period, many projects have tried to offer correct decompilation.

Today, there are some interesting projects in the wild. Unfortunately, most of them are not open-source[1] or depend on proprietary software[2] (e.g. IDA-Pro [15]) thus they can't be used for our project.

rev.ng

`rev.ng` [37] is a suite of tools for binary analysis based on QEMU [33] and LLVM. It is (*was*) developed by Alessandro Di Federico, a former PhD student at Politecnico di Milano [30]. The project is open-source and licensed under GPLv2 [12]. Each individual file is released under the terms of the MIT License [22].

This project relies on a few components but the most interesting one is its static binary translator. Provided an input ELF binary, it will analyse it and emit an equivalent LLVM IR. The currently supported architectures are `MIPS`, `ARM` and `x86-64`.

The main issue with this project is that it was last updated more than ten months ago, so it seems like it has been abandoned. Furthermore, this project was maintained by only one developer, which is quite a small team.

We ran some tests to see how good it decompiles a binary. Unfortunately, most of the time `rev.ng` would crash or produce empty output.

[1] Type of computer software whose source code is released under a license in which the copyright holder grants users the rights to study, change and distribute the software to anyone and for any purpose. — Wikipedia

[2] Proprietary software is non-free computer software for which the software's publisher or another person retains intellectual property rights—usually copyright of the source code, but sometimes patent rights. — Wikipedia

RetDec

RetDec [34], for Ret*argetable* Dec*ompiler*, is an open-source machine-code decompiler based on LLVM [19]. It is being developed by the famous company Avast Software [1] and is licensed under the MIT License [22]. Its development was internal to Avast for several years but in February 2018, they decided to release the code publicly. It currently supports the following architectures: `x86`, `ARM`, `MIPS`, `PIC32` and `PowerPC`. RetDec is also being actively developed by at least three people.

This project is composed of a plethora of libraries but we will mainly focus on the `bin2llvmir` library which aims to translate binaries into LLVM IR modules. The project also includes a tool named `bin2llvmirtool` which is a front-end for the `bin2llvmir` library.

The main feature that offers RetDec against other decompilers is that it is retargetable. That means that – thanks to the ISAC architecture description language [14], also developed by the RetDec team – it is not necessary to manually reconfigure the decompiler for a new architecture, making it compatible with all machines (but not all architectures, as stated before).

BOLT

The BOLT project [25] was developed by a Facebook team and interns. It aims to boost the performance of 64-bit ELF applications by implementing a post-link optimizer. BOLT is built on top of LLVM and its optimization techniques are based on reorganizing code so caches suffer less fragmentation and at reordering basic blocks to relieve pressure from the branch predictor unit of a processor.

BOLT was deployed in Facebook data-centers and improvements ranging between 2% and 8% were observed, which is quite remarkable giving the fact that data-centers' applications are already highly optimized. These optimizations are really important since they reduce energy consumption thus reducing environmental impacts and costs.

Even if BOLT is a really promising and interesting project, it does not decompile code. It disassembles it and constructs a control-flow graph based on the disassembled code. However, this project might be helpful for inspiration about our project's architecture and the techniques that might be used to reach our aim.

Conclusion

After this analysis, we chose RetDec as the best candidate for this project since its development is active and led by a professional team. The MIT License also allows us to freely use their code which could be useful for the development of this project.

Some useful resources are available on their website *retdec.com* such as publications and presentations.

Previously, we stated that RetDec only supports 32-bit architectures. That's right for its official version but the 64-bit decompilation can be enabled by switching to another version of RetDec, even if it's not ready yet. For this document, we will use the RetDec version which is *capable* of decompiling both 32- and 64-bit architectures.

Since our project will mostly rely on the decompilation done by RetDec, we will analyse in details how it does decompile machine code.

RetDec is composed of two main parts: the *pre-processing* and the *core*.

The pre-processing part is responsible of analysing the binary program to produce an image that will be later used by the core.
The first step of the pre-processing is to discover the format of the binary program. RetDec supports the following formats: ELF, HEX, PE, COFF, Mach-O and raw binary. Then, a uniform binary representation is produced. Thanks to this uniform representation, the other parts of RetDec don't need to care about the original format of the program. The next step focuses on passing the uniform representation to an *image loader* library which will emulates a loader. This is important since depending on the loader used, the data loaded into memory can look different than the data in the original binary.
The image is now emitted. RetDec will performs an additional step by looking for available debugging

information. This will help to produce a better LLVM IR but most of proprietary programs ship with no debug information to avoid being analysed and possibility modified by hackers.

The purpose of the core is to lift LLVM IR code from the image produced by the pre-processing.
To do so, it starts by doing initialization passes to perform dead global elimination, constant propagation, inlining, loop optimization, etc... in the machine code. Then, RetDec calls an third-party framework called Capstone [3], which actually lifts instructions to the LLVM intermediate representation. Afterwards, some low-level passes are performed to identify global and local variables, functions' arguments and return type, data types and so on. Finally, some high-level LLVM passes are done on the LLVM IR already at disposition for final touches. Then, the LLVM IR is emitted by RetDec for future use.

We had the chance to attend the *Pass the SALT* 2018 [27] conference, where the RetDec team gave a talk. Thanks to this talk, we had more overview about the RetDec architecture and how it internally works.

4 Successful recompilation

Now that we have reached the first objective of this project by identifying a decompilation tool that suits our needs, we can focus on the second objective: *find a way to successfully recompile a given function from a simple program*.

First of all, we code a function which simply adds two unsigned integers and returns the result:

simple.c

```
uint32_t simple_add(uint32_t *a, uint32_t *b) {
    return *a + *b;
}
```

Then, we create a simple main function which takes two integers as arguments, calls the `simple_add()` method and displays the result.

main.c

```
int main(int argc, char** argv) {
    uint32_t a, b;

    a = atoi(argv[1]);
    b = atoi(argv[2]);

    fprintf(stdout, "Result: %u\n", simple_add(&a, &b));

    return EXIT_SUCCESS;
}
```

Now that we have a fully functional program, we compile it:

```
gcc -std=c99 -Wall -Werror -pedantic -Iinclude -m32 main.c simple.c -o main
```

We now have a 32-`bit` binary. The most interesting part is not the compilation, but the recompilation. So let's get started. We invoke RetDec decompilation script to get the LLVM IR of the `simple_add()` method:

```
retdec-decompiler.sh --stop-after bin2llvmir --select-functions simple_add ./main
```

The command above calls the RetDec decompilation script and asks it to stop after it has translated the binary (`./main`) into the LLVM IR (`--stop-after bin2llvmir`). We also ask it to only decompile the `simple_add()` method (`--select-functions simple_add`). The following code is the resulting LLVM IR:

main.c.backend.ll

```
define i32 @simple_add(i32* %arg1, i32 %arg2) local_unnamed_addr {
entry:
    %v2_57d = load i32, i32* %arg1, align 4
    %v1_582 = inttoptr i32 %arg2 to i32*
    %v2_582 = load i32, i32* %v1_582, align 4
    %v2_584 = add i32 %v2_582, %v2_57d
    ret i32 %v2_584
}
```

You might have noted that the method's signature is not the same as the one defined in `simple.c`. It is an acknowledged bug that should be fixed in the future [35]. Apart from the signature, everything appears to be in order, which is already quite motivating. We will now recompile the LLVM IR into a new object file and link it with the previously created object file `main.o`.

```
llc-5.0 -march=x86 main.c.backend.ll -o main.c.backend.s
gcc -m32 -c main.c.backend.s -o main.c.backend.o
gcc -m32 main.o main.c.backend.o -o ./main.translated
```

The first line invokes the LLVM static compiler, which can compile LLVM IR code. We pass the option `-march=x86` to tell the compiler to compile into a `32-bit` version. This is required since RetDec can only decompile `32-bit` code at the moment. The resulting code is assembly code.

The second line calls GCC, a widely-used compiler, to compile the assembly code into an object file. Again, the `-m32` flag asks the compiler to produce a `32-bit` version of the binary.

The last line links the existing `main.o` and the newly created `main.c.backend.o` object files and creates a new binary named `main.translated`. By running it, we acknowledge that it still does its job:

```
./main.translated 5 4
Result: 9
```

This result demonstrates that a simple function can successfully be recompiled and linked with an existing file object, which is exactly the second objective of this project. Quite exciting!

Since the recompilation process requires quite a few commands, we created a `Makefile` to automatize the process. The source of the `Makefile` and of the `main.c` and `simple.c` files are available in appendix C.

5 Automatization tool

This section reviews the development of the third objective: *develop a program to automatize the recompilation process*. The tool is composed of a library, `liboptimizer` and a program to call the library, called `optimizer`.

5.1 Design

Above all, we have to define what the tool will do and how we will achieve it. The program will take several arguments for execution:

1. The function name to optimize or its offset ;
2. The target binary to execute ;
3. The arguments of the target binary.

The first argument is either the function name or its offset. When the function name is passed to the tool, we have to discover the offset of the function in the target binary and, reciprocally, when the function offset is passed to the tool, we must compute its symbol. So the first step of the library is to implement an ELF parser. The part of the library that will achieve this task will be referred as `elfparser`.

Once the target ELF has been parsed and that we have enough information, the next step is to call the RetDec's decompilation scripts to retrieve the LLVM IR of the function to optimize. This part is called `retdec`.

Then we must recompile it with a JIT compiler. The part of the library in charge of doing the JIT recompilation is named `jit`.

Finally, once we have the machine code of the recompiled function, the final step is to patch the target binary memory space so that it executes the new optimized function instead of the old one. To do that, we implemented a third part named `live-patcher`.

Each of these parts will be further discussed later. The figure 4 shows a simple overview of how the library is structured.

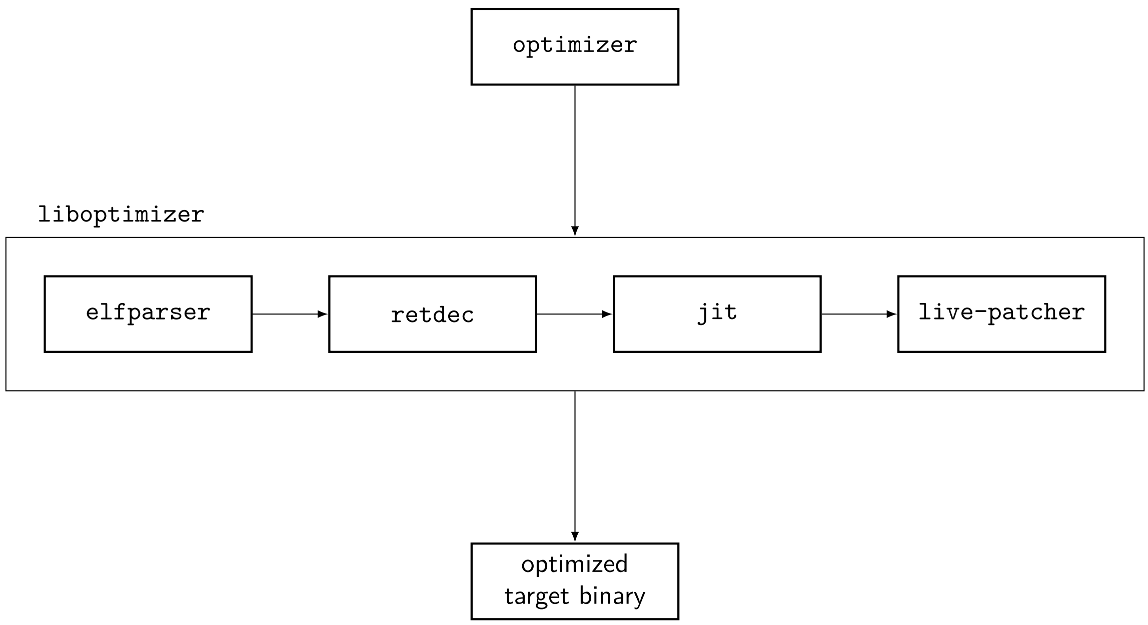

Figure 4: `liboptimizer` pipeline

5.2 elfparser

As said above, this part of the library is responsible for parsing the ELF binary passed to the library. Its purpose is to offer the possibility to retrieve the offset of a given symbol and vice-versa, given an offset, retrieve the corresponding symbol.

To implement this, some previous work is required: the ELF header has to be parsed so that we have information about the binary. The most wanted information to know is the class of the ELF file: is it 32-bit or 64-bit? Indeed, this information is the most important since the ELF is not structured the same way for the two classes. The first bytes of the ELF header are the same for 32- and 64-bit so we assume the ELF is 32-bit and then we adapt. The following `struct`s describe the structure of an ELF header for 32-bit (left) and 64-bit (right):

struct elf32_hdr

```c
typedef struct elf32_hdr {
    unsigned char e_ident[16];
    Elf32_Half    e_type;
    Elf32_Half    e_machine;
    Elf32_Word    e_version;
    Elf32_Addr    e_entry;
    Elf32_Off     e_phoff;
    Elf32_Off     e_shoff;
    Elf32_Word    e_flags;
    Elf32_Half    e_ehsize;
    Elf32_Half    e_phentsize;
    Elf32_Half    e_phnum;
    Elf32_Half    e_shentsize;
    Elf32_Half    e_shnum;
    Elf32_Half    e_shstrndx;
} Elf32_Ehdr;
```

struct elf64_hdr

```c
typedef struct elf64_hdr {
    unsigned char e_ident[16];
    Elf64_Half    e_type;
    Elf64_Half    e_machine;
    Elf64_Word    e_version;
    Elf64_Addr    e_entry;
    Elf64_Off     e_phoff;
    Elf64_Off     e_shoff;
    Elf64_Word    e_flags;
    Elf64_Half    e_ehsize;
    Elf64_Half    e_phentsize;
    Elf64_Half    e_phnum;
    Elf64_Half    e_shentsize;
    Elf64_Half    e_shnum;
    Elf64_Half    e_shstrndx;
} Elf64_Ehdr;
```

And the following `typedef`s define the types used for the above `struct`s:

Base types for 32-bit

```c
typedef uint32_t    Elf32_Addr;
typedef uint16_t    Elf32_Half;
typedef uint32_t    Elf32_Off;
typedef int32_t     Elf32_Sword;
typedef uint32_t    Elf32_Word;
```

Base types for 64-bit

```c
typedef uint64_t    Elf64_Addr;
typedef uint16_t    Elf64_Half;
typedef int16_t     Elf64_SHalf;
typedef uint64_t    Elf64_Off;
typedef int32_t     Elf64_Sword;
typedef uint32_t    Elf64_Word;
typedef uint64_t    Elf64_Xword;
typedef int64_t     Elf64_Sxword;
```

Thanks to the fields `e_shoff`, `e_shnum`, `e_shentsize`, we obtain information about the location of *Section Header Table*, the number of entries it contains and their size.
Additionally, the index of the *String Table* in the *Section Header Table* is stored into the field `e_shstrndx`. Thanks to this index, we already know where the *String Table* is located. The last information we need is the location of the *Symbol Table*.

The *Section Header Table* contains all the information necessary to locate each ELF section. But we only really need one section: the *Symbol Table*. An ELF file contains only one *Symbol Table* and the section has a unique type to identify it. This type has the value 2.

32-bit Section Header

```c
typedef struct elf32_shdr {
    Elf32_Word  sh_name;
    Elf32_Word  sh_type;
    Elf32_Word  sh_flags;
    Elf32_Addr  sh_addr;
    Elf32_Off   sh_offset;
    Elf32_Word  sh_size;
    Elf32_Word  sh_link;
    Elf32_Word  sh_info;
    Elf32_Word  sh_addralign;
    Elf32_Word  sh_entsize;
} Elf32_Shdr;
```

64-bit Section Header

```c
typedef struct elf64_shdr {
    Elf64_Word  sh_name;
    Elf64_Word  sh_type;
    Elf64_Xword sh_flags;
    Elf64_Addr  sh_addr;
    Elf64_Off   sh_offset;
    Elf64_Xword sh_size;
    Elf64_Word  sh_link;
    Elf64_Word  sh_info;
    Elf64_Xword sh_addralign;
    Elf64_Xword sh_entsize;
} Elf64_Shdr;
```

The field `sh_type` contains the type of the section. So we iterate over the table until we find a section with the type 2. The location of the symbols is stored by the field `sh_offset`. So what is left to do is to look at the symbols location, iterate over the list until we find the symbol we are interested in. A symbol is structured as follow:

32-bit symbol
```
typedef struct elf32_sym{
    Elf32_Word      st_name;
    Elf32_Addr      st_value;
    Elf32_Word      st_size;
    unsigned char   st_info;
    unsigned char   st_other;
    Elf32_Half      st_shndx;
} Elf32_Sym;
```

64-bit symbol
```
typedef struct elf64_sym {
    Elf64_Word      st_name;
    unsigned char   st_info;
    unsigned char   st_other;
    Elf64_Half      st_shndx;
    Elf64_Addr      st_value;
    Elf64_Xword     st_size;
} Elf64_Sym;
```

The field `st_name` contains an offset from the *String Table* where the symbol character string is located. The field `st_value` contains the offset of the symbol in the ELF file. So if we want to retrieve an offset from a given symbol, we iterate over all symbols until we find the corresponding character string and return the offset.
On the contrary, if we want to retrieve a symbol from a given offset, we iterate over all symbols until we find the corresponding offset and return the character string.

Summary

Developing this part of the library was not really difficult. The main *issue* was that there is a lot of duplicate code since 32-bit and 64-bit ELF have different structures.
The code relative to `elfparser` is available in appendix D.1.

5.3 `retdec`

`retdec` is responsible for calling the RetDec's decompilation scripts and retrieve the resulting LLVM IR file.

Since RetDec's scripts generate several files, we chose to create a temporary directory which is deleted once `liboptimizer` is done running. Then, the RetDec's scripts are called with corresponding arguments: stopping after the generation of the LLVM IR, the temporary directory as working directory, the name of the function to decompile and the target binary.

Summary

This part of the library is very lightweight so no problem were encountered during its development.
The code relative to `retdec` is available in appendix D.2.

5.4 `jit`

The purpose of the library `jit` part is to implement a JIT compiler so we can recompile the target function. It uses LLVM's On-Request-Compilation (ORC) APIs [2].

Since the offered APIs are really simple to use, the implementation is straight-forward. We simply initialize an `ExecutionEngine` [9], parse the LLVM IR file containing the function to optimize and tell the engine to compile it. It returns a pointer on the optimized function so we can use it later.

We have the optimized function machine code at our disposal and that's great. But we need one more information: the size (in bytes) of the optimized function. Unfortunately, the `ExecutionEngine` class doesn't provide a *simple* way to get the size.
The solution is to implement a subclass of the `JITEventListener` [17] class and register this new listener so that every time the JIT compiles a function, it notifies its registered listeners with more information than just a pointer to the new function's machine code.

Summary

The code for this part of `liboptimizer` is really simple thanks to the LLVM APIs. The main problem we came across was to get the size of the compiled function but after some research we achieve to overcome the issue.
The code relative to the JIT front-end and the `JITEventListener` subclass is available in appendix D.3.

5.5 `live-patcher`

The aim of the `live-patcher` is to modify the target process memory space so that it calls the optimized function instead of the old one during its execution.

It mostly relies on the `ptrace` [32] system call [38] which allows a *tracer* process (in our case, `liboptimizer`) to observe and control the execution of a *tracee* (the target binary).

First things first, we have to attach the tracer to the target process and stop its execution so that we can modify its memory.

The next step is to allocate a new memory segment into the target process memory space. This segment is used to store the machine code of the optimized function. To do so, we must inject a `mmap2` [23] system call. The injection consists of modifying some tracee registers with pre-defined values to ensure that we have the correct access rights, enough allocated space, etc... After the system call is injected, we retrieve the address of the newly allocated memory segment by reading back the value of the `RAX` register.

Now that we have a memory segment that we can execute and write into as we wish, we can write the optimized function's machine code into it.

The last but not least step is to hook the old function so that the process executes the optimized one. The hook consists of replacing the first bytes of the old function with an unconditional jump to the location of the optimized function, which is stored at the address of the memory segment we got before.
For the sake of clarity, we created two simple macros that allow to create the assembly code for 32- and 64-bit programs. The details of these macros are available in appendix D.4.

32-bit hook
```
unsigned char jump_32[] =
↪   MAKE_JUMP32(process->freesegment_address);
```

64-bit hook
```
unsigned char jump_64[] =
↪   MAKE_JUMP64(process->freesegment_address);
```

These macros take as argument the address of the memory segment and then append machine instructions to create a hook.
There are only two instructions: the first one loads the address of the memory segment into a register and the last one tells the processor to jump unconditionally to the location stored into the same register. Since we support both 32- and 64-bit binaries, we have two different jumps because the address length and the instruction sets are different. After all these memory replacements, we can finally let the tracee process continue its execution with the optimized function.

Summary

This part of the library was the most difficult but also the most interesting one to develop. It required to dig at a very low level into the tracee memory space, reading instructions byte per byte, find a way to inject a system call, etc... To ease debugging, we had to develop our very own debugging function. We learned a lot developing this part and the effort was rewarding.
The code relative to the `live-patcher` part is available in appendix D.4.

5.6 liboptimizer

All those parts work great individually but we need to wire them up so we can expose them to any user that wants to use the library. To keep things simple, we offer no more than four methods and a `struct` that is meant to contain all the information that is needed.

process_info_t

```
typedef struct {
    const char  *path;
    int         argc;
    char        **argv;
    pid_t       pid;
    const char  *function_name;
    uint64_t    function_offset;
    uint64_t    codesegment_address;
    uint64_t    freesegment_address;
    uint8_t     *optimized_function;
    size_t      optimized_function_size;
    uint8_t     is64;
} process_info_t;
```

- `*path`: Path of the target binary.
- `argc`: Arguments count of the target binary.
- `**argv`: Array of the arguments of the target binary.
- `pid`: The Process ID of the target binary.
- `*function_name`: The name of the function to optimize.
- `function_offset`: The offset of the function to optimize.
- `codesegment_address`: Address of the Code Segment of the target process.
- `freesegment_address`: Address of the newly allocated memory chunk.
- `*optimized_function`: Pointer to the optimized function machine code.
- `optimized_function_size`: Size in bytes of the optimized function.
- `is64`: Is the target process 32- or 64-bit?

Library methods:

- `char *symbol_at_address(const char *path, uint64_t address)`
  Resolves the symbol of the function at the given `address` in the binary located at `path`. Allocates and returns a null-terminated string containing the symbol.
  Returns `NULL` on error.

- `process_info_t *init_process(int argc, char **argv, const char *function_name)`
  Given `argc` and `argv` of the binary and the name of the function to optimize, allocates and returns a pointer to a `process_info_t` that contains basic information about the process.
  Returns `NULL` on error.
  This method calls the RetDec's scripts to decompile the target function, pass the resulting LLVM IR file to the JIT, retrieve the optimized function and attach the target binary so its memory space can be modified.

- `int modify_process(process_info_t *process)`
  Modifies the memory of the associated process of the argument. Basically hooks the function to optimize with the optimized one.
  Returns 0 on success, 1 otherwise.

- `int execute_process(process_info_t *process, bool wait_for_exit)`
  Starts the execution of the process. If `wait_for_exit` is `true`, waits for the process to exit and returns its exit status. If `wait_for_exit` is `false`, doesn't wait for the process to exit and returns 0 on success, 1 otherwise.

5.7 optimizer

Now that we have a library that works, we need a small program to show how to call it. The following code shows a simple use case of `liboptimizer`.

optimizer.c

```c
/*
 * File: optimizer.c
 *
 * Created by: Lucas Elisei <lucas.elisei@heig-vd.ch>
 *
 * Shows how to use the liboptimizer library.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "debug.h"
#include "liboptimizer.h"

static void print_help(char **argv) {
    fprintf(stderr, "Usage: %s {-f <function_name>, -a <function_offset>} <target_bin>
     ↪   [target_args]\n", argv[0]);
}

int main(int argc, char **argv) {
    process_info_t *process;
    char *function_name;
    uint64_t function_offset;
    int rc;

    if (argc < 4) {
        print_help(argv);

        return EXIT_FAILURE;
    }

    if (strcmp(argv[1], "-a") == 0) {
        function_offset = strtoull(argv[2], NULL, 16);
        function_name = symbol_at_address(argv[3], function_offset);
    }
    else if (strcmp(argv[1], "-f") == 0) {
        function_name = argv[2];
    }
    else {
        fprintf(stderr, "Unrecognized option: %s\n", argv[1]);
        print_help(argv);

        return EXIT_FAILURE;
    }

    process = init_process(argc - 3, argv + 3, function_name);
    if (process == NULL) {
        fprintf(stderr, "[optimizer] ERROR: Error during initialization.\n");

        return EXIT_FAILURE;
    }

#ifdef LIBOPTIMIZER_DEBUG
    print_process_info(process);
#endif

    rc = modify_process(process);
    if (rc < 0) {
        fprintf(stderr, "[optimizer] ERROR: Error while modifying process.\n");

        return EXIT_FAILURE;
    }

    rc = execute_process(process, true);
    if (rc < 0) {
        fprintf(stderr, "[optimizer] ERROR: Error while executing process.\n");
```

```
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

This simple program works as follow: the first argument is either the name of the function to decompile (specified by `-f`) or its offset (specified by `-a`). The rest of the arguments are the binary name and its arguments.
Remember the small program we decompiled in section 4? If we pass it to `optimizer` to decompile the `simple_add` method, it should look like something like the following:

```
    ./optimizer -f simple_add main 5 4
```

6 Examples

This section shows examples of using the `liboptimizer` and limitations of the *tool* (SPOILER: its dependencies).

Because we support both 32- and 64-bit binaries, we will perform those examples for both.

6.1 Simple addition

The first example focuses on a simple function that sums two integers and prints the result. The code of the target program is the following:

simple_add.c

```c
/*
 * File: simple_add.c
 *
 * Created by: lucas Elisei <lucas.elisei@heig-vd.ch>
 *
 * Sums two integers and prints the result.
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int simple_add(int a, int b) {
    return a + b;
}

int main(int argc, char **argv) {
    int a, b, rc;

    if (argc != 3) {
        fprintf(stderr, "[simple_add] Usage: %s <a> <b>\n", argv[0]);

        return EXIT_FAILURE;
    }

    // Retrieve arguments.
    a = atoi(argv[1]);
    b = atoi(argv[2]);

    // Sum the arguments.
    rc = simple_add(a, b);

    // Print the result.
    fprintf(stdout, "[simple_add] Result: %d\n", rc);

    return EXIT_SUCCESS;
}
```

For this first example, we will not recompile the `simple_add` function to optimize the program but to see if `liboptimizer` does its job. The function is too simple to measure its impact when optimized.
The 32- and 64-bit versions of this program are respectively named `simple_add32` and `simple_add64`.

6.1.1 32-bit

To compile the `simple_add32` program, we use the following command:

```
gcc -Wall -Werror -O0 -m32 simple_add.c -o simple_add32
```

The option `-O0` tells the compiler to avoid doing optimization. This way, we will clearly see if the function has been optimized after being processed by `liboptimizer`.

Below, the machine code corresponding to the non-optimized `simple_add` function:

```
000011bd <simple_add>:
    11bd:   55                      push   %ebp
    11be:   89 e5                   mov    %esp,%ebp
    11c0:   e8 cc 00 00 00          call   1291 <__x86.get_pc_thunk.ax>
    11c5:   05 3b 2e 00 00          add    $0x2e3b,%eax
    11ca:   8b 55 08                mov    0x8(%ebp),%edx
    11cd:   8b 45 0c                mov    0xc(%ebp),%eax
    11d0:   01 d0                   add    %edx,%eax
    11d2:   5d                      pop    %ebp
    11d3:   c3                      ret
```

We see that the size of the function is 22 bytes and is composed of 9 instructions (one of which is a call to another function – `__x86.get_pc_thunk.ax`). Now, we call the `optimizer` program to optimize the function.

```
./optimizer -f simple_add simple_add32 3 2
```

We are greeted with the message `[simple_add] Result: 5` which means that it returned the correct result, and that's great! Now, let's take a look to the machine code of the optimized function:

```
f7f70000 <simple_add>:
    0:  8b 44 24 08             mov    0x8(%esp),%eax
    4:  03 44 24 04             add    0x4(%esp),%eax
    8:  c3                      ret
```

We clearly see that the function has been optimized! The size of the `simple_add` function is now of 9 bytes and is composed of only 3 instructions.

6.1.2 64-bit

To compile the `simple_add64` program, we use the following command:

```
gcc -Wall -Werror -O0 simple_add.c -o simple_add64
```

Below, the machine code corresponding to the non-optimized `simple_add` function:

```
000000000000116a <simple_add>:
    116a:   55                      push   %rbp
    116b:   48 89 e5                mov    %rsp,%rbp
    116e:   89 7d fc                mov    %edi,-0x4(%rbp)
    1171:   89 75 f8                mov    %esi,-0x8(%rbp)
    1174:   8b 55 fc                mov    -0x4(%rbp),%edx
    1177:   8b 45 f8                mov    -0x8(%rbp),%eax
    117a:   01 d0                   add    %edx,%eax
    117c:   5d                      pop    %rbp
    117d:   c3                      retq
```

Before optimization, the size of the function is 19 bytes and is composed of 9 instructions. Now, we call the `optimizer` program with the same arguments as for the 32-bit version (only replacing `simple_add32` with `simple_add64`).

This time, the optimized program doesn't print any message... That's not good. Let's analyse the resulting machine code of the optimized function:

```
00007f0c7a90b000 <simple_add>:
    0:  48 c7 44 24 e8 00 00    movq   $0x0,-0x18(%rsp)
    7:  00 00
    9:  48 8b 44 24 e0          mov    -0x20(%rsp),%rax
    e:  48 89 04 25 f8 ff ff    mov    %rax,0xfffffffffffffff8
   15:  ff
   16:  48 c7 44 24 e0 f8 ff    movq   $0xfffffffffffffff8,-0x20(%rsp)
   1d:  ff ff
   1f:  8b 44 24 f8             mov    -0x8(%rsp),%eax
```

```
23:  89 04 25 f4 ff ff ff    mov    %eax,0xfffffffffffffff4
2a:  8b 44 24 f0             mov    -0x10(%rsp),%eax
2e:  48 8b 4c 24 e0          mov    -0x20(%rsp),%rcx
33:  89 41 f8                mov    %eax,-0x8(%rcx)
36:  48 8b 4c 24 e0          mov    -0x20(%rsp),%rcx
3b:  8b 41 f8                mov    -0x8(%rcx),%eax
3e:  03 41 fc                add    -0x4(%rcx),%eax
41:  48 8b 4c 24 e8          mov    -0x18(%rsp),%rcx
46:  48 8b 09                mov    (%rcx),%rcx
49:  48 89 4c 24 e0          mov    %rcx,-0x20(%rsp)
4e:  c3                      retq
```

First of all, the *optimized* function is bigger than the original one, quite odd. Second, the second instruction (at the offset 0x7), is invalid. Its opcode [24] doesn't correspond to any instruction that the processor supports... We can also see that at the offsets 0x15 and 0x1d, the opcode are not recognized.
To be sure of what causes the problem, we look at the optimized process' execution instruction per instruction. When the program reach the instruction at the offset 0x9, it crashes. So the problem was effectively this instruction (and the other two might cause a problem too).
The culprit is RetDec. As said in the section 3, the 64-bit version of RetDec has been used but it doesn't work as great as 32-bit decompilation. If we take a look at the LLVM IR generated by RetDec, we clearly see that there was some misunderstanding:

```
define i64 @simple_add() local_unnamed_addr {
dec_label_pc_116a:
    %rbp.global-to-local = alloca i64, align 8
    %rdi.global-to-local = alloca i64, align 8
    %rsi.global-to-local = alloca i64, align 8
    %rsp.global-to-local = alloca i64, align 8
    store i64 0, i64* %rsp.global-to-local, align 8
    %v0_116a = load i64, i64* %rbp.global-to-local, align 8
    %v1_116a = load i64, i64* %rsp.global-to-local, align 8
    %v2_116a = add i64 %v1_116a, -8
    %v3_116a = inttoptr i64 %v2_116a to i64*
    store i64 %v0_116a, i64* %v3_116a, align 8
    store i64 %v2_116a, i64* %rbp.global-to-local, align 8
    %v0_116e = load i64, i64* %rdi.global-to-local, align 8
    %v1_116e = trunc i64 %v0_116e to i32
    %v3_116e = add i64 %v1_116a, -12
    %v4_116e = inttoptr i64 %v3_116e to i32*
    store i32 %v1_116e, i32* %v4_116e, align 4
    %v0_1171 = load i64, i64* %rsi.global-to-local, align 8
    %v1_1171 = trunc i64 %v0_1171 to i32
    %v2_1171 = load i64, i64* %rbp.global-to-local, align 8
    %v3_1171 = add i64 %v2_1171, -8
    %v4_1171 = inttoptr i64 %v3_1171 to i32*
    store i32 %v1_1171, i32* %v4_1171, align 4
    %v0_1174 = load i64, i64* %rbp.global-to-local, align 8
    %v1_1174 = add i64 %v0_1174, -4
    %v2_1174 = inttoptr i64 %v1_1174 to i32*
    %v3_1174 = load i32, i32* %v2_1174, align 4
    %v1_1177 = add i64 %v0_1174, -8
    %v2_1177 = inttoptr i64 %v1_1177 to i32*
    %v3_1177 = load i32, i32* %v2_1177, align 4
    %v4_117a = add i32 %v3_1177, %v3_1174
    %v20_117a = zext i32 %v4_117a to i64
    %v0_117c = load i64, i64* %rsp.global-to-local, align 8
    %v1_117c = inttoptr i64 %v0_117c to i64*
    %v2_117c = load i64, i64* %v1_117c, align 8
    store i64 %v2_117c, i64* %rbp.global-to-local, align 8
    ret i64 %v20_117a

;   uselistorder directives
    uselistorder i64* %rbp.global-to-local, { 0, 2, 3, 4, 1 }
    uselistorder i64 -8, { 1, 2, 0 }
    uselistorder i32 1, { 0, 2, 1, 3 }
}
```

The LLVM IR is really big for a simple addition and the function signature is not interpreted correctly: RetDec seems to think that the function requires no arguments.

Conclusion

In conclusion for this simple addition function, the 32-bit version works great but the 64-bit doesn't because of RetDec. We consider this a normal behaviour since we use a modified version of RetDec with enabled 64-bit decompilation that is not officially supported. There is no solution but waiting for an official 64-bit support or implementing it, but that's not the scope of our work.

The 32-bit version could be optimized even more by implementing constant propagation, which means that the optimized function would pre-calculate the result and simply return the 5 instead of doing the operation itself.

6.2 Simple multiplication

This example is quite the same as the first one but this time, the target program multiplies two integers. The code is the following:

simple_mul.c

```c
/*
 * File: simple_mul.c
 *
 * Created by: Lucas Elisei <lucas.elisei@heig-vd.ch>
 *
 * Multiplies two integers and prints the result.
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int simple_mul(int a, int b) {
    return a * b;
}

int main(int argc, char **argv) {
    int a, b, rc;

    if (argc != 3) {
        fprintf(stderr, "[simple_mul] Usage: %s <a> <b>\n", argv[0]);

        return EXIT_FAILURE;
    }

    // Retrieve the arguments.
    a = atoi(argv[1]);
    b = atoi(argv[2]);

    // Multiply the arguments.
    rc = simple_mul(a, b);

    // Print the result.
    fprintf(stdout, "[simple_mul] Result: %d\n", rc);

    return EXIT_SUCCESS;
}
```

The 32- and 64-bit versions of this program are respectively named `simple_mul32` and `simple_mul64`.

6.2.1 32-bit

For compilation, the command is the same as the one used in the first example. Below, the machine code corresponding to the non-optimized `simple_mul` function:

```
000011bd <simple_mul>:
    11bd:    55                     push   %ebp
    11be:    89 e5                  mov    %esp,%ebp
    11c0:    e8 cb 00 00 00         call   1290 <__x86.get_pc_thunk.ax>
```

```
    11c5:   05 3b 2e 00 00          add    $0x2e3b,%eax
    11ca:   8b 45 08                mov    0x8(%ebp),%eax
    11cd:   0f af 45 0c             imul   0xc(%ebp),%eax
    11d1:   5d                      pop    %ebp
    11d2:   c3                      ret
```

We see that the size of the function is 21 bytes and is composed of 8 instructions. Now, we call the `optimizer` program to optimize the function:

```
./optimizer -f simple_mul simple_mul32 3 2
```

The program prints `[simple_mul] Result: 0`, which is not really the result we expected. Let's take a look at the optimized function's machine code:

```
f7f66000 <simple_mul>:
    0:  8b 44 24 04             mov    0x4(%esp),%eax
    4:  0f af 44 24 0c          imul   0xc(%esp),%eax
    9:  c3                      ret
```

At first glance, the machine code seems to not contain any error. But if we look closely at the second instruction, which loads an argument on the stack, the offset is wrong. It is `0xc` and should be `0x8` because the function arguments are 4-byte long, and not 8. There are two possibilities:

1. The compiler thinks that there are two arguments: one is 4-byte long and the other is 8-byte long ;
2. The compiler thinks that there are three arguments, each 4-byte long, and the second one is useless.

Taking a look at the generated LLVM IR from RetDec might help us solve the mystery. Here the LLVM IR representing the `simple_mul` function:

```
1 define i32 @simple_mul(i64 %arg1, i32 %arg2) local_unnamed_addr {
2 dec_label_pc_11bd:
3     %v4_11ca = trunc i64 %arg1 to i32
4     %v7_11cd = mul i32 %v4_11ca, %arg2
5     ret i32 %v7_11cd
6 }
```

If we look at the function's signature, we can see that it is wrong. The first argument is interpreted as 64-bit integer, which is false. At line 3, we see that the decompiler casts the first argument to a 32-bit integer. So RetDec gets the final type right but not the arguments.
The result printed by the optimized binary can be explained as follow: the compiler retrieves one of the two arguments 4 bytes farther than expected, resulting in an unexpected value.

6.2.2 64-bit

To compile the `simple_mul64` program, we use the following command:

```
gcc -Wall -Werror -O0 simple_mul.c -o simple_mul64
```

Below, the machine code corresponding to the non-optimized `simple_mul` function:

```
000000000000116a <simple_mul>:
    116a:   55                      push   %rbp
    116b:   48 89 e5                mov    %rsp,%rbp
    116e:   89 7d fc                mov    %edi,-0x4(%rbp)
    1171:   89 75 f8                mov    %esi,-0x8(%rbp)
    1174:   8b 45 fc                mov    -0x4(%rbp),%eax
    1177:   0f af 45 f8             imul   -0x8(%rbp),%eax
    117b:   5d                      pop    %rbp
    117c:   c3                      retq
```

Before optimization, the size of the function is 18 bytes and is composed of 8 instructions. The next step is to call the `optimizer` program with the same arguments as for the 32-bit version:

```
./optimizer -f simple_mul simple_mul64 3 2
```

As for the 64-bit version of the first example, the optimized doesn't print any message. Let's take a look at the optimized function's machine code:

```
7f807a326000 <simple_mul>:
    0:  48 c7 44 24 e8 00 00    movq   $0x0,-0x18(%rsp)
    7:  00 00
    9:  48 8b 44 24 e0          mov    -0x20(%rsp),%rax
    e:  48 89 04 25 f8 ff ff    mov    %rax,0xfffffffffffffff8
   15:  ff
   16:  48 c7 44 24 e0 f8 ff    movq   $0xfffffffffffffff8,-0x20(%rsp)
   1d:  ff ff
   1f:  8b 44 24 f8             mov    -0x8(%rsp),%eax
   23:  89 04 25 f4 ff ff ff    mov    %eax,0xfffffffffffffff4
   2a:  8b 44 24 f0             mov    -0x10(%rsp),%eax
   2e:  48 8b 4c 24 e0          mov    -0x20(%rsp),%rcx
   33:  89 41 f8                mov    %eax,-0x8(%rcx)
   36:  48 8b 44 24 e0          mov    -0x20(%rsp),%rax
   3b:  8b 48 fc                mov    -0x4(%rax),%ecx
   3e:  48 63 40 f8             movslq -0x8(%rax),%rax
   42:  48 0f af c1             imul   %rcx,%rax
   46:  48 8b 4c 24 e8          mov    -0x18(%rsp),%rcx
   4b:  48 8b 09                mov    (%rcx),%rcx
   4e:  48 89 4c 24 e0          mov    %rcx,-0x20(%rsp)
   53:  c3                      retq
```

We see the same behaviour as for the first example: one or more instructions are not recognized by the processor, resulting in the crash of the program.

Conclusion

The 32-bit version of the optimized binary does the good operation but with the wrong values since RetDec wrongly decompiles the function[1]. For 64-bit, the behaviour is the same as the first example.

The implementation of constant propagation might fix the problem encountered for the 32-bit version.

6.3 Matrices multiplication

For the third and last example, we will focus on a bigger program. This program performs the multiplication of two randomly generated matrices. It takes their dimensions as arguments. At the end of its execution, it prints the time taken by the target function to be executed.
The target function is named `matrix_mult` and its complexity is $\mathcal{O}(n^3)$, which means that the running time cubic grows as the input size grows.
The 32- and 64-bit versions of this program are respectively named `mmult32` and `mmult64`. The code is available at appendix D.5.

The correctness of the resulting matrix has been verified every time the program has been run.

6.3.1 32-bit

To compile the `mmult32` program, we use the following command:

```
gcc -Wall -Werror -O0 -m32 mmult.c main.c -o mmult32
```

Because the program measures the time the target function took to execute, we first do a run of the non-optimized program:

[1] We reported this bug on the RetDec's Github repository (issue #269).

```
./mmult32 1000 1000
Elapsed time for matrix multiplication (1000x1000): 9s 357689098ns
```

As said above, the function is quite complex so its machine code is long and hard to decrypt. However, it is available at appendix D.6.

The non-optimized function is 298 bytes long and is composed of 96 instructions. Now, we call the `optimizer` program to optimize the function with matrices of dimension `1000x1000` (which means 1 billion operations).

```
./optimizer -f matrix_mult mmult32 1000 1000
Elapsed time for matrix multiplication (1000x1000): 6s 260002895ns
```

Interesting: the target program doesn't crash, it prints a reasonable measurement and its value is less than the non-optimized run. Let's check if the resulting matrix is correct. And... yes, it is. Is this our first win?

Let's *joyfully* take a look at the optimized function machine code, available at the appendix D.6. It is 339 bytes long and is composed of 144 instructions. How come the optimized function is bigger than the normal one and still it is faster?
The instructions used might be the answer. Indeed, depending on their complexity, some instructions take more clock cycle to execute than other. For example, the optimized machine code contains 39 `NOP` instructions, which take only one clock cycle to execute.

Some measurements have been done on this example. The non-optimized binary and the optimized one have been run 30 times each and an average of the execution time for the `matrix_mult` function (on `1000x1000` matrices) has been calculated:
On average, the non-optimized function took 10.04 seconds execute. On average, the optimized function took 7.46 seconds to execute. Hence, the optimized function shows a 25% increase in performance.

However, it is interesting to measure the total time that the `optimizer` took to optimize the target binary. On average, it took 1.66 to optimize. Hence, counting in the time for `optimizer` to execute, the runs still show a 10% increase in performance.

6.3.2 64-bit

As for the previous examples, the 64-bit *optimized* machine code contains undefined instructions that make the program crash.

Conclusion

Thanks to this last example, we tested `liboptimizer` on a target program that does a lot of calculation. No surprises for the 64-bit version: the decompilation is in its early stages so it doesn't give satisfactory results but the 32-bit version shows a 25% increase in performance which is a really promising result.

More optimization could be done thanks to an LLVM pass named Polly [31].

7 Conclusion

The Moore's law is coming to an end and emerging computer science fields like machine learning or artificial intelligence require a lot of computational power.

To tackle such problems, we propose a method to optimize the bottlenecks of any binaries by identifying them, recompiling them specifically for the platform they are being run on and do it without the target program knowing it.

Through examples, we saw that our solution can enhance a binary performance up to 25%. Unfortunately, the decompiler that exist nowadays are still in an early stage of development, thus limiting the capacities of such a method.

Some improvements can be made on the library we developed for this project. For instance, it could be possible to merge the solicited parts of RetDec and our project to produce a standalone application that would not depend on RetDec being installed on the machine to run the library.
Another feature could be to save the patched binary to avoid calling the library each time the target process is being run.

With the constant development of RetDec, it might be possible that the 64-bit support of RetDec could come in next months. Testing the library with this hypothetically new version of RetDec might demonstrate better results on 64-bit binaries.

8 Acknowledgements

I would like to thank Alberto Dassatti for supervising this project and introducing me to the wonderful possibilities of recompilation.

I also would like to thank the HEIG-VD for giving me the opportunity to attend the Pass the SALT conference to meet and discuss with the RetDec team.

A last thank to all my friends and my family for being supportive during this project, taking the time to give me feedback about my work and being awesome.

References

[1] *Avast Software, Inc.* URL: https://www.avast.com/.

[2] *Building a JIT.* URL: https://llvm.org/docs/tutorial/BuildingAJIT1.html.

[3] *Capstone Framework.* URL: https://www.capstone-engine.org.

[4] *Dolphin Emulator.* URL: https://dolphin-emu.org/.

[5] L. Ďurfina et al. *Design of a Retargetable Decompiler for a Static Platform-Independent Malware Analysis.* URL: http://www.sersc.org/journals/IJSIA/vol5_no4_2011/8.pdf.

[6] Eldad Eilam. *Reversing: Secrets of Reverse Engineering.* John Wiley & Sons, 2005. URL: https://archive.org/details/reversing-secrets-of-reverse-engineering_2.

[7] *End of Moore's Law: It's not just about physics.* URL: https://www.scientificamerican.com/article/end-of-moores-law-its-not-just-about-physics/.

[8] *Executable and Linkable Format.* In: *Wikipedia.* URL: https://en.wikipedia.org/w/index.php?title=Executable_and_Linkable_Format&oldid=844622391.

[9] *ExecutionEngine class.* URL: http://llvm.org/doxygen/classllvm_1_1ExecutionEngine.html.

[10] *GCC, the GNU Compiler Collection.* URL: https://gcc.gnu.org/.

[11] *GIMPLE.* URL: https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html.

[12] *GPLv2.* URL: https://opensource.org/licenses/GPL-2.0.

[13] Maurice H. Halstead. *Machine-Independent Computer Programming.* Spartan Books, 1962.

[14] Adam Husár et al. *Automatic C Compiler Generation from Architecture Description Language ISAC.* URL: http://drops.dagstuhl.de/opus/volltexte/2011/3065/pdf/9.pdf.

[15] *IDA.* URL: https://www.hex-rays.com/products/ida/.

[16] Hex-Rays SA Ilfak Guilfanov. *Decompilers and beyond.* 2008. URL: https://www.hex-rays.com/products/ida/support/ppt/decompilers_and_beyond_white_paper.pdf.

[17] *JITEventListener class.* URL: http://llvm.org/doxygen/classllvm_1_1JITEventListener.html.

[18] J. Křoustek and F. Pokorný. "Reconstruction of instruction idioms in a retargetable decompiler". In: *2013 Federated Conference on Computer Science and Information Systems.* Sept. 2013, pp. 1519–1526.

[19] *LLVM.* URL: https://llvm.org.

[20] *LLVM Intermediate Representation.* URL: https://llvm.org/docs/LangRef.html.

[21] *Mac 68k emulator.* URL: https://en.wikipedia.org/wiki/Mac_68k_emulator.

[22] *MIT License.* URL: https://opensource.org/licenses/MIT.

[23] *mmap2.* URL: http://man7.org/linux/man-pages/man2/mmap2.2.html.

[24] *Opcode.* URL: https://en.wikipedia.org/wiki/Opcode.

[25] Maksim Panchenko et al. *BOLT: A Practical Binary Optimizer for Data Centers and Beyond.* URL: https://arxiv.org/abs/1807.06735.

[26] *Pandora console.* URL: https://en.wikipedia.org/wiki/Pandora_(console).

[27] *Pass the SALT 2018.* URL: https://2018.pass-the-salt.org/.

[28] *Pentium Pro.* URL: https://en.wikipedia.org/wiki/Pentium_Pro.

[29] *Playing StarCraft on an ARM.* URL: https://hackaday.com/2014/07/31/playing-starcraft-on-an-arm/.

[30] *Politecnico di Milano.* URL: https://www.polimi.it/.

[31] *Polly - Polyhedral optimizations for LLVM.* URL: https://polly.llvm.org/.

[32] *ptrace.* URL: http://man7.org/linux/man-pages/man2/ptrace.2.html.

[33] *QEMU.* URL: https://www.qemu.org.

[34] *RetDec.* URL: https://retdec.com.

[35] *RetDec Github Issue 269.* URL: https://github.com/avast-tl/retdec/issues/269.

[36] *RetDec Github Repository.* URL: https://github.com/avast-tl/retdec.

[37] *rev.ng*. URL: https://rev.ng.

[38] *System call*. URL: https://en.wikipedia.org/wiki/System_call.

[39] *TFA - Transparent Live Code Offloading on FPGA*. URL: http://reds.heig-vd.ch/en/rad/projects/tfa.

Appendices

A Authentication

I, Lucas Elisei, hereby declare having realized this work alone and not having used any other resources than those quoted in the bibliography

Par la présente, je soussigné, Lucas Elisei, déclare avoir réalisé seul ce travail et ne pas avoir utilisé d'autres sources que celles citées dans la bibliographie.

Date	Signature

Lucas Elisei

B Requirements

Since we mainly work on ELF binaries, it is highly recommended to use a Linux distribution to build and run the project.

The second step is to download a `32-bit` toolchain and the LLVM compiler. You can install them with your preferred package manager. We recommend using the LLVM version 5.

Finally, the most important step, you must download and build RetDec. To do so, clone the RetDec's Github repository [36] and follow the build instructions.

C Simple decompilation example

include/simple.h

```c
#ifndef LIB_SIMPLE_H
#define LIB_SIMPLE_H

#include <stdint.h>

/*
 * Simple addition of two integers.
 *
 * Returns the result of the addition.
 */
uint32_t simple_add(uint32_t *a, uint32_t *b);

/*
 * Simple addition of two integers. The result is stored at the address of the
 * third parameter.
 */
void simple_add_ref(uint32_t *a, uint32_t *b, uint32_t *result);

#endif
```

src/simple.c

```c
#include <stdint.h>

#include "simple.h"

uint32_t simple_add(uint32_t *a, uint32_t *b) {
    return *a + *b;
}

void simple_add_ref(uint32_t *a, uint32_t *b, uint32_t *result) {
    *result = *a + *b;
}
```

src/main.c

```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

#include "simple.h"

int main(int argc, char** argv) {
    uint32_t a, b;

    a = atoi(argv[1]);
    b = atoi(argv[2]);

    fprintf(stdout, "Result: %u\n", simple_add(&a, &b));

    return EXIT_SUCCESS;
}
```

Makefile

```
SHELL := /bin/bash

CC = gcc
override CFLAGS += -std=c99 -Wall -Werror -pedantic -Iinclude -m32

LDFLAGS = -m32

CLANG = clang-5.0
LLC = llc-5.0
LLC_FLAGS = -march=x86

SRC_DIR = src

# Logs directory.
LOGS_DIR = logs
# Rule to create logs.
LOGS_RULE = $(shell date "+%Y%m%d-%H%M%S")
LOGS_PATH := $(LOGS_DIR)/$(LOGS_RULE)

# Binary options.
BIN = main
BIN_SRC = $(wildcard $(SRC_DIR)/*.c)
BIN_OBJ = $(patsubst %.c,%.o,$(BIN_SRC))
# Arguments to pass to the translated binary.
BIN_ARGS = 4 5

# RetDec options.
RETDEC_DIR = $(HOME)/opt/retdec
RETDEC_BIN = $(RETDEC_DIR)/bin/retdec-decompiler.sh
RETDEC_FLAGS = --stop-after bin2llvmir
# Functions to decompile (temporary).
RETDEC_FUNCS = simple_add
# Only select some functions if asked to.
ifdef RETDEC_FUNCS
RETDEC_FLAGS += --select-functions $(RETDEC_FUNCS)
endif

.PHONY: all clean decompile recompile

all: recompile

decompile: $(BIN)
	$(RETDEC_BIN) $(RETDEC_FLAGS) $(BIN)
	@mkdir -p $(LOGS_PATH)
	@mv -f $(BIN)* $(LOGS_PATH)/

recompile: decompile
	@cp $(SRC_DIR)/$(BIN).o $(LOGS_PATH)/$(BIN).o
	@sed -i '/@__x86.get_pc_thunk.ax()/d' $(LOGS_PATH)/$(BIN).c.backend.ll
	$(LLC) $(LLC_FLAGS) $(LOGS_PATH)/$(BIN).c.backend.ll -o $(LOGS_PATH)/$(BIN).c.backend.s
	$(CC) $(CFLAGS) -c $(LOGS_PATH)/$(BIN).c.backend.s -o $(LOGS_PATH)/$(BIN).c.backend.o
	$(CC) $(LDFLAGS) $(LOGS_PATH)/$(BIN).o $(LOGS_PATH)/$(BIN).c.backend.o -o $(LOGS_PATH)/$(BIN).translated
	@echo ---
	@echo Testing translated binary with parameters: $(BIN_ARGS)
	@./$(LOGS_PATH)/$(BIN).translated $(BIN_ARGS)

$(BIN): $(BIN_OBJ)
	$(CROSS_COMPILE)$(CC) $(CFLAGS) $^ -o $@

%.o: %.c
	$(CROSS_COMPILE)$(CC) $(CFLAGS) -c $< -o $@

clean:
	rm -rf $(BIN_OBJ)
	rm -rf $(BIN)*
```

D liboptimizer

D.1 elfparser

elfparser.c

```
/*
 * File: elfparser.c
 *
 * Created by: Lucas Elisei <lucas.elisei@heig-vd.ch>
 */

#include <elf.h>
#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <sys/mman.h>

#include "debug.h"
#include "liboptimizer.h"
#include "elfparser.h"

typedef struct {
    uint8_t     *mem;       // Binary mapped into memory
    long        file_size;  // Binary size.
    uint8_t     is64;       // 32- or 64-bit
    uint64_t    sh_off;     // Points to the start of the section header table
    uint16_t    sh_num;     // Number of entries in the section header table
    uint16_t    sh_entsize; // Section header size
    uint16_t    sh_strndx;  // Index of SH String table.
    uint64_t    sym_off;    // Symbol table offset
    uint64_t    sym_size;   // Symbol table size
    uint64_t    str_off;    // String table size
} elf_file_t;

static void _elf_dump_file(elf_file_t *elf) {
    DBG("=== Printing elf_file_t at 0x%" PRIXPTR "\n", (uintptr_t)elf);
    DBG("  is64......: %"   PRIu8  "\n", elf->is64);
    DBG("  sh_off....: 0x%" PRIX64 "\n", elf->sh_off);
    DBG("  sh_num....: %"   PRIu16 "\n", elf->sh_num);
    DBG("  sh_entsize: %"   PRIu16 "\n", elf->sh_entsize);
    DBG("  sh_strndx.: %"   PRIu16 "\n", elf->sh_strndx);
    DBG("  sym_off...: 0x%" PRIX64 "\n", elf->sym_off);
    DBG("  sym_size..: %"   PRIu64 "\n", elf->sym_size);
    DBG("  str_off...: 0x%" PRIX64 "\n", elf->str_off);
    DBG("===================\n\n");
}

static char *_elf_resolve_symbol(elf_file_t *elf, uint64_t address) {
    int i;
    size_t offset;
    // Symbol header for 32- and 64-bit.
    Elf32_Sym sym32;
    Elf64_Sym sym64;
    char *symbol;
    size_t sym_len;

    // ELF32
    if (elf->is64 == 0) {
        // Iterate over all symbols.
        for (i = 0; i * sizeof(Elf32_Sym) < elf->sym_size; ++i) {
            // Calculate offset and get symbol.
            offset = elf->sym_off + (sizeof(Elf32_Sym) * i);
            memmove(&sym32, elf->mem + offset, sizeof(Elf32_Sym));

            // If the symbol is at the address we are looking for.
            if (sym32.st_value == address) {
                // Sanity check.
                if (sym32.st_name == 0) {
```

```
                    return NULL;
                }

                // Get symbol size for memory allocation.
                offset = elf->str_off + sym32.st_name;
                sym_len = strlen((char *)(elf->mem + offset));

                // Allocate memory for symbol.
                symbol = (char *)calloc(1, sizeof(char) * (sym_len + 1));
                if (symbol == NULL) {
                    return NULL;
                }

                // Copy symbol.
                sprintf(symbol, "%s", (char *)(elf->mem + offset));
            }
        }
    }
    // ELF64
    else {
        // Iterate over all symbols.
        for (i = 0; i * sizeof(Elf64_Sym) < elf->sym_size; ++i) {
            // Calculate offset and get symbol.
            offset = elf->sym_off + (sizeof(Elf64_Sym) * i);
            memmove(&sym64, elf->mem + offset, sizeof(Elf64_Sym));

            if (sym64.st_value == address) {
                // Sanity check.
                if (sym64.st_name == 0) {
                    return NULL;
                }

                // Get symbol size for memory allocation.
                offset = elf->str_off + sym64.st_name;
                sym_len = strlen((char *)(elf->mem + offset));

                // Allocate memory for symbol.
                symbol = (char *)calloc(1, sizeof(char) * (sym_len + 1));
                if (symbol == NULL) {
                    return NULL;
                }

                // Copy symbol.
                sprintf(symbol, "%s", (char *)(elf->mem + offset));
            }
        }
    }

    return symbol;
}

static uint64_t _elf_resolve_address(elf_file_t *elf, const char *symbol) {
    int i;
    size_t offset;
    // Symbol header for 32- and 64-bit.
    Elf32_Sym sym32;
    Elf64_Sym sym64;

    // ELF32
    if (elf->is64 == 0) {
        // Iterate over all symbols.
        for (i = 0; i * sizeof(Elf32_Sym) < elf->sym_size; ++i) {
            // Calculate offset and get symbol struct.
            offset = elf->sym_off + (sizeof(Elf32_Sym) * i);
            memmove(&sym32, elf->mem + offset, sizeof(Elf32_Sym));

            // Sanity check.
            if (sym32.st_name == 0) {
                continue;
            }

            // Calculate symbol offset.
            offset = elf->str_off + sym32.st_name;
```

```

            // If the symbol is equal to the one we are looking for, return
            // its value.
            if (!strcmp((char *)(elf->mem + offset), symbol)) {
                return (uint64_t)sym32.st_value;
            }
        }
    }
    // ELF64
    else {
        // Iterate over all symbols.
        for (i = 0; i * sizeof(Elf64_Sym) < elf->sym_size; ++i) {
            // Calculate offset and get symbol.
            offset = elf->sym_off + (sizeof(Elf64_Sym) * i);
            memmove(&sym64, elf->mem + offset, sizeof(Elf64_Sym));

            // Sanity check.
            if (sym64.st_name == 0) {
                continue;
            }

            // Calculate symbol offset.
            offset = elf->str_off + sym64.st_name;

            // If the symbol is equal to the one we are looking for, return
            // its value.
            if (!strcmp((char *)(elf->mem + offset), symbol)) {
                return (uint64_t)sym64.st_value;
            }
        }
    }

    return 0;
}

static int _elf_resolve_sections(elf_file_t *elf) {
    int i;
    size_t offset;
    uint64_t shstrtab_off;
    // Section header for 32- and 64-bit.
    Elf32_Shdr sec32;
    Elf64_Shdr sec64;

    // ELF32
    if (elf->is64 == 0) {
        // We need to get the Section Header STRing TABle offset before others.
        offset = elf->sh_off + (elf->sh_entsize * elf->sh_strndx);
        memmove(&sec32, elf->mem + offset, sizeof(Elf32_Shdr));
        shstrtab_off = sec32.sh_offset;

        // Iterate over all section headers.
        for (i = 0; i < elf->sh_num; ++i) {
            // Calculate offset and get section header.
            offset = elf->sh_off + (elf->sh_entsize * i);
            memmove(&sec32, elf->mem + offset, sizeof(Elf32_Shdr));

            switch (sec32.sh_type) {
                // Static symbols table.
                case SHT_SYMTAB:
                    elf->sym_off = sec32.sh_offset;
                    elf->sym_size = sec32.sh_size;
                    break;

                // String table. Since there are more than one string table, we
                // have to be sure to get the .strtab one.
                case SHT_STRTAB:
                    if (!strcmp((char *)(elf->mem + shstrtab_off + sec32.sh_name), ".strtab")) {
                        elf->str_off = sec32.sh_offset;
                    }
                    break;

                default:
                    break;
```

```
                }
            }
        }
        // ELF64
        else {
            // We need to get the Section Header STRing TABle offset before others.
            offset = elf->sh_off + (elf->sh_entsize * elf->sh_strndx);
            memmove(&sec64, elf->mem + offset, sizeof(Elf64_Shdr));
            shstrtab_off = sec64.sh_offset;

            // Iterate over all section headers.
            for (i = 0; i < elf->sh_num; ++i) {
                // Calculate offset and get section header.
                offset = elf->sh_off + (elf->sh_entsize * i);
                memmove(&sec64, elf->mem + offset, sizeof(Elf64_Shdr));

                switch (sec64.sh_type) {
                    // Static symbols table.
                    case SHT_SYMTAB:
                        elf->sym_off = sec64.sh_offset;
                        elf->sym_size = sec64.sh_size;
                        break;

                    // String table.
                    case SHT_STRTAB:
                        if (!strcmp((char *)(elf->mem + shstrtab_off + sec64.sh_name), ".strtab")) {
                            elf->str_off = sec64.sh_offset;
                        }
                        break;

                    default:
                        break;
                }
            }
        }

        return 0;
    }

    static int _elf_read_header(elf_file_t *elf) {
        // The first bytes of the header are same-sized for 32- and 64-bit archs.
        // To identify the file's class and magic number, we assume it's 32-bit.
        Elf32_Ehdr hdr32;
        Elf64_Ehdr hdr64;
        int rc;

        // Retrieve ELF header.
        memmove(&hdr32, elf->mem, sizeof(Elf32_Ehdr));

        // Check that the file is a valid ELF.
        rc = (hdr32.e_ident[EI_MAG0] == ELFMAG0 &&
              hdr32.e_ident[EI_MAG1] == ELFMAG1 &&
              hdr32.e_ident[EI_MAG2] == ELFMAG2 &&
              hdr32.e_ident[EI_MAG3] == ELFMAG3);
        if (rc == 0) {
            fprintf(stderr, "[liboptimizer] ERROR: File is not a valid ELF\n");

            rc = -2;
            goto _elf_not_valid;
        }

        // Check ELF class.
        switch (hdr32.e_ident[EI_CLASS]) {
            case ELFCLASS32:
                elf->is64 = 0;
                elf->sh_off = (uint64_t)hdr32.e_shoff;
                elf->sh_num = hdr32.e_shnum;
                elf->sh_entsize = hdr32.e_shentsize;
                elf->sh_strndx = hdr32.e_shstrndx;
                break;
            case ELFCLASS64:
                memmove(&hdr64, elf->mem, sizeof(Elf64_Ehdr));
                elf->is64 = 1;
```

```
            elf->sh_off = hdr64.e_shoff;
            elf->sh_num = hdr64.e_shnum;
            elf->sh_entsize = hdr64.e_shentsize;
            elf->sh_strndx = hdr64.e_shstrndx;
            break;
        default:
            fprintf(stderr, "[liboptimizer] ERROR: Invalid ELF class\n");
            rc = -2;
            goto _elf_not_valid;
    }

    rc = 0;

_elf_not_valid:
    return rc;
}

static elf_file_t *_elf_init(const char *path) {
    elf_file_t *elf;
    FILE *file;

    // Allocate memory.
    elf = (elf_file_t *)calloc(1, sizeof(elf_file_t));
    if (elf == NULL) {
        perror("[liboptimizer] calloc()");

        elf = NULL;
        goto _failed_calloc;
    }

    // Open binary.
    file = fopen(path, "rb");
    if (file == NULL) {
        perror("[liboptimizer] fopen()");

        elf = NULL;
        goto _failed_fopen;
    }

    // Get size.
    fseek(file, 0L, SEEK_END);
    elf->file_size = ftell(file);

    // Map binary into memory.
    elf->mem = mmap(NULL, elf->file_size, PROT_READ, MAP_PRIVATE, fileno(file), 0);
    if (elf->mem == NULL) {
        perror("[liboptimizer] mmap()");

        elf = NULL;
        goto _failed_mmap;
    }

    fclose(file);

    return elf;

_failed_mmap:
    fclose(file);
_failed_fopen:
    free(elf);
_failed_calloc:
    return NULL;
}

static elf_file_t *_parse_elf(const char *path) {
    elf_file_t *elf;
    int rc;

    elf = _elf_init(path);
    if (elf == NULL) {
        fprintf(stderr, "[liboptimizer] ERROR: Failed to allocate memory\n");

        goto _failed_init;
```

```c
    }

    rc = _elf_read_header(elf);
    if (rc < 0) {
        fprintf(stderr, "[liboptimizer] ERROR: Error while parsing ELF\n");

        goto _failed_read_header;
    }

    rc = _elf_resolve_sections(elf);
    if (rc < 0) {
        fprintf(stderr, "[liboptimizer] ERROR: Error while resolving sections\n");

        goto _failed_resolve_sections;
    }

    return elf;

_failed_resolve_sections:
    munmap(elf->mem, elf->file_size);
_failed_read_header:
    free(elf);
_failed_init:
    return NULL;
}

char *get_symbol_at_address(const char *path, uint64_t address) {
    elf_file_t *elf;
    char *symbol;

    elf = _parse_elf(path);
    if (elf == NULL) {
        symbol = NULL;

        goto _failed_parse;
    }

    _elf_dump_file(elf);

    symbol = _elf_resolve_symbol(elf, address);
    if (symbol == NULL) {
        fprintf(stderr, "[liboptimizer] ERROR: Error while retrieving symbols info\n");

        goto _failed_sym_info;
    }

_failed_sym_info:
    munmap(elf->mem, elf->file_size);
    free(elf);
_failed_parse:
    return symbol;
}

uint64_t get_symbol_address(const char *path, const char *symbol) {
    elf_file_t *elf;
    uint64_t address;

    elf = _parse_elf(path);
    if (elf == NULL) {
        address = 0;

        goto _failed_parse;
    }

    _elf_dump_file(elf);

    address = _elf_resolve_address(elf, symbol);
    if (address == 0) {
        fprintf(stderr, "[liboptimizer] ERROR: Error while retrieving address\n");

        goto _failed_resolve_addr;
    }

```

```c
_failed_resolve_addr:
    munmap(elf->mem, elf->file_size);
    free(elf);
_failed_parse:
    return address;
}

int8_t is64bit(const char *path) {
    elf_file_t *elf;
    int8_t is64;

    elf = _parse_elf(path);
    if (elf == NULL) {
        fprintf(stderr, "[liboptimizer] ERROR: Failed to allocate memory\n");

        return -1;
    }

    is64 = (int8_t)elf->is64;

    munmap(elf->mem, elf->file_size);
    free(elf);

    return is64;
}
```

D.2 retdec

retdec.c

```
/*
 * File: retdec.c
 *
 * Created by: Lucas Elisei <lucas.elisei@heig-vd.ch>
 */

#define _POSIX_C_SOURCE 200809L
#define _XOPEN_SOURCE 500

#include <fcntl.h>
#include <ftw.h>
#include <libgen.h>
#include <limits.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>

#include "debug.h"
#include "liboptimizer.h"
#include "retdec.h"
#include "TFAJITWrapper.h"

int _remove_file(const char *fpath, const struct stat *sb, int typeflag, struct FTW *ftwbuf) {
    int rv = remove(fpath);

    if (rv) {
        perror((char*)fpath);
    }

    return rv;
}

static int _compile_llvmir(process_info_t *process, char *tmp_directory) {
    int rc;

    rc = compile_llvmir_file(process, tmp_directory);

    return rc;
}

static int _execute_script(process_info_t *process, char *tmp_directory) {
    int log_fd;
    char log_path[PATH_MAX];
    char tmp_binary[PATH_MAX];
    pid_t pid;

    sprintf(log_path, "%s/decompile.log", tmp_directory);
    sprintf(tmp_binary, "%s/%s", tmp_directory, basename((char *)process->path));

    log_fd = open(log_path, O_RDWR | O_CREAT, 0666);
    if (log_fd < 0) {
        perror("[liboptimizer] open()");

        return -1;
    }

    pid = fork();
    if (pid == 0) {
        dup2(log_fd, STDOUT_FILENO);
        dup2(log_fd, STDERR_FILENO);
        close(log_fd);

        execl("/bin/bash", "sh", RETDEC_DECOMPILER,
            "--stop-after", "bin2llvmir",
```

```
            "--select-functions", process->function_name,
            "--output", tmp_binary,
            process->path, NULL);

        exit(0);
    }

    waitpid(pid, NULL, 0);

    return 0;
}

int retdec_recompile(process_info_t *process) {
    // Store return codes.
    int rc;
    // Temporary directory name.
    char *tmp_dir;
    // Buffer to store temporary directory template.
    char template_buffer[strlen(TMP_DIR_TEMPLATE) + 1];

    sprintf(template_buffer, TMP_DIR_TEMPLATE);

    // Create a temporary directory to store temporary files.
    tmp_dir = mkdtemp(template_buffer);
    if (tmp_dir == NULL) {
        perror("[liboptimizer] mkdtemp");

        return -1;
    }

    DBG("Created %s temporary directory\n", tmp_dir);

    // Execute RetDec's script into temporary directory.
    rc = _execute_script(process, tmp_dir);
    if (rc != 0) {
        fprintf(stderr, "[liboptimizer] ERROR: An error occured while executing decompilation
         ↪   script\n");

        goto _delete_tmp_dir;
    }

    // Call JIT compiler.
    rc = _compile_llvmir(process, tmp_dir);
    if (rc != 0) {
        fprintf(stderr, "[liboptimizer] ERROR: An error occured during compilation of the optimized
         ↪   function\n");

        goto _delete_tmp_dir;
    }

    rc = 0;


_delete_tmp_dir:
#ifndef LIBOPTIMIZER_DEBUG
    if (nftw(tmp_dir, _remove_file, 64, FTW_DEPTH | FTW_PHYS)) {
        perror("[liboptimizer] ntfw()");
    }
#endif

    return rc;
}
```

D.3 jit

TFAJITEventListener.hpp

```
/*
 * File: TFAJITEventListener.hpp
 *
 * Created by: Lucas Elisei <lucas.elisei@heig-vd.ch>
 */

#ifndef __LIBOPTIMIZER_INCLUDE_TFAJITEVENTLISTENER_H__
#define __LIBOPTIMIZER_INCLUDE_TFAJITEVENTLISTENER_H__

#include "llvm/ExecutionEngine/JITEventListener.h"
#include "llvm/Object/SymbolSize.h"

#include <iostream>
#include <map>

using namespace llvm;
using namespace llvm::object;

typedef struct {
    uint64_t address;
    uint64_t size;
} symbol_info_t;

class TFAJITEventListener : public JITEventListener {

private:
    // Map used to store symbols information.
    std::map<std::string, symbol_info_t *> symbolsMap;

public:
    // Default constructor.
    TFAJITEventListener() {}
    // Default destructor.
    ~TFAJITEventListener() {
        for (auto it = symbolsMap.begin(); it != symbolsMap.end(); ++it) {
            free(it->second);
        }
    }

    // Function called when the JIT has emitted an object file.
    virtual void NotifyObjectEmitted(const ObjectFile &obj, const RuntimeDyld::LoadedObjectInfo &L) {
        OwningBinary<ObjectFile> OWOF = L.getObjectForDebug(obj);
            ObjectFile &OF = *OWOF.getBinary();

        // Iterate over symbols and their respective size.
        for (const std::pair<SymbolRef, uint64_t> &pair : computeSymbolSizes(OF)) {
            SymbolRef symbolRef = std::get<0>(pair);
            uint64_t size = std::get<1>(pair);

            // Symbol with an empty size aren't interesting.
            if (size > 0) {
                symbol_info_t *symbol_info;

                // Allocate memory for symbol_info.
                symbol_info = (symbol_info_t *)calloc(1, sizeof(symbol_info_t));
                if (symbol_info == NULL) {
                    std::cerr << "ERROR: Could not allocate memory" << std::endl;

                    continue;
                }

                // Retrieve address of symbol.
                Expected<uint64_t> eAddr = symbolRef.getAddress();
                if (!eAddr) {
                    continue;
                }

                // Assign fields.
```

```
                symbol_info->size = size;
                symbol_info->address = *eAddr;

                // Insert symbol info into map.
                symbolsMap[symbolRef.getName().get().str()] = symbol_info;
            }
        }
    }

    // Returns a pointer on a symbol_info struct corresponding to the argument.
    // Returns NULL if the symbol could not be found.
    symbol_info_t *GetSymbolInfo(const std::string symbol) {
        return symbolsMap[symbol];
    }
};

#endif
```

TFAJITWrapper.cpp

```
/*
 * File: TFAJITWrapper.cpp
 *
 * Created by: Lucas Elisei <lucas.elisei@heig-vd.ch>
 */

#include "llvm/ExecutionEngine/ExecutionEngine.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Verifier.h"
#include "llvm/IRReader/IRReader.h"
#include "llvm/Support/CodeGen.h"
#include "llvm/Support/SourceMgr.h"

#include <climits>
#include <cstdlib>
#include <iostream>

#include "debug.h"
#include "liboptimizer.h"
#include "TFAJITEventListener.hpp"

using namespace llvm;

extern "C" {

int compile_llvmir_file(process_info_t *process, char *tmp_path) {
    SMDiagnostic err;
    static LLVMContext context;
    char path[PATH_MAX];
    char command[PATH_MAX];

    // Initialize targets.
    LLVMInitializeAllTargets();
    LLVMInitializeAllTargetMCs();
    LLVMInitializeAllTargetInfos();
    LLVMInitializeAllAsmPrinters();
    LLVMInitializeAllAsmParsers();
    LLVMInitializeAllDisassemblers();

    sprintf(path, "%s/%s.backend.ll", tmp_path, basename(process->path));

    // Remove occurences of __x86.get_pc_thunk.ax() function into LLVM IR file.
    sprintf(command, "sed -i '/@__x86.get_pc_thunk.ax/d' %s", path);
    system(command);

    // Parse LLVM IR file.
    std::unique_ptr<Module> module = parseIRFile(path, err, context);

    if (!module) {
        err.print(process->argv[0], llvm::errs());
```

```
        return -1;
    }

    // Verify that the module is valid.
    if (verifyModule(*module)) {
        std::cerr << "ERROR: The LLVM IR module is not valid" << std::endl;

        return -2;
    }

    // Initialize ExecutionEngine as a JIT Compiler.
    StringRef *arch;
    if (process->is64 == 0) {
        arch = new StringRef("x86");
    } else {
        arch = new StringRef("x86-64");
    }
    ExecutionEngine *EE = EngineBuilder(std::move(module))
            .setEngineKind(EngineKind::JIT)
            .setMArch(*arch)
            .setOptLevel(CodeGenOpt::Level::Aggressive)
            .setVerifyModules(true)
            .create();

    // Initialize JIT Event Listener.
    TFAJITEventListener *EL = new TFAJITEventListener();

    // Register JITEventListener.
    EE->RegisterJITEventListener(EL);

    // Compile module.
    EE->finalizeObject();

    DBG("=== ExecutionEngine %s\n", "dump");
    DBG("  triple: %s\n", EE->getTargetMachine()->getTargetTriple().str().c_str());
    DBG("  cpu...: %s\n", EE->getTargetMachine()->getTargetCPU().str().c_str());
    DBG("  layout: %s\n", EE->getDataLayout().getStringRepresentation().c_str());

    // Retrieve symbol info.
    std::string symbol_str(process->function_name);
    symbol_info_t *symbol_info = EL->GetSymbolInfo(symbol_str);
    if (symbol_info == NULL) {
        std::cerr << "ERROR: Could not retrieve recompiled symbol info" << std::endl;

        return -3;
    }

    symbol_info->address = EE->getFunctionAddress(process->function_name);

    // Assign fields.
    process->optimized_function_size = symbol_info->size;
    process->optimized_function = (uint8_t *)calloc(process->optimized_function_size, sizeof(uint8_t));
    if (process->optimized_function == NULL) {
        std::cerr << "ERROR: Could not allocate memory for optimized function" << std::endl;
    }
    memcpy(process->optimized_function, (void *)symbol_info->address,
     ↪   process->optimized_function_size);

    return 0;
}

}   // extern "C"
```

D.4 live-patcher

live-patcher.c

```
/*
 * File: ptrace.c
 *
 * Created by: Lucas Elisei <lucas.elisei@heig-vd.ch>
 */

#define _GNU_SOURCE

#include <fcntl.h>
#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include <sys/mman.h>
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/uio.h>
#include <sys/user.h>
#include <sys/wait.h>

#include "debug.h"
#include "liboptimizer.h"

#define MMAP2_SYSCALL_X32   192
#define MMAP2_SYSCALL_X64   9

/*
 * Creates an assembly unconditional jump for 32-bit binaries.
 *
 * MOV  _A_, %eax
 * JMP  *%eax
 */
#define MAKE_JUMP32(_A_)    {        \
    0xB8,                            \
    ((unsigned char *)&(_A_))[0],    \
    ((unsigned char *)&(_A_))[1],    \
    ((unsigned char *)&(_A_))[2],    \
    ((unsigned char *)&(_A_))[3],    \
    0xFF, 0xE0                       \
}

/*
 * Creates an assembly unconditional jump for 64-bit binaries.
 *
 * MOVABS   _A_, %rax
 * JMP      *%rax
 */
#define MAKE_JUMP64(_A_)    {        \
    0x48, 0xB8,                      \
    ((unsigned char *)&(_A_))[0],    \
    ((unsigned char *)&(_A_))[1],    \
    ((unsigned char *)&(_A_))[2],    \
    ((unsigned char *)&(_A_))[3],    \
    ((unsigned char *)&(_A_))[4],    \
    ((unsigned char *)&(_A_))[5],    \
    ((unsigned char *)&(_A_))[6],    \
    ((unsigned char *)&(_A_))[7],    \
    0xFF, 0xE0                       \
}

/*
 * Methods used for debugging purposes.
 */
#ifdef LIBOPTIMIZER_DEBUG
void print_regs(process_info_t *process) {
    struct user_regs_struct regs;
```

```
    ptrace(PTRACE_GETREGS, process->pid, NULL, &regs);

    DBG("Registers:\n");
    DBG("  ebp: 0x%" PRIX64 " (0x%llX)\n", ptrace(PTRACE_PEEKDATA, process->pid, regs.rbp, NULL), regs.rbp);
    DBG("  esp: 0x%" PRIX64 " (0x%llX)\n", ptrace(PTRACE_PEEKDATA, process->pid, regs.rsp, NULL), regs.rsp);
}

void print_stack(process_info_t *process) {
    long addr, value;
    int size;
    struct user_regs_struct regs;

    ptrace(PTRACE_GETREGS, process->pid, NULL, &regs);

    fprintf(stdout, "Stack:");

    if (process->is64 == 0) {
        size = 4;
    } else {
        size = 8;
    }

    for (addr = regs.rbp; addr > ((regs.rsp - 0x30) & ~0xFFu); addr -= size) {
        if (((addr ^ regs.rbp) & 0xFu) == 0) {
            fprintf(stdout, "\n");
            DBG("0x%0*lX: ", size * 2, addr);
        }
        value = ptrace(PTRACE_PEEKDATA, process->pid, addr, NULL);
        if (addr == regs.rsp) {
            fprintf(stdout, "*");
        }
        fprintf(stdout, "0x%0*lX ", size * 2, value);
    }
    fprintf(stdout, "\n");
}

void debug(process_info_t *process) {
    struct user_regs_struct regs;
    char c;
    uint64_t address;

    address = process->codesegment_address + process->function_offset;

    ptrace(PTRACE_GETREGS, process->pid, NULL, &regs);
    while (regs.rip != address) {
        ptrace(PTRACE_SINGLESTEP, process->pid, NULL, NULL);
        waitpid(process->pid, NULL, 0);
        ptrace(PTRACE_GETREGS, process->pid, NULL, &regs);
    }

    DBG("=== DEBUG COMMANDS ===\n");
    DBG("  p : print EBP and ESP registers\n");
    DBG("  s : print stack\n");
    DBG("  n : execute next instruction\n");
    DBG("  q : continue execution\n");
    DBG("======================\n");

    DBG("Got at 0x%" PRIX64 "\n", address);

    do {
        ptrace(PTRACE_GETREGS, process->pid, NULL, &regs);

        DBG("=== 0x%08llX (eip)\n", regs.rip);
        DBG("> ");
        c = getchar();
        while(getchar() != '\n');

        switch (c) {
            case 'p':
                print_regs(process);
```

```
                break;

            case 's':
                print_stack(process);
                break;

            case 'n':
                ptrace(PTRACE_SINGLESTEP, process->pid, NULL, NULL);
                waitpid(process->pid, NULL, 0);
                break;

            default:
                break;
        }
    } while (c != 'q');
}
#endif

/*
 * Waits for a syscall.
 *
 * Returns '0' if the child stopped on a syscall. Returns '1' if the child
 * exited.
 */
static int _wait_for_syscall(pid_t pid) {
    int status;

    while (1) {
        // Tell ptrace to wait for a syscall.
        ptrace(PTRACE_SYSCALL, pid, 0, 0);

        // Wait for child to be stopped.
        waitpid(pid, &status, 0);

        // Check status.
        if (WIFSTOPPED(status) && WSTOPSIG(status) & 0x80) {
            return 0;
        }

        if (WIFEXITED(status)) {
            return 1;
        }
    }
}

/*
 * Update the address of the code segment of the given process in its struct.
 *
 * On success, returns 0. Otherwise, prints an error and returns '-1'.
 */
static int _get_codesegment_address(process_info_t *process) {
    int rc;
    char file_path[255];
    char seg_addr_str[17];
    long seg_addr;
    FILE *file;

    // Get path of the `maps` file for the given pid.
    snprintf(file_path, sizeof(file_path), "/proc/%d/maps", process->pid);

    file = fopen(file_path, "r");
    if (file == NULL) {
        perror("[liboptimizer] fopen()");

        return -1;
    }

    // For now, we only retrieve the first 16 characters. Might need to check
    // permissions and all to be sure we parse the code segment.
    rc = fread(seg_addr_str, 1, sizeof(seg_addr_str) - 1, file);
    if (rc < 0) {
        perror("[liboptimizer] fread()");

```

```c
        return -1;
    }
    // Append end-of-string character.
    seg_addr_str[sizeof(seg_addr_str) - 1] = 0;

    fclose(file);

    // Convert string to long.
    seg_addr = strtol(seg_addr_str, NULL, 16);

    // Assign value.
    process->codesegment_address = seg_addr;

    return 0;
}

/*
 * Replaces the memory of the child process to hook the `func_addr` function
 * with the `free_addr` function.
 *
 * Returns '0' on success. Otherwise, prints an error and returns '-1'.
 */
static int _replace_mem(process_info_t *process) {
    unsigned char jump_32[] = MAKE_JUMP32(process->freesegment_address);
    unsigned char jump_64[] = MAKE_JUMP64(process->freesegment_address);

    int i, rc;
    uint64_t old_func_addr = process->function_offset + process->codesegment_address;

    // Put the new function into free allocated space.
    for (i = 0; i < process->optimized_function_size; i += 4) {
        rc = ptrace(PTRACE_POKEDATA,
                    process->pid,
                    (void *)process->freesegment_address + i,
                    ((unsigned int *)process->optimized_function)[i / 4]);
        if (rc < 0) {
            perror("[liboptimizer] PTRACE_POKEDATA");

            return -1;
        }
    }

    // 32-bit
    if (process->is64 == 0) {
        for (i = 0; i < sizeof(jump_32); i += 4) {
            rc = ptrace(PTRACE_POKEDATA,
                        process->pid,
                        (void *)old_func_addr + i, ((unsigned int *)jump_32)[i / 4]);
            if (rc < 0) {
                perror("[liboptimizer] PTRACE_POKEDATA");

                return -1;
            }
        }
    }
    // 64-bit
    else {
        for (i = 0; i < sizeof(jump_64); i += 4) {
            rc = ptrace(PTRACE_POKEDATA,
                        process->pid,
                        (void *)old_func_addr + i, ((unsigned int *)jump_64)[i / 4]);
            if (rc < 0) {
                perror("[liboptimizer] PTRACE_POKEDATA");

                return -1;
            }
        }
    }

    return 0;
}

/*
```

```
 * Injects a 'mmap2' syscall into a child process.
 *
 * On success, returns the address of the newly allocated memory segment.
 * If the child process unexpectedly stopped, returns '-1'.
 */
static int _inject_mmap(process_info_t *process) {
    struct user_regs_struct old_regs, new_regs;
    int rc;

    if (_wait_for_syscall(process->pid) != 0) {
        return -1;
    }

    rc = ptrace(PTRACE_GETREGS, process->pid, 0, &old_regs);
    if (rc < 0) {
        perror("[liboptimizer] PTRACE_GETREGS");

        return -1;
    }

    memcpy(&new_regs, &old_regs, sizeof(struct user_regs_struct));

    // 32-bit.
    if (process->is64 == 0) {
        // Registers for 32-bit binary on 64-bit machine.
        new_regs.rax = MMAP2_SYSCALL_X32;
        new_regs.rbx = 0;
        new_regs.rcx = process->optimized_function_size;
        new_regs.rdx = PROT_READ | PROT_WRITE | PROT_EXEC;
        new_regs.rsi = MAP_PRIVATE | MAP_ANONYMOUS;
        new_regs.rdi = -1;
        new_regs.rbp = 0;
        new_regs.orig_rax = MMAP2_SYSCALL_X32;
    }
    // 64-bit.
    else {
        // Registers for 64-bit binary on 64-bit machine.
        new_regs.rax = MMAP2_SYSCALL_X64;
        new_regs.rdi = 0;
        new_regs.rsi = process->optimized_function_size;
        new_regs.rdx = PROT_READ | PROT_WRITE | PROT_EXEC;
        new_regs.r10 = MAP_PRIVATE | MAP_ANONYMOUS;
        new_regs.r8  = -1;
        new_regs.r9  = 0;
        new_regs.orig_rax = MMAP2_SYSCALL_X64;
    }

    rc = ptrace(PTRACE_SETREGS, process->pid, NULL, &new_regs);
    if (rc < 0) {
        perror("[liboptimizer] PTRACE_SETREGS");

        return -1;
    }

    rc = ptrace(PTRACE_SINGLESTEP, process->pid, NULL, NULL);
    if (rc < 0) {
        perror("[liboptimizer] PTRACE_SINGLESTEP");

        return -1;
    }

    waitpid(process->pid, NULL, 0);

    rc = ptrace(PTRACE_GETREGS, process->pid, NULL, &new_regs);
    if (rc < 0) {
        perror("[liboptimizer] PTRACE_GETREGS");

        return -1;
    }

    rc = ptrace(PTRACE_SETREGS, process->pid, NULL, &old_regs);
    if (rc < 0) {
        perror("[liboptimizer] PTRACE_SETREGS");
```

```c

        return -1;
    }

    process->freesegment_address = new_regs.rax;

    return 0;
}

/*
 * Main function of parent process.
 *
 * On success, returns 0. Otherwise, prints an error message and returns the
 * corresponding value.
 */
static int _do_parent(process_info_t *process) {
    int rc;

    // Wait for child to be stopped.
    waitpid(process->pid, NULL, 0);

    // Set options for ptrace.
    ptrace(PTRACE_SETOPTIONS, process->pid, 0, PTRACE_O_TRACESYSGOOD);

    // Get address of the function to replace in the code segment.
    rc = _get_codesegment_address(process);
    if (rc < 0) {
        fprintf(stderr, "[liboptimizer] ERROR: Failed to get code segment address\n");

        return -1;
    }

    // Allocate new memory segment.
    rc = _inject_mmap(process);
    if (rc < 0) {
        fprintf(stderr, "[liboptimizer] ERROR: Failed to allocate memory segment\n");

        return -2;
    }

    return 0;
}

/*
 * Launches the child process.
 */
static int _do_child(process_info_t *process) {
    ptrace(PTRACE_TRACEME, 0, NULL, NULL);

    return execvp(process->path, process->argv);
}

int patcher_attach_process(process_info_t *process) {
    process->pid = fork();

    if (process->pid != 0) {
        DBG("child pid: %d\n", process->pid);
        return _do_parent(process);
    } else {
        return _do_child(process);
    }
}

int patcher_modify_process(process_info_t *process) {
    int rc;

    rc = _replace_mem(process);
    if (rc < 0) {
        fprintf(stderr, "[liboptimizer] ERROR: Failed to replace process memory\n");

        return -1;
    }

```

```c
    // NOTE: Uncomment if you want to debug the child process execution step
    // by step.
#ifdef LIBOPTIMIZER_DEBUG
    debug(process);
#endif

    return 0;
}

int patcher_continue_exec(process_info_t *process, bool wait_for_exit) {
    int rc;

    // Continue child execution.
    rc = ptrace(PTRACE_CONT, process->pid, NULL, NULL);
    if (rc < 0) {
        perror("[liboptimizer] PTRACE_CONT");

        return -1;
    }

    // Wait for child to quit if asked to.
    if (wait_for_exit) {
        waitpid(process->pid, NULL, 0);
    }

    return 0;
}
```

D.5 liboptimizer

liboptimizer.c

```c
/*
 * File: liboptimizer.c
 *
 * Created by: Lucas Elisei <lucas.elisei@heig-vd.ch>
 *
 * Main entry point of the library.
 */

#include <inttypes.h>
#include <limits.h>
#include <stdlib.h>

#include "debug.h"
#include "liboptimizer.h"
#include "live-patcher.h"
#include "elfparser.h"
#include "retdec.h"
#include "utils.h"

static int _recompile_function(process_info_t *process) {
    return retdec_recompile(process);
}

char *symbol_at_address(const char *path, uint64_t address) {
    return get_symbol_at_address(path, address);
}

process_info_t *init_process(int argc, char **argv, const char *function_name) {
    // Pointer to process information.
    process_info_t *process;
    // Store return codes.
    int rc;
    // Absolute path of the binary.
    char absolute_path[PATH_MAX];

    // Allocate memory for `process_info_t` struct and set the memory to zero.
    process = (process_info_t *)calloc(1, sizeof(process_info_t));
    if (process == NULL) {
        perror("[liboptimizer] calloc");

        return NULL;
    }

    // Retrieve absolute path of the binary.
    if (get_absolute_path(absolute_path, argv[0]) == NULL) {
        perror("[liboptimizer] get_absolute_path");

        goto _failed_abs_path;
    }

    DBG("absolute_path: %s\n", absolute_path);

    // Initialize the fields we already have information about.
    process->argc = argc;
    process->argv = argv;
    process->path = absolute_path;
    process->function_name = function_name;
    process->is64 = is64bit(process->path);

    // Get function offset from function_name.
    process->function_offset = get_symbol_address(process->path, process->function_name);
    if (process->function_offset == 0) {
        fprintf(stderr, "[liboptimizer] ERROR: Could not get function offset\n");

        goto _failed_func_offset;
    }

    // Call the RetDec's script to generate LLVM IR of the function passed as
```

```
    // argument.
    rc = _recompile_function(process);
    if (rc < 0) {
        fprintf(stderr, "[liboptimizer] ERROR: Error while recompiling function\n");

        goto _failed_recompilation;
    }

    // Attach the child process.
    rc = patcher_attach_process(process);
    if (rc < 0) {
        fprintf(stderr, "[liboptimizer] ERROR: Error while attaching process\n");

        goto _failed_attach;
    }

    return process;

// _not64bit:
_failed_func_offset:
_failed_abs_path:
_failed_recompilation:
_failed_attach:
    free(process);

    return NULL;
}

int modify_process(process_info_t *process) {
    int rc;

    rc = patcher_modify_process(process);
    if (rc < 0) {
        fprintf(stderr, "[liboptimizer] ERROR: Error while modifying process memory\n");

        return -1;
    }

    return 0;
}

int execute_process(process_info_t *process, bool wait_for_exit) {
    int rc;

    rc = patcher_continue_exec(process, wait_for_exit);
    if (rc < 0) {
        fprintf(stderr, "[liboptimizer] ERROR: Error while continuing process execution\n");

        return -1;
    }

    return 0;
}

#ifdef LIBOPTIMIZER_DEBUG
void print_process_info(process_info_t *process) {
    int i;
    FILE *fp;
    char filename[PATH_MAX];

    sprintf(filename, "%s%d.bin", process->function_name, process->is64 ? 64 : 32);

    fp = fopen(filename, "wb");

    DBG("=== Printing process_info_t at 0x%" PRIXPTR "\n", (uintptr_t)process);
    DBG("  path..................: %s\n", process->path);
    DBG("  argc..................: %d\n", process->argc);
    DBG("  argv..................:");
    for (i = 0; i < process->argc; ++i) {
        fprintf(stderr, " %s", process->argv[i]);
    }
    fprintf(stderr, "\n");
    DBG("  is64..................: %d\n", process->is64);
```

```c
    DBG("  pid....................: %d\n", process->pid);
    DBG("  function_name..........: %s\n", process->function_name);
    DBG("  function_offset........: 0x%" PRIX64 "\n", process->function_offset);
    DBG("  codesegment_address....: 0x%" PRIX64 "\n", process->codesegment_address);
    DBG("  freesegment_address....: 0x%" PRIX64 "\n", process->freesegment_address);
    DBG("  optimized_function_size: %zu\n", process->optimized_function_size);
    DBG("  optimized_function.....: 0x%" PRIXPTR "\n", (uintptr_t)process->optimized_function);
    if (process->optimized_function_size > 0) {
        DBG("    ");
        for (i = 0; i < process->optimized_function_size; ++i) {
            uint8_t byte = process->optimized_function[i];

            fputc(byte, fp);

            fprintf(stderr, " %02X", byte);
            if (i % 8 == 7) {
                fprintf(stderr, "\n");
                DBG("    ");
            }
        }
        fprintf(stderr, "\n");
    }
    DBG("===================\n\n");

    fclose(fp);
}
#endif // LIBOPTIMIZER_DEBUG
```

utils.c

```c
/*
 * File: utils.c
 *
 * Created by: Lucas Elisei <lucas.elisei@heig-vd.ch>
 */

#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include "utils.h"

char *get_absolute_path(char *dest, const char *path) {
    char buffer[PATH_MAX];

    if (*path == '/') {
        strcpy(buffer, path);
    }
    else {
        if (getcwd(buffer, PATH_MAX) == NULL) {
            perror("[liboptimizer] getcwd error");

            return NULL;
        }

        strcat(buffer, "/");
        strcat(buffer, path);
    }

    if (realpath(buffer, dest) == NULL) {
        perror("[liboptimizer] realpath error");

        return NULL;
    }

    return dest;
}
```

D.6 mmult

mmult.c

```c
/*
 * File: mmult.c
 *
 * Created by: Lucas Elisei <lucas.elisei@heig-vd.ch>
 */

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "mmult.h"

#define FILENAME    "/dev/urandom"

void matrix_init_data(matrix_t *m) {
    FILE* fp;
    size_t i;

    fp = fopen(FILENAME, "rb");
    if (fp == NULL) {
        fprintf(stderr, "Could not open %s\n", FILENAME);

        exit(EXIT_FAILURE);
    }

    for (i = 0; i < m->row * m->col; ++i) {
        uint8_t data;

        if (fread(&data, sizeof(uint8_t), 1, fp) != 1) {
            fprintf(stderr, "fread error\n");
            fclose(fp);

            exit(EXIT_FAILURE);
        }

        m->data[i] = data % 8;
    }

    fclose(fp);
}

matrix_t *matrix_init(size_t row, size_t col) {
    matrix_t *m;

    m = (matrix_t *)malloc(sizeof(matrix_t));
    if (m == NULL) {
        fprintf(stderr, "Could not allocate memory\n");

        return NULL;
    }

    m->data = (uint32_t *)malloc(row * col * sizeof(uint32_t));
    if (m->data == NULL) {
        fprintf(stderr, "Could not allocate memory\n");
        free(m);

        return NULL;
    }

    m->col = col;
    m->row = row;

    return m;
}

int matrix_mult(matrix_t *a, matrix_t *b, matrix_t *res) {
    size_t i, j, k;
```

```

    for (i = 0; i < res->row * res->col; ++i) {
        res->data[i] = 0;
    }

    for (i = 0; i < a->row; ++i) {
        for (j = 0; j < a->col; ++j) {
            for (k = 0; k < b->col; ++k) {
                res->data[i*res->col + j] += a->data[i*res->col + k] * b->data[k*res->col + j];
            }
        }
    }

    return 0;
}

void matrix_save(matrix_t *m, const char *filename) {
    size_t i, j;
    FILE *fp;

    fp = fopen(filename, "w");
    if (fp == NULL) {
        perror("fopen()");

        return;
    }

    for (i = 0; i < m->row; ++i) {
        for (j = 0; j < m->col; ++j) {
            fprintf(fp, "%3d ", m->data[i*m->col + j]);
        }
        fprintf(fp, "\n");
    }

    fclose(fp);
}
```

main.c

```
/*
 * File: main.c
 *
 * Created by: Lucas Elisei <lucas.elisei@heig-vd.ch>
 */

#define _POSIX_C_SOURCE 199309L

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

#include "mmult.h"

int main(int argc, char **argv) {
    matrix_t *m1, *m2, *res;
    size_t col, row;
    struct timespec start, stop;

    if (argc != 3) {
        fprintf(stdout, "usage: %s <row> <col>\n", argv[0]);

        return EXIT_FAILURE;
    }

    row = (size_t)atoi(argv[1]);
    col = (size_t)atoi(argv[2]);

    m1 = matrix_init(row, col);
    m2 = matrix_init(col, row);
```

```
    res = matrix_init(row, row);

    matrix_init_data(m1);
    matrix_init_data(m2);

    matrix_save(m1, "m1.mat");
    matrix_save(m2, "m2.mat");

    if (clock_gettime(CLOCK_REALTIME, &start) == -1) {
        perror("clock_gettime()");
        exit(EXIT_FAILURE);
    }
    matrix_mult(m1, m2, res);
    if (clock_gettime(CLOCK_REALTIME, &stop) == -1) {
        perror("clock_gettime()");
        exit(EXIT_FAILURE);
    }

    printf("Elapsed time for matrix multiplication (%zux%zu): %lus %luns\n",
            row, col, (stop.tv_sec - start.tv_sec),
            (stop.tv_nsec - start.tv_nsec));

    matrix_save(res, "res.mat");

    return EXIT_SUCCESS;
}
```

Machine code of non-optimized 32-bit matrix_mult function

```
00001424 <matrix_mult>:
    1424:    55                      push   %ebp
    1425:    89 e5                   mov    %esp,%ebp
    1427:    56                      push   %esi
    1428:    53                      push   %ebx
    1429:    83 ec 10                sub    $0x10,%esp
    142c:    e8 eb 01 00 00          call   161c <__x86.get_pc_thunk.ax>
    1431:    05 cf 2b 00 00          add    $0x2bcf,%eax
    1436:    c7 45 f4 00 00 00 00    movl   $0x0,-0xc(%ebp)
    143d:    eb 18                   jmp    1457 <matrix_mult+0x33>
    143f:    8b 45 10                mov    0x10(%ebp),%eax
    1442:    8b 40 08                mov    0x8(%eax),%eax
    1445:    8b 55 f4                mov    -0xc(%ebp),%edx
    1448:    c1 e2 02                shl    $0x2,%edx
    144b:    01 d0                   add    %edx,%eax
    144d:    c7 00 00 00 00 00       movl   $0x0,(%eax)
    1453:    83 45 f4 01             addl   $0x1,-0xc(%ebp)
    1457:    8b 45 10                mov    0x10(%ebp),%eax
    145a:    8b 10                   mov    (%eax),%edx
    145c:    8b 45 10                mov    0x10(%ebp),%eax
    145f:    8b 40 04                mov    0x4(%eax),%eax
    1462:    0f af c2                imul   %edx,%eax
    1465:    39 45 f4                cmp    %eax,-0xc(%ebp)
    1468:    72 d5                   jb     143f <matrix_mult+0x1b>
    146a:    c7 45 f4 00 00 00 00    movl   $0x0,-0xc(%ebp)
    1471:    e9 bf 00 00 00          jmp    1535 <matrix_mult+0x111>
    1476:    c7 45 f0 00 00 00 00    movl   $0x0,-0x10(%ebp)
    147d:    e9 a0 00 00 00          jmp    1522 <matrix_mult+0xfe>
    1482:    c7 45 ec 00 00 00 00    movl   $0x0,-0x14(%ebp)
    1489:    e9 81 00 00 00          jmp    150f <matrix_mult+0xeb>
    148e:    8b 45 10                mov    0x10(%ebp),%eax
    1491:    8b 50 08                mov    0x8(%eax),%edx
    1494:    8b 45 10                mov    0x10(%ebp),%eax
    1497:    8b 40 04                mov    0x4(%eax),%eax
    149a:    0f af 45 f4             imul   -0xc(%ebp),%eax
    149e:    89 c1                   mov    %eax,%ecx
    14a0:    8b 45 f0                mov    -0x10(%ebp),%eax
    14a3:    01 c8                   add    %ecx,%eax
    14a5:    c1 e0 02                shl    $0x2,%eax
    14a8:    01 d0                   add    %edx,%eax
    14aa:    8b 08                   mov    (%eax),%ecx
```

```
    14ac:    8b 45 08                mov    0x8(%ebp),%eax
    14af:    8b 50 08                mov    0x8(%eax),%edx
    14b2:    8b 45 10                mov    0x10(%ebp),%eax
    14b5:    8b 40 04                mov    0x4(%eax),%eax
    14b8:    0f af 45 f4             imul   -0xc(%ebp),%eax
    14bc:    89 c3                   mov    %eax,%ebx
    14be:    8b 45 ec                mov    -0x14(%ebp),%eax
    14c1:    01 d8                   add    %ebx,%eax
    14c3:    c1 e0 02                shl    $0x2,%eax
    14c6:    01 d0                   add    %edx,%eax
    14c8:    8b 10                   mov    (%eax),%edx
    14ca:    8b 45 0c                mov    0xc(%ebp),%eax
    14cd:    8b 58 08                mov    0x8(%eax),%ebx
    14d0:    8b 45 10                mov    0x10(%ebp),%eax
    14d3:    8b 40 04                mov    0x4(%eax),%eax
    14d6:    0f af 45 ec             imul   -0x14(%ebp),%eax
    14da:    89 c6                   mov    %eax,%esi
    14dc:    8b 45 f0                mov    -0x10(%ebp),%eax
    14df:    01 f0                   add    %esi,%eax
    14e1:    c1 e0 02                shl    $0x2,%eax
    14e4:    01 d8                   add    %ebx,%eax
    14e6:    8b 00                   mov    (%eax),%eax
    14e8:    0f af d0                imul   %eax,%edx
    14eb:    8b 45 10                mov    0x10(%ebp),%eax
    14ee:    8b 58 08                mov    0x8(%eax),%ebx
    14f1:    8b 45 10                mov    0x10(%ebp),%eax
    14f4:    8b 40 04                mov    0x4(%eax),%eax
    14f7:    0f af 45 f4             imul   -0xc(%ebp),%eax
    14fb:    89 c6                   mov    %eax,%esi
    14fd:    8b 45 f0                mov    -0x10(%ebp),%eax
    1500:    01 f0                   add    %esi,%eax
    1502:    c1 e0 02                shl    $0x2,%eax
    1505:    01 d8                   add    %ebx,%eax
    1507:    01 ca                   add    %ecx,%edx
    1509:    89 10                   mov    %edx,(%eax)
    150b:    83 45 ec 01             addl   $0x1,-0x14(%ebp)
    150f:    8b 45 0c                mov    0xc(%ebp),%eax
    1512:    8b 40 04                mov    0x4(%eax),%eax
    1515:    39 45 ec                cmp    %eax,-0x14(%ebp)
    1518:    0f 82 70 ff ff ff       jb     148e <matrix_mult+0x6a>
    151e:    83 45 f0 01             addl   $0x1,-0x10(%ebp)
    1522:    8b 45 08                mov    0x8(%ebp),%eax
    1525:    8b 40 04                mov    0x4(%eax),%eax
    1528:    39 45 f0                cmp    %eax,-0x10(%ebp)
    152b:    0f 82 51 ff ff ff       jb     1482 <matrix_mult+0x5e>
    1531:    83 45 f4 01             addl   $0x1,-0xc(%ebp)
    1535:    8b 45 08                mov    0x8(%ebp),%eax
    1538:    8b 00                   mov    (%eax),%eax
    153a:    39 45 f4                cmp    %eax,-0xc(%ebp)
    153d:    0f 82 33 ff ff ff       jb     1476 <matrix_mult+0x52>
    1543:    b8 00 00 00 00          mov    $0x0,%eax
    1548:    83 c4 10                add    $0x10,%esp
    154b:    5b                      pop    %ebx
    154c:    5e                      pop    %esi
    154d:    5d                      pop    %ebp
    154e:    c3                      ret
```

Machine code of optimized 32-bit `matrix_mult` function

```
00000000 <.data>:
    0:    55                      push   %ebp
    1:    53                      push   %ebx
    2:    57                      push   %edi
    3:    56                      push   %esi
    4:    83 ec 18                sub    $0x18,%esp
    7:    8b 4c 24 34             mov    0x34(%esp),%ecx
    b:    8b 44 24 04             mov    0x4(%esp),%eax
    f:    89 44 24 0c             mov    %eax,0xc(%esp)
   13:    8b 04 24                mov    (%esp),%eax
   16:    89 44 24 14             mov    %eax,0x14(%esp)
```

```
    1a:   8b 41 04                 mov    0x4(%ecx),%eax
    1d:   0f af 01                 imul   (%ecx),%eax
    20:   85 c0                    test   %eax,%eax
    22:   74 24                    je     0x48
    24:   31 c0                    xor    %eax,%eax
    26:   31 f6                    xor    %esi,%esi
    28:   90                       nop
    29:   90                       nop
    2a:   90                       nop
    2b:   90                       nop
    2c:   90                       nop
    2d:   90                       nop
    2e:   90                       nop
    2f:   90                       nop
    30:   8b 79 08                 mov    0x8(%ecx),%edi
    33:   c7 04 38 00 00 00 00     movl   $0x0,(%eax,%edi,1)
    3a:   46                       inc    %esi
    3b:   8b 79 04                 mov    0x4(%ecx),%edi
    3e:   0f af 39                 imul   (%ecx),%edi
    41:   83 c0 04                 add    $0x4,%eax
    44:   39 fe                    cmp    %edi,%esi
    46:   72 e8                    jb     0x30
    48:   8b 4c 24 2c              mov    0x2c(%esp),%ecx
    4c:   8b 19                    mov    (%ecx),%ebx
    4e:   85 db                    test   %ebx,%ebx
    50:   0f 84 e5 00 00 00        je     0x13b
    56:   8b 54 24 30              mov    0x30(%esp),%edx
    5a:   8b 69 04                 mov    0x4(%ecx),%ebp
    5d:   31 f6                    xor    %esi,%esi
    5f:   89 e8                    mov    %ebp,%eax
    61:   90                       nop
    62:   90                       nop
    63:   90                       nop
    64:   90                       nop
    65:   90                       nop
    66:   90                       nop
    67:   90                       nop
    68:   90                       nop
    69:   90                       nop
    6a:   90                       nop
    6b:   90                       nop
    6c:   90                       nop
    6d:   90                       nop
    6e:   90                       nop
    6f:   90                       nop
    70:   85 c0                    test   %eax,%eax
    72:   0f 84 b8 00 00 00        je     0x130
    78:   8b 42 04                 mov    0x4(%edx),%eax
    7b:   31 ff                    xor    %edi,%edi
    7d:   89 74 24 10              mov    %esi,0x10(%esp)
    81:   90                       nop
    82:   90                       nop
    83:   90                       nop
    84:   90                       nop
    85:   90                       nop
    86:   90                       nop
    87:   90                       nop
    88:   90                       nop
    89:   90                       nop
    8a:   90                       nop
    8b:   90                       nop
    8c:   90                       nop
    8d:   90                       nop
    8e:   90                       nop
    8f:   90                       nop
    90:   85 c0                    test   %eax,%eax
    92:   b8 00 00 00 00           mov    $0x0,%eax
    97:   74 7f                    je     0x118
    99:   31 ed                    xor    %ebp,%ebp
    9b:   89 7c 24 08              mov    %edi,0x8(%esp)
    9f:   90                       nop
    a0:   8b 44 24 34              mov    0x34(%esp),%eax
    a4:   89 c2                    mov    %eax,%edx
```

```
  a6:   8b 42 08                mov    0x8(%edx),%eax
  a9:   8b 52 04                mov    0x4(%edx),%edx
  ac:   0f af d6                imul   %esi,%edx
  af:   8b 7c 24 08             mov    0x8(%esp),%edi
  b3:   8d 34 3a                lea    (%edx,%edi,1),%esi
  b6:   8b 04 b0                mov    (%eax,%esi,4),%eax
  b9:   8b 71 08                mov    0x8(%ecx),%esi
  bc:   89 14 24                mov    %edx,(%esp)
  bf:   01 ea                   add    %ebp,%edx
  c1:   8b 14 96                mov    (%esi,%edx,4),%edx
  c4:   8b 5c 24 30             mov    0x30(%esp),%ebx
  c8:   8b 73 08                mov    0x8(%ebx),%esi
  cb:   89 34 24                mov    %esi,(%esp)
  ce:   8b 4c 24 34             mov    0x34(%esp),%ecx
  d2:   8b 49 04                mov    0x4(%ecx),%ecx
  d5:   0f af cd                imul   %ebp,%ecx
  d8:   89 4c 24 04             mov    %ecx,0x4(%esp)
  dc:   01 f9                   add    %edi,%ecx
  de:   0f af 14 8e             imul   (%esi,%ecx,4),%edx
  e2:   8b 74 24 10             mov    0x10(%esp),%esi
  e6:   01 c2                   add    %eax,%edx
  e8:   8b 4c 24 34             mov    0x34(%esp),%ecx
  ec:   8b 41 08                mov    0x8(%ecx),%eax
  ef:   89 04 24                mov    %eax,(%esp)
  f2:   8b 49 04                mov    0x4(%ecx),%ecx
  f5:   0f af ce                imul   %esi,%ecx
  f8:   89 4c 24 04             mov    %ecx,0x4(%esp)
  fc:   01 f9                   add    %edi,%ecx
  fe:   89 14 88                mov    %edx,(%eax,%ecx,4)
 101:   8b 4c 24 2c             mov    0x2c(%esp),%ecx
 105:   45                      inc    %ebp
 106:   8b 43 04                mov    0x4(%ebx),%eax
 109:   39 c5                   cmp    %eax,%ebp
 10b:   72 93                   jb     0xa0
 10d:   8b 69 04                mov    0x4(%ecx),%ebp
 110:   8b 54 24 30             mov    0x30(%esp),%edx
 114:   8b 7c 24 08             mov    0x8(%esp),%edi
 118:   47                      inc    %edi
 119:   39 ef                   cmp    %ebp,%edi
 11b:   0f 82 6f ff ff ff       jb     0x90
 121:   8b 19                   mov    (%ecx),%ebx
 123:   89 e8                   mov    %ebp,%eax
 125:   46                      inc    %esi
 126:   39 de                   cmp    %ebx,%esi
 128:   0f 82 42 ff ff ff       jb     0x70
 12e:   eb 0b                   jmp    0x13b
 130:   31 c0                   xor    %eax,%eax
 132:   46                      inc    %esi
 133:   39 de                   cmp    %ebx,%esi
 135:   0f 82 35 ff ff ff       jb     0x70
 13b:   8b 44 24 14             mov    0x14(%esp),%eax
 13f:   89 04 24                mov    %eax,(%esp)
 142:   8b 44 24 0c             mov    0xc(%esp),%eax
 146:   89 44 24 04             mov    %eax,0x4(%esp)
 14a:   31 c0                   xor    %eax,%eax
 14c:   83 c4 18                add    $0x18,%esp
 14f:   5e                      pop    %esi
 150:   5f                      pop    %edi
 151:   5b                      pop    %ebx
 152:   5d                      pop    %ebp
 153:   c3                      ret
```