Master Cryptis
Faculté des Sciences et Techniques
123 avenue Albert Thomas
87060 Limoges Cedex, France

Thomson R&D
1 avenue de Belle Fontaine
35576 Cesson-Sévigné, France

Internship Report

Analysis and Binary Transformation

Public version

Author:
Guillaume Bouffard

Supervisor:
Antoine Monsifrot

Academic tutor:
Jean-Louis Lanet

Universitary years: 2009–2010

Abstract

This document describes the work done during the internship for my master degree Information Technologies Security and Cryptography at the University of Limoges, France, carried out with Technicolor at the Technicolor Security and Content Protection Laboratories at Rennes, France.

This internship aims to protect the sensitive piece of code of a binary executable without application source code knowledge. A sensitive piece of code of a binary application is a set of the most uses instructions. In order to protect this binary part, I extracted this sensitive piece of code of the binary application and I replaced with that provides a way to communicate to a dongle. On this dongle, the extracted protected instructions are putted and executed. Thus, to correctly execute a protected application, the user needs a dongle which contains the missed part of the executed application.

After a short presentation of the context and the company, I will present each step to my internship. In a first time, I will describe how to find the sensitive binary part of an executable program. When this part is found, I will try to translate this piece of code in another language to protect it. Next, I will modify the binary application, without the source code, to change the protected piece of code by a set of instructions designed to communicate to the dongle containing the extracted piece of code. That will take us to a proof of concept. To conclude, I will sum up the completed objectives, the possible improvements and the difficulties I have encountered.

Acknowledgments

First, I would like to thank Antoine Monsifrot, my supervisor during these six months, for giving me the chance to do this internship, his help, his patience, his kindness and his confidence during each moment of this training period.

Second, I want to thank the team of Technicolor Security and Content Protect Laboratories for their welcoming, their availability, their help and their good humour where I have done my internship on a high note.

Moreover, I would like to thank all the teaching personnel of the master Information Technologies Security and Cryptography for their advices, lessons and exciting projects during these two years.

I cannot forget Marion Floury, Javier Franco-Contreras and Julien Devigne, other interns in the same laboratory as me, for their help, advices, patience and the good times done in the *interns' office*.

Finally, I would like to thank Michaël Bouygues, Christian Chung and Ludovic Courgnaud for their help during my internship.

Contents

Preface

At the end of my Master of Sciences in Information Technology Security, I need to make an internship to validate my degree. This training period must be made in a company or a laboratory during four to six months. This work experience allows me to discover, for the first time, the world of work.

I rather wanted to make my internship in a laboratory. After my university formation, I wanted work in a research and development laboratory to computer science because I wished to contribute to the computer science theory. Doing my internship in the Technicolor Security and Content Protection Laboratories was a way to discover, with a research subject, if this career path was suitable for me.

Chapter 1

The Internship

1.1 Technicolor

With more than 95 years of experience in entertainment innovation, Technicolor serves as international base of entertainment, software, and gaming customers. The company is a leading provider in production, postproduction, and distribution services to content creators, network service providers and broadcasters. Technicolor is one of the world's largest film processors; the largest independent manufacturer and DvDs distributor (including Blu-ray Disc); and a leading global supplier of set-top boxes and gateways. The company also operates an Intellectual Property and Licensing business unit. Technicolor Rennes makes its substantial contribution to the Thomson portfolio.

Technicolor Rennes (Thomson R&D France) is the largest technology Center of Technicolor. The Rennes center designs and develops innovative solutions for the Communication, Media and Entertainment Industries. Technicolor Rennes studies new technologies in the fields of compression, security, protection, transmission, production and management of high definition contents. The Rennes Center prepares the future generation of digital cable, satellite and IP equipment and associated services for broadband network operators.

Established in Rennes for nearly 30 years and located within the Rennes Atalante Science Park, Technicolor Rennes has built partnerships with top academic institutions, public institutes, and industrial partners throughout Europe.The centre is involved in all French and European research programs and is a cofounder of the Media and Networks global Competitiveness Cluster, in Brittany and Pays de la Loire.

1.2 Technicolor Security and Content Protect Laboratories

Technicolor Security Laboratories is formed with 30 security experts and one PhD student split between Rennes (France) and Hanover (Germany). The Laboratories work on:

- Cryptography
- Signal processing for security
- DRM - content protection
- Network security
- Tamper resistance

It is within this surprising and interesting environment that I did my internship.

1.3 My Internship

1.3.1 Context

For software and game manufacturers in particular, illegal software duplication and intellectual property theft are two important problems without any acceptable solution. So far according to the Japan's Computer Entertainment Suppliers Association (CESA) video game piracy for portable consoles like the Nintendo DS and PSP around the world costs the gaming industry at least $44.15 billion between 2004 and 2009 [1].

Unlike hardware protection, software protections are user friendly (as there is no need to take care about a hardware token) and provide renewable security. However, software protection could be analyzed and broken without expensive equipment. It just requires skills and time. Therefore, software protections are a lot more subject to attacks.

In this context, the use of hardware component seems to be interesting. Some solutions like dongles already exist and are used to prevent illegal copies. However, all the solutions we have found just verify the presence of the dongle. The verification of this presence only relies on software protection and by consequence is no more robust than software protection.

1.3.2 Internship subject

This internship aims to find a robust solution to the problem previously stated. This subject can divided in three parts (figure 1.1):

- First step, we need to find the interesting piece of code which can be executed in a dongle. This piece of code needs a comprise between its instructions size and how many times this piece of code is executed.
- Second step, when the piece of code to be protected is found, we need to proceed to a binary transformation (computer assembler to a language understood by a dongle).
- Finally, in the third step, we need to modify, statically, the application to be protected, in order to replace the protected instructions by communication with the dongle. This step also provides a binary for the dongle which contains the extracted pieces of code.

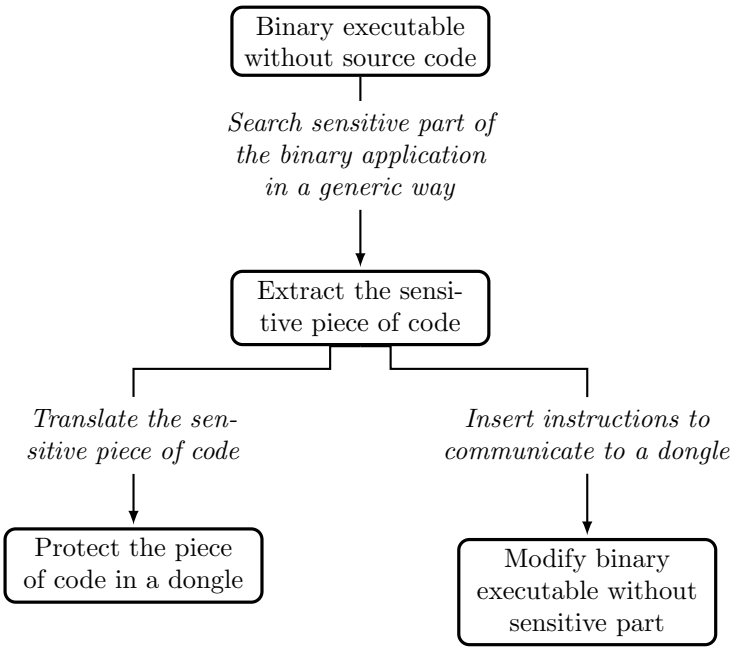


Figure 1.1: Internship objectives

1.3.3 Motivations

Every step during my formation, I discovered a new stage in the process of creating a clean and optimized application, through some development tricks, and also pragmatism. Over the last 2 years, I had the occasion to work with smart cards, Java cards, binary files [2] and hacked into some applications in order to improve my understanding of inside story of computer sciences.

Nonetheless, until this internship, I never really worked at the compilation stage of an application nor an executable structure. Thus, this training period allowed me to discover a brand new unknown world, the beautiful world of compilation.

1.3.4 Organisation

Like I explained in the subsection 1.3.2, my internship was split in three parts. Each part did not take me a same amount of time to realize. Thus, the study of the profile applications and the translate steps took me about a fourth of internship time. The modification of the binary application took about four mouths. The last weeks were reserved to made a proof of concept.

Chapter 2

Profiling applications to find their sensitive binary part

> Dark, profound it was, and cloudy, so that though I fixed my sight on the bottom I did not discern anything there.
>
> Canto IV: First Circle
> Dante's Inferno

In order to dynamically find the application sensitive binary part with an automatic mean, I realized, in a first time, a state of the art, where, I explain, for each profiling applications, its characteristics. In a second time, I performed a few tests in order to find the better profiling application.

2.1 State of the Art

2.1.1 Objective

The sensitive part of a binary application is a set of the most uses Basic Blocks (BB). To found this piece of code, I need to dynamically check for each instruction how many times is it executed. To perform this step, I should to use a profiler. A profiler reports each event do by a specific application. Thus, a profiling application measures each instruction and resource used during this execution.

In my case, I need to count for each instruction how many times did they executed.

To find the best profiling application, this profiler can detect, with a finest granularity, each executed assembler instructions. Moreover, this profiler does not need any special compilation parameters.

Now, I describe two profiling applications (OProfile [3] and Valgrind [4] with two different tools, Callgrind and BBV).

2.1.2 OProfile

OProfile is a profiler using a client/server architecture. A dæmon running in the background, in kernel land, to analyse each action doing by the profiled application and its external components. On other hand, a "client" dialog with this dæmon to get the report of analysed application. According to OProfile website [3]:

- Profiling application do not need any special recompilation,
- OProfile dynamically creates Control Flow Graph (CFG),
- The OProfile overhead is 1-8% in comparison to normal application overhead,
- The profile data can be produced on the function-level or instruction-level detail.

2.1.3 Valgrind

Valgrind [4] is a famous dynamic analysis tool. Now, it provides many features like a memory error detector, two thread error detectors, a cache and branch-prediction profiler, a call-graph generating cache profiler and a heap profiler. Also, there are two experimental tools: a heap/stack/global array overburn detector and the SimPoint [5] basic block generator.

To dynamically find sensitive binary part, I will describe Callgrind, a call-graph generating cache profiler, and BBV, the SimPoint basic blocks generator.

Valgrind Callgrind tool

Callgrind [4] provides a simulation of the I1, D1 and L2 caches memory references and instructions executed for each line of source code, with per-function, per-module and whole-program summaries, and more information about Control Flow Graph. In opposition to OProfile, Callgrind execute programs about 20–100x slower than normal.

Valgrind BBV tool

BBV is a Valgrind experimental tool [6]. It analyses an application during its execution and provides a list of basic blocks. Each BB is linked to its function and its address in two files, a bb and a pc file. On one hand, a bb file contains every basic block and, on the other hand, a pc file holds, for each BB, the begin addresses, instructions number and the function name where it is contained. These files use the SimPoint [5] file structure. This file organization is simple, documented [6] and it is easy to automatically parse these files.

2.2 Tests, comparison and choice

In order to test each application, I used a simple modulus operandi. First, I implemented a simple, and unoptimized, square 1024 matrix product in C language (available on appendix A). Second, I profiled my matrix product with each profiler and the best profiling application can detect, without source code and not special compilation options, the most used basic block. Now, looking what really makes each application.

2.2.1 OProfile

For test OProfile, I set it up with the commands describe in the listing 2.1. In a first time, I ran a dæmon in the background. Next, I execute the program to analyse. Finally, I recover the profiling report in order to found the most used instructions.

```
# opcontrol --no-vmlinux \
            --separate=all       # Set-up OProfile
# opcontrol --start              # Daemon started

/* Run application to analyse */

# opcontrol --stop               # Daemon stopped
# opcontrol --dump               # Check every things are doing
                                 # for application was ran

$ opreport   -l <ANALYSED_APPLICATION_NAME> \
             --merge=all         # Profiling report

$ opannotate --assembly <ANALYSED_APPLICATION_NAME> \
             --merge=all         # Get a profiling in assembler
                                 # instruction level
# opcontrol --reset              # Delete all profiling data
# opcontrol --shutdown           # Daemon killed
```

Listing 2.1: OProfile commands

When the matrix product was profiled, I was able to read two types of OProfile report. The first provides a report with the cost, in percentage and its numbers of call, for each function (appendix B.1.1). The second type (appendix B.1.2) describes for each instruction, its cost on call number and percentage. Moreover these files are human readable.

2.2.2 Valgrind

Valgrind Callgrind tool

In order to test Callgrind tool of Valgrind, I used the explained parameters in the listing 2.2.

At the end of the profiling step, Callgrind provides a report file. The file format is described in the Valgrind documentation [7]. The used grammar is more complex than OProfile report, I need to analyse statically the analysed binary executable in order to determine correctly the sensitive piece of code.

```
valgrind --tool=callgrind      \   # Tools to create application
                                   # call graph
         --dump-line=no        \
         --dump-instr=yes      \   # Analyse each application
                                   # instructions
         --simulate-cache=yes  \   # Simulate a cache
         --collect-jumps=yes   \   # Save every jumps applications
         ApplicationToAnalyse      # Binary to analyse
```

Listing 2.2: Valgrind configuration for Callgrind

Valgrind BBV tool

Finally, in order to test this tool, I used the parameters described in the listing 2.3.

```
valgrind --tool=exp-bbv        \   # Tools to create application
                                   # basic block vector

         --bb-out-file=bb.out  \   # Output basic block vector file

         --pc-out-file=pc.out  \   # This file holds program counter
                                   # addresses and function name
                                     info
                                   # for the various basic blocks

         --interval-size=5     \   # Size of the interval to use
                                   # (in instruction number)
         ApplicationToAnalyse      # Binary to analyse
```

Listing 2.3: Valgrind configuration for BBV

Profile with this tool needs a long time (few days to profile my matrix product) and many storage space (report files need at least 2 gigabytes).

2.2.3 Comparison and Choice

To compare each profiler application, I measured the amount of time necessary to run my matrix product (available on appendix A) with the same precision. The table 2.1 shows the results.

Profiler	Amount time to profile the matrix product
Without profiler	16 seconds
OProfile	18 seconds
Valgrind - Callgrind	26 minutes and 02 seconds
Valgrind - BBV	3 days 5 hours 15 minutes and 45 seconds

Table 2.1: Comparative of each profiler application.

With this table, OProfile is the quickest profiler. Moreover, with its dæmon running in the background, the profiling step is invisible for the analysed application.

The OProfile report (appendix B.1.1 and B.1.2) has a same granularity than Valgrind BBV tool. It provides a same `objdump` disassembler output extend

with profiling information (for each instruction, its number of execution, etc.).

Each profiling tools monitored each external libraries uses by the profiled application. In the internship goal, I need not this information.

Thus, I chose OProfile because of:

- Quicker profiling
- Invisible for the analysed application
- Report with a one instruction precision
- Finest granularity

Now, I have a way to determine the sensitive piece of code of a binary application. In order to protect this set of instructions, I need to translate this piece of code to execute this content on a dongle.

Chapter 3

Translating a computer piece of code to a dongle

> Nothing lost, nothing is created,
> everything is transformed
>
> Antoine Lavoisier

In the chapter 2, I found a mean to decide the sensitive part of a binary executable. If I can extract a piece of code of a binary application, I need to find a way to protect this part in an external component. Usually, this kind of component has less resources than a computer (less memory, capacities, etc.). Thus, this translation step should take into account the dongle technical capacities.

In the interest of keeping a simple test system, I have used a smart card like a dongle. In the world of smart card, the Java Card has been the most used. This success comes from an easily development environment, very similar to Java with a hardware abstraction and security. Thus, a Java Card Applet can be run on a different Java Card. Moreover, a Cap File, a file containing an Applet for Java Card, is normalized in a specification. You can modify, and add a translated piece of code, with a generic means, before sending this file to the smart card. There is a CapFileManipulator [2] can add piece of code, and guarantee the file coherence.

3.1 State of art of assembler to Java translator

Automatically translating a language (here assembler) to another (Java) can be provided by some frameworks. This conversion is a way to execute the protected binary part in the smart card. Thus, the application cannot work without the smart card containing the missing piece of code.

I found some language translators, but only one can translate assembler instructions to another language. Most of these language translators convert C

language to another language.

3.1.1 UQBT

UQBT [8, 9], for a Resourceable and Retargetable Binary Translator, can translate a specific architecture binary executable to another (figure 3.1).

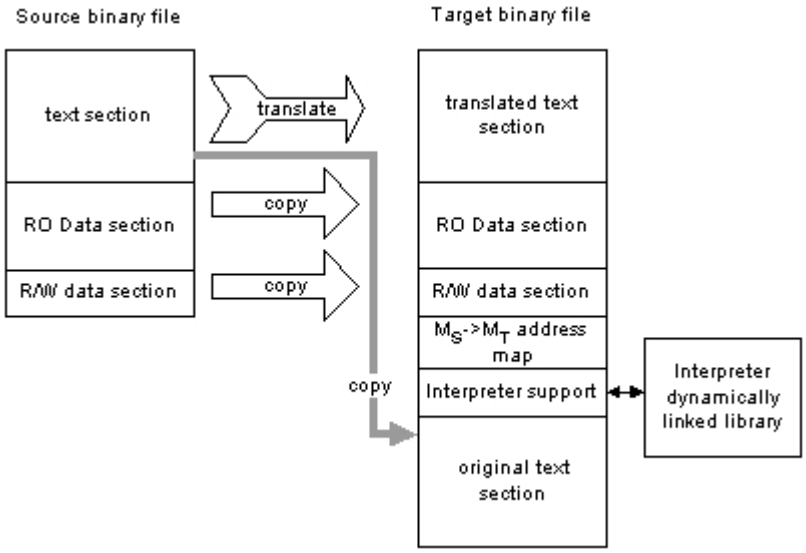


Figure 3.1: UQBT generated target binary.

In order to improve UQBT, Cristina Cifuentes and her team [8] added a Java Backend for GNU Compiler Collection (GCC). Based to `egcs`, an ancestor of GCC, this project converts a piece of binary application to a Java `.class` file using `Jasmin`. `Jasmin` is assembler for the Java Virtual Machine (JVM). It takes as input a text description of Java classes, written in a simple assembler JVM instructions. It converts them to a binary class files, loading by the JVM.

Unfortunately, I could not use UQBT and the Java Backend. The main problem came from an incompatibility between `egcs` and the UQBT patch. After an unsuccessfull researching step to resolve this trouble, I had to to suspend this part in order to concentrate on the modification of the binary executable.

Chapter 4

Modification of the binary application

> Any sufficiently advanced technology is indistinguishable from magic.
>
> Arthur C. Clarke

With the chapter 2, I can find an application sensitive part with a profiling step. Now, I want a clean way to take the most important instructions of a binary application in order to change it by a piece of code which provides a smart card communication.

I will quickly explain how Executable and Linking Format (ELF) is used, by GNU/Linux, to structure binary applications. Next, I will be doing a state of art of binary modifier frameworks which will lead to the choice of the best framework. Finally, I will explain how to use it with few simple examples.

4.1 ELF Format

Since 1995 [10], in the GNU/Linux kernel, ELF specification [11] is used to describe object files, shared libraries and binary executables. If you look at figure 4.1, you can see there are many connected sections (or segments), but ELF files have undetermined and unordering sections. The main sections are [10]:

- **ELF header** containing informations which explains how to read this file,
- **Program Header Table**: this section, gives informations for Operating System (OS) application loader. A binary executable must have it.
- `.text` **section** containing the program instructions,
- `.rodata` **section** is the place where constant data (strings value, etc.) are saved,
- `.data` **section** hold all initialized data here by default.

- **Section Header Table (SHT)** locates all sections in the ELF file.

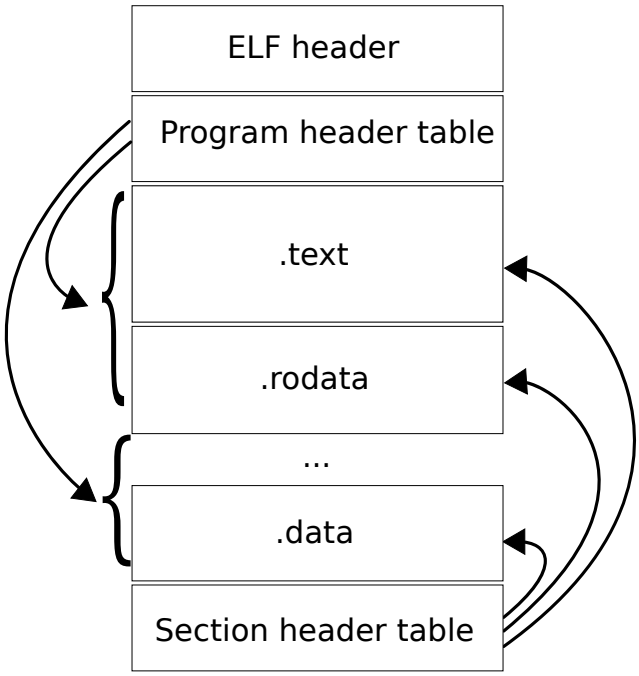


Figure 4.1: An ELF executable example [12].

You can see, on the figure 4.1, there are dependencies with each section. Thus, if you want modify one section, you should correctly link sections (section sizes, offsets, addresses, etc.). In order to easily modify an ELF file, I need to use a framework.

4.2 State of the art of binaries modifier frameworks

4.2.1 MetAsm

MetAsm [13] is a binary manipulator suite written in Ruby language.

Advantages

- MetAsm supports a lot of architectures (x86 (16/32/64bits), MIPS, PPC),
- MetAsm supports many executable format (Raw, MZ and PE/COFF (32 and 64 bits), ELF (32 and 64 bits), Mach-O (incomplete) and Universal-Binary, a few other (a.out, xcoff, nds)),
- MetAms has advance features like live process manipulation, GCC/Microsoft Visual Studio-compatible preprocessor, automatic backtracking in the dis-

assembler, C headers shrinking, linux/windows/remote debugging API interface, a C compiler/decompiler, a gdb-server compatible debugger, and various advanced features.

Drawback

- Framework implemented in pure Ruby language.

4.2.2 Diablo

Diablo [14] is a framework, developed in C language that provided binary ELF modifications for lot of architectures.

Advantages

Diablo has three main advantages:

- Diablo is a rewriting linker: it takes object files and libraries from which the modified program is built
- Diablo is safe: the extra informations is available at link time (in particular relocation information), it is possible to correctly interpret the complete binary, something is that not always possible without this information,
- Diablo is retargetable. Now, it supports many architectures (ARM, Alpha, IA64, MIPS, PowerPC (32 & 64 bits), i386 and amd64)

Drawbacks

- Diablo only works on statically linked programs, need special compilation parameter (`-static` for GCC),
- Diablo needs more informations about a standard compilation program. In order to obtains this extra informations, we need to use the patched GCC, glibc and binutils version.

Due to a lack of time, I just tested Diablo because it provides the features than I need.

4.3 Diablo

In order to understand how Diablo works, I needed to discover, by some example, this framework. Because of the lake of documentation, this step took me a long time.

4.3.1 How does Diablo work ?

The figure 4.2 describes Diablo process, I can cut it in three parts. First, in the blue zone, this framework parse at least one statically linked executable and your map file, which contains formation about program's global symbols, disassemble the application and analyse statically each input file in order to build internal structures. These internal elements, stored as an oriented graph, are:

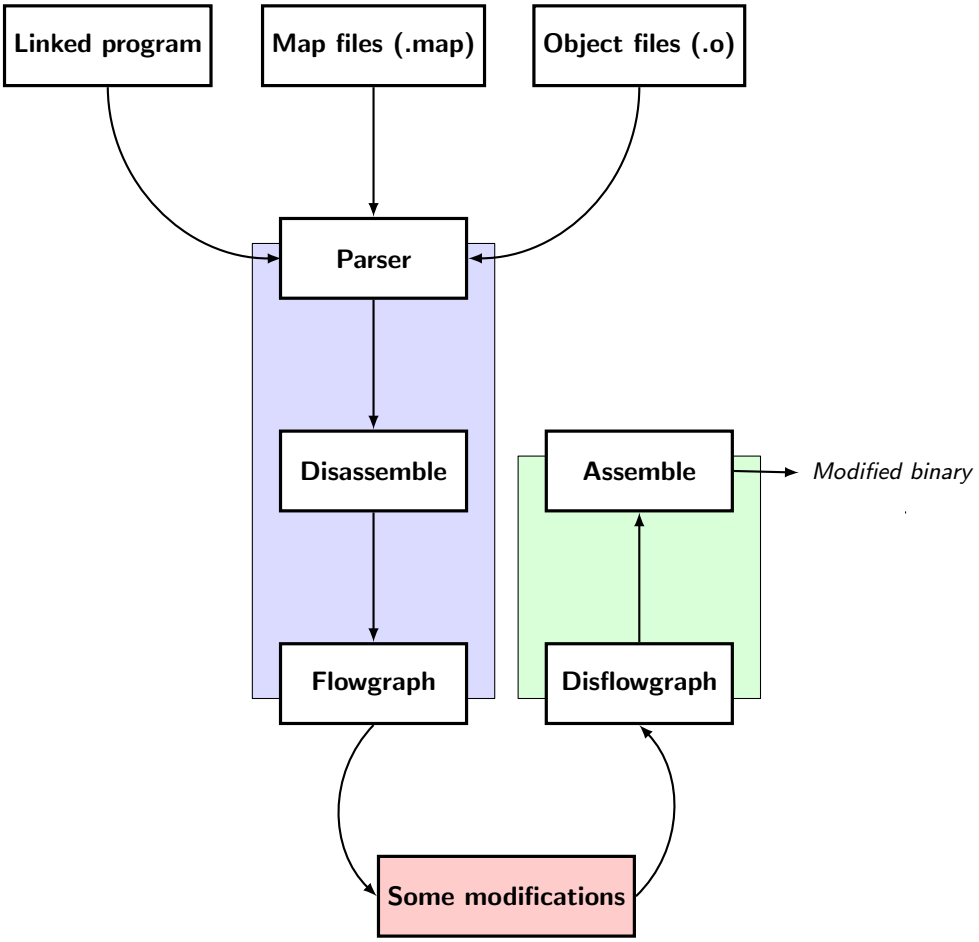


Figure 4.2: Diablo working explication.

- An object which contains a group of parsed binary files (Diablo input(s)),
- A set of ELF sections (see section 4.1),
- CFG is a set of functions, basic blocks, and instructions of Diablo input file(s). All components are statically decided. For each information, its successor and predecessor are defined like a double linked list,
- A set of functions contains basic blocks and its instructions,
- A set of basic blocks,
- A set of instructions,
- A set of Operation Code (opcode) with its parameters.

In the second part, the red part, when the internal structures are loaded, We can interact with any instructions, basic blocks and/or functions. Diablo provides some functions to:

- search a function, basic block or instruction by type, name or address,
- modify an Opcode and its parameters,
- add functions, basic blocks or instructions...
- ...or delete it.

Finally, in the green part, Diablo reconstruct the binary program with our modifications.

4.3.2 My First Hello World

In order to discover Diablo in-depth, I implemented a simple hello world program in C language in the listing 4.1.

```
#include <stdio.h>

int main ( int argc , char **argv ) {
  printf("hello world\n");
  return EXIT_SUCCESS ;
}
```

Listing 4.1: My Hello World in C language

In the subsection 4.3.1, I explained that Diablo needs special compilation options. Indeed, with that, and a patched toolchain [14], I can have some extra information needed by Diablo (information added in the built binary and a needing linked map file). To build correctly my "hello world", I need to build and link my application like in the listing 4.2.

```
# hello.c contains My Hello World code
gcc -c hello.c # We obtains hello.o
gcc -static -Wl,-Map,hello.map -o hello hello.o
```

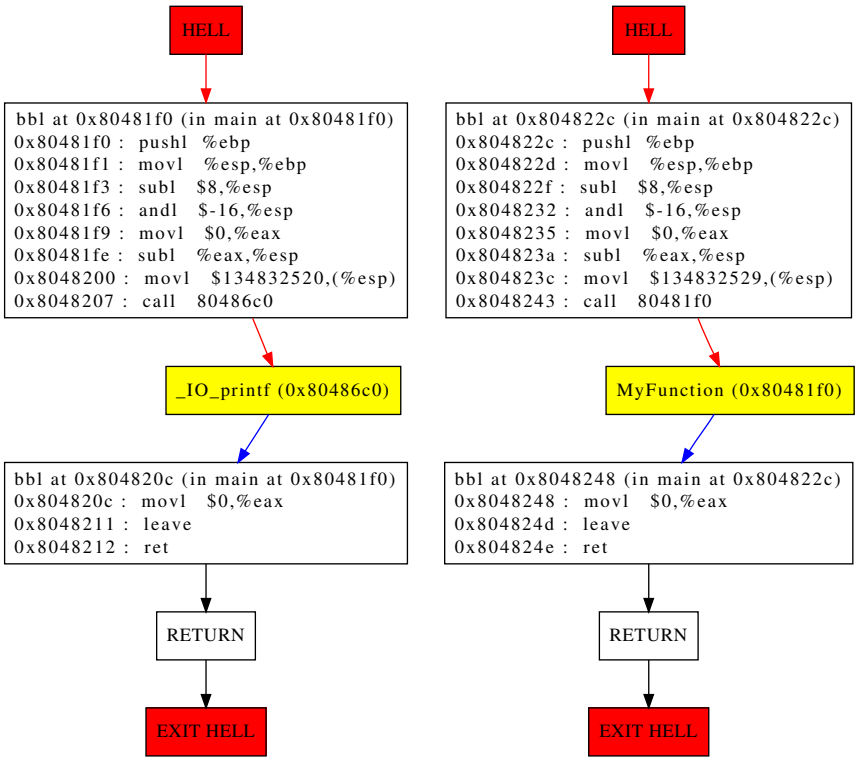
Listing 4.2: Compilation steps

When the "hello world" binary is created, I disassemble it to see this assembler instructions. To disassemble a binary executable, I used `objdump` [15] with Intel syntax mode. In the listing 4.3, We have the disassemble version of the main function.

```
080481f0 <main>:
 80481f0: 55                    push   ebp
 80481f1: 89 e5                 mov    ebp,esp
 80481f3: 83 ec 08              sub    esp,0x8
 80481f6: 83 e4 f0              and    esp,0xfffffff0
 80481f9: b8 00 00 00 00        mov    eax,0x0
 80481fe: 29 c4                 sub    esp,eax
 8048200: c7 04 24 88 61 09 08  mov    DWORD PTR [esp],0x8096188 ;
     push "Hello World"
  return 0 ;
 8048207: e8 b4 04 00 00        call   80486c0 <_IO_printf> ; call
     printf
 804820c: b8 00 00 00 00        mov    eax,0x0  ; push 0x0
 8048211: c9                    leave  ; ready to stop application
 8048212: c3                    ret    ; return eax (0x00) and stop
```

Listing 4.3: My built and linked Hello World

The assembler version of the main function (listing 4.3), instructions, at 0x 08048200 (stack up "Hello World") and 0x 08048207 (call `printf` function), are the most important. To try to discover the Diablo potential, I want to modify `printf` call by a file opening, write "Hello World" in the opened file, and close it.



(a) Original Hello World (b) My File Handling

Figure 4.3: CFG of each binary file main function

In order to do these modifications, Diablo offers two possibilities: you can insert, one by one, each assembler instruction or import a function or a set of functions contained by an external object file. I have chosen the second solution because it is the most simple, and, a quicker solution. So, I have implemented a simple C file handling.

In the listing 4.4, I used two functions:

- the main function to have a working executable,
- the function `MyFunction` which created a file "output", write the `msg` parameter in the opened file, and closed it. This function will replace the `printf` call.

```c
#include <stdio.h>
void MyFunction ( char * msg )
{
  FILE * file = fopen ( "output" , "w" );
  fprintf(file , msg);
  fclose(file);
}

int main ( int argc , char **argv )
{
  MyFunction ( "Congratulation !\n" );
  return EXIT_SUCCESS ;
}
```

Listing 4.4: A Simple write data in C language

Usually, the function parameters are stacked up before the function is called (like figure 4.3). Thus, if I just modify the call instruction, the called function has same parameters on the stack. Diablo provides this feature and it can resolve the dependencies of the new inserted instructions.

If you remember the figure 4.2, Diablo parses your executable before disassembling it. Between these two steps, you can to add extra object files in order to add some extra functions in the input binary application.

```c
/*
 * Create an object with relocation information by
 * emulating the link of the object.
 */
obj = LinkEmulate (
     /* relative address of the original executable to modify it */
        "HelloWorld" ,
        true            // Reads the debug informations
      );
LinkObjectFileNew (  /* Link in an extra file */
    obj                 , // Defined original executable
    "./file_handling.o" , // Link in file name
    SYMBOL_PREFIX       , // Defines the prefix
    TRUE                , // Reads the debug informations
    FALSE                 // Don't prefix undefined symbols
);
ObjectDisassemble (obj); /* disassemble */
```

Listing 4.5: Add extra binary file with Diablo

In the listing 4.5, We can see the Diablo function (line 10) to add an extra binary file. After having linked an object file, I can use all imported functions (here, `MyFunction` and `main`). Next, I just redirect the incoming and outgoing edges which linked to called function by our inserted function. In this example, and in order to redirect the edges, I deleted the outgoing edge of `_IO_printf` calling basic block, listing 4.6, and the incoming edges of last basic block (listing 4.7).

```
while
  (BBL_SUC_FIRST (outgoing_bbl))
{
  /*
   * Get me the first
   * outgoing edge
   */
  t_cfg_edge * outgoing_edge =
    BBL_SUC_FIRST(input_bbl);
  /*
   * Deleting first
   * outgoing edge
   */
  CfgEdgeKill
    (BBL_PRED_FIRST
      (input_bbl)) ;
}
```

Listing 4.6: Diablo instructions to delete outgoing edges of a basic block

```
while
 (BBL_PRED_FIRST (incoming_bbl))
{
  /*
   * Get me the first
   * incoming edge
   */
  t_cfg_edge * incoming_edge =
    BBL_PRED_FIRST (output_bbl);
  /*
   * Deleting first
   * incoming edge
   */
  CfgEdgeKill
    (BBL_PRED_FIRST
      (output_bbl)) ;
}
```

Listing 4.7: Diablo instructions to delete incoming edges of a basic block

Finally, I recreate (listing 4.8), incoming and outgoing edges linked to the inserted function between each basic block.

```
CfgEdgeCreateCall(
  /* cfg = OBJECT_CFG(obj); <= obtains Diablo Flowgraph step */
  cfg,
  /* Representing the call site */
  input_bbl,
  /* Representing the first block of the called function */
  FUNCTION_BBL_FIRST(MyFunction),
  /* Representing the return site (can be NULL) */
  outpout_bbl,
  /* Representing the exit block of the called function (can be
      NULL) */
  FunctionGetExitBlock(MyFunction)
);
```

Listing 4.8: Diablo instructions to delete incoming edges of a basic block

With this modification, Diablo provides the modified binary with "Hello World" written in a file instead of printing on the standard output.

4.3.3 The CouCou World

Previously, I explained how to modify a classic "Hello World" changing an instruction (`printf` call) by a call to our inserted function. Now, I will replace "Hello World" `printf` parameter by "CouCou World" message. In the section 4.1, I explained that the constant data are saved in the `.rodata` section.

In the following part, I use the modifications made in the previous subsection.

In order to modify data pushed on the stack before the function is called, I need to add a new message in the `.rodata` section. Next, I need to stack up this new data before the function is called.

First, I need to add the message "CouCou World" in the `.rodata` section. With Diablo, I can add data on each section contents by the input binary executable.

```
1  t_section* AddData2Rodata
2    ( t_cfg * cfg         ,
3      char * data         , // Data to add in .rodata section
4      t_uint32 data_size  , // Size of data to add
5      char * data_name    ) // Data name
6  {
7    // Initialisation step
8    ...
9    /* Give me .rodata section please */
10   srodata = SectionGetFromObjectByName (CFG_OBJECT(cfg),".rodata");
11   /* Create a Linker to my binary program */
12   linker  = ObjectGetLinkerSubObject   (CFG_OBJECT(cfg)          );
13   /* Create a .rodata child section to add our new data */
14   MyData  = SectionCreateForObject
15     ( linker                  , // Linker to the original object
16       RODATA_SECTION          , // Section type
17       srodata                 , // Parent section
18       AddressNew32(data_size) , // Data size
19       data_name               );// Section name
20
21   /* Copy data in the new .rodata child section */
22   memcpy(SECTION_DATA(MyData), data, data_size*sizeof(char));
23   return MyData;
24 }
```

Listing 4.9: Diablo instructions to add data in `.rodata` section

To add new data in the `.rodata` section, see listing 4.9, I need to follow four steps:

- First, in the line 10, Diablo gives a pointer to the `.rodata` section.
- Second, I need a link to my original object, line 12.
- Third, line 14, I created a subsection of `.rodata` section in order to save my new data.
- Finally, I copied, line 22, the new data in the new subsection.

Now, our message is saved in the `.rodata` section. It remains to stack up this data before call the inserted function (`MyFunction`). Here, I would like to change the "Hello World" address, such as function parameter, on a `esp` register. Thus, I have, like on the listing 4.3, at the assembler instruction 0x 08048200:

```
mov DWORD PTR [esp] , 0x8096188
```

In this example, I pushed the value at the address 0x 08096188 on the `esp` register. The stack up value corresponds to the "Hello World" address on the binary executable.

In this case, I would like to modify this instruction, but Diablo does not know the "CouCou World" address during the modification step. In order to modify with the correct address, I need to declare, explicitly, the second `mov` instruction parameter like relocatable for Diablo. For this, I need to use the `RelocTableAddRelocToRelocatable` function.

```
reloc = RelocTableAddRelocToRelocatable
  ( OBJECT_RELOC_TABLE(CFG_OBJECT(cfg)),
    /* relocate offset value */
    AddressNew32  (    0 ) ,
    /* Instruction to modify */
    T_RELOCATABLE (  mov ) ,
    /* Address position to modify */
    AddressNew32  (    3 ) ,
    /* it points to your message */
    T_RELOCATABLE (  msg ) ,
    /* the message is at offset 0 in the subsection you created */
    AddressNew32  (    0 ) ,
    /* not a relocation to hell
     * (hell = "it might point anywhere")
     */
    FALSE                  ,
    /* not related to an edge of the control flow graph */
    NULL                   ,
    /* this parameter is no longer used and should be removed */
    NULL                   ,
    /* we don't need a symbol in the address calculation of our */
    NULL                   ,
    /* Calculation of the value */
    "R00A00+" "\\" WRITE_32
  );
/* force relocate mov instruction */
I386_OP_FLAGS(I386_INS_SOURCE1(mov)) = OPFLAG_ISRELOCATED;
```

Listing 4.10: Diablo instruction to relocate data address

The actual calculation of the value, for the last function argument, to be written at this place:

- take the address of relocatable (R00 = `msg`; `mov` is the "from" relocatable and hence not counted in this list),
- add the value of addend 0 (A00+),
- and then write the resulting 32 bits.

When Diablo has made this modification, the output binary application write the message "CouCou World" in a file.

Diablo is complex, powerful, but poorly documented. Now, with my explications, I can, at least, add a external function, and new data, and change a `call` instruction and these values stacked up before it. With these modifications, I have enough knowledge to create a proof of concept.

Chapter 5

Proof Of Concept

> Never trust anything that can think for itself if you can't see where it keeps its brain.
>
> Harry Potter and the Chamber of Secrets
> J.K. Rowling

Because of a lack time, I made the choice to create a simple proof of concept. This proof aims to modify each multiplication instructions, in a 1024 square matrix, in order to calculate these operations by our Java Card.

This proof of concept is realised on a x86 (32-bit) computer architecture. In this architecture, an integer is stored on 32 bits. Most multiplication instructions use, at least, two 32-bit registers. For the following, I use an integer number is the same to 32-bit number.

5.1 Java Card side

The *JCOP 31/36k v2.2* uses Java Card 2.2.1. In this version, I can only make multiplication operations with `byte` (8-bit) and `short` (16-bit) numbers. Moreover, Java Card API 2.2.2 [16] specifies a `math.BigNumber` class which provides elementary operations on a n-byte numbers (all implementations must support at least 8-byte length internal representation capacity). Unfortunately, this class is not present in our smart card Java Card Virtual Machine.

In the security laboratory, there is an implementation of a integer library for Java Card with few elementary operations for 32-bit numbers. This class provides only integers addition and subtraction but not integers multiplication. Try to implement it!

5.1.1 Integers multiplication on Java Card

I have two 32-bit numbers a and b which can be written like:

$$\begin{cases} a = a_3x^3 + a_2x^2 + a_1x + a_0 \\ b = b_3x^3 + b_2x^2 + b_1x + b_0 \end{cases}$$

So

$$a = \sum_{i=0}^{3} a_i x^i \text{ and } b = \sum_{j=0}^{3} b_j x^j$$

Then

$$\begin{aligned} a \times b &= \sum_{i=0}^{3} a_i x^i \times \sum_{j=0}^{3} b_j x^j = \sum_{k=0}^{6} C_k x^k \\ \text{with } C_k &= \sum_{i+j=k}^{3} a_i b_j = \sum_{i=0}^{k} a_i b_{k-i} \end{aligned}$$

With a computer point of view, a and b sent by a APDU request and received by the smart card such as a 4-byte array. Thus, I can store these values like:

$$a = [a_3, a_2, a_1, a_0] \text{ and } b = [b_3, b_2, b_1, b_0]$$

After I have selected my multiplication applet, and the card received the correct parameters, the smart card makes a classic and unoptimised multiplication (algorithm 1).

Algorithm 1: Integers multiplication on Java Card

```
Input: a and b: unsigned numbers stored in a 4-byte array each
Data: ret: an array of byte with 7 elements
Result: a × b mod 2^32 in a 4-byte array
begin
    temp ⟵ 0  // This variable is a 16-bit number
    arrayFill(ret, 0)
    for i = (Lenght(a) − 1) to 0 do
        for j = (Lenght(b) − 1) to 0 do
            temp ⟵ a_i × b_i
            ret[i + j] ⟵ LowPart(temp)
            if HighPart(temp) ≠ 0 then // carry propagation
                if (i + j) ≠ 0 then
                    ret[i + j − 1] ⟵ HighPart(temp)
                end
            end
        end
    end
    return ret
end
```

Now, the smart card can receive two 32-bit numbers and return their multiplication. For the next step, I need to find a way to send, at the smart card, the 32-bit numbers to multiply, and receive the result.

5.2 Communication between binary application and the smart card

The Java Card implements the ISO7816 specification. The ISO7816 is a standard describes a protocol which provides a structure to the APDU request.

There is an open source library, `libpcsc-lite` [17], providing low level functions to communicate with a smart card. In order to use the Java Card functions, it provides a complex C++ classes, implemented `libpcsc-lite`, and add many features (multi management of smart card, cryptographic functions, etc.).

Of course, for our proof of concept, this kind of library is complex, and I need just a function to select the multiplication applet, send 32-bit numbers to multiply and receive the multiplication result.

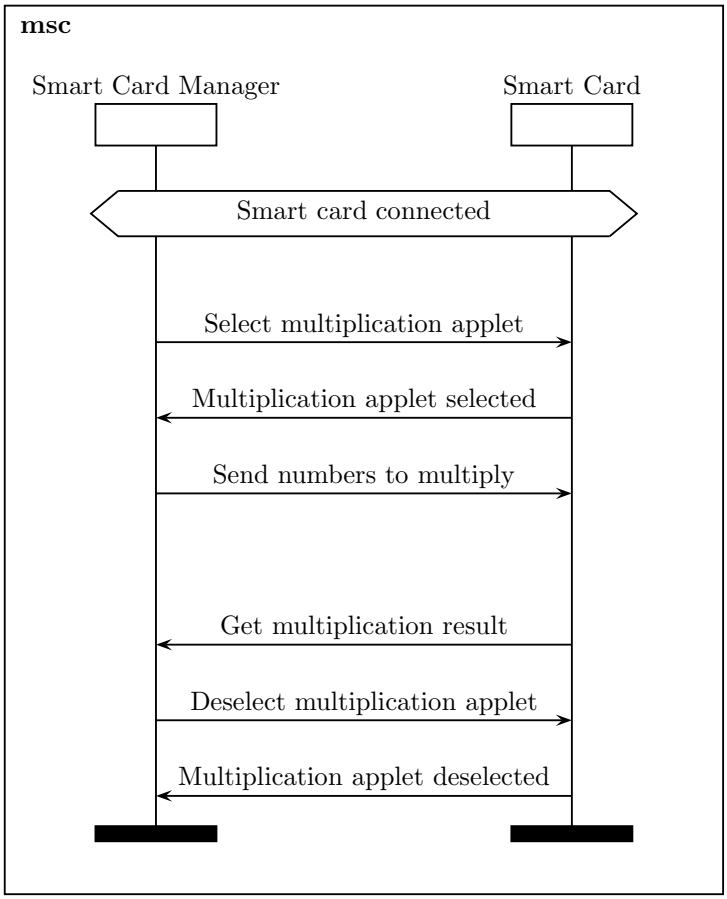


Figure 5.1: Communication between Smart Card Manager and the Smart Card.

To communicate with the Java Card, and our multiplication applet, I need to follow this simple protocol (figure 5.1):

- Select multiplication applet
- Send 32-bit numbers to multiply to the multiplication applet
- Receive the multiplication result
- Deselect multiplication applet

Once this protocol is implemented, I can change, with Diablo, each multiplication instruction by a call to the Smart Cart Manager.

5.3 Binary Modification

The first solution to communicate with the smart card is not conclusive because of Diablo cannot parses correctly the object files results from C++-language source compilation. In order to have a working proof of concept, I implemented a simple smart card manager in C-language which respects the protocol define in the figure 5.1 using only `libpcsc-lite` dependency.

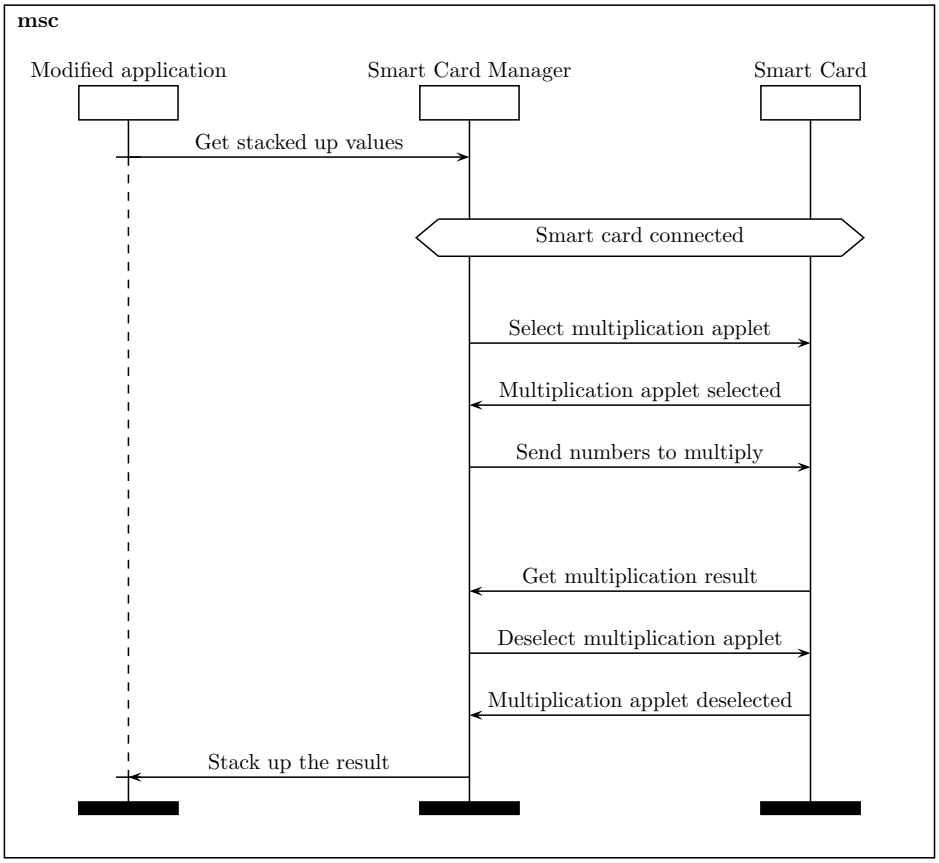


Figure 5.2: Communication between the modified application and the smart card contains multiplication operation.

Now, the main idea is to change each multiplication instruction in my matrix product (appendix A) by the function for communicate to the smart card. In order to modify the binary application, I should to follow the protocol define in the figure 5.2.

These modifications are very similar to these described for Diablo in the subsection 4.3.2 and 4.3.3. Unfortunately, when I am writing these lines, Diablo cannot parse the `libpcsc-lite`. I sent an email to the Diablo developers to find a solution. Now, the problem is always here...

Conclusion

First, I describe the objectives accomplished this internship. Next, I explained my difficulties. After, I suggest eventual perspectives for this work and to finish I will make a personal conclusion about this internship.

Objectives accomplished

At the end of my internship, I can to profile binary application with a tool like OProfile without program source code. This profiling application dynamically described the cost of each executed instructions. Thus, with this information, I will can determinate with the specific heuristics, the sensitive piece of code. Theses heuristics will should study in another work.

Next, to translate the extracted sensitive piece of code in order to protect it in a dongle, I tried to use UQBT. An external component cannot understand the sensitive instructions. Because of the problems due to the age of UQBT, this step has allowed to study a way for execute the sensitive set of instructions on the dongle.

Finally, with Diablo, I can modify a piece of code in a binary executable but this step needs at least the mapping files of the application to modify. Thus, I can to find a set of instructions (the sensitive piece of code) in order to replace that by the instructions to communicate to the dongle which contains the protected piece of code. So as to give this modification consistent, the used registers value should be sent to the dongle in order to the protect piece of code does correctly executed.

Difficulties encountered

The main difficulty was my English level. During each step of my university formation, I neglected to learn it. When I began my internship, I need to improve himself the knowledges in this language. In order to take the tiger by the tail, I practiced with emails exchange to Diablo developers later, reported the conference sum up in English and, as you will see, I wrote my internship report in the Shakespeare language.

Before this project, I did not really try to discover how binary executables do was in an operating system. During I progressed my internship, I can under-

stand in-depth this side when I cannot to modify correctly a binary application. Moreover, these difficulties allowed me to acquire how each compilation steps do work.

Finally, Diablo is an undocumented framework. This lack off information forced me by a trial and error method to discover how its working. Thus, with progressive difficulty binary modification, and the helps of Diablo developers, I succeed to understand Diablo working. Unfortunately, I cannot successful to make the proof of concept because of Diablo cannot to parse the `libpcsc-lite`. Now, this problem is persisted.

Possible improvements

In order to combine the advantages to Java Card Security and Java development facilities, I had chose a Java Card such as a dongle to protect the sensitive piece of code. This support is weak at the side channel attack. The works described in [18] and [19] explains that executed instructions in a Java Card 2.2 and Java Card 3 can be determined by a correlation attack. A strong type of dongle of this attack should be used to protect the sensitive piece of code executed.

Next, in order to measure the cost of this protection, a complete proof of concept should be implemented. Thus, with a knew application, you can test each protection step (profiling, extraction of the sensitive piece of code, translation of the extracted instructions and the binary modification) for having a global view. So, you can to determine the viability and the constraints of this type of protection.

Finally, in order to improve the security, we should obfuscate the piece of code that communicate with the dongle. You can obfuscate too the APDU requests. Moreover, Diablo uses a older toolchain for make the binary modifications. Now, Diablo need a patched GCC-3.3 (GCC-4.5 is yet used). The used version provides a security flaw on the generated binary. After a mail exchange with the Diablo developers, the GCC-4.5 specifications managed by Diablo need to a long time modification. Now, the developers preferred to improved the different architectures support.

Personal impact

During my internship, I discovered the aspects of a research final year project in laboratory. This training period is the first time in a private laboratory. Thus, I assessed the different dynamic than a public research center. The public laboratory bases your researches on a thematic which that unnecessary linked with the Companies needs. On the contrary, in a private lab, the researches are going to the Company objectives and a project have to aims to be final product. I liked contribute to the begin step of this project.

With this internship, I can discovered the reality of this subject type. Thus, in a research, you preferred found a generic solution instead an only working

solution for the same problem. The second approach, the engineer method, is the hander solution. Unlike that, the first approach provides the main line of the future engineer works.

At the end of my studies, I would like to contribute to the computer science theory with an innovative project. Thus, I am searching a thesis in order to involve, on a research way, to the future computer science project.

Bibliography

[1] CESA. Piracy cost game industry over 3.8 trillion Yen. http://www.andriasang.com/e/blog/2010/06/04/cesa_piracy_report/, June 2010.

[2] A.C. Noubissi, A.A.K. Séré, J. Iguchi-Cartigny, J.L. Lanet, G. Bouffard, and J. Boutet. Cartes à puce: Attaques et contremesures. In *MajecSTIC*. University of Limoges – XLim, November 2009.

[3] OProfile. http://oprofile.sourceforge.net.

[4] Valgrind. http://valgrind.org.

[5] SimPoint. http://cseweb.ucsd.edu/~calder/simpoint/.

[6] BBV: an experimental basic block vector generation tool. http://valgrind.org/docs/manual/bbv-manual.html.

[7] Callgrind Format Specification. http://valgrind.org/docs/manual/cl-format.html.

[8] Cristina Cifuentes, Mike Van Emmerik, Norman Ramsey, and Brian Lewis. a retargetable static binary translation framework. Technical report, University of Queensland, 2002.

[9] Cristina Cifuentes, Mike Van Emmerik, and Norman Ramsey. Uqbt: A resourceable and retargetable binary translator. http://www.itee.uq.edu.au/~cristina/uqbt.html.

[10] Thomas Garnier. Introduction au format elf. http://www.supinfo-projects.com/fr/2005/introduction_elf_fr/, Janvier 2005.

[11] ATT. System V Application Binary Interface. http://www.sco.com/developers/devspecs/gabi41.pdf, March 1997.

[12] Wikipedia. Executable and linkable format. https://secure.wikimedia.org/wikipedia/en/wiki/Executable_and_Linkable_Format, Juin 2010.

[13] Yoann Guillot and Julien Tinnes. The metasm assembly manipulation suite. http://metasm.cr0.org/.

[14] PARIS research group, ELIS department, and Ghent University. Diablo is a better link-time optimizer. https://diablo.elis.ugent.be/.

[15] The GNU binutils. http://www.gnu.org/software/binutils/.

[16] Oracle. Java card 2.2.2 api. http://www.win.tue.nl/pinpasjc/docs/apis/jc222/index.html.

[17] Pcsc-lite. http://pcsclite.alioth.debian.org/.

[18] Dennis Vermoen, Marc Witteman, and Georgi N. Gaydadjiev. Reverse engineering java card applets using power analysis. In *WISTP'07: Proceedings of the 1st IFIP TC6 /WG8.8 /WG11.2 international conference on Information security theory and practices*, pages 138–149, Berlin, Heidelberg, 2007. Springer-Verlag.

[19] G. Barbu, H. Thiebeauld, and V. Guerin. Attacks on Java Card 3.0 Combining Fault and Logical Attacks. *Smart Card Research and Advanced Application*, pages 148–163, 2010.

[20] Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[21] J. Caballero, N.M. Johnson, S. McCamant, and D. Song. Binary code extraction and interface identification for security applications. In *Network and Distributed Systems Symposium (NDSS)*, 2010.

[22] A. Maña, J. Lopez, J.J. Ortega, E. Pimentel, and J.M. Troya. A framework for secure execution of software. *International Journal of Information Security*, 3(2):99–112, 2004.

[23] Chenxi Wang. *A security architecture for survivability mechanisms*. PhD thesis, University of Virginia, Charlottesville, VA, USA, 2001. Adviser-Knight, John.

[24] Èric Petit. *Vers un partitionnement automatique d'applications en codelets spéculatifs pour les systèmes hétérogènes à mémoires distribuées*. PhD thesis, University of Rennes 1, Rennes, France, 2009.

[25] Wikipedia. Basic block. https://secure.wikimedia.org/wikipedia/en/wiki/Basic_block.

[26] Wikipedia. Opcode. https://secure.wikimedia.org/wikipedia/en/wiki/Opcode.

[27] Wikipedia. Apdu. https://secure.wikimedia.org/wikipedia/en/wiki/Apdu.

Glossary

APDU
The Application Protocol Data Unit (APDU) is the communication unit between a smartcard reader and a smartcard. The structure of an APDU is primarily defined by ISO/IEC 7816-4 [27] . 22, 23, 27

basic block
A basic block (BB) is a sequence of instructions with ony one entry point and only one exit point [25] . 4–6, 14, 15, 17, 18

Control Flow Graph
A Control Flow Graph is a graph representation of the all paths can that be used by a program. 5, 14

Diablo
Diablo is a retargetable link-time binary rewriting framework. A full description is giving in the chapter 4, section 4.2.2 . 13–20, 24–27

ELF
The Executable and Linkable Format is a common standard file format for executables, object code, shared libraries, and core dumps [12]. 11, 12, 14

GCC
The GNU Compiler Collection is a collection of compilation system supporting various programming language and provided by GNU project. 10, 12, 13, 27

Opcode
An operation code is the portion of a machine instruction that specifies the operation to be performed [26] . 14, 15

List of Figures

List of Listings

Appendices

Appendix A

Matrix Product

Listing A.1: matrix_product.c

```
/*
 * ========================================================
 *
 *       Filename:  matrice_product.c
 *
 *    Description:  An unoptimised matrix multiplication
 *
 *        Version:  1.0
 *        Created:  03/04/2010 04:12:16 PM
 *       Revision:  08/10/2010 11:24:53 AM
 *       Compiler:  gcc
 *
 *         Author:  Guillaume Bouffard
 *           Mail:  guillaume.bouffard@technicolor.com
 *        Company:  Technicolor
 *
 * ========================================================
 */

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

#define SIZE_MATRIX     1024

/*
 * {{{ Matrix initialisation
 * \param matrix  Matrix to allocate
 * \param m       Number of lines
 * \param n       Number of rows
 * \return        New created matrix
 */
int **
initialisation (int **matrix, size_t m, size_t n)
{
  size_t i = 0; int **t = NULL;
  matrix = (int **) malloc (m * sizeof (int *));
  if (matrix == NULL)
    {
      perror ("Matrix not allocated");
      exit (errno);
```

```
    }
  t = matrix;
  for (i = 0; i < m; ++i)
    {
      *t = (int *) calloc (n, sizeof (int));
      if (*t == NULL)
        {
          perror ("Matrix not allocated");
          exit (errno);
        }
      ++t;
    }
  return matrix;
}
/* }}} */

/*
 * {{{ Delete a matrix
 * \param matrix  Matrix to delete
 * \param m       Number of lines
 */
void
delete_matrix (int **matrix, size_t m)
{
  int **t = matrix; int i = 0;
  for (i = 0; i < m; ++i)
    {
      free (*t);
      ++t;
    }
  free (matrix);
}
/* }}} */

/*
 * {{{ Multiply two matrices
 * \param m1  Matrix 1
 * \param m2  Matrix 2
 * \param m   Number of matrix 1 lines
 * \param n   Number of matrix 2 rows
 * \param p   Number of matrix 1 rows == Number of matrix 2 lines
 * \return    Product of matrix 1 & 2
 */
int **
matrix_multiplication (int **m1, int **m2, size_t m, size_t n,
    size_t p)
{
  size_t i, j, k ; int **multiplication ;
  if (!initialisation (multiplication, m, p)) return NULL;
  for (i = 0; i < m; ++i)
      for (j = 0; j < p; ++j)
          for (k = 0; k < n; ++k)
              multiplication[i][j] += m1[i][k] * m2[k][j];
  return multiplication;
}
/* }}} */

/*
 * {{{ Randomize a matrix contents
 * \param m1  Matrix to randomize
 * \param m   Number of matrix lines
 * \param p   Number of matrix rows
```

```
 * \return    Matrix randomized
 */
int **
randomize_matrix (int **matrix, size_t m, size_t p)
{
  int i, j;
  srandom (random ());
  for (i = 0; i < m; ++i)
    {
      for (j = 0; j < p; ++j)
        {
          matrix[i][j] = random ();
        }
    }
  return matrix;
}
/* }}} */

/*
 * {{{ Main function
 */
int
main (int argc, char **argv)
{
  int **m1 = NULL, **m2 = NULL, **multi = NULL;
  m1 = initialisation (m1, SIZE_MATRIX, SIZE_MATRIX);
  if (m1 == NULL)
    {
      perror ("M1 Created");
      exit (1);
    }
  m2 = initialisation (m2, SIZE_MATRIX, SIZE_MATRIX);
  if (m2 == NULL)
    {
      perror ("M2 Created");
      delete_matrix (m1, SIZE_MATRIX);
      exit (1);
    }
  randomize_matrix (m1, SIZE_MATRIX, SIZE_MATRIX);
  randomize_matrix (m2, SIZE_MATRIX, SIZE_MATRIX);
  multi = matrix_multiplication (m1, m2,
                                 SIZE_MATRIX ,
                                 SIZE_MATRIX ,
                                 SIZE_MATRIX );
  if (multi == NULL)
    {
      perror ("Multiplication done");
      exit (2);
    }
  delete_matrix (m1   , SIZE_MATRIX);
  delete_matrix (m2   , SIZE_MATRIX);
  delete_matrix (multi, SIZE_MATRIX);
  return 0;
}
/* }}} */
```

Appendix B

Profiling matrix product results

B.1 OProfile results

B.1.1 Cost for each function

```
opreport -l Produit_Matrice --merge=all
CPU: Core 2, speed 2600 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Clock cycles when not halted)
  with a unit mask of 0x00 (Unhalted core cycles) count 400000
samples  %        app name                 symbol name
106009   99.2594  Produit_Matrice          matrix_multiplication
553       0.5178  no-vmlinux               /no-vmlinux
77        0.0721  libc-2.11.1.so           random_r
72        0.0674  libc-2.11.1.so           random
52        0.0487  Produit_Matrice          randomize_matrix
26        0.0243  libc-2.11.1.so           __i686.get_pc_thunk.bx
3         0.0028  libc-2.11.1.so           __memset_ia32
2         0.0019  libc-2.11.1.so           _int_free
2         0.0019  libc-2.11.1.so           _int_malloc
1        9.4e-04  Produit_Matrice          initialisation
1        9.4e-04  ld-2.11.1.so             _dl_fixup
1        9.4e-04  ld-2.11.1.so             do_lookup_x
1        9.4e-04  libc-2.11.1.so           calloc
```

B.1.2 Cost for each instructions

```
/*
 * Command line: opannotate -a -t 50 Produit_Matrice --merge=all
 *
 * Interpretation of command line:
 * Output annotated assembly listing with samples
 *
 * CPU: Core 2, speed 2600 MHz (estimated)
 * Counted CPU_CLK_UNHALTED events (Clock cycles when not halted)
 * with a unit mask of 0x00 (Unhalted core cycles) count 400000
 */
[...]
               :
08048642 <matrix_multiplication>: /* matrix_multiplication
                                     total: 106009 99.2594  */
```

```
samples %
[...]
               : 804867d: jmp  804870b <matrix_multiplication+0xc9>
               : 8048682: movl $0x0,0xfffffff0(%ebp)
               : 8048689: jmp  80486ff <matrix_multiplication+0xbd>
    2   0.0019 : 804868b: movl $0x0,0xffffffec(%ebp)
               : 8048692: jmp  80486f3 <matrix_multiplication+0xb1>
 2744   2.5693 : 8048694: mov  0xfffffff4(%ebp),%eax
   29   0.0272 : 8048697: shl  $0x2,%eax
    2   0.0019 : 804869a: add  0xfffffff8(%ebp),%eax
   32   0.0300 : 804869d: mov  (%eax),%eax
 2845   2.6639 : 804869f: mov  0xfffffff0(%ebp),%edx
   18   0.0169 : 80486a2: shl  $0x2,%edx
    2   0.0019 : 80486a5: lea  (%eax,%edx,1),%edx
   24   0.0225 : 80486a8: mov  0xfffffff4(%ebp),%eax
 2626   2.4588 : 80486ab: shl  $0x2,%eax
    1  9.4e-04 : 80486ae: add  0xfffffff8(%ebp),%eax
               : 80486b1: mov  (%eax),%eax
  151   0.1414 : 80486b3: mov  0xfffffff0(%ebp),%ecx
 2631   2.4635 : 80486b6: shl  $0x2,%ecx
               : 80486b9: add  %ecx,%eax
   54   0.0506 : 80486bb: mov  (%eax),%ecx
 1276   1.1948 : 80486bd: mov  0xfffffff4(%ebp),%eax
 2213   2.0721 : 80486c0: shl  $0x2,%eax
               : 80486c3: add  0x8(%ebp),%eax
   25   0.0234 : 80486c6: mov  (%eax),%eax
  633   0.5927 : 80486c8: mov  0xffffffec(%ebp),%ebx
 2021   1.8923 : 80486cb: shl  $0x2,%ebx
               : 80486ce: add  %ebx,%eax
   74   0.0693 : 80486d0: mov  (%eax),%ebx
 1787   1.6732 : 80486d2: mov  0xffffffec(%ebp),%eax
 2057   1.9260 : 80486d5: shl  $0x2,%eax
               : 80486d8: add  0xc(%ebp),%eax
   12   0.0112 : 80486db: mov  (%eax),%eax
  862   0.8071 : 80486dd: mov  0xfffffff0(%ebp),%esi
 1975   1.8493 : 80486e0: shl  $0x2,%esi
    2   0.0019 : 80486e3: add  %esi,%eax
  135   0.1264 : 80486e5: mov  (%eax),%eax
64249  60.1582 : 80486e7: imul %ebx,%eax
 7855   7.3549 : 80486ea: lea  (%ecx,%eax,1),%eax
 2585   2.4204 : 80486ed: mov  %eax,(%edx)
 3876   3.6292 : 80486ef: addl $0x1,0xffffffec(%ebp)
 3021   2.8287 : 80486f3: mov  0xffffffec(%ebp),%eax
   14   0.0131 : 80486f6: cmp  0x14(%ebp),%eax
  134   0.1255 : 80486f9: jb   8048694 <matrix_multiplication+0x52>
    1  9.4e-04 : 80486fb: addl $0x1,0xfffffff0(%ebp)
   29   0.0272 : 80486ff: mov  0xfffffff0(%ebp),%eax
    8   0.0075 : 8048702: cmp  0x18(%ebp),%eax
    4   0.0037 : 8048705: jb   804868b <matrix_multiplication+0x49>
[...]
```

B.2 Valgrind

B.2.1 Valgrind Callgrind tool result

```
version: 1
creator: callgrind-3.6.0.SVN-Debian
pid: 23048
cmd:  ./Produit_Matrice
part: 1
```

```
desc: I1 cache: 32768 B, 64 B, 8-way associative
desc: D1 cache: 32768 B, 64 B, 8-way associative
desc: L2 cache: 2097152 B, 64 B, 8-way associative

desc: Timerange: Basic block 0 - 1118197422
desc: Trigger: Program termination

positions: instr
events: Ir Dr Dw I1mr D1mr D1mw I2mr D2mr D2mw
summary: 41001103990 22624609395 2199170632 806 1209805220 327135
    800 67376142 327037

[...]

cob=(5) /home/bouffardg/stage/test_Profiling/Programmes_tests/
    Produit_Matrice/Produit_Matrice
cfi=(58) ???
cfn=(374) 0x080484d0
calls=1 0x80484d0
* 18 5 3 0 1
+5 1 1
+1 1 1
+1 1 1
+1 1 1

fn=(190) 0x0401f31c
0x401f31c 1 0 1 1
+1 1
+2 1 0 1
+1 1
+3 1
+6 1
+6 1 1
+6 1
+2 1
jcnd=1/1 +7
*
+7 1 0 1 1 0 0 1
cob=(3) /usr/lib/valgrind/vgpreload_core-x86-linux.so
cfi=(42) ???
cfn=(194) 0x00000410
calls=1 0x410
* 15 6 3 2 0 0 2

ob=(4)
fl=(63)
fn=(298)
0xc6715 1
+6 1 0 1
+1 1
+2 1 1
+1 1 1
+6 1 1
+3 1
+5 1
+2 1
+3 1 1

[...]

fl=(37)
fn=(134)
```

```
0x15ff0 5 0 5 1 0 0 1
+1 5 5
+4 5 5
+4 5 5
+4 5
+5 5
+2 5 5
+1 5
+5 5
+2 5 5

fn=(178)
0x15fb0 1
+2 1 1
+4 1 1
+4 1
+5 1
+2 1
+2 1
+5 1
+2 1 1

totals: 41001104082 22624609211 2199170540 990 1209805220 327135
    800 67376142 327037
```

B.2.2 Valgrind BBV tool

BB out file

```
[...]
T:1941:2   :1942:3
T:1943:2   :1944:3
T:1945:2   :1946:3
T:1947:5
T:1947:5
T:1947:5
T:1947:5
T:1947:5
T:1947:5
T:1947:5
T:1947:5
T:1947:5
T:1947:5
T:1947:5
T:1947:5
T:1947:5
T:1947:5
T:1947:5
T:1947:5
T:1947:5
T:1947:5
T:1947:5
T:1947:5
T:1947:5
T:1947:5
T:1947:5
T:1947:5
T:1947:5
T:1947:5
T:1947:5
[...]
```

PC out file

```
 :ID:basic block address:function name
[...]
F:1627:8048554:initialisation
F:1777:8048576:initialisation
F:1778:804859a:initialisation
F:1780:80485a9:initialisation
F:1810:80485bc:initialisation
F:1811:80485e7:initialisation
F:1779:80485ef:initialisation
F:1821:80485f7:initialisation
F:1926:8048642:matrix_multiplicati
F:1940:8048663:matrix_multiplicati
F:1941:8048676:matrix_multiplicati
F:1943:8048682:matrix_multiplicati
F:1945:804868b:matrix_multiplicati
F:1947:8048694:matrix_multiplicati
F:1946:80486f3:matrix_multiplicati
F:1948:80486fb:matrix_multiplicati
F:1944:80486ff:matrix_multiplicati
F:1949:8048707:matrix_multiplicati
F:1942:804870b:matrix_multiplicati
F:1873:8048721:randomize_matrix
F:1889:804872d:randomize_matrix
F:1916:8048735:randomize_matrix
F:1918:804873e:randomize_matrix
F:1920:8048747:randomize_matrix
F:1921:8048760:randomize_matrix
F:1919:8048766:randomize_matrix
F:1922:804876e:randomize_matrix
F:1917:8048772:randomize_matrix
F:1923:804877a:randomize_matrix
F:1626:8048783:main
F:1822:80487c0:main
F:1823:80487e3:main
F:1871:80487ff:main
F:1872:8048836:main
F:1924:8048852:main
F:1925:804886e:main
F:1607:8048910:__libc_csu_init
F:1609:804891b:__libc_csu_init
F:1618:8048929:__libc_csu_init
F:1619:8048962:__libc_csu_init
F:1608:804896a:
F:1615:8048970:
F:1616:8048994:
```