Command line replacements for GNU Binutils

Proposal for GSOC 2018
Paul Semel

Name: Paul Semel
Email: semelpaul@gmail.com
Phone: +33 6 65 31 53 44
Programming languages: C / C++
Past open source contributions: Xen hypervisor, Xen testing framework
Pending open source contributions: LLVM (https://reviews.llvm.org/D44093)
Sample source code, hobby projects, etc (GitHub): https://github.com/paulsemel
(LDSO - a simple dynamic linker for linux)

Summary

Overview

The “*LLVM Binutils*” is a suite of powerful tools that permits user to do some operations on binaries. Despite the fact that it might sometimes have more features than its sister, the “*GNU Binutils*”, this last one remains more used than the LLVM one. This might probably be explainable by the ancientness of the GNU one.
However, the fact is that, people are just accustomed to the GNU Binutils, and almost every build toolchains are using those ones. To make those toolchains switch to LLVM Binutils, we definitely need to provide tools that are behaving the same as the GNU Binutils ones, so that people don’t have to change the whole code behind the toolchain to make it work.
To sum it up, we can distinguish two parts of refactoring : the command line and the output compliance.
Indeed, a lot of toolchains/bash scripts are relying on the output of those binaries. For example, I used in the past a runtime binary patching toolchain that was relying on the output of the **GNU readelf**, which is in fact totally different from the **llvm-readelf** one.
All of this led me to think that it is a really bad thing that people aren’t using LLVM tools, that are in my opinion more extendable (see below my thoughts about **llvm-objcopy**).
As a first shot, it would thus be needed to add the missing features in the different LLVM Binutils so that we won’t lack those ones anymore, and we would at least be able to use the toolchain without having any technical impossibilities.
Then, the second objective will be to make the command lines compliant with the GNU one. This will basically consist of linking the correct options with the correct features.
Last but not least, if there is some time remaining for this GSOC, it would be great to adjust the output format of the suite. This last one might not be a hard task to complete, but might somehow be time consuming depending on how far the LLVM Binutils formating is from the GNU Binutils.
Thus, the main objective of the project would be that people/organisations are able to switch from GNU Binutils to LLVM Binutils with the less efforts as possible.

Who am I ?

Before exposing my motivation, I thought it would be great that I present myself.
I am a 4th year student currently studying at EPITA. Aside from my school, I am also part of the system and security laboratory of EPITA, where we are working on various system related projects.
At first, I would consider myself as a kernel developer student, but my recent works made me think that it’s probably not the thing I really like to do.
I enrolled the laboratory last year, where I first worked on Valgrind; my goal was to detect security vulnerabilities by playing with the Valgrind Core IR.

Then, I started working on the development of a 64 bits kernel, on which I worked for about 5 months.
I am kind of used to the ELF internal structure, as I have done multiple projects that were implying parsing an ELF binary. First, as you can see on my github, I have implemented a dynamic linker, which was actually the project I did to enter in my laboratory. Then, I implemented a readelf like tool, but my goal was to be able to dump the pretty printed content of the ELF in absolutely any format (like JSON, XML etc..). I created an interface so that I could add new output formats really easily (this project is also present on my github).
After that, I worked on the Xen hypervisor, and more precisely on the Xen Testing Framework (http://xenbits.xen.org/docs/xtf/), where I worked on implementing some new features in the micro-kernel. This work was done in the context of my mandatory internship for my school, which I did at Amazon AWS, in Germany.
This is during this time that I started working on the LLVM/CLang projects.

Why applying ?

It's been now a few months that I'm working on the LLVM project. I actually started to work on this project because of my needs.
I first wanted to be able to detect unsequenced modifications in a code base. Indeed, I remarked that there was no good open source tools that were doing this, and the clang warning was not efficient enough for my needs.
The first step was to learn the LLVM code base, which was not that easy, even if I have to admit that working on the IR of Valgrind the past year helped me. My goal for this project was just to have a PoC, to ensure that, yes, it was possible to detect this undefined behavior. (https://reviews.llvm.org/D44154)
After that, I started working on a structure pretty printer CLang built-in. Indeed, as I am originally doing kernel development, it's been at least a year that I am thinking of this kind of feature (I think a lot of kernel developer have needed this feature at a moment or an other). This gave me the will to get involved in the LLVM/CLang projects. (https://reviews.llvm.org/D44093).
This feature has been a real opportunity for me, because it has permitted me to learn a lot, both while implementing it and while being reviewed on Phabricator.
Then, I took a look at the LLVM subjects for this GSOC 2018, and I got really interested by this subject. Indeed I find really bad that people are not using LLVM tools for their projects, and I actually had the bad experience of someone telling me that "No one is using the clang toolchain for doing kernel development" while talking about the built-in I wanted to implement.
As I really think that the LLVM toolchain must be used, I am really interested in making those tools more GNU Binutils compliant so that it pushes people to use the toolchain.

What did I learn so far ?

General assessments

Although the tools are doing really different things, we can still draw a general picture of how those tools are working.

The option parsing

This step is actually the first step a tool need to do. To make it more unified and reliable, the tools are using the same “library” for their command line parsing (it will for example permit to “easily” add the feature of using shortened options like “**llvm-objdump -ldS**”).
To do so, those are using the “*/llvm/Support/CommandLine.h*” headers, that is defining this command line parsing library.
It basically work in two steps : the registration and the parsing.
So first, the tools will have to registered their options as static variables, so that we can user them everywhere else after this.
As you can see in the **Figure 1**, the way we can declare the options is very smart. Indeed, we are basically constructing the command line type templated over a “usable” type like **bool** or a **string list**. This is very handy to use because once the command line is parsed, we can just access the variables to the variables values without having to care about the command line parsing library.
The second thing is aliases. This is also very useful when we have short options, because we are able to alias the short option on the long one, so that we can only care about one option, even if this one can be setted by multiple ways.

```
static cl::opt<bool> StripDebug("strip-debug",
                     cl::desc("Removes all debug information"));

static cl::alias OnlyKeepA("j", cl::desc("Alias for only-keep"),
                     cl::aliasopt(OnlyKeep));

static cl::list<std::string> ToRemove("remove-section",
          cl::desc("Remove <section>"), cl::value_desc("section"));
```

Figure 1: Example of declaring options using the LLVM Command Line parser.

After declaring our variables, we can just call the parsing function in our main function (see **Fig. 2**). After this call, our static variables are setted correctly, and we can use them to execute correctly the requested options.

```
cl::ParseCommandLineOptions(argc, argv, "llvm objcopy utility\n");
```

Figure 2: Example of calling the ParsCommandLineOptions for objcopy.

The tools

During my researches though the different LLVM tools, I've learned much about how they are working. As you will see, some of those are really far from being compliant with the **GNU Binutils**, but other are needing a really few changes to be subtracted to its GNU equivalent in a real life toolchains.

llvm-objcopy

After having a talk with Jake Ehrlich, this "How llvm-objcopy works" part might change in the future, as this is actively discussed at the time I am writing those lines.

This is the binary that needs the more changes to be GNU compliant.
But first, let's expose quickly how it works. First, after parsing the options, a **Reader** is created. What it does is finding the type of the input file (for the moment, it only handles ELF), and create an object with the correct reader.
Then, the **Writer** is created (this one will of course write to the output file) by passing a reference on the object file (which is of type **Object**) that was previously generated by the reader. This is, in my opinion, the important thing to understand.
De facto, as the objcopy options are basically removing sections/symbols/etc.. from the input object, we we can directly do those operations on the **Object**, and then, as the writer already have a reference on it, we will only need to write its content to the output file.
Anyway, there is a lot of missing features/options compared to the GNU Binutils **objcopy**. To better present those, let's divide it into two parts : the ELF part, and the COFF/PE part.

The Elf part

First, for the ELF part, I basically noticed two problems. First one, **llvm-objcopy** is not able to copy the binary from ELF file of different formats (even if it is not supposed to run, this is actually working on GNU **objcopy**). However, even if this works, I am pretty sure this feature is almost never used.
Secondly, there is also some missing options. Here is a few of those (non exhaustive list):

- -L --localize-symbol - make a symbol local
- -W --weaken-symbol - make a symbol weak
- -K --keep-symbol - keep the symbol

- -M --merge-notes - remove redundant entries in note sections

Some short options are also missing, but this part might not be that complicated, as it only consists of adding aliases.
This is on this second part I will focus at first.

The COFF/PE part

After having a discussion with Jake Ehrlich on this subject, it appears that the existing code is too much “ELF design oriented”, and trying to make it more generically usable would be too difficult and painful. The idea exposed by Jake is to develop a whole new backend system for the COFF/PE binary files, so that we won’t have problem on this part.
Then, we would need to find a way to call the correct subset depending on which binary format we are processing. For this part, I was actually thinking about only detecting the binary format with the help of the LLVM **Binary** class, and then call the correct subset of functions/backend. Of course, this way of doing things wouldn’t be acceptable if we wanted to handle the cross format translations (ELF to COFF), but it appears that this task was decided not to be done because this feature was mainly not used by people.

The other tools

Then, I took a look at the **llvm-objdump** binary and I noticed that a lot of **objdump** features were missing. Indeed, some of those (like -archive-headers) are arch specifics, which is not the case for the GNU objdump. Same, some short options are not available, which can be problematic for inserting this tool in a real toolchain. Finally, some are just not present, like the “-R”, which displays the dynamic relocations.
About the **llvm-readelf** tool, which is actually a symlink to **llvm-readobj**, it appears that there is a few work to do, especially for “-R”, “-p”, “-x” and options that are shortcuts for multiple options.
There is also some other binaries that are not needed that much changes compared to those previous ones, like **llvm-strings**. For this one, it seems that the options are present but are not binded to the right keys as for the GNU **strings**.

Timeline

Mid March - Mid April
Doing some command lines comparisons of the different LLVM tools and their GNU Binutils equivalents.

Mid April - Mid May
Start working on the **llvm-objcopy** tool.
The ELF part will be done. I will focus on the command line missing options.

Mid May - Mid July
Working on the COFF/PE part of **llvm-objcopy**.
I think we can make the assumption that the actual small subset of options currently present in **llvm-objcopy** have to work. If it appears that the porting is faster than expected, it would still be really great to have all the generic options to work for COFF/PE.

Mid July - Early August
Working on the **llvm-objdump**

Early August - End of GSOC
Working on multiple binaries that (I think) need less rework (I will try to do as much as possible):

- **llvm-strings**
- **llvm-ar** has a few missing options
- **llvm-nm** (only -l option seems to be missing)
- Working on the **llvm-readelf**