

Extracting Information About Software Build Process and Source Code

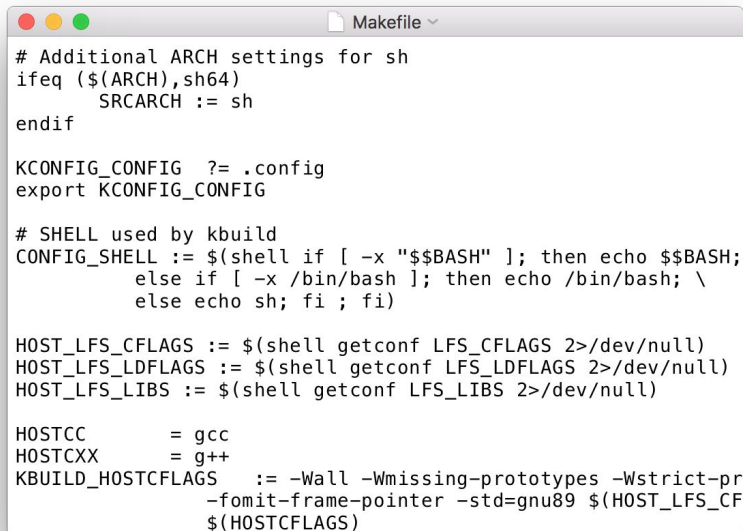
Ilya Shchepetkov, ISP RAS
shchepetkov@ispras.ru

CPAchecker & LDV Workshop 2018
26 September, Moscow

Motivation

Most software verification tools require preprocessed source code as their input, which can be obtained only knowing correct compilation options

Such options may be obtained by *intercepting build commands* of the target program

A screenshot of a terminal window titled "Makefile" showing the following content:

```
# Additional ARCH settings for sh
ifeq ($(ARCH),sh64)
    SRCARCH := sh
endif

KCONFIG_CONFIG ?= .config
export KCONFIG_CONFIG

# SHELL used by kbuild
CONFIG_SHELL := $(shell if [ -x "$$BASH" ]; then echo $$BASH;
    else if [ -x /bin/bash ]; then echo /bin/bash; \
    else echo sh; fi ; fi)

HOST_LFS_CFLAGS := $(shell getconf LFS_CFLAGS 2>/dev/null)
HOST_LFS_LDFLAGS := $(shell getconf LFS_LDFLAGS 2>/dev/null)
HOST_LFS_LIBS := $(shell getconf LFS_LIBS 2>/dev/null)

HOSTCC          = gcc
HOSTCXX         = g++
KBUILD_HOSTCFLAGS := -Wall -Wmissing-prototypes -Wstrict-pr
                  -fomit-frame-pointer -std=gnu89 $(HOST_LFS_CF
                  $(HOSTCFLAGS)
```

Motivation

Large programs are difficult to verify as a whole, so they are usually decomposed into separate fragments, each of which require an entry point. The decomposition algorithm, as well as the entry point generator, may require some additional information, like:

- Dependencies between source and object files;
- Some inner knowledge: function and macros definitions, declarations, functions calls, and so on

Previous Solutions

LDV Tools [2010 - 2015] intercepted build commands by directly modifying the main project Makefile

- Only the Linux kernel was supported;
- Only some command types were intercepted (gcc, ld, mv) - and probably some commands were missed;
- Very little information about the source code was gathered;
- There were compatibility issues

Previous Solutions

Klever - [up until now] - have used *wrappers* for the commands that needed to be intercepted:

- Wrappers were created for several tools (gcc, ld, mv);
- The build process was forced to execute a wrapper by modifying the PATH environment variable

The entry point generator additionally obtained information about the source code using CIF

Goals

To develop a new standalone tool that would be able to:

- Intercept *all* commands that are executed during the build process;
- Parse selected subset of these commands to identify input and output files, options;
- Be extendible to support parsing of additional commands;
- Connect parsed commands to each other;
- Collect information about the source code;
- All obtained information must be transferable between various computers;

Intercepting Build Commands

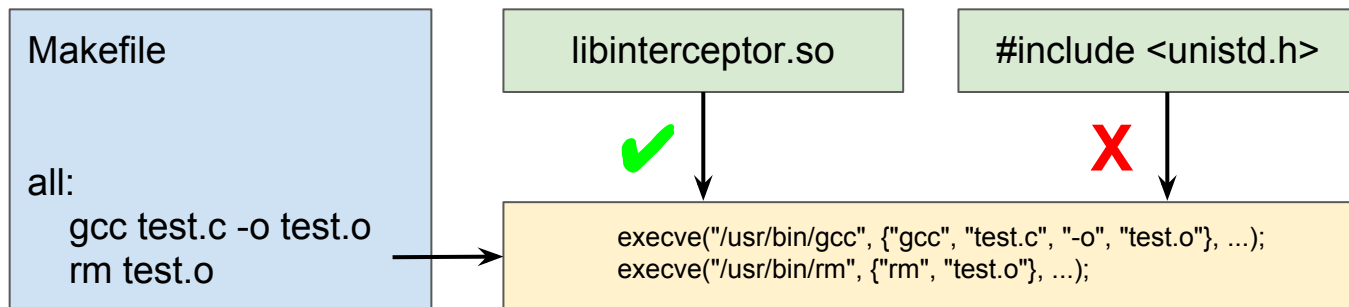
The approach is to intercept the *exec* calls issued by the build tool for each build command

To do this we have developed a shared library that redefine such *exec* functions: before creating a new process our *exec* functions store the information about the command into separate file

The library is then injected into the build process using **LD_PRELOAD** (Linux) and **DYLD_INSERT_LIBRARIES** (macOS) mechanisms provided by the dynamic linker

Intercepting Build Commands

```
/work/test$ LD_PRELOAD=libinterceptor.so make
```



Intercepting Build Commands

But: access control mechanisms on different operating systems might disable library injection:

- SELinux on Fedora, CentOS, RHEL;
- System Integrity Protection on macOS;
- Mandatory Integrity Control on Windows (disables similar mechanisms)

Intercepting Build Commands

We have implemented an additional "fallback" intercepting mechanism similar to one that was used in Klever before. It is also based on wrappers, but they are generated automatically for every executable program that can be found in PATH

- Works everywhere;
- Do not intercept some commands
 - ✓ `gcc test.c`
 - ✗ `/usr/bin/gcc test.c`

Build Commands Parsing

Next step is to parse intercepted commands to find their input and output values, options. It is easy to do for simple commands, and not so easy for commands like this one:

```
/Library/Developer/CommandLineTools/usr/bin/clang -cc1 -triple x86_64-apple-macosx10.13.0
-Wdeprecated-objc-isa-usage -Werror=deprecated-objc-isa-usage -Eonly -disable-free -disable-llvm-verifier
-discard-value-names -main-file-name server.c -mrelocation-model pic -pic-level 2 -mthread-model posix -mdisable-fp-elim
-fno-strict-return -masm-verbose -munwind-tables -target-cpu penryn -target-linker-version 351.8 -dwarf-column-info
-debug-info-kind=standalone -dwarf-version=4 -debugger-tuning=gdb -resource-dir
/Library/Developer/CommandLineTools/usr/lib/clang/9.1.0 -dependency-file - -w -MT server.deps.c -D REDIS_STATIC= -I
../deps/hiredis -I ../deps/linenoise -I ../deps/lua/src -O2 -Wall -W -Wno-missing-field-initializers -pedantic -std=c99
-fdebug-compilation-dir /Users/siddhartha/work/git/redis/src -ferror-limit 19 -fmessage-length 0 -stack-protector 1 -fblocks
-fobjc-runtime=macosx-10.13.0 -fencode-extended-block-signature -fmax-type-align=16 -fdiagnostics-show-option
-vectorize-loops -vectorize-slp -x c server.c
```

Build Commands Parsing

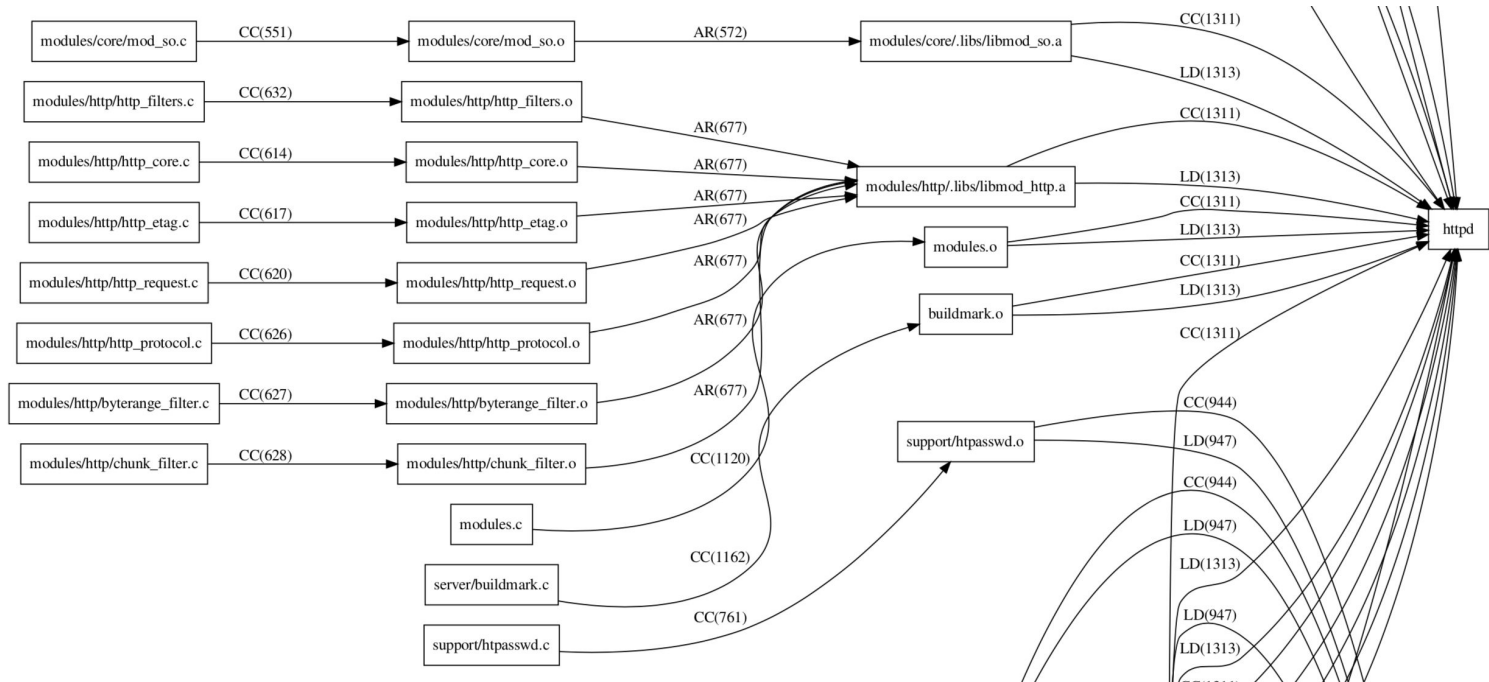
Our parsing algorithm is quite simple and is based on *options that require values* (-MT server.deps.c). We split command string into array of separate words:

- If a word starts with "-" - it is an option;
- If it is an option and it require values, that the next word is also an option;
- Value after "-o" option is an output file;
- Every word that left (is not an option or an output file) - is input file

There are some exceptions, but it works without much tinkering for all types of build commands that are interesting for us (cc, ld, mv, objcopy, ar, as)

Graph of commands

We can use their input and output values of all parsed commands to create graph of commands, which in turn can be used for program decomposition into fragments



Source code querying

Final step: obtaining information about the source code using CIF:

- **Functions:** definitions, declarations, calls, calls via a function pointer;
- **Macros and macro functions:** definitions, expansions;
- Limited information about initializations of global variables and typedefs

Using this information together with command graph we can create a pretty accurate call graph

Future work

- Consider Windows support
- Maybe replace CIF by Clang LibTooling for source code querying

Thanks!

Build Commands Parsing

For compilation commands we additionally do the following:

- Get dependencies of the input source file using pre-processor;
- Store input source file together with the dependencies for future reuse