Research Internship Report

Packaging applications on a pruned Linux Kernel

Paul Naert
X 2015

Tutor : Ashish Gehani

April 2 2018 - August 17 2018

1 Abstract

Sharing an application can often be a tricky process, due to the differences of environments between the source and the targets. When publishing research, one may need to distribute multiple libraries with the binary, out of which few functions may actually be used. To guarantee compatibility between different systems, one may even want to share an entire virtual machine.

However, these inclusions can result in very large files being shared for very few lines of code actually used. OCCAM is a tool developed at SRI International to trim binaries and their environment at build time to facilitate the sharing of application. Here, we focus mainly on packaging an entire virtual machine, with our goal being the specialization of the Linux kernel, pruning any unnecessary function and argument while maintaining full functionality for the application.

This report was written 7 weeks before the end of my internship, and my research is still ongoing. It covers the state of my work so far, which is far from a finished product.

Contents

2 Tools used

2.1 LLVM and clang

LLVM is a general compiler framework that makes a clear distinction between the three phases of compiling: the front-end, the optimization and the back-end. By introducing a specific, self-contained intermediate representation of the code (LLVM IR) common to all compiled languages and target architectures, LLVM allows most of the work (the optimizer) to be shared between different source languages and targets.

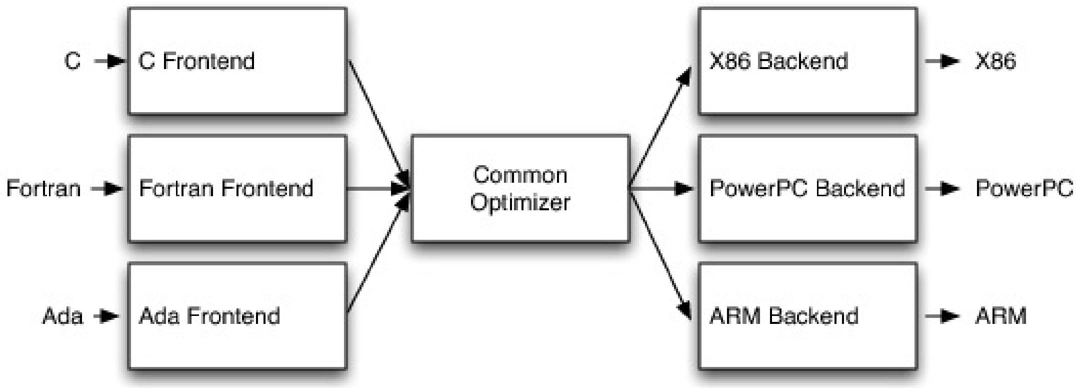


Figure 1: LLVM retargetable compiler structure [1]

When compiling code, LLVM offers the possibility to choose optimization passes from the current library, or to write new ones. Common passes include dead code elimination and constant propagation, for instance.

LLVM IR is human-readable, and as such quite expansive in terms of size. A smaller, compressed equivalent form is LLVM bitcode, and one can alternate between the two representations using `llvm-dis` and `llvm-as`.

Clang is the default C compiler of the LLVM project. It aspires to become faster than its main competitor `gcc`, while offering better static analysis tools and maintaining compatibility. Currently however, some `gcc` options are not supported by clang, and some will never be (see VLAIS page 8)

Because of its transparency and modular design, several companies are now switching to Clang from `gcc`. For instance, Clang has been the default compiler on OSX as of 2011 in XCode 4.2[2], and there is an ongoing effort to build Android user space and kernel using it[3].

For most of my work at SRI, I used LLVM version 5.0, which was the latest stable version in April 2018.

2.2 OCCAM

OCCAM[4] is an large scale project at the Computer Science Lab of SRI. Ultimately, the goal is to produce a tool called `razor` that would act as a code debloating optimization pass for the LLVM toolchain.

When including libraries into a program, or even when using a more general program with recurring arguments, many segments of code are packaged but not used in any way, inducing larger files and potential vulnerabilities to return oriented programming (ROP) attacks.

What OCCAM does is trying to specialize functions, mostly by propagating constants known at compile time and eliminating dead code. By specializing functions, i.e. getting rid of constant arguments, the tool allows a better optimization of register assignations.

Compared to simpler specialization algorithms, OCCAM often duplicates functions when it can specialize one of the branches, greatly expanding the possibilities for specialization at the risk of creating even more code than there was in the first place. As a result, the choice has been made not to specialize recursive functions in order not to have the size of the file explode.

2.3 gllvm

In order to do non conventional optimization passes such as OCCAM's `razor`, we need to have a direct access to the project's code in LLVM IR or bitcode. However, although Clang can compile a file to bitcode using the flag `--emit-llvm`, it does not offer the possibility to convert a large project into bitcode, mainly because building one object often requires having an object version of some previously compiled file.

gllvm[5] is an ongoing project at SRI of a go wrapper for clang that builds bitcode for each compiled file in parallel with the standard build process. It allows the user to retrieve the bitcode for a file by calling a simple function `get-bc` that will link all the bitcode files compiled for its predecessors.

During the course of my internship, I made several contributions to the tool, adding compiler options to the parsing function and allowing it to handle large bitcode files by splitting them into temporary files.

3 Pruning the Linux Kernel

3.1 Motivation

The goal of my internship at SRI was to explore the potential application of OCCAM's `razor` on the Linux kernel. The main goal of that pruning is to specialize the kernel to the application it is packed with. What we gain from doing so is some disk space and the elimination of ROP gadgets which may cause security flaws.

3.1.1 Applications

This process of trimming the kernel could be used for several applications, but my project has two main goals, corresponding to either running the kernel on bare metal or running it inside a virtual machine.

Running this kernel on bare metal means having a physical machine dedicated to one specific application. While that may be of little use for a personal computer, it makes a lot more sense when running programs on the cloud. This pruning could take place for specialized servers for instance. However, there are alternatives for this type of applications, most notably unikernels. Running a trimmed Linux kernel will probably be slower than a unikernel built specifically for this type of application.

The second option is to have our kernel run inside a virtual machine, and this offers very promising applications. An important one is executable sharing. When publishing research, one may want to publish an executable in order to have other people experiment with it. However, sharing the sources might be complicated either because of property concerns or because the build process is very long. What a tool like OCCAM could offer is a way to select which level of packaging would come with an executable. For instance, one may want to ship some dynamically linked libraries, or even a full virtual machine. Having a tool to reduce the size of the file to what is necessary could prove very handy.

3.2 The Linux Kernel

3.2.1 General Picture

The Linux kernel constitutes the core of all Linux distributions (Ubuntu, Fedora...). It is responsible for interfacing the *user space* with the hardware, itself operating in *kernel space*. Historically, it was created by Linus Torvalds in 1991 as a side project, but it quickly gained importance due to it being one of the first openly available kernels.

We can separate the kernel into three main components, from the more abstract to the hardware-specific. First we have the system call interface, which is the interface between *user space* and *kernel space*. It stands on top of architecture independent code , which encompasses most of Process Management, File System, Memory Management and Network Stack. Finally, we have the hardware specific code, which includes device drivers and architecture dependent code.

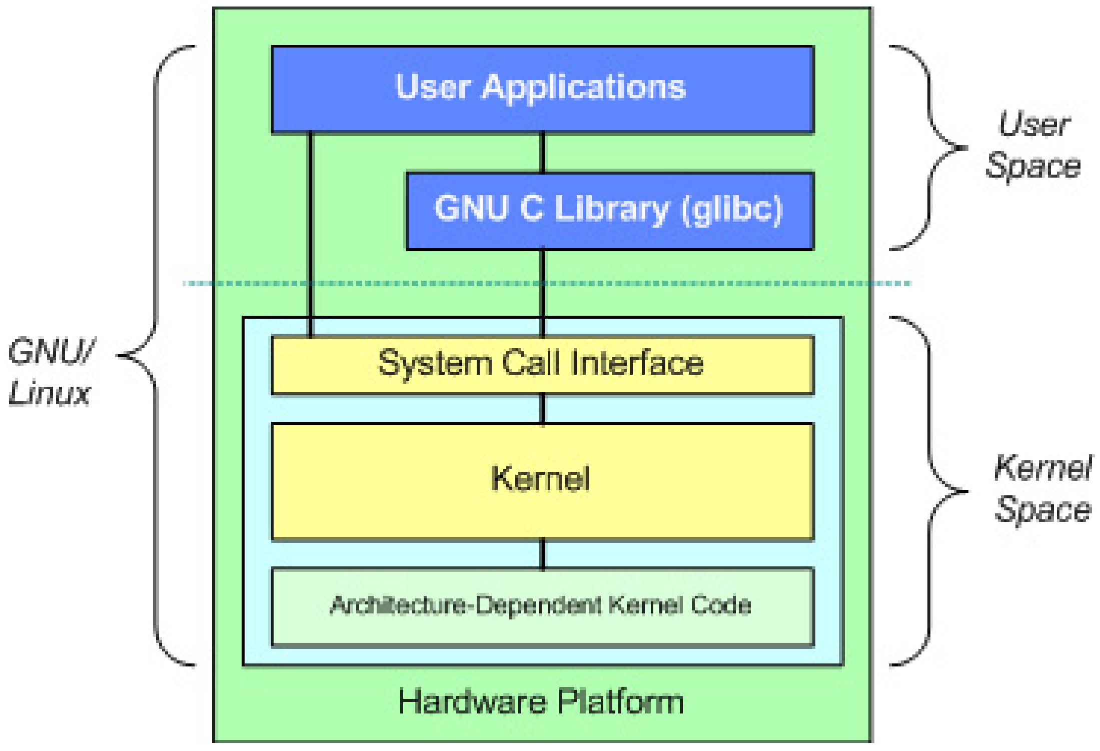


Figure 2: Linux kernel general architecture [6]

3.2.2 Kernel entry points

In order to prune the kernel, we determine what functions we cannot delete by following the call graphs of an initial set of functions that we know will be called from the outside of the kernel, either by the boot loader or the application. Determining what functions serve as the entry points into the kernel is thus essential to prevent the deletion of important code.

Most of this paragraph is adapted from Linux Inside[7], a Git based book written by 0xAX. The paths I refer to are either from the kernel build folder root or from `arch/x86/`.

Booting In order to boot, the Linux kernel relies on a boot loader, the most common being GRUB 2. The boot loader writes information into the kernel header and then transfers control to the kernel setup code, starting in `boot/header.S` at symbol `_start`. After some checks, control is transferred to the C code through the `main` function of the `boot/main.c` file. The setup will then continue, switching the processor from *Real mode* to *Protected mode* and then *Long mode*. After that, the kernel will finally be decompressed and the `start_kernel` function from `init/main.c` will be called to take control of the kernel initialization process.

Exceptions, interrupts and system calls Exceptions and interrupts are special signals given to the processor that trigger the interruption of the current task and the execution of a specific piece of code. Interrupts are asynchronous, being generally generated by hardware, while Exceptions are generated by the processor itself synchronously, when it encounters a fault in the code or when the code requests it. System calls are a specific type of exception, generally raised with the code 0x80 on Linux. Compared to regular code executed in *user space*, interrupts are handled in *kernel space* and thus can be used for hardware interaction, such as reading or writing files.

In the Linux kernel, interruptions are handled through a 256 entries long Interrupt Descriptor Table, on which we can find an extract in `asm/traps.h`:

```c
/* Interrupts/Exceptions */
enum {
   X86_TRAP_DE = 0, /* 0, Divide-by-zero */
   X86_TRAP_DB,     /*  1, Debug */
   X86_TRAP_NMI,    /*  2, Non-maskable Interrupt */
   X86_TRAP_BP,     /*  3, Breakpoint */
   X86_TRAP_OF,     /*  4, Overflow */
   X86_TRAP_BR,     /*  5, Bound Range Exceeded */
   X86_TRAP_UD,     /*  6, Invalid Opcode */
   X86_TRAP_NM,     /*  7, Device Not Available */
   X86_TRAP_DF,     /*  8, Double Fault */
   X86_TRAP_OLD_MF, /* 9, Coprocessor Segment Overrun */
   X86_TRAP_TS,     /* 10, Invalid TSS */
   X86_TRAP_NP,     /* 11, Segment Not Present */
   X86_TRAP_SS,     /* 12, Stack Segment Fault */
   X86_TRAP_GP,     /* 13, General Protection Fault */
    ...
   X86_TRAP_XF,     /* 19, SIMD Floating-Point Exception */
   X86_TRAP_IRET = 32, /* 32, IRET Exception */
};
```

As mentioned above, interrupt 0x80 calls the system call handler, located in `entry/entry_64.S`, which essentially does `call *sys_call_table(, %rax, 8)`. If we look into `entry/syscall_64.tbl`, we can see the whole table consisting of 322 entries. Here are the first 16:

```
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The abi is "common", "64" or "x32" for this file.
#
0  common   read        sys_read
1  common   write       sys_write
2  common   open        sys_open
3  common   close       sys_close
4  common   stat        sys_newstat
5  common   fstat       sys_newfstat
6  common   lstat       sys_newlstat
7  common   poll        sys_poll
8  common   lseek       sys_lseek
9  common   mmap        sys_mmap
10 common   mprotect    sys_mprotect
11 common   munmap      sys_munmap
12 common   brk         sys_brk
13 64 rt_sigaction      sys_rt_sigaction
14 common   rt_sigprocmask   sys_rt_sigprocmask
15 64 rt_sigreturn      sys_rt_sigreturn/ptregs
```

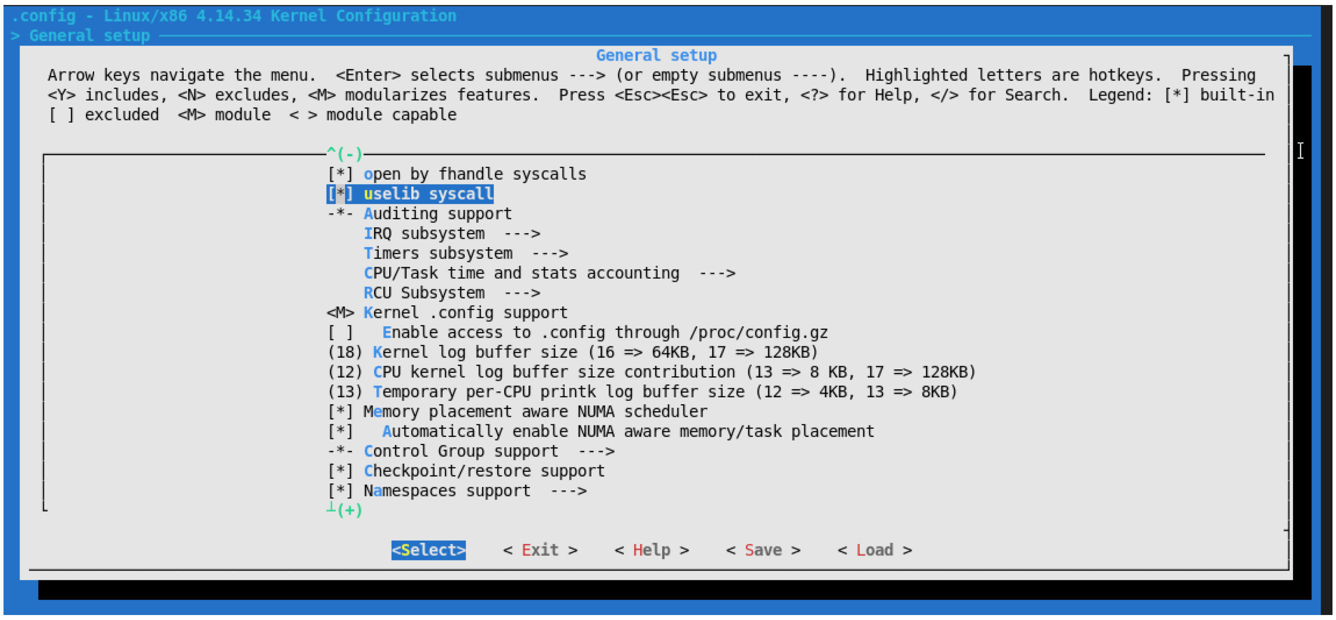
3.2.3 Build Process

The kernel is mostly written in C, some hardware specific code being written directly in assembly. It is designed to be compiled with `gcc`, although some efforts are made to make it compatible with clang[8]. Specific `gcc` features include variable length arrays in structures (VLAIS), a practice both forbidden by the C standard[9] and frowned upon by Linus Torvalds[10].

The configuration of a kernel build is made through a .config file. The Linux kernel offers several ways of editing it. Most notably, calling `localmodconfig` will fetch the configuration of the kernel running on the current machine, and calling `menuconfig` will allow its edition through a graphical user interface (see below). Most options can either be incorporated ([*]) or left out ([ ]), but some can be built as modules (<M>), which means they can be dynamically loaded and unloaded without rebooting the machine.

The kernel first builds all the necessary object files for the given configuration, and then links them into a binary called `vmlinux`. From there, it compresses `vmlinux` into `bzImage`, which is what is used to boot on.

During my internship, I worked mostly on version 4.14 of the kernel, a long-term support version released in November 2017. In June, version 4.17 was released, introducing assembly goto calls (asm-goto), which is not supported by clang yet[11]. Working on the most recent kernel with clang is now impossible. Choosing the right configuration to boot with clang is a lengthy process consisting mostly of trials and errors. Additionally to forbidden syntax such as VLAIS, some features including Virtualization will not prevent the kernel from building but will prevent the machine from booting.



```
.config - Linux/x86 4.14.34 Kernel Configuration
> General setup
                                         General setup
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus ----).  Highlighted letters are hotkeys.  Pressing
  <Y> includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] built-in
  [ ] excluded  <M> module  < > module capable

                              ^(-)
                              [*] open by fhandle syscalls
                              [*] uselib syscall
                              -*- Auditing support
                                  IRQ subsystem  --->
                                  Timers subsystem  --->
                                  CPU/Task time and stats accounting  --->
                                  RCU Subsystem  --->
                              <M> Kernel .config support
                              [ ]   Enable access to .config through /proc/config.gz
                              (18) Kernel log buffer size (16 => 64KB, 17 => 128KB)
                              (12) CPU kernel log buffer size contribution (13 => 8 KB, 17 => 128KB)
                              (13) Temporary per-CPU printk log buffer size (12 => 4KB, 13 => 8KB)
                              [*] Memory placement aware NUMA scheduler
                              [*]   Automatically enable NUMA aware memory/task placement
                              -*- Control Group support  --->
                              [*] Checkpoint/restore support
                              [*] Namespaces support  --->
                              ┴(+)

                   <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure 3: Linux kernel `menuconfig` window

3.3 bitcode extraction

In order to apply OCCAM's `razor` on the Linux kernel, we need to have a clean basis of bitcode on which we can experiment. The tricky part here, is not getting bitcode though, it is getting it to recompile into a working binary.

3.3.1 Manual Extraction

Once I was able to compile the kernel with Clang, compiling it with `gllvm`'s wrapper is transparent. From that we get bitcode file dotted all around the repertory. Using `get-bc vmlinux` creates a vmlinux.bc file, but trying to recompile that raises missing symbols errors. Going one step before in the build process, we can look in `link-vmlinux.sh` for what is linked into `vmlinux`:

```
# Link of vmlinux
# ${1} - optional extra .o files
# ${2} - output file
vmlinux_link()
{
   local lds="${objtree}/${KBUILD_LDS}"
   local objects

   objects="--whole-archive   \
      built-in.o            \
      --no-whole-archive      \
      --start-group         \
      ${KBUILD_VMLINUX_LIBS}   \
      --end-group             \
      ${1}"

   ${LD} ${LDFLAGS} ${LDFLAGS_vmlinux} -o ${2} \
        -T ${lds} ${objects}
}
# ld --build-id -T vmlinux.lds -o vmlinux --whole-archive built-in.o
    --no-whole-archive --start-group lib/lib.a arch/x86/lib/lib.a
    --end-group .tmp_kallsyms2.o
```

I have added the actual command called at the end, and we can see that most of the sources are in a file called `built-in.o`. It is actually a thin archive, referencing object files all over the source tree. After updating `gllvm` to handle thin archives, I tried building bitcode from `built-in.o`, but there were some files missing: all the files written in assembly, which makes sense as the compiler will not generate bitcode for them. By refining the extraction process I was finally able to compile a new `vmlinux` for `defconfig` and boot on it, excluding the folders `drivers/` and `fs/` from the back and forth compiling.

3.3.2 Creating an extraction tool

This process of picking which files to include and which not to for each configuration was extremely tedious, so after achieving the first successful build, I wrote a python script to do all the sh script writing for me. By analysing the `built-in.o` archive and the error logs of `gllvm`, this script writes a `build-script.sh` script which in turn copies all the necessary files and links them together. With that script, I was able to seemlessly change the configuration without having to rewrite my scripts.

Here is about half of the main function, the file is available on the gllvm Github repository [12].

```python
# This is where most of the script gets written.
# This function takes as argument:
#    a list of path that should not be translated into bitcode
#    the depth from the root (this is a recursive function)
#    the path from the root to the current folder (its length should be depth)
def write_script(excluded_paths, depth, base_dir):
    builtin = open (base_dir+"arbi","w")
    subprocess.call(["ar", "-t", base_dir+"built-in.o"],stdout=builtin)
    builtin.close()
    builtin = open(base_dir+"arbi","r")
    out.writelines("mkdir -p $build_home/built-ins/"+base_dir+" \n")
    # In directories we store the list of depth 1 directories (and files) found in builtin
    directories=[]
    for line in builtin.readlines():
        line=line.split('\n')[0]
        words=line.split('/')
        if words[0]=="arch":  # We had to make an exception for the arch folder which does not have a built-in.o at the root.
            words[0]=words[0]+'/'+words[1]
            if words[2] in archbi: #A second exception for arch, not all files in the folder are referenced in the arch/x86/built-in.o. We store the exceptions in the archi list
                words[0]=words[0]+'/'+words[2]
        if words[depth] not in directories and line not in standalone_objects: #Some objects are not listed in a built-in.o except at the root, so we include them separately in order not to mess the order of linking
            directories.append(words[depth])

    # folders in which there are excluded folders which we will need to act on recursively
    excluded_roots = [path[depth] for path in excluded_paths]

    for direc in directories:
        print base_dir+direc
        # the folder contains an excluded folder
        if direc in excluded_roots:
            # Check if we have an excluded folder
            if base_dir+direc in excluded_dirs:
                out.writelines("convert-thin-archive.sh "+base_dir+direc+"/built-in.o \n")
                out.writelines("cp "+ base_dir +direc+"/built-in.o.new $build_home/built-ins/"+base_dir+direc+"bi.o \n \n")
                link_args.writelines("built-ins/"+base_dir + direc +"bi.o ")
            # Else we filter the excluded_path list for only relevant stuff and we call the recursion on that folder
            else:
                relevant_excluded = [path for path in excluded_paths if (path[depth]==direc and len(path)>depth+1) ]
                if relevant_excluded:
                    write_script(relevant_excluded,depth+1,base_dir+direc+"/")
        else:
            #If the "directory" is a file, we copy it into the relevant objects folder and compile it
            if direc[-2:] == ".o":
                out.writelines("get-bc -b "+base_dir+direc+"\n")
                out.writelines("mkdir -p $build_home/built-ins/"+base_dir+"objects \n")
                out.writelines("cp "+ base_dir+direc+".bc $build_home/built-ins/"+base_dir+"objects \n")
                out.writelines("clang -c -no-integrated-as -mcmodel=kernel -o $build_home/built-ins/"+base_dir+ direc + " $build_home/built-ins/"+base_dir+"objects/" + direc+".bc \n \n")
                # We then add it to the linker arguments file
                link_args.writelines("built-ins/"+base_dir + direc +" ")
            # When dealing with a folder, we get the bitcode from the built-in.o file and check for errors in the log.
            # For each file that does not have a bitcode version (compiled straight from assembly) we copy it into the build folder directly and add it to the linker args
```

3.4 Pruning the bitcode

Once a stable basis of bitcode is established, we can start pruning it by eliminating dead code and specializing the input.

3.4.1 Extracting the kernel call graph

In order to get a good idea of the call graph architecture of the kernel that OCCAM will be working on, I had to extract the call graph for the entire kernel.

There are three options for extracting a function call graph. It can be done on the code, at compile time, or on the binary.

The first tool I used was the LLVM's `opt --dot-callgraph`. Given a bitcode file, this tool writes a Graphviz dot file which lists all the links between functions. Writing a python script to parse this file, I was able to find 40723 functions in the kernel, with 3056 of those called indirectly by the system call handler. However, the main issue with that approach is that I miss all the files for which I have no bitcode, and in particular the assembly files, very useful for the bootup phase.

What I tried next was using a `gcc` plugin called Kayrebt[13][14] designed specifically for the static analysis of the Linux kernel. Its Callgraphs module builds a SQLite database of all encountered functions and their calls. However, the tool is quite painful to install and only works with `gcc 4.8`, it needs a separate compiling from the regular `gclang` build and brings little to no benefit. Indeed, it does not track either calls made through inline assembly or in assembly files, as I was able to test using simple examples.

The final option is analyzing the binary after it is compiled. For this purpose I have started working with `angr`[15], a python library designed to manipulate binaries. On small examples, I get the correct call graph for inline assembly calls, so I am hoping it will perform well on the full kernel binary. For now however, given the size of the kernel binary, execution times are extremely long and I do not have a result for it yet.

3.4.2 Pruning the code while maintaining all the useful entry points

At the moment, I have not started pruning code. However, I think a good assumption would be to consider that except for a few C files called by assembly at bootup (`boot/main.c` for instance), most calls from assembly are to assembly, and no calls are made within C inline assembly. This is probably false, but hopefully, once I try pruning C files there will not be too much useful code left out, and I can manually check where the errors originate.

The first thing I will try to do is parse the bitcode for any system call or interrupt that may trigger access to the kernel. Then, I will feed all the calls to OCCAM, which should do the pruning. In order not to prune necessary functions, I think that I will set a "kernel" mode to it, which should whitelist all the necessary functions.

4 Delivering a packaged application

4.1 Packaging the application

As mentioned above, the general goal of my internship is to produce a tool to package one application with possibly no external dependency while keeping a minimal size. The two main approaches that I explored were unikernels and tiny virtual machines, which I detail below.

4.1.1 Unikernels

Unikernels are the output of library operating systems. This type of OS builds application-specific machines by integrating most of the OS code into libraries and linking those to the executable, adding a small runtime environment on top. This allows a deep specialization of kernel features, similar to what we would like to achieve with OCCAM.

They achieve this level of simplification by getting rid of the user space and context switching, meaning all processes run in the same address space. This generally makes unikernel much more proficient than usual multi user machines, with faster boot times and generally faster application execution[16].

One of such library OS is OSv[17], an operating system developed by Cloudius Systems for use on the cloud. I experimented on this one because it is able to run native Linux applications, provided they are compiled as position independent code and shared objects (`-shared -fPIC`). I was able to compile and run simple applications, for instance a terminal Battleship game[18], but I could not run more advanced programs, running a graphical user interface for example.

I think unikernels have a lot of potential for specialized use such as in clouds, but their reach is too restricted for the general use we plan to give to our application.

4.1.2 Tiny Virtual Machines

Another possibility was using tiny virtual machines, which are a lot more flexible than unikernels. This type of machine consists of a Linux kernel on top of which lies a small file system. Such a file system can be built through an *initramfs*. This consists of a file system compressed in a cpio archive that the system decompresses at boot. The OS will then execute the `init` file at the root. While this is generally used to build a preliminary file system before loading kernel modules and the main disk, it can be used as a final step if working with a small environment.

Combining this with BusyBox[19] makes all basic features of an OS available, for instance a bash console. I am currently beginning to experiment with it, so I have yet to try adding an X window system to use GUI programs with, but it should be feasible, as some existing OS such as TinyCore[20] provide this.

4.2 Web Browsers

Having an integrated web browser in an application can sometimes be useful, but the build processes for modern browsers tend to be quite complicated. Trying to build Firefox and Chromium into LLVM bitcode was a way for me to understand how to use the tools that I had at my disposal, in order to build the Linux kernel more efficiently while adding features to the final product. However, those builds turned out to be more complicated than the kernel build, and I did not succeed in building either of those into bitcode and then into binary.

4.2.1 Mozilla Firefox

Mozilla Firefox is one of the most popular web browsers for desktop and mobile use. Compared to most of its competitors, it is fully open-source, which made it the first choice for my experiments. It is programmed both in C++, JavaScript and in Rust, an LLVM based language bearing similarities with both C and ML. The design of Firefox is highly modular, each feature being separated from the others. As of 2017, Firefox officially supports Clang to build the c++ code.

Clang supports makes it easy to build Firefox from source with `gclang` (gllvm's wrapper for Clang), but the modular design and the multiple languages used make it very hard to actually extract the bitcode and recompile the executable using those. The first problem is that there is not, strictly speaking, one executable. The `firefox-bin` binary is very small and relies on libraries dotted all around the build folder and the source. I spent a large amount of time trying to determine what libraries were being used, but this is not made easy as a library may only be used for not-crucial features. For instance, moving the build folder out of the source folder will not stop Firefox from booting, but will prevent the user from using Google from the address bar, and will add small black rectangles around the tabs. Furthermore, experiments on this build were extremely slow considering it took my computer one full day to compile the source.

Considering all of this, I decided not to create a wrapper for the Rust compiler, and rather switch to another open-source browser a bit more friendly.

4.2.2 Chromium

Chromium is the open source version of Google Chrome, the most used browser worldwide, with about 60% of the market. It shares the vast majority of its code with its proprietary version, except from some DRM and patent bound modules. Chromium is coded almost entirely in C++, and clang is the default compiler since 2014. Compared to Mozilla Firefox, Chromium has the big advantage of being mostly integrated into a big binary, which made it easier to build bitcode for it.

Here I encountered an unexpected issue, where LLVM functions did not accept to read the bitcode file, raising a `LLVM Error : malformed block`. What I did to circumvent this was modify the code for gllvm's `get-bc` to have it split the bitcode into smaller files, and I was able to determine that the bug occurred when operating on files larger than 580 MB, with the bitcode for Chromium being around 1.3GB. I tried compiling small parts separately, but linking failed for reasons I was not able to determine.

Here is some code I wrote to split the final bitcode file, from `gllvm`'s `extractor.go`[5].

```go
func linkBitcodeFilesIncrementally(ea extractionArgs, filesToLink []string, argMax int, linkArgs []string) {
  var tmpFileList []string
  // Create tmp dir
  tmpDirName, err := ioutil.TempDir(".", "glinking")
  if err != nil {
    LogFatal("The temporary directory in which to put temporary linking files could not be created.")
  }
  if !ea.KeepTemp { // delete temporary folder after used unless told otherwise
    LogInfo("Temporary folder will be deleted")
    defer CheckDefer(func() error { return os.RemoveAll(tmpDirName) })
  } else {
    LogInfo("Keeping the temporary folder")
  }

  tmpFile, err := ioutil.TempFile(tmpDirName, "tmp")
  if err != nil {
    LogFatal("The temporary linking file could not be created.")
  }
  tmpFileList = append(tmpFileList, tmpFile.Name())
  linkArgs = append(linkArgs, "-o", tmpFile.Name())

  LogInfo("llvm-link argument size : %d", getsize(filesToLink))
  for _, file := range filesToLink {
    linkArgs = append(linkArgs, file)
    if getsize(linkArgs) > argMax {
      LogInfo("Linking command size exceeding system capacity : splitting the command")
      var success bool
      success, err = execCmd(ea.LinkerName, linkArgs, "")
      if !success || err != nil {
        LogFatal("There was an error linking input files into %s because %v, on file %s.\n", ea.OutputFile, err, file)
      }
      linkArgs = nil

      if ea.Verbose {
        linkArgs = append(linkArgs, "-v")
      }
```

```go
            tmpFile, err = ioutil.TempFile(tmpDirName, "tmp")
            if err != nil {
                LogFatal("Could not generate a temp file in %s because %v.\n", tmpDirName, err)
            }
            tmpFileList = append(tmpFileList, tmpFile.Name())
            linkArgs = append(linkArgs, "-o", tmpFile.Name())
        }

    }
    success, err := execCmd(ea.LinkerName, linkArgs, "")
    if !success {
        LogFatal("There was an error linking input files into %s because %v.\n", tmpFile.Name(), err)
    }
    linkArgs = nil
    if ea.Verbose {
        linkArgs = append(linkArgs, "-v")
    }
    linkArgs = append(linkArgs, tmpFileList...)

    linkArgs = append(linkArgs, "-o", ea.OutputFile)

    success, err = execCmd(ea.LinkerName, linkArgs, "")
    if !success {
        LogFatal("There was an error linking input files into %s because %v.\n", ea.OutputFile, err)
    }
    LogInfo("Bitcode file extracted to: %s, from files %v \n", ea.OutputFile, tmpFileList)
}
```

5 Conclusion

As I experienced firsthand when trying to install different tools during my internship, sharing applications is often a complicated process when the program relies on specific version of libraries or executables. I hope that a tool like OCCAM will be able to make it easier by including parts of the execution path with the binary, reducing the compatibility burden. With my work, I hope to be able to extend the tool to the very root of the execution stack, the kernel, without adding too much overhead in terms of size or vulnerability. This is done by pruning code in the Linux kernel that the application does not need.

Regarding my internship, I think that working on the Linux kernel is not as engaging as I though it would be, for several reasons. First, the size of the project makes compile times very long, lasting between half an hour and one day. This creates an irregular work schedule, where I constantly have to work in parallel on different problems on several virtual machines, and I can never really dive into one problem. Second is the overall complexity of the kernel. When I was building the kernel, I got a lot more familiar with the build process than I did with the kernel itself, and I did not really dive below the surface before having to analyze the entry points of the kernel. With the kernel being such a huge project, I could not fix any problem that I encountered, but instead I had to build around them. This is especially true when working on the kernel with LLVM, which is still a work in progress tool, with many bugs. I rarely felt like I was building something clean. I hope that the tool I build by the end of my internship will be clean enough to be usable.

References

[1] Introduction to llvm, . URL `http://www.aosabook.org/en/llvm.html`. 1

[2] Clang website. URL `https://clang.llvm.org`. 2.1

[3] Greg Hackmann Nick Desaulniers and Stephen Hines. Compiling android userspace and linux kernel with llvm. URL `http://llvm.org/devmtg/2017-10/slides/Hines-CompilingAndroidKeynote.pdf`. 2.1

[4] Ashish Gehani Gregory Malecha and Natarajan Shankar. Automated software winnowing. URL `http://www.csl.sri.com/users/gehani/papers/SAC-2015.Winnow.pdf`. 2.2

[5] gllvm github repository. URL `https://github.com/SRI-CSL/gllvm`. 2.3, 4.2.2

[6] Kernel architecture presentationkernel, . URL `https://www.ibm.com/developerworks/library/l-linux-kernel/index.html`. 2

[7] 0xAX. *Linux Inside.* in progress. URL `https://legacy.gitbook.com/book/0xax/linux-insides/details`. 3.2.2

[8] The llvmlinux project, . URL `https://wiki.linuxfoundation.org/llvmlinux`. 3.2.3

[9] The c11 standard - *structures are defined on pages 129-130.* URL `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1548.pdf`. 3.2.3

[10] Linus torvalds on vlais. URL `https://lkml.org/lkml/2013/9/23/500`. 3.2.3

[11] Discussion on asm-goto support for clang. URL `https://bugs.llvm.org/show_bug.cgi?id=9295`. 3.2.3

[12] gllvm example: the linux kernel, . URL `https://github.com/SRI-CSL/gllvm/blob/master/examples/linux-kernel/`. 3.3.2

[13] Laurent Georget Frédéric Tronel Valérie Viet Triem Tong. Kayrebt: An activity diagram extraction and visualization toolset designed for the linux codebase. *2015 IEEE 3rd Working Conference on Software Visualization.* URL `http://kayrebt.gforge.inria.fr/assets/pdf/Georget_et_al_2015_Kayrebt.pdf`. 3.4.1

[14] The kayrebt toolset. URL `http://kayrebt.gforge.inria.fr/index.html`. 3.4.1

[15] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War:

Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016. 3.4.1

[16] Benchmarks for the osv library os. URL `http://osv.io/benchmarks/`. 4.1.1

[17] Glauber Costa Pekka Enberg Nadav Har'El Don Marti and Vlad Zolotarov Avi Kivity, Dor Laor. Osv—optimizing the operating system for virtual machines. *2014 USENIX Annual Technical Conference*. URL `https://www.usenix.org/system/files/conference/atc14/atc14-paper-kivity.pdf`. 4.1.1

[18] Battleship game repository. URL `https://github.com/SergiyShvets/BattleShip/`. 4.1.1

[19] Busybox: The swiss army knife of embedded linux. URL `https://busybox.net/about.html`. 4.1.2

[20] Lauri Kasanen. *Into the Core*. URL `https://distro.ibiblio.org/tinycorelinux/corebook.pdf`. 4.1.2