Analyzing the CMake Build System

KimHao Nguyen
University of Nebraska-Lincoln, USA

ThanhVu Nguyen
George Mason University, USA

Quoc-Sang Phan
Facebook, USA

ABSTRACT

CMake is one of the most widely used build automation tools in the industry. Facebook engineers often rely on examining large and complex CMake build files for various program analyses tasks. In this paper, we report on some of the unique challenges when analyzing CMake files at Facebook.

KEYWORDS

build system, cmake, build conditions, symbolic execution

ACM Reference Format:
KimHao Nguyen, ThanhVu Nguyen, and Quoc-Sang Phan. 2022. Analyzing the CMake Build System. In *ICSE '22: International Conference on Software Engineering, May 21–29, 2022, Pittsburgh, PA.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3510457.3513064

1 INTRODUCTION

CMake is a well-known build automation tool used in many large and influential projects such as MySQL, Boost, Webkit, KDE, and Android Studio. CMake itself is not a build system such as Make, but instead works by generating the appropriate build scripts for various build systems in different platforms. For example, it generates GNU Makefiles for Linux and Visual Studio project files for Windows.

Similar to many organizations and companies such as INRIA and Netflix, Facebook relies on CMake to build many of its internal and public projects. While Facebook is actively developing Buck, another build system with more advanced features and Python-like language, Facebook engineers still work directly with CMake to build projects, especially those from companies acquired by Facebook (e.g., WhatsApp) as they do not use Buck and rely on the more standard CMake automation tool.

Other than the obvious use of building software, Facebook engineers have a couple of uses that require proper CMake analysis.

Use 1: Converting CMake to Buck. Buck is the default system to build, run, and test programs at Facebook. Thus, most internal tools are built to support and extract information from Buck files, often named `BUCK`. For example, to fuzz a function `foo`, the fuzzing infrastructure consumes a harness of `foo` and a `BUCK` file to compile the harness. Bottom-up data flow analysis tools extract target dependency in `BUCK` files to build their call graphs. Thus, if a team wants to run these internal tools on a CMake project, the first step is converting `CMakeLists.txt` files into `BUCK` files.

Currently, there are no existing tools to convert CMake files to Buck and thus such conversions are painfully done and maintained by hand. Converting a 500 LoC, average-sized `CMakeLists.txt` file to `BUCK` is often a one-week project for an engineer highly experienced with both CMake and Buck (and the initial conversion will likely contain bugs).

Use 2: Extracting Build Conditions. The flexibility of CMake language allows a file, a directory, or a whole library to be added or removed at CMake runtime depending on the configuration. Here are some scenarios where we need to find the configuration that a specific file is compiled during the build.

- *Crash dump analysis.* When WhatsApp crashes on a user's device, the security team analyze the stack trace to determine, for example, if the crash is exploitable or an adversary was actively fuzzing the app. When such a stack trace starts from the function `foo` in the file `bar.cc`, the first step to reproduce the crash is to figure out the set of compilation flags that builds `bar.cc`.
- *Static analysis.* Static analyzers are run automatically when engineers submit patches for code review. Suppose the patch consists a change in `bar.cc`, the static analyzer needs need full information how to parse `bar.cc` into an abstract syntax tree. Hence, it needs to find a build configuration that includes `bar.cc`. This is not possible at Facebook at the moment, and many files have been missed by the tool, leading to potential false negatives.

While there is some recent work on analyzing certain forms of Makefiles to extract build information, there is no work on analyzing CMake scripts, which are different from Makefiles in both syntax and semantics[1].

In addition to the unique uses of CMake at Facebook, the ability to analyze CMake and especially its build conditions has many advantages. Build conditions can help developers find orphan code sections, files, or compilation options that are never used and determine what patches or code changes affect a compilation configuration. They also allow engineers to estimate the project's build time and size. Moreover, these conditions can reveal interesting properties, e.g., the complexity of the build conditions and "influential" compilation or linkage options affecting how files are built.

Analyzing CMake. We are developing a static analysis for CMake that targets the above uses. The analysis consists of two main phases: lexing/parsing the CMake code into AST and recursively visiting and transforming nodes in the AST into equivalent Buck scripts or into logical formulae representing build conditions. Thus, we aim to use a static analysis to solve both problems of converting to Buck and collecting build conditions.

In our experience, we find that parsing CMake is relatively straightforward as we can reuse the CMake parser or an existing

[1]We could analyze Makefiles generated from CMake, but we would miss many build conditions only presented in the original CMake files and discarded in the Makefiles.

ICSE '22, May 21–29, 2022, Pittsburgh, PA

ACM ISBN 978-1-4503-9226-6/22/05…$15.00
https://doi.org/10.1145/3510457.3513064

CMake parser. However, we have encountered several challenges in analyzing and transforming CMake. Below we describe these challenges and hope that they will inspire new program techniques for the software research community. The code snippets below are adapted from various CMake files used in the FOLLY open-source library from Facebook.

2 CHALLENGES IN ANALYZING CMAKE

The CMake language is Turing-complete and can theoretically perform any computation. In contrast, Buck (more specifically the Skylark language) is not Turing-complete and intentionally designed to be not as powerful. Similarly, to facilitate automate reasoning, the fragment of logical formula we use to capture build conditions is also less powerful (e.g., quantifier free, no uninterpreted functions).

While theoretically powerful, CMake is not intended for general programming and in practice, we can replace common uses in CMake with Buck (as Facebook engineers have done manually) or use logical formulae to capture CMake's build conditions. Nonetheless, to automate CMake analysis and transformation, we need to overcome several challenges, some of which also appear in Makefiles and other shell scripting languages. While there are existing works on analyzing very restricted subsets of Makefiles such as Linux KBuild Makefiles (e.g., [1–3]), these do not apply to general Makefiles and especially CMake.

Scoping. CMake has different scoping rules compared to most languages. The assertion in the code on the right would hold in languages such as Python and Buck because `bar` cannot modify variables declared in the parent scope. However, in CMake, `bar` has a copy of *a* in its scope and can opt to modify *a* in its parent scope and violate the assertion. Thus, analyzing CMake is difficult as we need to determine and track parent variables that are exposed and modified by the callee. The task is further complicated due to indirect changes (e.g., `bar` modifies *a* through a sequence of variable expansions).

```
def foo():
 a = True
 bar()
 assert(a)
```

Variable Expansion. Similar to a scripting language such as Make, variable expansions are used frequently in CMake. However, due to CMake's treatment of lists being semi-colon separated lists (e.g., the list "a; b; c" has three elements), its expansion can be tricky to analyze. For example, the code below demonstrates how expansions in CMake can produce potentially surprising and confusing results.

```
set(A x y) # A has 2 elements ["x","y"]
set(B z k) # B has 2 elements ["z","k"]
set(C ${A} ${B}) # C has 4 elements ["x","y","z","k"]
set(D "${A} ${B}") # D has 3 elements ["x","y z","k"]
set(E ${A}${B}) # E has 3 elements ["x","yz","k"]
foo(x ${A} "${A}") # foo has 4 arguments: ["x","x","y","x;y"]
```

Moreover, variable expansions are sometimes nested, making it difficult to reason about data flow, e.g., to determine the destination of `set()` or how variables are used as command arguments. The code below demonstrates how nesting can make it difficult in determining the value of *X* through nesting.

```
set(FOO_AND_BAR X)
set(A INNER)
set(VAR_INNER AND)
set(${FOO_${VAR_${A}}_BAR} 7)
message(${X}) # X is 7
```

Reasoning over Strings. CMake uses string to represent all kinds of values (e.g., numbers, bools, lists). In contrast, Buck and logical formulae have richer types. Thus, transforming CMake to Buck or logical formulae would require either emulating the string handling mechanism in Buck or inferring proper variable types.

```
set(FLAGS -O2 -m -s) # FLAGS is a string "-O2;-m;-s"
if("-O" IN_LIST FLAGS) # FLAGS treated as list
 # False
endif()
if(FLAGS MATCHES "-O") # FLAGS treated as string
 # True
endif()
```

Moreover, depending on how strings are used, CMake can interpret them differently (e.g., hybrid of strings and lists) as demonstrated in the code above. Variable expansion is also used in conditions (in fact, it can be used anywhere). CMake uses another mini-language, independent of the CMake grammar to parse conditional expressions. This makes analyzing and collecting build conditions non-trivial as we might have to do multiple passing to fully instantiate the correct conditions. In general, while these reasonings can be achieved with careful analysis, the transformation can be difficult due to indirect complex variable expansions and modifications.

Symbolic Inputs. Static analysis often represents input variables symbolically to reason about all possible input values. However, representing a CMake variable symbolically is challenging. Using a single string like CMake does internally is precise, but would be expensive for current constraint solvers (e.g., split string by semicolon character to test if an element is in a list or not). Using an array or list of strings would be less expensive, but we have to handle various forms of string expansions and manipulations used in CMake. More importantly, these arrays would be unbounded or have large bounds, which makes automated reasoning inefficient. We note that recent existing static and symbolic techniques for KBuild Makefiles only work for input options that have three values (here we are dealing with arbitrary strings).

Side-Effects. CMake allows side-effects and typical CMake usage often invokes arbitrary shell commands with unpredictable results (e.g., using complex regex's, compiling programs, deleting or modifying files). These behaviors are in general discouraged in Buck as it aims to have predictable behaviors for reproducible builds and accurate artifact caching. Moreover, while many of these commands can be abstracted or modeled, providing logical models or mocks for system commands often invoked in CMake is notoriously manual and difficult (as shown in the Klee symbolic execution tools).

ACKNOWLEDGMENT

This material is based in part upon work supported by the National Science Foundation under grant numbers 1948536, 2107035 and a gift from Facebook.

REFERENCES

[1] Paul Gazzillo. 2017. Kmax: Finding all configurations of kbuild makefiles statically. In *Foundations of Software Engineering*. 279–290.
[2] ThanhVu Nguyen and KimHao Nguyen. 2020. Using Symbolic Execution to Analyze Linux KBuild Makefiles. In *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 712–716.
[3] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2014. Static analysis of variability in system software: The 90,000# ifdefs issue. In *USENIX Annual Technical Conference*. 421–432.