The Devil Is in the Command Line: Associating the Compiler Flags With the Binary and Build Metadata

Gunnar Kudrjavets
University of Groningen
Groningen, Netherlands
g.kudrjavets@rug.nl

Aditya Kumar
Google
Mountain View, CA, USA
appujee@google.com

Jeff Thomas
Meta Platforms, Inc.
Menlo Park, CA, USA
jeffdthomas@meta.com

Ayushi Rastogi
University of Groningen
Groningen, Netherlands
a.rastogi@rug.nl

ABSTRACT

Engineers build large software systems for multiple architectures, operating systems, and configurations. A set of inconsistent or missing compiler flags generates code that catastrophically impacts the system's behavior. In the authors' industry experience, defects caused by an undesired combination of compiler flags are common in nontrivial software projects. We are unaware of any build and CI/CD systems that track how the compiler produces a specific binary in a structured manner. We postulate that a queryable database of how the compiler compiled and linked the software system will help to detect defects earlier and reduce the debugging time.

CCS CONCEPTS

• **Software and its engineering** → *Software defect analysis*; *Software design trade-offs*; *Empirical software validation.*

KEYWORDS

Defect prevention, compiler flag, Clang, GCC, MSVC

ACM Reference Format:
Gunnar Kudrjavets, Aditya Kumar, Jeff Thomas, and Ayushi Rastogi. 2023. The Devil Is in the Command Line: Associating the Compiler Flags With the Binary and Build Metadata. In *Proceedings of 46th International Conference on Software Engineering (ICSE 2024).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION AND BACKGROUND

Compilers are software systems that translate programs "into a form in which it can be executed by a computer" [1]. A C or C++ compiler such as Clang, GCC, or MSVC supports hundreds of command-line arguments (flags, options, switches). The *compiler flags* instruct compiler on different aspects of code generation, types of error detection, compliance to a specific version of the programming language standard, or target platform-specific nuances. *An incorrect combination of compiler flags can have disastrous consequences for the resulting software system.* For example, accidentally turning off the compiler flag to enable checks for buffer security to catch stack overflows (e.g., omitting the `/GS` option in MSVC) can expose a zero-day vulnerability [11, 12].

Engineers can compile the same version of a software system to target different platforms and intents, such as debugging, profiling, or an official release. Typical implicit variables that influence the final set of compiler flags are the host operating system where the compiler executes, the target operating system where the code will run, the compiler version, dependencies available on the host system, and the desired build type [15, 21].

Both commercial and open-source software utilizes a variety of *build systems.* The build system determines the conditions under which a compiler runs and what combination of compiler flags it passes to the compiler. Some of the most popular build systems are: Ant [22], Bazel [7], Buck2 [17], CMake [13], GNU Make [6], Ninja [16], and NMAKE [3]. Each build system has different means of specifying the dependency graph, defining the rules for build actions, and how it initializes the default set of compiler flags.

Listing 1 displays a simple conditional statement that modifies the set of dependent libraries based on the host operating system.[1]

Listing 1: Excerpt from the Redis Makefile.

```
# Linux ARM32 needs -latomic at linking time
ifneq (,$(findstring armv,$(uname_M)))
        FINAL_LIBS+=-latomic
endif
```

In Listing 2, we see a more complex conditional logic.[2] The build system disables the usage of a critical dependency (the jemalloc memory allocator) based on what the host and target platforms are.

Listing 2: Excerpt from the RocksDB Makefile.

```
ifeq ($(PLATFORM), OS_MACOSX)
ifeq ($(ARCHFLAG), -arch arm64)
ifneq ($(MACHINE), arm64)
  ...
  DISABLE_JEMALLOC=1
  PLATFORM_CCFLAGS := $(filter-out -march=native, ...)
  PLATFORM_CXXFLAGS := $(filter-out -march=native, ...)
endif
endif
endif
```

[1] https://github.com/redis/redis/blob/unstable/src/Makefile
[2] https://github.com/facebook/rocksdb/blob/main/Makefile

ICSE 2024, April 2024, Lisbon, Portugal

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

By design, a modern build system lets users define build instructions at a higher abstraction level than the formal compiler command lines. In Listing 3, we can see how to define a C++ executable `foo` in Buck2 [17]. The executable `foo` contains one source code file and one header file. It also depends on another C or C++ library called `bar`.

Listing 3: Sample Buck2 build definition for a C++ binary.

```
cxx_binary(
  name = 'foo',
  srcs = [ 'foo.cxx', ],
  headers = [ 'foo.hxx', ],
  deps = [ ':bar', ],
)
```

For an engineer to understand the details about how *exactly* the compiler generates code, they need to either intercept the compiler execution or inspect the resulting log files with the final command line that the compiler interprets.

2 INDUSTRY CHALLENGES

Our primary motivation for this paper comes from observing, debugging, and fixing the repeating patterns of defects. The root cause for these defects are the incorrect assumptions about how a compiler generated binaries for a particular software system. A possible negative interaction between compiler flags is a known problem [19]. In our industry experience, *the defects caused by incorrect (extraneous, missing, unsuitable) compiler flags have high consequences, are hard to detect and stealthy, and are time-consuming to investigate and replicate.*

The primary categories of problems that we have encountered during the last two decades in the industry are as follows:

(1) **The differences between engineers' development environment and the official build servers**. The official build servers can run on a different operating system, use a different compiler version, have different environment variables set, or use a different set of preprocessor directives.
(2) **A lack of formal alerting mechanism when the compiler flags change**. Any seemingly unrelated commit can influence how the compiler generates the code or what are the application's dependencies. The resulting compiler flags can change because of confounding variables, such as environment settings, compiler configuration files, or corporate device management policies.
(3) **A lack of tools to detect anomalies in the resulting build**. A compiler can generate a subset of a software system differently than the rest. Sample issues include (a) a release build that targets the production environment includes a component with debug tracing enabled, (b) global optimization options do not propagate correctly to all the dependencies, and (c) a consumer that uses different versions of the same dependency in parallel (some things are worse than the infamous "DLL hell" [4]).
(4) **Inability to easily detect syntactic mistakes**. An engineer may mistype a `CXFLAGS` macro in the makefile instead of `CXXFLAGS` (extra flags passed to a C++ compiler). Most build systems lack the means to detect and notify engineers of these mistakes.
(5) **Third-party software components rarely provide the build configuration used to generate the binaries**. For example, a dependency can turn off the support for exception handling (e.g., specifying the `-fno-exceptions` flag in GCC). If the consumer assumes that it can catch exceptions, it invalidates the application's ability to handle errors.

2.1 Non-deterministic builds

Popular C++ compilers can generate code in a non-deterministic manner. Recompiling a translation unit using the same build configuration can result in a different binary [9, 10]. Similarly, modern build systems process the build graph efficiently by enforcing directed acyclic graph-like build dependencies [7, 17]. As a result, the order of object files listed during the linking stage will depend on which translation unit the compiler built first. That, in turn, can cause non-deterministic behavior [5].

Tracking the usage of compiler flags will decrease a subset of problems related to the reproducibility of the build environment [14]. It will *help with debugging and early detection of complex defects that influence the behavior of an entire software system.*

3 INDUSTRY NEEDS

While the "[b]uild systems are awesome, terrifying—and unloved," they are something that each engineer uses daily [18]. Modern build tools such as Bazel [7] and Buck2 [17], have made significant progress towards the build hermeticity [8] and making builds reproducible. However, developing a fully self-contained and deterministic build system has been a complex problem, even for a company like Microsoft [15, 20, 21]. Popular utilities such as GNU Autotools are not hermetic by design and rely on the dependencies from the current execution environment [2].

Engineers need to have the means to understand the evolution of the final set of compiler flags for each binary throughout the project's history for each configuration. Currently, the "state-of-the-art solution" involves engineers inspecting the build logs as a text file and using tools such as `diff` to compare the results between two builds. If the build logs are not systematically archived, engineers must rebuild the entire system to understand the final set of compiler flags. Building multiple product versions to isolate a problem can take days or longer for complex software systems, such as an operating system.

4 POTENTIAL RESEARCH DIRECTIONS

An obvious solution is to parse the build logs, extract the necessary information, store it in the metadata associated with each resulting build, and provide a query interface for engineers to solve problems similar to the ones we enumerate in Section 2. Most industrial CI/CD systems enable associating an individual build with the test case results, the state of the source code repository when the compiler generated the binaries, and other related metadata. Adding the information about compiler flags is just another dimension of the metadata.

Another option is storing the compiler flags the build system uses inside each binary. For example, an ELF file format supports the `.comment` and `.note` sections in the final binary [23]. However, that approach will require making changes to each used compiler.

REFERENCES

[1] Alfred V. Aho, Jeffrey D. Ullman, Ravi Sethi, and Monica S. Lam. 2007. *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison Wesley, Boston, MA, USA.

[2] John Calcote. 2020. *Autotools: A Practitioner's Guide to GNU Autoconf, Automake, and Libtool* (2nd ed.). No Starch Press, San Francisco, CA, USA.

[3] Microsoft Corporation. 2021. *NMAKE Reference.* Retrieved October 5, 2023 from https://docs.microsoft.com/en-us/cpp/build/reference/nmake-reference

[4] Stephanie Dick and Daniel Volmar. 2018. DLL Hell: Software Dependencies, Failure, and the Maintenance of Microsoft Windows. *IEEE Annals of the History of Computing* 40, 4 (Oct. 2018), 28–51. https://doi.org/10.1109/MAHC.2018.2877913

[5] Free Software Foundation, Inc. 2009. *Ld(1): GNU linker—Linux man page.* Retrieved October 5, 2023 from https://linux.die.net/man/1/ld

[6] Free Software Foundation, Inc. 2023. *Make—GNU Project—Free Software Foundation.* Retrieved October 5, 2023 from https://www.gnu.org/software/make/

[7] Google. 2023. *Bazel—A Fast, Scalable, Multi-Language and Extensible Build System.* Retrieved October 5, 2023 from https://bazel.build/

[8] Google. 2023. *Hermeticity.* Retrieved October 5, 2023 from https://bazel.build/basics/hermeticity

[9] Mandeep Singh Grang. 2017. Non-determinism in LLVM Code Generation. Talk presented at the 11th meeting of LLVM developers and users. San Jose, CA, USA. Retrieved October 5, 2023 from https://github.com/mgrang/non-determinism/blob/master/poster__nondeterminism_in_llvm_code_generation__llvmdevmeet_2017.pdf

[10] Mandeep Singh Grang. 2018. Fighting Non-determinism in C++ Compilers. Talk presented at the CppCon 2018. Bellevue, WA, USA. Retrieved October 5, 2023 from https://github.com/CppCon/CppCon2018/blob/master/Posters/fighting_nondeterminism_in_cpp_compilers/fighting_nondeterminism_in_cpp_compilers__mandeep_singh_grang__cppcon_2018.pdf

[11] Michael Howard and David Leblanc. 2002. *Writing Secure Code* (2nd ed.). Microsoft Press, Redmond, WA, USA.

[12] Michael Howard and Steve Lipner. 2006. *The Security Development Lifecycle.* Microsoft Press, Redmond, WA, USA.

[13] Kitware, Inc. 2023. *CMake.* Retrieved October 5, 2023 from https://cmake.org/

[14] Chris Lamb and Stefano Zacchiroli. 2022. Reproducible Builds: Increasing the Integrity of Software Supply Chains. *IEEE Software* 39, 2 (2022), 62–70. https://doi.org/10.1109/MS.2021.3073045

[15] Vincent Maraia. 2005. *The Build Master: Microsoft's Software Configuration Management Best Practices.* Addison-Wesley Professional, Reading, MA, USA.

[16] Evan Martin. 2022. *Ninja, a Small Build System With a Focus on Speed.* Retrieved October 5, 2023 from https://ninja-build.org/

[17] Meta Platforms, Inc. 2023. *Buck2.* Retrieved October 5, 2023 from https://buck2.build/

[18] Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. 2018. Build Systems à La Carte. *Proceedings of the ACM on Programming Languages* 2, ICFP, Article 79 (July 2018), 29 pages. https://doi.org/10.1145/3236774

[19] R.P.J. Pinkers, P.M.W. Knijnenburg, M. Haneda, and H.A.G. Wijshoff. 2004. Statistical selection of compiler options, In *The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004. (MASCOTS 2004). Proceedings.* 494–501. https://doi.org/10.1109/MASCOT.2004.1348305

[20] Wolfram Schulte. 2016. Changing Microsoft's Build: Revolution or Evolution. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore) *(ASE '16).* Association for Computing Machinery, New York, NY, USA, 2. https://doi.org/10.1145/2970276.2985779

[21] Peter Smith. 2011. *Software Build Systems: Principles and Experience* (1st ed.). Addison-Wesley Professional, Upper Saddle River, NJ, USA.

[22] The Apache Software Foundation. 2022. *Apache Ant.* Retrieved October 5, 2023 from https://ant.apache.org/

[23] TIS Committee. 1995. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification.* Retrieved October 5, 2023 from https://refspecs.linuxfoundation.org/elf/elf.pdf