Detecting Incorrect Build Rules

Nándor Licker, Andrew Rice
Department of Computer Science and Technology
University of Cambridge
Cambridge, UK
{nl364, acr31}@cam.ac.uk

***Abstract*—Automated build systems are routinely used by software engineers to minimize the number of objects that need to be recompiled after incremental changes to the source files of a project. In order to achieve efficient and correct builds, developers must provide the build tools with dependency information between the files and modules of a project, usually expressed in a macro language specific to each build tool. Most build systems offer good support for well-known languages and compilers, but as projects grow larger, engineers tend to include source files generated using custom tools. In order to guarantee correctness, the authors of these tools are responsible for enumerating all the files whose contents an output depends on. Unfortunately, this is a tedious process and not all dependencies are captured in practice, which leads to incorrect builds. We automatically uncover such missing dependencies through a novel method that we call build fuzzing. The correctness of build definitions is verified by modifying files in a project, triggering incremental builds and comparing the set of changed files to the set of expected changes. These sets are determined using a dependency graph inferred by tracing the system calls executed during a clean build. We evaluate our method by exhaustively testing build rules of open-source projects, uncovering issues leading to race conditions and faulty builds in 30 of them. We provide a discussion of the bugs we detect, identifying anti-patterns in the use of the macro languages. We fix some of the issues in projects where the features of build systems allow a clean solution.**

***Index Terms*—build tools, exhaustive testing, verification**

I. Introduction

Automated build systems are at the heart of all software projects, executing the set of actions required to build applications either from scratch or by incorporating incremental changes into temporary outputs. Clean builds of modern software engineering projects, such as the Linux kernel or MySQL Server, can take a significant amount of time, but developers rarely change more than a handful of files between subsequent builds. In order to maintain productivity, incremental builds are vital: whenever some files change, automated build systems recompile only the minimal subset of outputs which depend on the changes. To enable incremental builds, build tools must be made aware of the dependencies among the files and modules in a project. Our goal is to automatically find errors in the definitions of these dependencies.

In the case of projects which span hundreds of thousands of lines of code, contain multiple modules, and include a large number of dependencies, the dependency graphs are complex and tedious to describe manually. Languages such as C, C++, Java, and Rust carry, in their syntax, dependency information which can be exploited to construct dependency graphs. Instead of manually specifying arguments to compiler invocations in the macro languages of tools like GNU Make [1] or Ninja [2], developers provide brief definitions to build file generators such as CMake [3], Autotools [4], and SCons [5], describing the high-level structure of a project. Build files are generated out of these definitions, along with dependency information extracted by partially parsing the source code of languages known to the build system generator. Such generators also permit the integration of custom tools, but require developers to manually enumerate dependencies.

When engineers write build definitions manually or integrate custom tools into definitions generated by build system generators, they might forget to enumerate all dependencies correctly. These mistakes lead to incorrect incremental builds, where the final executables are linked with stale objects which were not recompiled, despite changes to their sources. Parallel clean builds can also fail because of race conditions: if the build systems does not know about a dependency between an input and an output, the job generating the input can be scheduled at the same time as the job reading it, failing the build. Detecting and identifying such issues is a time-consuming process — GNU Make itself was developed because of frustration with stale builds [6]. A single-threaded clean build which discards all temporary objects is the easy, yet timewise expensive fix, wasting engineering resources which could be used more productively. Lack of proper tooling increases the difficulty of creating correct build definitions and integrating custom tools which generate source files into builds.

Consider the following CMake rule from an open-source project with over 200 stars[1] on GitHub:

```
add_custom_command(
  OUTPUT ${expanded_files_h}
  DEPENDS ${CMAKE_CURRENT_SOURCE_DIR}/${root}.h.t
  COMMAND
    ${PYTHON_EXECUTABLE} ${PYTHON_DASH_B}
    ${CMAKE_CURRENT_BINARY_DIR}/generate_helper.py
    ${root} ${root}.h.t ${ARGN})
```

Even though the input file is mentioned among the dependencies, the Python script is not included, nor are the other Python files imported into the root script. A change to any of those scripts will fail re-generate the desired header and recompile the binaries that might include it, leading to bugs which are hard to detect. This is just one example of a buggy build definition: throughout our evaluation, we uncover issues with out-of-source builds, dependencies on tools,

[1]https://github.com/bastibl/gr-ieee802-11

complex build scripts, the handling of external dependencies, and manual dependency management, as well as attempts to address these problems which lead to redundant work.

We describe a new technique, which we call build fuzzing, to detect these problems. We treat a build as a black box, without inspecting any of the user-provided or automatically generated definitions, relying solely on observing the interactions between the build process and the underlying operating system. We identify the input files of a project and we update each of them in turn, triggering an incremental build afterwards. The set of outputs changed during a build is compared with the set of files expected to change based on the inferred dependency graph. We report whether the build tool fails to rebuild any of the expected targets or rebuilds targets which do not actually depend on the changed input. We identify race conditions by locating rules which can be triggered before all of their dependencies are built. We evaluate our method using open-source projects from GitHub.

The contributions of this paper are:

- A novel method to infer dependency graphs, check the correctness of incremental and parallel builds, discovering missing dependencies and race conditions
- Evidence that our approach solves a significant problem, we tested a large number of open-source projects and found redundant or missing dependencies in 30 of them, including the Linux kernel, MySQL Server and Redis
- An analysis and characterisation of the uncovered bugs, along with a discussion of the deficiencies of build systems and engineering practices which cause those bugs

All our tools are released under the MIT License and available at https://github.com/nandor/mkcheck. Artefacts are available at https://doi.org/10.17863/CAM.35270.

II. Background and Related Work

Build systems come in many flavours, but at their core they parse a set of definitions typically defined in macro languages with imperative or functional syntax [1], [7], [8]. These definitions describe the dependency graph among the modules and files of a project statically before a build (Make, Ninja, SCons), or specify the steps required to determine the graphs dynamically during the build (Bazel). Relying on these dependency graphs, build tools can compile projects from scratch or identify the minimal set of actions to be executed to correctly rebuild all outputs after incremental changes.

In the past, there has been little motivation to study issues with build rules, especially with the goal of finding systematic problems in build definitions. Previous research considered the correlation between the size and complexity of projects and the size and complexity of the build rules that compile them, as well as the evolution of the build definitions along with the evolution of the source code compiled by them [9], [10]. Such studies show that most developers frequently fix build issues they encounter, but only a few knowledgeable contributors create new build rules or improve existing ones. The high impact of build definition maintenance on developer productivity was quantified without considering the root cause of the bugs on which developers waste time [11], [12].

We are mostly concerned with extensibility: even though most solutions satisfy the needs of end-users and offer good support for the languages they were designed for, the macro languages they expose to developers of custom tools are fairly limited and difficult to use. Projects frequently include sources generated from domain-specific languages using external tools, requiring custom definitions to be integrated with other build rules. Unfortunately, even modern build systems assume that those definitions are developed by a few, knowledgeable people and fail to offer features to ensure correctness. Only a few of them offer built-in features which allow the underlying build graphs to be visualised. We describe the 5 widely used build systems and build system generators we inspected, along with their debugging facilities, in Section II-A.

Existing verification methods which analyse the actions executed by build systems along with the source code of the rules defining the actions, explored in Section II-B, do not generalise well because of the difficulties involved in parsing and understanding the static definitions, which are fundamentally different across all the build tools.

A. Build Systems

Make [1] has been around for the past 50 years and still sees widespread use. It trakcs the dependencies of individual modules, rebuilding the minimal set of targets which depend on a particular file. Nowadays developers can avoid writing raw Makefiles, relying on build system generators which can output Makefiles from brief definitions instead, automatically detecting and tracking certain dependencies. Make has multiple implementations and variations available on GNU, BSD and Microsoft systems. *bmake* [13], available on BSD, incorporates dynamic information from system call traces to rebuild targets whenever system libraries or utilities change, in addition to the dependencies specified in build rules. The target of our evaluation was the canonical GNU Make implementation, available on most Linux distributions.

Ninja [2] is a more recent build tool which performs similar functions to make, except it was not designed to be human-readable. Instead, rules are meant to be generated from higher-level definitions: the tool was first used in the Chromium project alongside the gyp (generate-your-project) generator. Ninja outperforms GNU Make when comparing incremental build speeds and most build system generators support it. The tool aids debugging by producing a visual representation of the graph it parses from definitions.

SCons [5] is a highly extensible build system which does not rely on a custom macro language, allowing developers to write definitions in Python instead. It has built-in support for automatically discovering C and C++ dependencies dynamically and can dump a textual representation of dependency graphs.

CMake [3] is a widely-used build system generator. It natively supports a large number of languages such as C, C++, Fortran and various assembly dialects. CMake generates complex recursive Makefiles from concise definitions, detecting and updating C and C++ dependencies [14]. Both Ninja and GNU Make definitions can be generated. The macro language allows developers to specify custom rules, but requires all dependencies to be manually enumerated in order to ensure the correctness of incremental builds. Build system generators require significantly less effort from developers: for example, in the case of MySQL server, 4000 lines of CMake definitions generate more than 130,000 lines of GNU Make rules.

Bazel [15] and *Buck* [16] are modern distributed build systems. They distinguish themselves by allowing developers to define their build rules in languages similar to Python and by building individual targets in the cloud, instead of a single local workstation. They offer some support for automatically detecting dependencies between C++ files, however they encourage programmers to specify the full dependency graphs manually. In the case of custom rules, they still require dependencies to be manually enumerated. Similarly to Ninja, dependency graphs can be queried and visualised.

B. Dependency Graph Inference

In a typical build system the actions required to build some output are specified along with all the inputs to the underlying tools. The dependency information is used to enforce the order in which actions are executed and to identify the set of outdated targets during an incremental build. Since the tools are arbitrary processes, the build system has no means of preventing those tools from inspecting files other than the ones listed as dependencies. Because of this, dependency graphs defined by users of build tools cannot be trusted, however more accurate information can be extracted by inspecting the interactions among processes and the file system by capturing the system calls executed during a build. System call tracing has been previously used to recover true dependency graphs, by connecting the files written by a process to files produced by processes which read them. These graphs were applied to build system migration, linting and refactoring.

CloudMake [17] is a distributed build system by Microsoft which exposes an interface based on TypeScript to specify build definitions. As with all new build systems, this project faced adoption issues as well: the company already had millions of lines of code compiled using definitions based on another system. To ease the transition, existing build scripts were automatically migrated. In order to translate an existing set of build definitions, dependency graphs were recovered through system call tracing by instrumenting calls to the Win32 APIs. Through pattern matching, the nodes represented by processes were replaced with human-readable build rules, defined in a higher-level language. Automatic refactoring was applied to simplify and coalesce build rules. The build language of CloudMake was formally specified and verified, however the proofs are limited to a subset of known tools specified through axioms [18].

depslint [19] is a Python script for linting build rules of projects using the Ninja build system. It traces system calls during clean builds and validates incremental builds, emitting warnings whenever files other than the ones mentioned among the dependencies are read from during the execution of a build action. This tool is limited by the fact that it only supports a single build system whose definitions were not meant to be written manually, by design. This creates difficulties in correlating the warnings emitted by the tool with the high-level definitions which yield the erroneous definitions.

MAKAO [20] is a dependency graph inference tool successfully used to reverse engineer the build definitions of the Linux kernel [21]. This tool is tailored for the GNU Make build systems and recovers dependency information through a hybrid approach, by parsing Makefiles and the command line strings used to invoke the `gcc` compiler, captured through the Bourne shell's `xtrace` option. The dependency information was used to gain a deeper understanding of the Linux kernel's source code by visualising the dependency graph.

III. Our Approach

Our testing approach is inspired by existing tools which inspect system calls in order to gain more information about the underlying dependencies. Instead of using this information to migrate build rules to a new language or refactor build rules, we check the correctness of build definitions, regardless of the build system, in order to find mistakes causing incorrect incremental builds or race conditions during clean parallel builds. This approach applies to build systems which performs compilation in a separate process invoked by the build tool. We assume that clean builds succeed and that all actions defined in the rules are correct, if that is not the case, tools such as MkFault [22] can be used to localise errors.

In order to obtain accurate dependency information for a project, we perform system call tracing, presented in Section III-A, to infer the dependency graphs outlined in Section III-B. These graphs can be used to check the correctness of incremental builds interactively, as developers work on a project, however we focus on fuzz testing open-source projects. We describe a method to automatically detect missing dependencies in Section III-C and race conditions in Section III-D. Our system call tracer can infer dependency graphs for arbitrary projects, provided clean builds are correct.

A. System Call Tracing

Any data persisted by a process must be either an argument to a system call, written to a memory-mapped files or passed to another process using an inter-process communication mechanism, such as shared memory or pipes. A process can be viewed as a pure function, transforming input files into one or more output files. The goal of system call tracing is to recover the inputs and outputs which link the functions together.

Most systems offer facilities to trace the system calls executed by a process, through the operating system (on Linux), by instrumenting library calls (on Windows through the Win32 APIs and on OS X through `libc`) or by instrumenting the kernel (dtrace on OS X and eBPF on Linux). Performance and complexity vary wildly. Dependency graph reconstruction requires capturing a subset of all system calls, those which perform I/O (`read`, `write`, `mmap`, ...) and those which transfer data between processes (`pipes`, `shmget`, ...).

`ptrace` is a Linux system call, allowing a tracer process to stop and inspect the registers and memory of a tracee process whenever a system call is entered or exited in the tracee. Through `ptrace`, all system calls executed by a process can be captured. This approach is expensive: besides the two context switches between the kernel and the tracee, at least 4 other context switches are required between the tracer and the kernel to inspect and resume a syscall. The tracee must be stopped even when the syscall is not relevant to the graph.

Dtrace and eBPF probes can be used to trace relevant system calls on OS X, Linux and Solaris. Unfortunately, this method requires disabling System Integrity Protection on OS X and the installation of additional kernel modules on Linux. Even though this approach offers significantly better performance compared to `ptrace`, it is highly inconvenient. Furthermore, dtrace does not guarantee intercepting all system calls since information might be dropped in order to maintain performance under heavy loads.

On Windows and OS X, system calls are performed through the standard library: manually invoking the `syscall` instruction is highly discouraged. By relying on the functionality of dynamic loaders or other readily-available instrumentation tools, the interfaces between builds tools and the system libraries can be modified in order to capture all system calls. Unfortunately, this approach is more complex as it not only requires changes to all system call wrappers (`open`, `close`, `write`, etc.), but most standard C functions must be modified as well (`fopen`, `fread`, etc.) since the system call wrappers might be inlined inside the library.

Our inference method uses `ptrace`, targeting Unix-based build environments. Unlike existing tools such as BSD make's meta mode, our approach correctly traces and handles pipes and the close-on-exec behaviour of file descriptors. Support for pipes is important since tools which dump their output to `stdout` are commonly used to generate code during builds.

Some build systems make network requests to download packages (for example, it is fairly common for a build based on CMake to download external dependencies through `wget`). In such cases, we assume that the remote resources never change once they are downloaded. We do not consider system calls which read system parameters (such as `getrlimit` and `getrusage`) since these parameters seldom change, nor do we intercept calls returning the current time, which in the `glibc` implementation rely on vDSO instead of executing actual system calls (`gettimeofday`).

B. Dependency Graphs

Relying on the arguments and return values of system call intercepted by `ptrace`, we use Algorithm 1 to capture information about all files and processes involved in a build. The result is a graph with two kinds of nodes, representing files and processes along with specific metadata. File nodes track whether the file is a temporary or a persisted object, along with a list of dependencies to represent renamed files and symbolic links. Process nodes track their inputs, outputs and parent processes. During the execution of the algorithm, the file descriptors introduced by `open` are tracked and files are added to the set of inputs or outputs of a process when an I/O action is performed (`read`, `mmap`, ...). The close-on-execute (`CLOEXEC`) flag is tracked in order to correctly propagate file descriptors from parent to child processes.

For example, when the `pipe` system call is encountered in `cat a.txt | md5 > b.txt`, the tracer adds two virtual files to the state of the process, corresponding to the read and write ends of the pipe. An additional dependency is added between the read end and the write end of the pipe. `md5` receives the read end of the pipe and calls `read` on it, adding the virtual file to its inputs. `cat` receives the write end and outputs to it using `write`, adding the virtual file to its outputs, allowing the dependency from `a.txt` to `b.txt` to be followed transitively through the virtual files and the additional dependency between them.

Algorithm 1 Dependency Graph Inference

```
procs ← { traced process }, files ← ∅
while a child is running do
    pid, call details ← PTRACE
    p ← procs[pid]
    switch call details do
        case child = fork():
            procs[child] ← p
            procs[child].pid ← child
            procs[child].parent ← p.pid
        case exit(): p.running ← false
        case execve(image):
            p.ins ← ∅, p.outs ← ∅, p.image ← image
            p.files ← files without cloexec from p.files
        case fd = open(path, cloexec):
            files[path] ← { deleted: false }
            p.files[fd] ← { path, cloexec }
        case rd, wr = pipe(cloexec):
            rdn, wrn ← create unique names for pipes
            p.files[rd] ← { rdn, cloexec }
            p.files[wr] ← { wrn, cloexec }
            files[rdn].deps ← files[rdn].deps ∪ wrn
        case read(fd): p.ins ← p.ins ∪ p.files[fd].path
        case write(fd): p.outs ← p.outs ∪ p.files[fd].path
        case unlink(path): files[path].deleted ← true
        case rename(src, dst):
            files[src].deleted ← true
            files[dst].deps ← files[dst].deps ∪ src
end while
```

Figure 1 illustrates a dependency graph recovered through system call tracing during a typical build. Nodes and edges in green represent the process hierarchy, encoded in the *parent* field of the process objects, of the tools invoked by GNU Make, as well as the various other tools invoked by the `gcc` compiler driver. Incoming black edges (*ins* field) originate from inputs, while outgoing edges point to outputs (*outs* field). Some output files are temporary, indicated by red text: these are temporary objects deleted by the compiler after emitting the object files. They are still relevant to the graph as the input sources are connected transitively to the output objects through the intermediary assembly files. Additionally, dependencies between files, stored in the *deps* field of files, exist between the two ends of a Unix pipe and also occur when files are moved or

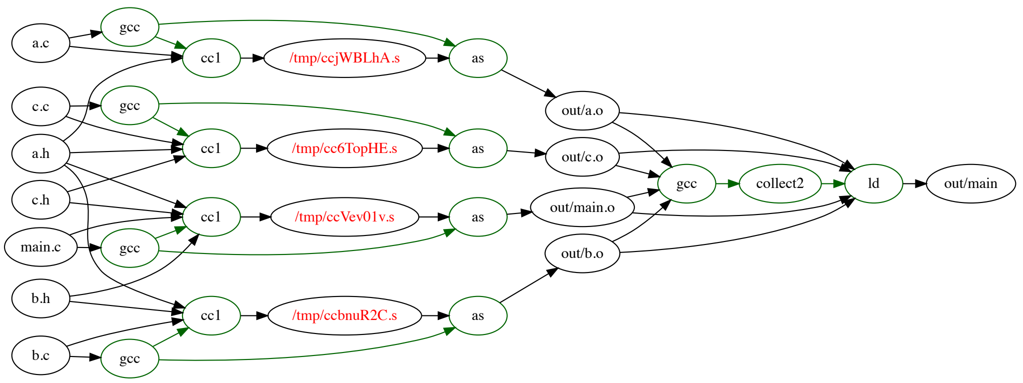

Fig. 1: Dependency graph inferred from a C project built by GNU Make. Green edges and nodes represent the process hierarchy of the invoked build tools, the black nodes and edges represent the files which are the inputs and outputs of processes. The files in red are not persisted after the build. The process running GNU Make itself was omitted.

symbolic links are created. This allows us to correctly handle pipes, ignored by existing tools such as *depslint* or *bmake*.

For the purpose of our analyses, process nodes are only used to identify the name of the executable which generated a file. Such nodes are otherwise collapsed, forming a dependency graph of files. Even though inter-process edges carry a data dependency through the arguments and environment passed from a parent to a child process, they are ignored. In a typical build, the process belonging to the build tool reads from every file without writing to any, creating a cycle from outputs to inputs through the process edge which connects the build system to the tools it invokes. Ignoring these edges excludes dependencies on build files read by the build tool, potentially containing the command line arguments passed on to compilers. Since build files are usually generated from third-party sources before a build, we are not including them in our analyses as they would uncover issues with generators, which are unlikely to occur, instead of validating handwritten rules.

Our graphs are only an approximation of all data dependencies between all objects involved in a build. Since some edges might be missing (underconstraining) or some outputs might be redundantly connected to some inputs (overconstraining), our analyses can result in false positives or negatives.

1) Overconstraining: An overconstrained graph introduces false positives and negatives: the analysis might report that a file should be changed when an input is touched, while in reality data from the touched file does not actually determine the output. If the build system unnecessarily rebuilds an output after an input is changed and a redundant path is present in the graph, the analysis fails to identify the problem.

The granularity of the dependency graph is limited to processes: if a process produces multiple outputs, dependency inference cannot distinguish which inputs determine which output, considering all outputs to be dependants of all inputs. This is usually not a concern in C/C++/Assembly projects compiled using GNU Make or CMake since most tools involved in a build either generate a single outputs or all outputs they generate depend on all inputs.

The lack of information about the flow of data inside a process prevents our analyses from running Java projects. Java build systems, such as Maven [23] and Gradle [24], invoke the compiler as a library instead of creating a separate process. Since our analysis would create a node having all source files as inputs and all class files as outputs, the results would not be relevant. An example of such a graph, inferred by our tool, is shown in Figure 2. In large Java projects which rely on the Java Native Interface (JNI), our approach could still be used to determine the correctness of the native components.

Another source of false positives are Unix pipes. Even though only data derived from a subset of the inputs might be transmitted over the pipe, all inputs are dependencies of the outputs of the reading process, due to transitivity. Pipes are routinely used in shell scripts, but we have not encountered a script large enough to introduce redundant edges.

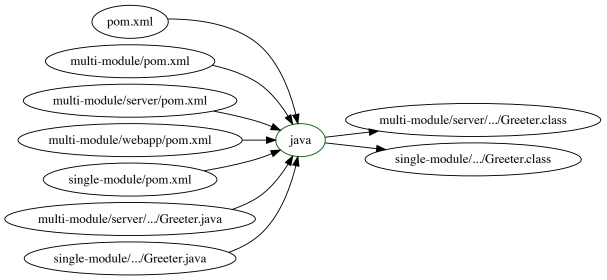
pom.xml
multi-module/pom.xml
multi-module/server/pom.xml
multi-module/webapp/pom.xml
single-module/pom.xml
multi-module/server/.../Greeter.java
single-module/.../Greeter.java
java
multi-module/server/.../Greeter.class
single-module/.../Greeter.class

Fig. 2: An example of an overconstrained graph, obtained by tracing a Maven build on a sample project. Since Java builds run in a single process, all inputs are connected to all outputs through the single process running the Java Virtual Machine.

2) Underconstraining: The inferred dependency graphs might lack edges between files whose contents depend on each other, leading to both false negatives and positives in our analyses. If an edge is missing but the build system specifies a dependency between the two files, the output is considered to be redundantly rebuilt. If a dependency exists, but neither our graph, nor the build system are aware of it, then we fail to emit a warning if the dependant object is not recompiled.

Despite capturing all interactions between a process and the operating system, some edges might be missing from the recovered graph. Our analysis only considers system calls which succeed, however failing ones are also important. For example, C++ compilers search through a large number of directories in order to locate input files during their preprocessing step: creating a file in such a directory should trigger a build. We do not consider the addition or deletion of files to a project, as we are only concerned with checking existing dependencies, thus this issue does not affect our results.

Another source of problems are arguments and environment variables passed to child processes through `execve` and its variants: the parameters passed introduce a data dependency, which we omit to avoid cycles in our graphs, potentially introduced by the build tools which spawn processes. This prevents our analysis of builds which involve tools reading arguments from files, passing them on to child processes.

Algorithm 2 Fuzz Testing Procedure

```
dependencies ← ∅, ins← ∅, outs← ∅, built_by ← ∅
for file ∈ files do
    for dep ∈ file.deps do
        dependencies[dep] ← dependencies[dep] ∪ file
    end for
end for
for proc ∈ procs do
    ins ← ins ∪ proc.ins, outs ← outs ∪ proc.outs
    for out ∈ proc.outs do
        built_by[out] ← proc.image
        for in ∈ proc.ins do
            dependencies[in] = dependencies[in] ∪ out
        end for
    end for
end for
function ALL-DEPS(file, deps = ∅)
    if file ∉ deps then
        deps ← deps ∪ {file}
        for f ∈ dependencies[file] do
            deps ← deps ∪ ALL-DEPS(f, deps)
        end for
    end if
    return deps
end function
for file ∈ ins \ outs do
    t0 ← READ-TIMESTAMPS(outs)
    TOUCH-AND-BUILD(file)
    t1 ← READ-TIMESTAMPS(outs)
    changed ← {t|t ← outs, t1[t] > t0[t]}
    for f ∈ ALL-DEPS(file) \ changed do
        REPORT-MISSING(f, built_by[f])
    end for
    for f ∈ changed \ ALL-DEPS(file) do
        REPORT-REDUNDANT(f, built_by[f])
    end for
end for
```

C. Fuzz Testing

Our build fuzzing method, which identifies the files in a project which trigger incorrect incremental builds, is outlined in Algorithm 2. First, the dependency graph is simplified by collapsing process nodes and temporary files, retaining a mapping from output files to the processes that write them. Next, all relevant input files are identified and incremental builds are triggered by modifying each file in turn. In the case of build systems which rely on content hashing, an additional null terminator or newline is added to the file. Otherwise, the modification timestamp is simply updated. The set of files affected by a build is identified by comparing the timestamps of files before and after the build, retaining the changed ones. If the set of changed files does not match the set of expected changes, computed using a depth-first traversal of the dependency graph starting from the changed file, a diagnostic message is emitted, identifying the dependencies which were not rebuilt or were redundantly rewritten, along with the tools which should have been invoked to build them.

The set of tested files is determined based on the type of the tested project. Only those files which were read without being modified during a clean build are considered: in the dependency graph, these are the file nodes without any incoming edges. Files involved in built-in rules, as files created by the configuration step of the project, specific to each build system, are excluded to speed up testing, as they are likely correct.

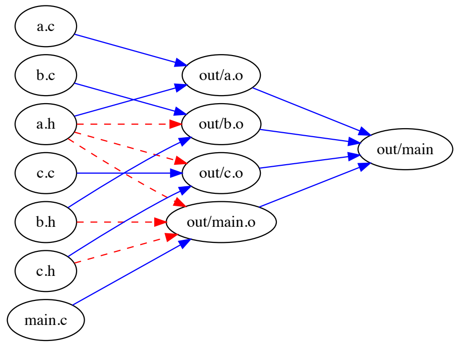

Fig. 3: Missing dependencies: blue edges represent the dependencies defined in the Makefile, red edges the missing ones

Figure 3 illustrates a dependency graph used by build fuzzing, obtained by collapsing the processes and temporary files of the inferred graph shown in Figure 1. Fuzzing triggers an incremental build after modifying each of the inputs: the C header and source files. Incorrect builds are triggered after 3 of the headers are modified, as they are not enumerated in the dependencies of the object files in the Makefile. The missing edges, obtained by intersecting the set of outgoing edges of a node with the set of stale dependants, are highlighted in red.

D. Race Condition Detection

In addition to identifying files which trigger incorrect incremental builds, our method can be extended to detect potential race conditions in builds. We consider that a race condition occurs during a build if the job building an object is executed before all of its dependencies are built. Because of race conditions, clean parallel builds can fail spuriously, as rules can be scheduled to read from missing files. During incremental builds, objects might be rebuilt from stale inputs.

Our method to detect races is outlined in Algorithm 3. Through fuzzing, we first find the missing outgoing edges by intersecting the set of objects reported missing by Algorithm 2 with the set of outgoing edges of a node for all files involved in a build, not only inputs. For each object we find the set of all of its transitive dependencies, excluding inputs, in the inferred dependency graph: the size of the sets indicates how many objects need to be built before the command generating an object can be triggered. If the execution order of all commands was linearised, the size of the set represents the earliest point in time an object can be built. We compute this index in the graph with the missing edges removed as well: if the index of a file is less in this graph, it means that the build system might trigger the rule generating it sooner in time, before all dependencies are generated, indicating a potential race condition.

Algorithm 3 Race Condition Detection

```
function FIND-SCHEDULE(g)
    gt ← TRANSPOSE(g), deps ← $\emptyset$, schedule ← $\emptyset$
    for file ∈ TOPO-SORT(g) do
        for pred ∈ gt ∩ outs do
            deps[file] ← deps[file] ∪ {pred} ∪ deps[pred]
        end for
    end for
    for file ∈ g do
        schedule[file] = |deps[node]|
    end for
    return schedule
end function
partial ← dependencies
for from, to ∈ missing_edges do
    partial[from] = partial[from] \ {to}
end for
s0 ← FIND-SCHEDULE(dependencies)
s1 ← FIND-SCHEDULE(partial)
for f ∈ files do
    if s then1[f] < s0[f]
        REPORT-RACE(f)
    end if
end for
```

Figure 4 illustrates a dependency graph with a race condition detected by our method: `b.c` includes `a.h`, but the dependency is not present in the build file. The build tool is free to schedule the compilation of `b.c` before the generation of `a.h`, leading to an error. In the correct graph, `a.o` can be scheduled earliest at time 3 and `b.o` can be scheduled at time 4. In the graph defined in the build file, both objects can be scheduled at time 3, indicating the possibility of a race.

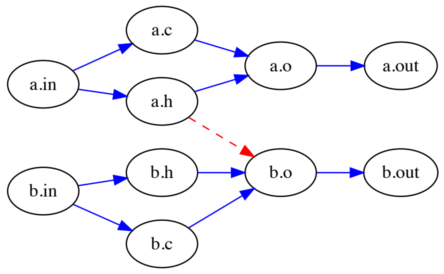

Fig. 4: Race condition: the missing dependency, shown in red, allows `b.o` to be built before `a.h`, failing a parallel build

IV. Evaluation

A. Supported Build Systems

Even though we only subjected projects relying on CMake, GNU Make and SCons to build fuzzing, we tested our tools on a wide range of build systems to ensure we can recover useful dependency graphs. For various languages and build systems, we evaluated existing projects or we created small projects to compile multiple isolated modules of the same language into an executable. We inspected the dependency graphs to check whether they had any redundant edges connecting the two isolated modules, negatively affecting our analyses.

Build Tool	Language	Supported/Issues
GNU Make	C	✓
Ninja	C	✓
SCons	C	✓
Bazel	C	deadlock under `ptrace`
Maven	Java	overconstraining
Ocamlbuild	Ocaml	✓
`go build`	Go	✓
cargo	Rust	✓
cabal	Haskell	✓

TABLE I: Dependency graph inference from build systems

Table I reports the automated build tools we have evaluated and the issues we encountered with dependency graph inference. The list does not include build system generators, only the tools for which they generate inputs. Our method generalises well across various build systems, allowing a larger range of projects to be tested, compared to what was possible with existing tools. We encounter problems with build systems that invoke compilers as libraries instead of shelling out to a separate process, as in the case of Java. Due to our reliance on `ptrace`, we serialize system calls executed by all parallel threads in a process, deadlocking tracees such as Bazel. Using another provider which does not alter the tracees behaviour, such as `dtrace`, would avoid such problems.

B. Build Fuzzing

We evaluated the effectiveness of our tool by automatically identifying incorrect builds and race conditions in a large number of projects built using CMake, GNU Make and SCons. We searched for projects using the GitHub code search API to find roughly 500 projects which relied on handwritten Makefiles or included custom rules into CMake or SCons builds using `add_custom_command` or `env.Command`. We built and tested projects satisfying the following criteria:

- Projects had to contain a `CMakeLists.txt`, a `Makefile` or a `SConstruct` in the root folder: this excluded some poorly written projects with no documentation, or large projects with complex build instructions
- Projects which bundled large external dependencies were excluded: issues found in the dependencies would have outnumbered the issues in the smaller project and would have been representative of the build system of the dependency, not the one used to build the project itself
- Projects had to compile out of the box or with minimal changes: we excluded old projects which were outdated to the point of irrelevance and relied on deprecated libraries.

CMake can generate both Ninja builds files and GNU Makefiles, however some projects rely on features which can only be translated to GNU Make. Where possible, we have tested projects using both build systems, obtaining identical results. We filtered CMake-specific configuration files and C/C++ header and source files from the inputs, as the built-in rules emitted by CMake can correctly handle these. For the same reason, we also excluded C/C++ sources and headers from SCons projects. SCons can detect changes by checking both timestamps and content hashes: we have modified SCons to force the use of the timestamp mechanism for our tests. All files were considered in projects relying on handwritten Makefiles, with the exception of the Linux kernel where we omitted reliable configuration files and C source files.

Tables II, III and IV present the list of projects where we encountered errors or false positives. Each row indicates the authors and names of the projects, the number of stars on GitHub which is correlated with the relevance of a project, the number of tested files, the number of files which triggered incorrect incremental builds and the number of files which triggered builds that recomputed unnecessary targets, a check mark indicating whether we fixed the build or an explanation why a fix was not possible otherwise, along with a short explanation of the root cause of the uncovered build problems.

In order to confirm the validity of our results, we identified the faulty build rules associated with each of the files reported by our evaluation, aided by the information emitted in diagnostic messages. Wherever a fix possible in a reasonable amount of time without substantially re-engineering a project, we modified the Makefiles or CMake definitions to fix the

Project	Stars	Files	Errors	Fixed	Issues
torvalds/linux	61 462	27	6 / *11*	✗[1]	missing dependencies, false positives
antirez/redis	28 241	545	87 / 4	✗[1,2]	races, subproject dependencies
tinyCC/tcc	163	26	6 / 1	✗[4]	races, manual C dependencies
pyssling/namespaced_parser	159	4	2 / 0	✓	races, manual C/C++ dependencies
jkbenaim/cboy	17	67	42 / 0	✗[4]	races, manual C/C++ dependencies
dcdelia/tinyvm	6	21	7 / 0	✓	manual C/C++ dependencies
nicknytko/x86-thing	2	52	20 / 52	✗[1]	unconditional rules, manual deps
sadiredd-sv/CacheSimulator	1	7	2 / 0	✓	manual C/C++ dependencies
coldbloodx/lec	1	74	1 / 0	✓	manual C/C++ dependencies
kostrahb/Generic-C-Project	0	3	1 / 0	✗[2]	manual C/C++ dependencies
percivalgambit/hindsight-is-8080	0	19	8 / 0	✗[2]	manual C/C++ dependencies
radekvit/reon	0	15	1 / 2	✗[4]	manual deps and globbing
apron	-	190	147 / 69	✗[4]	races, manual C dependencies

TABLE II: GNU Make stale/redundant incremental builds

Project	Stars	Files	Errors	Fixed	Issues
mysql/mysql-server	2 533	21	1 / 20	✗[1]	inputs to custom tools
anbox/anbox	656	24	9 / 23	✗[1,2]	unconditional, inputs to custom tools
ICTeam28/PiFox	326	41	23 / 0	✗[2]	races, manual dependencies
bastibl/gr-ieee802-11	239	7	2 / 0	✓	python import
uwsampa/grappa	106	89	0 / 9	✗[1]	unconditional rules
qknight/automate	11	86	23 / 0	✗[3]	races, manual deps
regmi007/ALang	11	3	2 / 0	✓	manual deps
geodynamics/specfem3d_geotech	9	25	20 / 0	✗[3]	races, unsupported language
DavidPeicho/tiny3Dloader	5	26	*20* / 6	✗	unconditional rules, false positives
davidzchen/decaf	6	2	2 / 0	✓	in-source
prozum/sppl	3	5	0 / 1	✗[3]	unconditional rules
lukedodd/Pixslam	2	20	4 / 2	✓	races, in-source, tool dependency
leidav/tetris	0	6	2 / 0	✓	no dependency on tool
calendarium-romanum/libcalrom	0	4	2 / 0	✓	in-source

TABLE III: CMake stale/redundant incremental builds

Project	Stars	Files	Errors	Fixed	Issues
nieklinnenbank/FreeNOS	219	54	*54* / 54	✗	false positives due to scons
blitz/baresifter	12	27	0 / 12	✗[2]	unconditional rules
profmaad/steppinrazor	1	15	13 / 0	✗[2]	no dependencies between ASM files
brouhaha/nonpareil	-	18	1 / 0	✗[2]	custom build rules
fsp/fsp	-	40	*40* / 40	✗	unconditional rule, false positives due to scons

TABLE IV: SCons stale/redundant incremental builds

[1] Requires extensive changes to project structure
[2] Build system lacks features or language support
[3] Broken third-party build rule
[4] Project too large for correct manual dependency management

build rules and re-tested the project, identifying no issues and proving that the initial original rules were faulty. Out of the 32 projects we found problems in, we managed to fix 10. We added additional dependencies to makefiles an adjusted `add_custom_command` macros in CMake to include all outputs, even transitive dependencies of the generator scripts. Some projects were too large to fix, manually tracking dependencies for a significant number of sources, or required extensive reorganisation to eliminate certain targets which were unconditionally rebuilt at all times. Others required features which were not present in the build systems: discovering dependencies dynamically as targets are built, extracting them from the contents of the files they are building. In certain CMake projects, we found issues in third-party rules we did not fix. Most notably, in projects using *flex* and *bison*, we found that the build rules generating the C headers and sources do not enumerate all outputs. Where we could not fix issues, we confirmed them by adding breaking changes to files and triggering incremental builds which succeeded since nothing was recompiled due to missing dependency information.

The projects we tested contained too many files for us to manually check for the presence of false negatives, however we encountered false positives due to missing or unnecessary edges in the inferred graphs. In 3 projects, namely *DavidPeicho/tiny3Dloader*, *nieklinnenbank/FreeNOS*, and *fsp/fsp*, we report false positives due to the granularity of processes: the build system's process performs a significant amount of work, copying files from one location to another. Based on our graph we expect the whole batch to unnecessarily change, even though only one file is updated. In the Linux kernel, a header file is regenerated every minute, containing a minute-accurate timestamp. We do not expect files dependant on the header to change as we do not model the time as an input to our graph.

C. Classes of Bugs

By manually inspecting and checking the issues reported through testing, we discovered some patterns in the bugs, allowing us to relate the issues to problems with the feature sets of modern build systems. We discuss why these bugs occur in some build systems and how other tools or environments might be able to solve them. Since we do not correlate problems with their definitions in the macro languages of build systems, this process is not automated.

Issues with Out-of-source Builds This issue was uncovered before testing: even though one of the selling points of CMake are out-of-source builds which allow compilation artefacts to be placed in a directory tree separate from the sources, some of the projects did not compile correctly if the build folder was separate. The developers of the projects did not use the proper macros to point to the expected location of inputs and outputs to their tools, leading to failed builds. More recent build systems such as Buck and Bazel provide more intuitive syntax to define the inputs and outputs to the rules, automatically formatting them to point to a correct isolated location. Encountered in: *davidzchen/decaf*, *lukedodd/Pixslam*, *calendarium-romanum/libcalrom*

Dependencies on Tools Build rules for automatically generated files must include dependencies on the inputs to the tools, as well as on the tools themselves. Even if the tool is external and provided by the system, a dependency on it should exist to ensure that outputs are rebuilt when the system is updated. In some cases, this issue is avoided by running tools unconditionally during each build or by generating files once, when the project is configured. None of these solutions are satisfactory as generating files and recompiling files generated during each build can be time consuming. Some build systems solve this issue by offering more complex macro languages, requiring the tool to be a target defined in the project, not an arbitrary shell command passed to a shell. Encountered in: *prozum/sppl*, *lukedodd/Pixslam*, *leidav/tetris*, *ghewgill/emulino*

Transitive Dependencies Adding a dependency on the entry point of an interpreted script is not enough: scripts include other modules from inside the project which are not marked as dependencies. This problem exists when using languages which do not bundle all their sources into compact executables. Unfortunately, Python, Perl and Ruby are popular choices for generator scripts and they fall into this category. Some build system generators, such as CMake, bundle template engines to generate configuration files which solve this issue, however the languages are not powerful or intuitive enough to ensure wide use. Encountered in: *torvalds/linux*, *bastibl/gr-ieee802-11*, *tinyCC/tcc*, *prozum/sppl*, *brouhaha/nonpareil*

Handling of External Dependencies C and C++ do not have standard package management systems, thus most projects rely on the package managers of operating systems to place headers and shared objects in locations known to the project. If the project requires a specific version of the package, developers then to include it as a submodule or as a copy in their projects. Build fuzzing is likely to identify all files from the dependency as erroneous, as there no link between the shared object built from them and the final executable into which they are linked. Languages such as Haskell, Rust and Go integrate package management in their build systems and distributed tools systems such as Buck and Bazel offer only very limited support for external dependencies, adhering to the monorepo philosophy [25]. Encountered in: *antirez/redis*

Unconditional Rules In order to avoid all the problems that arise from having to correctly integrate automatically generated sources and track all dependencies, some create unconditional rules that recompile everything, without considering the changes since the last build at all. Such a design increases the cost of incremental builds, as the files might be expensive to generate, as is the case with Thrift protocols. Encountered in: *nicknytko/x86-thing*, *prozum/sppl*, *uwsampa/grappa*, *fsp/fsp*, *anbox/anbox*, *DavidPeicho/tiny3Dloader*

Manual and Static Dependency Management We found issues in all non-trivial projects we considered which defined their build rules in handwritten Makefiles. Most build systems rely on static dependency graphs: the dependencies of a file cannot be defined in the file itself, as is the case with C/C++ headers. The presence of these bugs in Makefiles justifies the existence of build system generators such as CMake or Autotools and the use of build systems with dynamic graphs, such as Bazel. Encountered in: *pyssling/namespaced_parser*, *jkbenaim/cboy*, *dcdelia/tinyvm*, *mysql/mysql-server*, *tinyCC/tcc*, *sadiredd-sv/CacheSimulator*, *kostrahb/Generic-C-Project*, *qknight/automate*, *anbox/anbox*, *apron*, *coldbloodx/lec*, *percivalgambit/hindsight-is-8080*, *radekvit/reon*, *geodynamics/specfem3d_geotech*, *regmi007/ALang*

Race Conditions This issue does not usually affect clean, single-threaded builds since the scheduling algorithm in most build systems is likely to deterministically and accidentally order jobs correctly, however the non-determinism introduced by parallelism is likely to reveal these problems, causing builds to fail. Usually developers avoid these problems by disabling parallelism, wasting valuable time. Encountered in: *tinyCC/tcc*, *antirez/redis*, *pyssling/namespaced_parser*, *jkbenaim/cboy*, *apron*, *ICTeam28/PiFox*, *qknight/automate*, *geodynamics/specfem3d_geotech*, *lukedodd/Pixslam*

D. Severity

The severity of the bugs which can be detected using our tool varies: if a buggy target has few dependants, developer time is wasted since a time consuming clean build is required after each change. In such a case, a solution is valuable. If a large portion of a project depends on the target, then an incremental build might be as slow as a clean one and investing time in a fix might not be worthwhile. The detected race conditions reduce parallelism, increasing build times and wasting resources due to randomly failing builds.

E. Performance

All tracing methods involve some overhead on the traced process. Our tool, which relies on `ptrace`, slows down builds by a factor of two and is around 20% faster than `strace`, as measured on clean builds running on 4 threads. Other tracing approaches, such as dtrace or instrumentation, should reduce this overhead to around 10%. This should not impact developers who create custom build rules as we only trace clean builds once to infer the dependency graph.

Fuzzing times depend on project incremental build times. Even though it is valuable to subject a project to such tests, they do not need to be executed often as build rules are seldom changed, thus it is acceptable to wait for some amount of time for the tests to finish. Since build definitions might contain race conditions and multiple projects are free to write to shared files located outside their project folders, we did not parallelise the clean builds to ensure the correctness of build graphs.

V. Alternative Methods

A. Dynamic Taint Analysis

Since system call tracing cannot track how a process uses its inputs to produce outputs, we considered finer-grained instrumentation in order to refine the dependency graph and enable the verification of Java builds, which run in a single process. In order to test the idea, we created a custom LLVM pass and a support library to instrument GNU Make with dynamic taint tracking in an attempt to identify which files are inspected before a build is triggered.

Unfortunately, we identified issues with build systems that prevent this method from yielding useful results. Such tools are likely to rely on traversing a directed acyclic graph, building targets after all of their dependencies. A build stops whenever a target fails to build, which means that the set of taint values affecting control flow constantly increases due to the conditional branches which determine whether a target succeeded or not. Control flow information is both absolutely necessary and useless: taint must be propagated from the comparison which inspects the timestamps of the `stat` system call, but the control flow decisions involved graph traversals lead to a large amount of overtainting.

B. Parsing Build Definitions

In an attempt to increase the efficiency of the verification process and to verify our results, we considered parsing build definitions in order to compare the defined dependency graphs with the inferred ones. Even though the two graphs can be trivially compared, statically analysing Makefiles can be quite problematic, especially when they are handwritten and not generated. Some projects contain Makefiles for multiple build systems (mixing Python's setuptools with GNU Make, for example), preventing dependencies to be traced through arbitrary processes. Creating custom parsers also involves substantial engineering effort since a new parser must be created for each build system (GNU Make, Ninja, SCons, etc).

VI. Concluding Remarks

We developed a novel method, build fuzzing, to test the correctness of the build definitions in automated build systems by finding missing dependencies and race conditions. We evaluated our implementation of build fuzzing and race condition detection on publicly available open-source projects relying on 3 different build systems and build system generators. Based on the diagnostic messages emitted by our tools, we provided solutions to the issues in some of the evaluated projects.

Unlike existing systems, our approach relies solely on capturing the interactions of build tools with the underlying operating system or file system. We do not rely on parsing build definitions, thus our tool can be used to test a large number of projects, built using arbitrary code generators and compilers managed by a wide range of build systems.

Our search revealed numerous problems in 30 projects, producing only a small number of false positives. We found issues in both small hobby projects as well as in mature ones such as the Linux kernel. We analysed the bugs to categorise the underlying problems, identifying anti-patterns and scenarios where the feature sets of existing build systems do not provide adequate support. The issues we found can potentially waste valuable developer time if users are forced to run single-threaded builds, to constantly perform clean builds after changing files or to debug issues introduced by stale files. Our tools can be integrated into the workflows of developers, saving on time spent debugging or waiting for builds.

References

[1] S. I. Feldman, "Make a program for maintaining computer programs," *Software: Practice and experience*, vol. 9, no. 4, pp. 255–265, 1979.

[2] E. Martin, "Ninja, a small build system with a focus on speed," https://ninja-build.org, 2012.

[3] K. Martin and B. Hoffman, *Mastering CMake: a cross-platform build system*. Kitware, 2010.

[4] J. Calcote, *Autotools: A Practitioner's Guide to GNU Autoconf, Automake, and Libtool*. No Starch Press, 2010.

[5] S. Knight, "Scons design and implementation," in *10th International Python Conference*, 2002.

[6] E. S. Raymond, *The Art of Unix programming*. Addison-Wesley Professional, 2003.

[7] R. Levin and P. R. McJones, "The vesta approach to precise configuration of large software systems," in *SRC Research Report 105, Digital Equipment Corporation, Systems Research*. Citeseer, 1993.

[8] A. Mokhov, N. Mitchell, and S. Peyton Jones, "Build systems à la carte," *Proceedings of the ACM on Programming Languages*, vol. 2, no. ICFP, p. 79, 2018.

[9] S. McIntosh, B. Adams, and A. E. Hassan, "The evolution of ant build systems," in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*. IEEE, 2010, pp. 42–51.

[10] M. Shridhar, B. Adams, and F. Khomh, "A qualitative analysis of software build system changes and build ownership styles," in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2014, p. 29.

[11] S. McIntosh, B. Adams, T. H. Nguyen, Y. Kamei, and A. E. Hassan, "An empirical study of build maintenance effort," in *Proceedings of the 33rd international conference on software engineering*. ACM, 2011, pp. 141–150.

[12] G. Kumfert and T. Epperly, "Software in the doe: The hidden overhead of"the build"," Lawrence Livermore National Lab., CA (US), Tech. Rep., 2002.

[13] A. de Boor, "Bsd make meta mode," http://www.crufty.net/help/sjg/bmake-meta-mode.htm, 2013.

[14] P. Miller, "Recursive make considered harmful," *AUUGN Journal of AUUG Inc*, vol. 19, no. 1, pp. 14–25, 1998.

[15] Google, "Bazel - a fast, scalable, multi-language and extensible build system," https://bazel.build/, 2015.

[16] Facebook, "Buck: A fast build tool," https://buckbuild.com/, 2015.

[17] M. Gligoric, W. Schulte, C. Prasad, D. van Velzen, I. Narasamdya, and B. Livshits, "Automated migration of build scripts using dynamic analysis and search-based refactoring," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14. New York, NY, USA: ACM, 2014, pp. 599–616. [Online]. Available: http://doi.acm.org/10.1145/2660193.2660239

[18] M. Christakis, K. R. M. Leino, and W. Schulte, "Formalizing and verifying a modern build language," in *International Symposium on Formal Methods*. Springer, 2014, pp. 643–657.

[19] M. Kalaev, "Depslint," https://github.com/maximuska/depslint, 2013.

[20] B. Adams, H. Tromp, K. De Schutter, and W. De Meuter, "Design recovery and maintenance of build systems," in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*. IEEE, 2007, pp. 114–123.

[21] B. Adams, K. De Schutter, H. Tromp, and W. De Meuter, "The evolution of the linux build system," *Electronic Communications of the EASST*, vol. 8, 2008.

[22] J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen, "Fault localization for build code errors in makefiles," in *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 600–601.

[23] F. P. Miller, A. F. Vandome, and J. McBrewster, "Apache maven," 2010.

[24] B. Muschko, *Gradle in action*. Manning, 2014.

[25] R. Potvin and J. Levenberg, "Why googl stores billions of lines of code in a single repository," *Communications of the ACM*, vol. 59, no. 7, pp. 78–87, 2016.