A Model for Detecting Faults in Build Specifications

THODORIS SOTIROPOULOS, Athens University of Economics and Business, Greece
STEFANOS CHALIASOS, Athens University of Economics and Business, Greece
DIMITRIS MITROPOULOS, Athens University of Economics and Business, Greece
DIOMIDIS SPINELLIS, Athens University of Economics and Business, Greece

Incremental and parallel builds are crucial features of modern build systems. Parallelism enables fast builds by running independent tasks simultaneously, while incrementality saves time and computing resources by processing the build operations that were affected by a particular code change. Writing build definitions that lead to error-free incremental and parallel builds is a challenging task. This is mainly because developers are often unable to predict the effects of build operations on the file system and how different build operations interact with each other. Faulty build scripts may seriously degrade the reliability of automated builds, as they cause build failures, and non-deterministic and incorrect outputs.

To reason about arbitrary build executions, we present buildFS, a generally-applicable model that takes into account the specification (as declared in build scripts) and the actual behavior (low-level file system operation) of build operations. We then formally define different types of faults related to incremental and parallel builds in terms of the conditions under which a file system operation violates the specification of a build operation. Our testing approach, which relies on the proposed model, analyzes the execution of single full build, translates it into buildFS, and uncovers faults by checking for corresponding violations.

We evaluate the effectiveness, efficiency, and applicability of our approach by examining 612 Make and Gradle projects. Notably, thanks to our treatment of build executions, our method is the first to handle JVM-oriented build systems. The results indicate that our approach is (1) able to uncover several important issues (247 issues found in 47 open-source projects have been confirmed and fixed by the upstream developers), and (2) much faster than a state-of-the-art tool for Make builds (the median and average speedup is 39× and 74× respectively).

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Gradle, Make, parallel builds, incremental builds, JVM-based builds

ACM Reference Format:
Thodoris Sotiropoulos, Stefanos Chaliasos, Dimitris Mitropoulos, and Diomidis Spinellis. 2020. A Model for Detecting Faults in Build Specifications. 1, 1 (September 2020), 29 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Automated builds are an integral part of software development. Developers spend considerable amounts of time on writing and maintaining scripts [McIntosh et al., 2011, 2015] that implement the build logic of their project. Such scripts may involve the compilation of source files, application testing, and the construction of software artifacts such as libraries and executables. The advent of

Authors' addresses: Thodoris Sotiropoulos, Athens University of Economics and Business, Greece, theosotr@aueb.gr; Stefanos Chaliasos, Athens University of Economics and Business, Greece, schaliasos@aueb.gr; Dimitris Mitropoulos, Athens University of Economics and Business, Greece, dimitro@aueb.gr; Diomidis Spinellis, Athens University of Economics and Business, Greece, dds@aueb.gr.

XXXX-XXXX/2020/9-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Continuous Integration (CI) together with the complexity of modern software systems have made prominent two important properties related to automated builds: efficiency and reliability [Gligoric et al., 2014; Hilton et al., 2016; Vakilian et al., 2015; Visser et al., 2016]. To save computing resources and development time [Hilton et al., 2016; Licker and Rice, 2019], build tools must be capable of coping with complex systems quickly, but without sacrificing the reliability of the final deliverables. Following this direction, new build systems have emerged providing features such as parallelism [Bazel, 2020; Coetzee et al., 2011; Gligoric et al., 2014], caching [Gradle Inc., 2020a], incrementality [Erdweg et al., 2015; Konat et al., 2018], and the lazy retrieval of project dependencies [Celik et al., 2016].

Among these features, parallelism and incrementality are in the heart of almost every modern build system. Parallel builds reduce build times by processing independent build operations on multiple CPU cores. Incrementality saves time and resources by executing only those build operations affected by a specific change in the codebase. Both features are vital for a smooth development process, as they significantly shorten feedback loops [Konat et al., 2018; Visser et al., 2016]. For example, thanks to parallelism, building systems, which consist of million lines of code and thousands of source files, such as the Linux Kernel or LLVM, can be a matter of a few minutes.

Parallel and incremental builds though, pose threats to the reliability of the build process when they are not used with caution. Conceptually, a build is a sequence of tasks that work on some input files, and produce results (output files), potentially used by other tasks. To avoid failures and race conditions, developers must specify all dependencies in their build scripts, so that the underlying build system does not process dependent tasks in the wrong sequence or in parallel (e.g., linking before compilation is erroneous). Similarly, for correct incremental builds, developers need to enumerate all source files that a build task relies on. This ensures that after an update to a source file, all the necessary tasks are re-executed to generate the new build artifacts reflecting this change. Build scripts are susceptible to faults because declaring all task dependencies is a challenging and error-prone task [Licker and Rice, 2019; Morgenthaler et al., 2012; Vakilian et al., 2015]. Even best practices [GNU Make, 2020a], and tools [Martin and Hoffman, 2010] for managing dependencies automatically are often insufficient for preserving correctness [Licker and Rice, 2019]. Build failures, non-deterministic and inconsistent build outputs, or time-consuming builds, are inevitably the result of such faults [Licker and Rice, 2019; McIntosh et al., 2011].

There is little prior work focusing on detecting incorrect build definitions, and the existing approaches suffer from two major shortcomings [Bezemer et al., 2017; Licker and Rice, 2019] (as we discuss in 3.1) that prevent them from being useful in practice. First, the existing tools are tailored to analyze Make-based builds [Feldman, 1979] only, and applying the technique behind these tools to other build systems, such as JVM-based build tools, is not possible. One of the main reasons is that prior work makes strong assumptions about the internal behavior of build systems that are only relevant to Make builds. Unfortunately, there are not any techniques available to examine the reliability of builds originated from other systems beyond Make (e.g. Gradle), even though such build systems are extensively used [Hassan et al., 2017; McIntosh et al., 2011] and suffer from similar issues [Dashenkov, 2020; Greene, 2015]. Second, fault localization using prior methods requires a large amount of time that hinder their adoption. For example, employing *mkcheck* [Licker and Rice, 2019] to detect build-related issues in a Make-based project consisting of a hundred files can take hours (or even days).

We propose an *effective* and *efficient* dynamic method for detecting faults in parallel and incremental builds. Our method is based on a model (buildfs) that treats a build execution stemming from an *arbitrary* build system as a sequence of tasks, where each task receives a set of input files, performs a number of file system operations, and finally produces a number of output files. buildfs takes into account (1) the specification (as declared in build scripts) and (2) the definition (as

observed during a build through file accesses) of each build task. By combining the two elements, we formally define three different types of faults related to incremental and parallel builds that arise when a file access violates the specification of build. Our testing approach operates as follows. First, it monitors the execution of a build script, and models this execution in buildfs. Our method then verifies the correctness of the build execution by ensuring that there is no file access that leads to any fault concerning incrementality or parallelism. Note that to uncover faults, our method only requires a single full build.

We demonstrate the applicability of our approach on build scripts written in two popular build automation systems, namely, Make and Gradle. Make is one of the most well-established build tools [McIntosh et al., 2015], while Gradle is a modern JVM-based system that has become the de-facto build tool for Android and Kotlin programs [Derr et al., 2017; Karanpuria and Roy, 2018; Pelgrims, 2015]. Our approach is also applicable to other build systems such as Ninja, Bazel or Scala's *sbt*. To the best of our knowledge, our approach is the first treatment of JVM-oriented build executions.

Contributions. Our work makes the following contributions:

- We propose buildfs, a model for specifying and verifying arbitrary build executions, which is the key for applying our fault detection approach in both traditional (e.g., Make), and modern (e.g., Gradle) build systems (Section 3).
- We introduce a dynamic method that relies on buildfs, and is able to uncover issues in parallel and incremental builds by analyzing the execution of a single clean build. (Section 4).
- We evaluate the effectiveness and the applicability of our approach by detecting issues in 324 out of 612 Make and Gradle projects. Notably, 237 issues found in 47 open-source projects were confirmed and fixed by upstream developers. Furthermore, our approach is more effective, and 74× faster than the state-of-the-art, on average, when analyzing Make projects (Section 5).

Availability. Our system is available as open-source software under the GNU General Public License v3.0 at https://github.com/theosotr/buildfs. The research artifact is available at https://github.com/theosotr/buildfs-eval. A DOI URL will be also provided in the camera-ready version of the paper.

2 BACKGROUND

We provide the basic elements of Make and Gradle. Then, we discuss the types of fault that may occur in the corresponding scripts and are related to incremental and parallel builds.

2.1 Build Systems

Make. Make is the oldest build system used today [Feldman, 1979; Licker and Rice, 2019]. It provides a Domain-specific Language (DSL) that allows developers to write definitions of rules that instruct the system how to build certain targets. For example, the following rule states that building the target `source.o`, which depends on the file `source.c`, requires to invoke the `gcc` command (line 2):

```
1  source.o: source.c
2      gcc -c $^
```

By default, Make builds every target incrementally, meaning that it generates targets only when they are missing or when their dependent files are more recent than the target. Make uses file timestamps to determine whether a file has changed or not. Also, it provides some built-in variables starting with the symbol "$". The most common ones are `$@` and `$^`, which refer to the name of the target (e.g., `source.o`), and the dependencies (e.g., `source.c`) of the current rule respectively. Developers write their Make rules in files called *Makefiles*. In particular, developers can either

```
1  CXXFLAGS=-MD
2  OBJS=CMetricsCalculator.o QualityMetrics.o
3  qmcalc: $(OBJS) qmcalc.o
4      gcc -g qmcalc.o $(OBJS) -o $@
5  -include $(OBJS:.o=.d)
```

Fig. 1. A Make definition that does not capture the dependencies of the object file `qmcalc.o`.

write their own Makefiles, or use higher-level tools, such as CMake [Martin and Hoffman, 2010] or GNU Autotools [Calcote, 2020], that automatically generate Makefile definitions. CMake offers its own DSL, and enables programmers to write rules which in turn are translated into Makefiles. CMake is useful for managing systems with complex structure. Autotools is a collection of tools that configure and generate Makefiles from templates.

Gradle. Although newer than other JVM-based build tools, such as Ant and Maven, Gradle has gained much popularity recently. Currently, around 55% of the most popular Java Github projects use Gradle [Hassan et al., 2017], and it has become the preferred build tool for Kotlin and Android programs [Derr et al., 2017; Karanpuria and Roy, 2018; Pelgrims, 2015]. Gradle is at least two times faster than Maven [Gradle Inc., 2020c], as it offers features, such as parallelism, and a build cache.

Gradle provides a Groovy- and a Kotlin-based DSL which adopts a task-based programming model. In this sense, Gradle programmers assemble build logic in a set of tasks. A task is a fundamental component in Gradle that describes a piece of work needed to be done as part of a build. Developers can impose constraints on the execution order of tasks. Then, Gradle represents the build workflow as a directed acyclic graph and processes every task in topological ordering. To enable incremental builds, developers need to enumerate the files consumed and produced by each task. In this context, a task is executed only when there is a change to any of its input or output files. Gradle adopts a content-based approach to identify updates: it compares the checksum of the input / output files with that coming from the last build. Consider the following snippet:

```
1  task extractZip {
2     inputs.file "/file.zip"
3     outputs.dir "/extractedZip"
4     from zipTree("/file.zip")
5     into "/extractedZip"}
```

The listing above demonstrates a task named `extractZip` written in Gradle. This task extracts the contents of an archive, namely `/file.zip`, into the directory `/extractedZip`. The input and the output files of this task are declared at lines 2 and 3 respectively. Declaring the input / output makes the task `extractZip` incremental. In this context, Gradle re-executes this task only when any of those files are modified. Notice that an input or an output file can be a directory (See line 3). In this case, Gradle recursively examines the contents of the directory for updates.

Gradle provides a rich API that developers can rely on to customize their builds or create plugins. A plugin consists of a set of common tasks that can be reused across multiple projects, e.g., consider a plugin that applies a linter to the source files of a project. Up to now, there are more than 3,600 Gradle plugins available for use [Gradle Inc., 2020d].

2.2 Faults in Incremental & Parallel Builds

Three types of faults can occur due to incorrect build definitions: *missing inputs*, *missing outputs*, and *ordering violations*. The first two are associated with incremental builds, while the last one concerns parallelism.

Missing Inputs. A build definition manifests a missing input issue, when a developer fails to define all input files of a particular build task. This leads to faulty incremental builds because

```
1  apply plugin: "java"
2  apply plugin: "com.github.johnrengelman.shadow"
3  shadowJar { classifier = "" }
```

Fig. 2. A Gradle script with an ordering violation.

whenever there is an update to any of the missing input files, the dependent build task is not executed by the build system. Consequently, the build system produces stale targets and outputs.

As an example, consider the fragment of a Make definition taken from the `cqmetrics` project, illustrated in Figure 1. This build creates the executable `qmcalc` by linking the object files `CMetricsCalculator.o`, `QualityMetrics.o`, and `qmcalc.o` (lines 3–4). Every object file is created by a built-in Make rule that compiles each implementation file `.c` with a command of the form `$(CC) $(CXXFLAGS) -c`. By default, the input file of these built-in rules is only the underlying implementation file, e.g., the input file of the rule `qmcalc.o` is `qmcalc.c`. However, an object file might also depend on a set of header files. Thus, changing a dependent header file requires the re-generation of the object file. The developers tackle this issue by compiling every object file with the `-MD` flag (line 1). This flag stores all header files that a target relies on into a dedicated dependency file whose suffix is `.d`. The developers include these dependency files in their Makefile on line 5. Although compiling source files with `-MD` follows the best practices for managing Make dependencies automatically [GNU Make, 2020a], the above script is faulty, because only the dependency files of the object files included in the variable `$OBJS` (line 2) are considered. The issue here is that when there is an update to a header file that the rule `qmcalc.o` depends on, the object file is not re-created. Thus, the final executable `qmcalc` can be linked with stale object files. Note that we have identified the aforementioned issue by using our approach. In addition, we reported the fault to the upstream developers who confirmed and fixed it.

Missing Outputs. Missing output faults are similar to missing input faults. However, this time the cause of the problem is that a developer does not properly enumerate the output files of a task. As with missing inputs, this issue makes incremental builds skip the execution of some build tasks even if their outputs have changed. Note that missing outputs do not appear in Make builds, because Make considers only the timestamp of input files to decide if a target rule must be re-executed.

In case of Gradle, missing outputs also affect the efficiency of a build. Gradle caches the output files of a task from previous builds, and reuses them in subsequent ones when input files remain the same. Hence, missing outputs make a Gradle build run slower. Notably, this build cache is an important Gradle feature that makes Gradle much more efficient than other build systems [Gradle Inc., 2020c].

Beyond preserving the correctness and efficiency of incremental builds, Gradle also uses the declared inputs and outputs of tasks to improve the reliability of parallel builds by implicitly specifying ordering constraints between dependent tasks. In particular, Gradle examines the declared outputs / inputs of tasks and adds an *auto-dependency* between a task that creates a file (this file is declared as an output of the task), and another task that consumes the same file (the task declares this file as its input). This prevents Gradle from running two conflicting tasks (e.g, the one produces something consumed by the other) in parallel.

Ordering Violations. Every build tool supporting parallelism runs independent tasks *non-deterministically*. This means that the build system is free to process unrelated operations in any order for achieving high performance. Non-determinism does not cause any problems to the build process, when two tasks are indeed independent, and the one does not depend on the other. However, race conditions emerge, when two tasks are conflicting, but are executed concurrently. Developers can introduce ordering constraints in their build definitions as a side effect of explicitly defining

dependencies among conflicting build tasks. An ordering violation occurs when a developer does not specify ordering constraints between two dependent tasks. Note that an ordering violation does not relate to the incremental issues discussed above. That is, there can be a task declared with the correct input / output relations, but it races with another conflicting task.

Figure 2 shows an example of an ordering violation. Here we have an excerpt of a real-world Gradle script (from the `nf-tower` project) whose goal is to create the fat JAR of a Java application. A fat JAR packages all `.class` files of the current project along with the `.class` files of project dependencies, forming the executable distribution of the project. The code first applies the built-in Gradle plugin `"java"` (line 1). This plugin—among other things—runs two tasks: (1) the task `classes` that compiles all Java files into their corresponding `.class` files, and (2) the task `jar` that generates a JAR file containing only the classes of the current project. In turn, the code employs an external plugin (line 2) containing the task `shadowJar` (line 3) that eventually generates the fat JAR of the project. The problem here is that the name of the fat JAR generated by the task `shadowJar` conflicts with the name of the naive JAR produced by the task `jar`. The tasks `jar` and `shadowJar` do not depend on each other, so Gradle is free to schedule `jar` after `shadowJar`. This erroneous ordering results in incorrect output, i.e., the task `jar` overrides the contents of the JAR file produced by the task `shadowJar`. A fix to this problem is to create a fat JAR with a different name (e.g, changing its classifier at line 3 to `"-all"`). Through our work we have identified this issue and reported it to the the developers of `nf-tower` who confirmed fixed it.

As we will see in Section 5, such issues are widespread and affect the reliability of many software deliverables. This motivates the design of a generic approach for easing the adoption of incremental and parallel builds in practice.

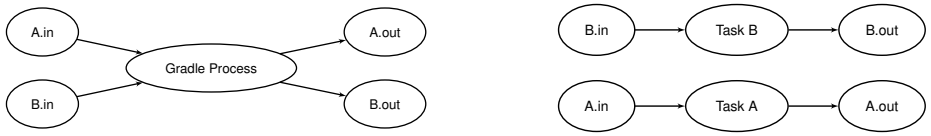
3 A MODEL FOR BUILD EXECUTIONS

Designing a technique that is able to locate faults in incremental and parallel builds regardless of the underlying build system requires a generally-applicable and precise model for reasoning about build executions. Existing models make assumptions that are relevant to a specific target of builds. As a result, the testing approaches that rely on these models become *ineffective* when applied to build tools not satisfying these assumptions (Section 3.1). To address this, we propose buildFS, a model for understanding build executions (Section 3.2). Then, we introduce the notion of task graph (Section 3.3), a component that serves as a basis for ensuring the correctness of a build execution (Section 3.4).

The buildFS model is the core component of our testing approach for builds. Our approach, which we present in Section 4, utilizes buildFS in the following manner: it first monitors the execution of a build (written in an arbitrary build system), and models it in buildFS. Then, based on the notion of correctness for build executions (Section 3.4), our approach verifies whether the examined build is incorrect, and then reports all violations (if any).

3.1 Motivation

Prior work on detecting faults in Make incremental builds, namely mkcheck [Licker and Rice, 2019], models build execution as a set of system processes created by the build system during execution, e.g., Make creates a new `gcc` process for compiling every source file. This model treats every process as a function that takes an input, and produces an output. The input stands for the set of files that are read by the process, while the output is the set of files written by it. The inputs and outputs of every process are computed by analyzing the system call trace of a build. Through this model, they infer the inter-dependencies among files by considering each output to be dependent on every input file. All dependencies are then transitively propagated using the process hierarchy. However, modeling build execution as a set of system processes is problematic for the following reasons.



(a) Overconstrained graph (b) Precise graph

Fig. 3. Overconstrained dependency graph computed by an approach modeling build execution as sequence of processes vs. precise graph produced by an approach modeling build execution as a sequence of tasks.

Low Precision. The main assumption made by Make-based tools is that the build system *always* spawns a separate process when proceeding to a new build task. However, this assumption is *no* longer valid in modern build systems such as Gradle, Maven, or Scala's sbt, where the same system process (e.g., JVM process) involves multiple build tasks. Tools that model builds as a sequence of processes become ineffective when applied to such build systems, as their analysis precision significantly drops.

To highlight how this feature affects the precision of existing work we provide a representative example. Consider a Gradle task `A` that reads a file `A.in` and creates a file `A.out`, and a Gradle task `B` with file `B.in` as its input, and `B.out` as its output. An approach that works on granularity of processes produces the dependency graph of Figure 3a. The main Gradle process runs both tasks A and B; therefore, the analysis considers files `A.in` and `B.in` as the inputs of that process, and files `A.out` and `B.out` as its outputs. Conceptually, this *merges* the two tasks into a single task. The resulting graph is overconstrained [Licker and Rice, 2019], because the analysis *over-approximates* the set of dependencies. For example, when there is a change in the input file `A.in`, the analysis incorrectly considers that both output files (i.e., `A.out` and `B.out`) must be updated, even if `A.in` only affects `A.out`. Overconstrained graphs lead to dozens of false positives and negatives [Licker and Rice, 2019].

Efficiency and Applicability. Another core limitation of existing work is that it only captures OS-level facts (e.g., file accesses and file dependencies) which are computed while analyzing the build trace. To verify the inferred file dependencies against the specification of build scripts, prior work (i.e., mkcheck) triggers incremental builds by touching each source file, and checking whether the expected output files are re-generated in response to the updated input files. This makes the verification task extremely slow as it requires substantial resources when applying multiple incremental builds in large-scale projects [Licker and Rice, 2019] (see also Section 5.6).

A critical reader may think that combining static analysis with dynamic analysis is a workaround for this efficiency issue [Bezemer et al., 2017]. Specifically, another approach could perform static analysis on build scripts to extract task specification, and then compare this specification against the actual behavior of task observed during build execution. Nevertheless, reliably extracting task specifications from build scripts through static analysis is particularly challenging (and in many cases not possible) for multiple reasons [Licker and Rice, 2019]. First, static analysis cannot reason about tasks whose inputs / outputs are dynamically computed and are not known in build scripts. The same applies for tasks not explicitly mentioned in build scripts, e.g., tasks defined in external Gradle plugins as illustrated in Section 2.2. Second, static analysis needs to reason about the complex semantics of build system's DSL. This limits generalizability, as applying the approach to a new build system requires implementing a new static analyzer which involves a lot of engineering effort. Third, even when a static analyzer is available, OS-level facts (inferred dynamically by the existing model) are not comparable with task specifications (computed statically), when the build system does not follow the *target-prerequisites* pattern for expressing custom tasks, as in the case of Gradle or Scala's sbt. To further clarify this, consider the following example.

```
task A {
  inputs.file "/file/A"
  // Arbitrary operation...
}
task B {
  inputs.files ("/file/A", "/file/B")
  // Arbitrary operation...
}
```

```
open("/file/A")
... // other file system operations
open("/file/B")
... // other file system operations
open("/file/A")
```

In the Gradle script on the left, we have two incremental tasks (task `A` and `B`) performing some arbitrary operations. The specification of the task `A` says that this task is expected to consume the file `/file/A`, while the task `B` reads the files `/file/A` and `/file/B`. Note that the specification only indicates the intent of the developer, and *not* the actual interactions of task with the system. The latter is shown in the execution trace on the right. In this scenario, it is not possible to compare the actual behavior of tasks (inferred by analyzing the execution trace on the right) against build specification (extracted statically from the build script on the left). This is because existing dynamic analysis techniques are unable to map the file accesses shown on the right to the task they belong to. This is necessary for deciding correctness. For example, if the first access comes from task `A` while the remaining ones stem from task `B`, the build script is not faulty. On the other hand, if it is the other way around (i.e., the last two accesses belong to the first task), the task `A` manifests a missing input on file `/file/B`, as it consumes a file not mentioned in the build script.

Make adopts the target-prerequisites pattern (i.e., each Make task is uniquely identified by a target file), and this is why it is supported by the existing work. Specifically, it is easy for the existing work to verify the dynamic file dependencies against the static build specification as both are expressed in terms of file paths.

3.2 Modeling Builds

All the points discussed in the previous section are fundamental issues associated with the method's design and underlying model, and *not* with its implementation. We introduce BuildFS, a model for thinking about build executions that addresses the main limitations of existing work.

The proposed model treats every build as a sequence of *tasks* rather than system processes. Every task corresponds to the execution of a build operation. For example, a task in BuildFS stands for the execution of a target rule in Make and Ninja, a goal in Java Maven, or a Gradle task in Gradle. This tackles low precision introduced by prior work, because it enables us to relate every build task to its correct input and output files regardless of the internal behavior of the build tool (e.g., whether it spawns a separate process or not). For example, unlike the overconstrained dependency graph of Figure 3a, BuildFS allows us to infer the precise graph shown in Figure 3b. BuildFS separates file accesses based on which task they belong to. Therefore, it does not perform unnecessary merges when encountering tasks governed by the same process, which is the main source of imprecision in previous work [Licker and Rice, 2019].

For dealing with efficiency and applicability, BuildFS provides each task with a specification that consists of (1) a set of files that the task is expected to consume, (2) a set of files that the task is expected to produce, and (3) a set of task dependencies. A task dependency indicates that a task depends on another, i.e., it is executed only after the dependent task. Beyond specification, every task has a definition containing all the (low-level) file system operations performed while executing the task, e.g., reading and writing files, or changing the OS transient structures, such as the file descriptor table. Combining the (high-level) specification and actual behavior (low-level file system operations) of each task makes our approach efficient and applicable, for we can verify correctness

$$
\begin{array}{rcll}
b \in \mathit{Build} & ::= & t^* & \texttt{[build execution]} \\
t \in \mathit{Task} & ::= & \texttt{task}\ \tau\ k\texttt{:}\ k\ \texttt{after}\ d = s & \texttt{[task]} \\
d \in \mathit{Dep} & ::= & \bot \mid \tau \mid (\tau \ldots) & \texttt{[task deps]} \\
k \in \mathit{FileSpec} & ::= & p \mid (p \ldots) \mid \bot \mid \top & \texttt{[file spec]} \\
s \in \mathit{Statements} & ::= & \texttt{sysOp in}\ z = o & \texttt{[operation]} \\
 & & \mid \texttt{newproc}\ z & \texttt{[new process]} \\
 & & \mid \texttt{newproc}\ z\ \texttt{from}\ z & \texttt{[fork]} \\
 & & \mid s\texttt{;}s & \texttt{[compound stmt]} \\
o \in \mathit{Op} & ::= & \texttt{fd}_f = e & \texttt{[create fd]} \\
 & & \mid \texttt{del(fd}_f\texttt{)} & \texttt{[destroy fd]} \\
 & & \mid \texttt{consume(}e\texttt{)} & \texttt{[consume path]} \\
 & & \mid \texttt{produce(}e\texttt{)} & \texttt{[produce path]} \\
 & & \mid o\texttt{;}o & \texttt{[compound op]} \\
e \in \mathit{Expr} & ::= & p & \texttt{[path]} \\
 & & \mid \texttt{fd}_f & \texttt{[fd]} \\
 & & \mid p\ \texttt{at}\ e & \texttt{[}p\ \texttt{relative to}\ e\texttt{]} \\
p \in \mathit{Path} & ::= & \textit{is the set of paths} & \\
\tau \in \mathit{TaskName} & ::= & \textit{is the set of task names} & \\
f \in \mathit{FileDesc} & ::= & \textit{is the set of file descriptors} & \\
z \in \mathit{Proc} & ::= & \textit{is the set of processes} &
\end{array}
$$

(a) Syntax

$$
\begin{aligned}
r \in \mathit{FileAcc} &= \mathcal{P}(\mathit{Path}) \times \mathcal{P}(\mathit{Path}) \\
\pi \in \mathit{ProcFD} &= \mathit{FileDesc} \hookrightarrow \mathit{Path} \\
\sigma \in \mathit{State} &= \mathit{Proc} \hookrightarrow \mathit{ProcFD}
\end{aligned}
$$

(b) Domains

$$
\begin{aligned}
r_{\downarrow c} = r_{\downarrow 1}\ \ r_{\downarrow p} &= r_{\downarrow 2} \\
[\![e]\!] &\in \pi \to \mathit{Path} \\
[\![p]\!]_\pi &\Rightarrow p \\
[\![\texttt{fd}_f]\!]_\pi &\Rightarrow \pi(f) \\
[\![p\ \texttt{at}\ \texttt{fd}_f]\!]_\pi &\Rightarrow \textsc{join}(\pi(f), p) \\
[\![o]\!] &\in \pi \times r \to \pi \times r \\
[\![\texttt{fd}_f = e]\!]_{\pi,r} &\Rightarrow (\pi[f \to [\![e]\!]_\pi], r) \\
[\![\texttt{del(fd}_f\texttt{)}]\!]_{\pi,r} &\Rightarrow (\pi[f \to \bot], r) \\
[\![\texttt{consume(}e\texttt{)}]\!]_{\pi,r} &\Rightarrow (\pi, (r_{\downarrow c} \cdot [\![e]\!]_\pi\ ,\ r_{\downarrow p})) \\
[\![\texttt{produce(}e\texttt{)}]\!]_{\pi,r} &\Rightarrow (\pi, (r_{\downarrow c}\ ,\ r_{\downarrow p} \cdot [\![e]\!]_\pi)) \\
[\![o_1; o_2]\!]_{\pi,r} &\Rightarrow [\![o_2]\!]_{\pi',r'}\ \ (\pi', r') = [\![o_1]\!]_{\pi,r} \\
[\![s]\!] &\in \sigma \times r \to \sigma \times r \\
[\![\texttt{sysOp in}\ z = s]\!]_{\sigma,r} &\Rightarrow (\sigma[z \to \pi], r)\ \ (\pi, r) = [\![s]\!]_{\sigma(z),r} \\
[\![\texttt{newproc}\ z]\!]_{\sigma,r} &\Rightarrow (\sigma[z \to \bot], r) \\
[\![\texttt{newproc}\ z_1\ \texttt{from}\ z_2]\!]_{\sigma,r} &\Rightarrow (\sigma[z_1 \to \sigma(z_2)], r) \\
[\![s_1; s_2]\!]_{\sigma,r} &\Rightarrow [\![s_2]\!]_{\sigma',r'}\ \ (\sigma', r') = [\![s_1]\!]_{\sigma,r} \\
[\![t]\!] &\in \sigma \to \sigma \times r \\
[\![\texttt{task}\ \tau\ k_1\texttt{:}k_2\ \texttt{after}\ d = s]\!]_\sigma &\Rightarrow [\![s]\!]_{\sigma,\bot}
\end{aligned}
$$

(c) Semantics

Fig. 4. The syntax for representing build executions (a), the domains of buildfs (b), and the semantics of buildfs expressions, operations, statements, and tasks (c).

(i.e., whether the actual behavior conforms to the specification), by analyzing the execution of a single clean build (i.e., no need to run incremental builds or static analysis on build scripts).

Figure 4 shows the complete model for build executions. A build execution $b = \langle t^1, t^2 \ldots \rangle$ consists of a sequence of tasks. Every task t is described by a unique name ($\tau \in \mathit{TaskName}$), and contains a specification and a definition. The specification declares the input / output files and the dependencies of each task, while its definition consists of statements. For example, `task A (/file/in): /file/out after` $\bot = s$ means that the task named `A` consumes the file `/file/in`, produces the file `/file/out`, has no dependencies (`after` $\bot$), while its definition is given by s. The symbol $\bot$ is used to indicate the absence of a value, while $\top$ is used as a wildcard symbol representing any value. For example, when we say that the declared output of a task is $\top$, we mean that this task is valid to produce *any* file.

A definition is one or more statements. There are two types of statements. First, the `sysOp in` $z = o$ statement executes a system operation o in a process given by z. Every process defines a scope for file descriptor variables ($\texttt{fd}_f$), where each file descriptor variable points to a file path. An operation ($o \in \mathit{Op}$) executed inside a process z may introduce new file descriptor variables in the current process (scope) ($\texttt{fd}_f = \ldots$), or delete existing ones (`del`). Moreover, an operation may perform various file system updates, including file creation (`produce`) and file consumption (`consume`). An expression ($e \in \mathit{Expr}$) can be a constant path, a file descriptor variable, or p `at` e. The latter allows us to interpret the path p relative to the path given by the e expression (note that the result of an expression is a path). Finally, the `newproc` z_1 statement creates a fresh process (scope) z_1, and it

```
# Copying file using a Make rule.
target: "/source"
    cp $^ $@
```

```
// Copying file using a Gradle task.
task target {
  inputs.file "/source"
  outputs.file "/target"
  from file("/source")
  into file("/target") }
```

Fig. 5. Copying the contents of source into target.

```
1  task target ("/source"): "/target" after
       ⊥ =
2    newproc p
3    sysOp in p =
4      fd3 = "/source"
5      consume(fd3)
6      fd4 = "/target"
7      produce(fd4)
8      del(fd4)
9      del(fd3)
```

Fig. 6. Modeling the execution of the task target that stems from Make and Gradle scripts of Figure 5.

optionally copies all file descriptor variables of an existing process z_2 to z_1 (newproc z_1 from z_2). This models process forking.

As an example of modeling, consider a simple build scenario where we want to copy the contents of the file /source into the file /target. Figure 5 shows how we can express this using Make and Gradle. When we execute these build scripts, the build system first opens the file /source, reads its contents, then opens the file /target, and finally writes the contents of /source to the file descriptor corresponding to the second file. Figure 6 illustrates how we model the execution stemming from these scripts. Every build consists of a single task named target. This task consumes /source to create /target. The definition of the task target creates a new process (line 2) and uses it to execute all the file-related operations performed when running Make and Gradle (lines 3–9). For instance, the operation $\texttt{fd}_3$ = /source creates a new file descriptor (in the current process) pointing to file /source, while the operation at line 5 consumes this file descriptor. These two operations model file opening. On the other hand, the operation $\texttt{del(fd}_4\texttt{)}$ deletes the given file descriptor once the task closes the corresponding file (line 8).

The semantics of buildfs are shown in Figure 4. Every task $t \in \mathit{Task}$ is evaluated on a state $\sigma \in \mathit{State} = \mathit{Proc} \hookrightarrow \mathit{ProcFD}$. A state is a map that provides the file descriptor table (*ProcFD*) of every process. A file descriptor table $\pi \in \mathit{ProcFD} = \mathit{FileDesc} \hookrightarrow \mathit{Path}$ is a map that provides the file path that each file descriptor of a process points to. Conceptually, the state models the file descriptor feature of POSIX-compliant operating systems. The result of a task evaluation $[\![t]\!]_\sigma \in \sigma \to \sigma \times r$ is a new state along with the file accesses performed by this task. An instance of the domain of file accesses $r \in \mathit{FileAcc} = \mathcal{P}(\mathit{Path}) \times \mathcal{P}(\mathit{Path})$ is a pair that contains the set of files consumed and produced by the task. The first element of the pair is the set of consumed files, while the second one stands for the set of produced files. For convenience, we also define the projections $r_{\downarrow_c}$ and $r_{\downarrow_p}$ that give the set of files consumed, and produced by the task respectively. Statements, operations, and expressions are evaluated accordingly. Notably, operations and expressions are evaluated using the file descriptor table (i.e., $\pi \in \mathit{ProcFD}$) of the process where they take place. As an example, after evaluating the following buildfs task on the state $\sigma = \bot$, we get a new state $\sigma' = z_1 \to (1 \to \texttt{"/f1"}, 2 \to \texttt{"/f1/f3"})$, while the set of consumed files $r_{\downarrow_c}$ is {"/f1/f3"}, and the set of produced files $r_{\downarrow_p}$ is {"/f2/f4"}.

```
task τ ("/f1"): "/f2" after ⊥ =
  newproc z1
  sysOp in z1 =
    fd1 = "/f1"
    fd2 = "f3" at fd1
    consume(fd2)
    produce("f4" at "/f2")
```

```
1  task t1 ("/f1"): "/f2"
2    after ⊥ = s1
3  task t2 ("/f2"): ⊥
4    after t1 = s2
5  task t3 ("/f3", "/f4"): ("/f2", "/f5")
6    after ⊥ = s3
```

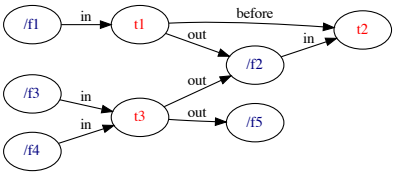


Fig. 7. An example of a build execution and its task graph.

3.3 Task Graph

We introduce the notion of *task graph*. The task graph is a component that stores the input files, output files, and the dependencies of every task as declared by the developers in build scripts. The task graph is computed by traversing the specification of every task found in a BUILDFS program (Section 3.2), and collecting all input / output files and task dependencies. We later use the task graph for ensuring the correctness of a build execution (Section 3.4).

We define the task graph as $G = (V, E)$. A node $v \in V$ in the task graph is either a task $t \in \mathit{Task}$ or a file $p \in \mathit{Path}$. The set of edges $E \subseteq V \times V \times L$, where $L = \{\texttt{in}, \texttt{out}, \texttt{before}\}$, determine the following relationships. Given a task graph G, the edge $p \xrightarrow{\texttt{in}}_G t$ indicates that the file p has been declared as an input of the task t. The edge $t \xrightarrow{\texttt{out}}_G p$ states that the task t produces the file p. Finally, the edge $t_1 \xrightarrow{\texttt{before}}_G t_2$ shows a task dependency, i.e., the execution of t_1 precedes that of t_2.

Given a build execution modeled as a BUILDFS program (Figure 4), we gradually compute the task graph by inspecting the specification of every task entry. The $\xrightarrow{l}$ edges, where $l \in \{\texttt{in}, \texttt{out}, \texttt{before}\}$, are constructed by examining the header part of a `task` construct. For instance, whenever we encounter a task entry of the form $\texttt{task}\ t_1\ (p_1)\texttt{:}\ p_2\ \texttt{after}\ t_2$, we add the following edges to the task graph G: (1) an $p_1 \xrightarrow{\texttt{in}}_G t_1$ edge, (2) an $t_1 \xrightarrow{\texttt{out}}_G p_2$ edge, and (3) an $t_2 \xrightarrow{\texttt{before}}_G t_1$ edge.

A complete example is shown in Figure 7, where we have a build execution in BUILDFS on the left, and its resulting task graph on the right. Red nodes denote tasks while blue nodes indicate files.

3.4 Correctness of Build Executions

Having proposed our model for build executions and the concept of task graph, we now formalize the property of correctness for build executions. To do so, we exploit the task graph and define the *subsumption* and *happens-before* relations that we use as a base for verifying correctness.

Definition 3.1. (Subsumption). Given a task graph G, we define the reflexive, binary relation $\sqsubseteq_G$ and its transitive closure $\sqsubseteq_G^+$ on two file paths $p_1, p_2 \in \mathit{Path}$. The definition is shown in Figure 8.

The subsumption relation $p_1 \sqsubseteq_G p_2$ says that the file path p_1 is subsumed within the file path p_2. This relation is reflexive ([SELF]), and for every file path $p \in \mathit{Path}$, we have $p \sqsubseteq_G \top$. The relation $p_1 \sqsubseteq_G p_2$ holds when p_2 is the parent directory of p_1 ([PAR-DIR]), or when p_2 relies on p_1, i.e., there is at least one task in the task graph G that produces p_2 using p_1 ([INDIRECT]). As we will see later the subsumption relation is important for ensuring that a file access made while executing a build task matches the task's specification.

Definition 3.2. (Happens-Before). Given a task graph G, we define the binary relation $\prec_G$ and its transitive closure $\prec_G^+$ on two tasks $t_1, t_2 \in \mathit{Task}$. The definition is shown in Figure 9.

The happens-before relation $t_1 \prec_G t_2$ states that the task t_1 is executed before t_2. The definition of this relation consults the task graph G to identify tasks that are connected with each other through

$$\textsc{self}\ \frac{p \in \mathit{Path}}{p \sqsubseteq_G p} \qquad \textsc{top}\ \frac{p \in \mathit{Path}}{p \sqsubseteq_G \top} \qquad \textsc{par-dir}\ \frac{p_1, p_2 \in \mathit{Path} \qquad \textsc{isParentDir}(p_1, p_2)}{p_1 \sqsubseteq_G p_2}$$

$$\textsc{indirect}\ \frac{p_1, p_2 \in \mathit{Path} \qquad t \in \textsc{getNodes}(\textsc{g}) \qquad p_1 \sqsubseteq_G p_1' \qquad p_1' \overset{\text{in}}{\rightarrow}_G t \qquad t \overset{\text{out}}{\rightarrow}_G p_2}{p_1 \sqsubseteq_G p_2}$$

$$\textsc{trans-clos}\ \frac{p_1 \sqsubseteq_G p_2}{p_1 \sqsubseteq_G^+ p_2} \qquad \textsc{trans-clos}\ \frac{p_1 \sqsubseteq_G^+ p_2 \qquad p_2 \sqsubseteq_G^+ p_3}{p_1 \sqsubseteq_G^+ p_3} \qquad \textsc{mul}\ \frac{m = (p_1, p_2 \ldots) \qquad \exists p' \in m.\ p \sqsubseteq_G^+ p'}{p \sqsubseteq_G^+ m}$$

Fig. 8. Definition of the $\sqsubseteq_G$ relation through inference rules.

$$\textsc{dep}\ \frac{t_1, t_2 \in \mathit{Task} \qquad t_1 \overset{\text{before}}{\rightarrow}_G t_2}{t_1 \prec_G t_2} \qquad \textsc{trans-clos}\ \frac{t_1 \prec_G t_2}{t_1 \prec_G^+ t_2} \qquad \textsc{trans-clos}\ \frac{t_1 \prec_G^+ t_2 \qquad t_2 \prec_G^+ t_3}{t_1 \prec_G^+ t_3}$$

Fig. 9. Definition of the $\prec_G$ relation through inference rules.

an $\overset{\text{before}}{\rightarrow}_G$ edge, which indicates a dependency between two tasks. Finally, the transitive closure of $\prec_G$ gives indirect task dependencies. The happens-before relation enables us to verify that two dependent tasks are always executed in the correct order.

3.4.1 Verifying Correctness of Tasks. Using the subsumption relation, we now formalize what the property of correctness means for a BUILDFS task.

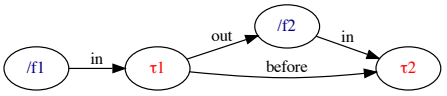
Definition 3.3. (Missing Input). Given a task graph G and a state σ, a task $t \in \mathit{Task} = \texttt{task}\ \tau\ k_1$: k_2 `after` $d = s$ manifests a *missing input* on state σ, when

- $(\sigma', r) = [\![t]\!]_\sigma$
- $\exists p \in r_{\downarrow_c}\ p \not\sqsubseteq_G^+ k_1$

In other words, to verify that a task does not contain a missing input issue, we first compute all file accesses made by the task on the given state σ, i.e., $[\![t]\!]_\sigma$. Then, we check that every file consumed by this task (i.e., $r_{\downarrow_c}$) matches the input files k_1 declared in the specification. To do so, we exploit the subsumption relation. In particular, when there exists a file path p consumed by this task for which $p \not\sqsubseteq_G k_1$, we say that the task has a missing input on the state σ. In practice, this means that although the build task relies on p (as the definition of task consumes p), the build system does not trigger the execution of the task, whenever p is modified.

Example. Consider the following build execution and its task graph.

```
1  task τ1 ("/f1"): "/f2" after ⊥ =
2    newproc z1
3    sysOp in z1 =
4      consume "/f1/f3"
5      produce "/f2"
6  task τ2 ("/f2"): ⊥ after τ1 =
7    sysOp in z1 =
8      consume "/f2"
9      consume "/f1/f3"
10     consume "/f3"
```



When examining the task τ_1, we presume that it does not contain any missing input issues, as it only consumes the file `"/f1/f3"` (line 4) which is subsumed within the input file `/f1` declared at line 1 (recall the [PAR-DIR] rule from Figure 8). On the other hand, the task τ_2 consumes three

files (lines 8–10). For the first access (line 8), the $\sqsubseteq_G^+$ relation holds, as the path consumed by τ_2 is the same with that declared in the specification. The subsumption relation also holds for the second access, as the file `f1/f3` is an input of the first task τ_1 whose output is used as an input for the target task τ_2. Therefore, changing this file will first trigger the execution of task τ_1. This will eventually cause the invocation of task τ_2, because the first task τ_1 updates the inputs of τ_2. This behavior is captured by the [INDIRECT] rule (Figure 8). Finally, the task τ_2 manifests a missing input for the third access (line 10), because we have `/f3` $\not\sqsubseteq_G^+$ `/f2`.

Definition 3.4. (Missing Output). Given a task graph G and a state σ, a task $t \in \mathit{Task} = \texttt{task}\ \tau\ k_1$: k_2 `after` $d = s$ manifests a *missing output* on state σ, when

- $(\sigma', r) = [\![t]\!]_\sigma$
- $\exists p \in r_{\downarrow_p}\ p \not\sqsubseteq_G^+ k_2$

The definition for missing outputs is conceptually similar to that for missing inputs. This time however, we check that for every file $p \in r_{\downarrow_p}$ produced by the examined task, the $p \sqsubseteq_G^+ k_2$ relation holds, where k_2 stands for the declared output files found in the specification of the task.

Given the above definitions, we now introduce the notion of correctness for a certain BUILDFS task.

Definition 3.5. (Correctness of Task). Given a task graph G and a state σ, a task $t \in \mathit{Task}$ is correct on state σ, when it does not manifest a missing input or missing output on state σ.

3.4.2 Verifying Correctness of Build Executions. Recall that a build execution in BUILDFS is a sequence of tasks $b = \langle t^1, t^2 \ldots \rangle$. A build execution may manifest an ordering violation, when there are pairs of tasks that access a file p, at least one of them produces p, and there is no ordering constraint between these tasks, i.e., they can be executed in any order.

Definition 3.6. (Ordering Violation.) Given a task graph G and an initial state $\sigma^0 = \bot$, a build execution $b = \langle t^1, t^2 \ldots t^n \rangle$ manifests an *ordering violation* on state σ_i, when $\exists j$ with $1 \leq j < i < n$ such that

- $(\sigma^{i+1}, r^{i+1}) = [\![t^{i+1}]\!]_{\sigma^i}$ and $(\sigma^j, r^j) = [\![t^j]\!]_{\sigma^{j-1}}$
- $|r^{i+1}_{\downarrow_p} \cap (r^j_{\downarrow_c} \cup r^j_{\downarrow_p})| \geq 1 \vee |r^j_{\downarrow_p} \cap (r^{i+1}_{\downarrow_c} \cup r^{i+1}_{\downarrow_p})| \geq 1$
- $t^{i+1} \not\prec_G^+ t^j \wedge t^j \not\prec_G^+ t^{i+1}$

Contrary to missing inputs and outputs, the definition for ordering violations checks whether two tasks with a conflicting file access are executed in the right order. To achieve this, we use the happens-before relation $\prec_G^+$. For example, consider a task t_1 that creates a file p. When the same file is consumed by a task t_2, the $t_1 \prec_G^+ t_2$ must hold. Otherwise, the build system is free to execute t_2 before t_1. Therefore, t_2 may access a file that does not exist, resulting in a build failure.

Definition 3.7. (Correctness of Build Execution). Given a task graph G and an initial state $\sigma^0 = \bot$, a build execution $b = \langle t^1, t^2 \ldots t^n \rangle$ is correct, when

- the task t^i is correct on state σ^{i-1}, and when $(\sigma^i, r^i) = [\![t^i]\!]_{\sigma^{i-1}}$ the task t^{i+1} is also correct on state σ^i for $1 \leq i < n$.
- the build execution does not manifest an ordering violation on state σ^i for $1 \leq i < n$.

Definition 3.7 summarizes our approach for verifying a build execution. We begin with examining and evaluating tasks in the order they appear in a BUILDFS program according to the semantics of Figure 4. The initial state is $\sigma_0 = \bot$. Evaluating a build task gives us a new state, and the set of files consumed and produced by the task. We then verify that the task is correct, that is, it does not contain any missing inputs or outputs while we also check that it does not conflict with any previous task based on the Definition 3.6 for ordering violations. Finally, we use the fresh state to evaluate the next task and perform the same verification task.

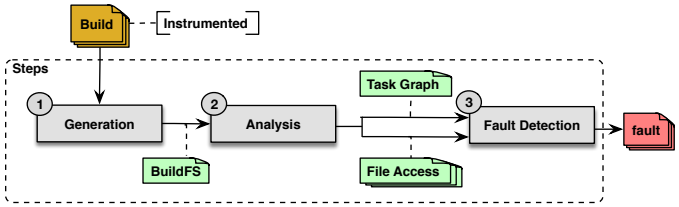


Fig. 10. The steps of our testing approach. First, we monitor the execution of an instrumented build, and then we report faults by analyzing the generated BuildFS programs.

```
1  + write(1, "#BuildFS#: Begin target");
2  + write(1, "#BuildFS#: target input /source");
3  + write(1, "#BuildFS#: target output /target");
4  open("/source", O_RDONLY) = 3;
5  open("/target", O_WRONLY|O_CREAT) = 4;
6  read(3, "content");
7  write(4, "content");
8  close(4);
9  close(3);
10 + write(1, "#BuildFS#: End target");
```

Fig. 11. Example of the native function calls observed, while executing the build scripts of Figure 5. The highlighted function calls are those inserted by our instrumentation.

4 TESTING APPROACH

We now present the practical realization of our model, which works on the three phases shown in Figure 10. During the first phase (*Generation*) we monitor the execution of an instrumented build script, and generate a BuildFS representation that models this execution. As we will explain shortly, the instrumentation performed on build scripts provides the generation step with all the necessary high-level information to produce a valid BuildFS program, such as, the execution boundaries and specification of every build task.

In the second phase (*Analysis*), we analyze the generated BuildFS program, and produce two outcomes. First, we construct the task graph capturing the input / output files, and the dependencies of every BuildFS task. Second, we track all file accesses of every task, by evaluating each task t^i in build execution $b = \langle t^1, t^2 \dots t^n \rangle$ as $[\![t^i]\!]_{\sigma_{i-1}}$ for $1 \leq i \leq n$, and with $\sigma^0 = \bot$.

The final step (*Fault Detection*) verifies the correctness of the given build execution (modeled in BuildFS) using the file accesses and task graph computed by the previous step. Specifically, this phase reports those file accesses that violate the correctness of build execution, i.e., they lead to missing inputs, missing outputs, or ordering violations according to the definitions of Section 3.4.

4.1 Generating BuildFS programs

To model a build execution in BuildFS, our dynamic approach takes an *instrumented* build script as input and monitors its execution. The goal of the instrumentation applied to build scripts is to provide the execution boundaries of every task, and other information coming from build definitions (i.e., declared input / output files and dependencies). To do so, we place instrumentation points before and after the execution of each build task, and augment their execution by calling special native functions. These functions take a string argument that either contains the information originated from build definitions, or indicate when the execution of a task begins or ends. Then, our dynamic analysis identifies these calls, and extracts their arguments to construct the specification of

each task, and map the intermediate file operations to the corresponding task. Through monitoring these special native functions calls and all the other file system operations (e.g., a call to `open`) that take place while building, we are able to construct buildFS programs.

An example of native function calls inserted by our instrumentation are writes to the standard output. These calls are triggered by inserting simple `print` statements as part of the instrumentation. Consider again the build scripts of Figure 5. When monitoring their base execution (no instrumentation is added), we observe the file system operations at lines 4–9 (see Figure 11) that reflect file copying. When instrumenting these scripts, we augment their execution by adding the native function calls at lines 1–3, before the execution of the task `target`, along with the function call at line 10 after file copying. These calls enables us to identify the points where the task `target` begins and ends (lines 1, 10), along with its input / output files (lines 2, 3). Our dynamic analysis detects and examines these calls, and finally produces the buildFS representation shown in Figure 6. Note that without this instrumentation, we are unable to map the intermediate file system operations (lines 4–9) to the task they come from, and to build the specification of the currently executed buildFS task.

Based on this instrumentation, we extract task specifications while executing the build, and *not* through statically analyzing the build scripts. To perform a sound execution, every build system is aware of all the dependencies and input / output files of each task at runtime. For example, the build system can recognize all task dependencies to schedule the execution of a task in the correct order. Similarly, in an incremental build, the build system is aware of all declared file inputs to determine which tasks must be executed in response to some identified file updates. Our approach benefits from extracting this information from the execution engine of build systems at runtime for two reasons. First, we do not have to perform static analysis (a challenging task as we discussed in 3.1) to extract this information from build scripts. Second, we can recognize all dependencies and input / output files, including the ones computed dynamically and not explicitly mentioned in build scripts. Details regarding the instrumentation of build scripts are implementation-specific and depend on the underlying build system as we show in Section 4.4.

Despite its simplicity, buildFS is expressive, and allows us to model several OS features that are related to or affect the file system, including operations on file paths, symbolic links, working directories, process cloning, operations on file descriptors, and more. Below, we discuss how we express such operations in terms of buildFS.

Operations on paths. A system operation that works on paths is translated to either `consume` or `produce` operations, depending on its effect on the file system. For example, when a build task creates a new directory through the `mkdir("/dir")` system call, we emit a `produce("/dir")` operation. Another example is when the build system creates a hard link to an existing file by invoking the `link("/source", "/target")` system call. In this case, we yield two buildFS operations: `consume("/source")`, and `produce("/target")`.

Similarly, we treat symbolic links as operations on file paths. We first track symbolic links, and the files they reference by monitoring the `symlink`-family system calls. Then, when we encounter an access to a symbolic link, we emit two `consume` constructs. The first `consume` statement concerns the symbolic link, while the second one is associated with the file pointed to by the symbolic link.

Operations on file descriptors. When the build process creates a new file descriptor, we use the $\texttt{fd}_f$ = ... operation. For example, when creating a new file descriptor through opening a file `open("/file") = 3`, we emit $\texttt{fd}_3$ = `"/file"`. We do the same, when copying an existing file descriptor to a new one, e.g., `dup2(3, 4)` turns into $\texttt{fd}_4 = \texttt{fd}_3$. Finally, closing a file descriptor leads to `del` operations.

Note that we do not need to model pipes, as file descriptors are tracked across system processes by modeling the `clone` and dup-family system calls.

```
1  binary: $(glob dir/*.c)
2    cc -o binary dir/*.c
```

(a) glob in Make.

```
1  task binary (type: Exec) {
2    inputs.files fileTree(dir: "dir", include: "*.c")
3    commandLine "cc -o binary dir/*.c"
4  }
```

(b) glob in Gradle.

Fig. 12. Examples of glob operations.

Working Directory. Each system process operates on a specific directory. In BuildFS, we use a special file descriptor variable, namely $\texttt{fd}_0$, that points to the working directory of the current process. Whenever, the working directory of a process changes (through the `chdir` system call), we emit a $\texttt{fd}_0 = \ldots$ operation to model this effect.

Relative Paths. Some file system operations operate on relative paths. For example, the call to `mkdir("dir")` creates the directory `dir` inside the current working directory. We handle relative paths through the p `at` e expression. Specifically, we model the above example as `produce("dir" at` $\texttt{fd}_0$`)`.

Forking Processes. When the build system creates a new process from an existing one (e.g., by calling the `clone` system call), we generate a `newproc` z_2 `from` z_1 statement, where z_2 refers to the id of the new process, while z_1 is the parent process. Finally, we model the main build process using a `newproc` z statement.

Operations on Directories. Beyond modeling working directories, our approach does not make any other special treatment on directories. Operations on directories, such as indexing or globbing, are irrelevant and they do not affect the effectiveness of our approach.

Consider the two equivalent Make and Gradle tasks shown in Figure 12. Both tasks use build system-specific functions to determine their input files through a `glob` pattern. In particular, the Make task (Figure 12a) uses the function `$(glob dir/*.c)` (line 1), while the Gradle task (Figure 12b) employs the function `fileTree` (line 2). By the time our approach runs the instrumented code to dump the declared inputs of each task to the standard output, the functions `glob` and `fileTree` have already been evaluated, and have resulted in a list of file paths that match the input pattern `dir/*.c`. As a result, our approach is able to print the *concrete* input files to the standard output (recall Section 4.1). In a similar manner, the `glob` pattern passed as input to an operation (i.e., `cc -o binary *.c`—line 2 in Figure 12a and line 3 in Figure 12b) also leads to a list of matching files which we capture and model through their individual accesses (i.e., file opens) while monitoring the build (i.e., there is no a Unix system call operating on `glob` patterns).

4.2 Analyzing BuildFS Programs & Detecting Faults

After modeling a build execution in a BuildFS representation, our method performs a linear pass over the representation and produces two types of output. First, it generates the corresponding task graph, and second, it computes all file accesses that take place in every BuildFS task based on the semantics presented in Figure 4.

In the final step of our method (fault detection), we verify the correctness of a build execution based on the task graph and the file accesses computed in the analysis step. In particular, we examine the file accesses of every task t, and we proceed as follows. If a file access p is of type "consumed" ("produced"), and the subsumption relation (Section 3.4.1) between p and the file inputs (outputs) of t does not hold, we report a missing input (output) on p. For ordering violations, we check whether p was accessed elsewhere (say another task t') in the given BuildFS program. If this is the case, we verify whether the execution order between t and t' is deterministic using the happens-before relation. If the happens-before relation between these tasks is undefined, we report

an ordering violation. Our fault detection approach eventually reports all file accesses that violate the correctness of the given build execution according to the definitions of 3.4.

The faults related to parallelism manifest themselves non-deterministically (depending on the execution schedule of the build system), while the ones associated with incrementality do not appear in full builds. However, our technique is capable of detecting subtle and future latent faults, because it does not require the build to crash and then reason about the root cause of the failure.

4.3 False Negatives and False Positives

We discuss potential false positive and negative errors focusing on specific examples.

False Negatives. Monitoring and reasoning about a single build execution (as our approach does) produces false negatives when incorrectly specified file accesses are not observed in the monitored build, i.e., they are conditionally defined. For example, their execution may depend on the environment (e.g., OS, environment variables, build parameters, and other external factors) when the build takes place. If these condition criteria are not met under the monitored build, our approach will be unable to identify the fault. As an example, consider the following Make script:

```
1  ifeq ($(LANG),en_US)
2    options = --input file_en_US.in
3  else
4    options = --input file.in
5  endif
6  target: file.in
7    cmd $(options) --output $@
```

This script defines a task (lines 6, 7) that executes a command (`cmd`). The input of this command depends on the current locale settings, i.e. `--input file_en_US.in` in the case of English (lines 1, 2), and `--input file.in` in any other case (lines 3, 4). When we run this build script in an environment where the locale settings are not set to English, a false negative occurs. Even though the script is incorrect, our approach misses the fault because it is unable to infer that in addition to `file.in` (line 6), the build task can also depend on the file `file_en_US.in`.

False negatives may also occur in the case of file system operations not modelled in buildFS. For example, our approach does not model the case when a process passes a file descriptor to another process via sockets (through an `SCM_RIGHTS` message).[1] In practice, it is highly unlikely for different build tasks to communicate with each other in such a way. That said, we leave supporting this case as future work. Note that we can support the aforementioned operation without extending our buildFS model. Specifically, we can express the operation in terms of a $\texttt{fd}_f$ `=` p statement in the process that receives the message (i.e., file descriptor). In this scenario, f corresponds to the file descriptor that was passed via a socket message, while p is the file path pointed to by the file descriptor f defined in the process that sent the message.

Similarly, not modelling the data blocks for read / write operations (e.g., `read` system call) may lead to an ordering violation issue not captured by our approach. Consider the case where a task must read from an existing file (not produced by the build process), after the contents of this file get modified by another task. If this ordering constraint is not specified by the developer, our approach will miss the issue because it does not track the contents read from and written to the file. Therefore, it cannot capture the fact that the first task may read a different value than the one written by the second one. In the context of builds though, file modifications are not as common as file creations.

False Positives. We can identify the missing inputs / outputs of build tasks without reporting false positives. This is justified by two reasons. First, by determining the exact execution boundaries

[1] https://www.man7.org/linux/man-pages/man7/unix.7.html

of every build task, our approach correlates the files that are read and written with the corresponding build tasks. When we detect an access on a file named `file.txt` during the execution of a specific task, this task indeed accesses `file.txt`. Second, the observed file accesses can be reliably verified against build specifications. Thanks to the instrumentation performed on build scripts, we know *all* task inputs and outputs that are declared by developers. Hence, our approach never reports false positives caused due to incomplete knowledge of the declared dependencies.

However, we may report false alarms when dealing with ordering violations. Specifically, a false alarm occurs when there are two independent build tasks that conflict with each other (i.e., they access the same file), but their execution order does not affect the correctness and the output of the build. Consider the next Gradle script.

```
1  task A {
2    f = file("file.txt")
3    doFirst { f.text = "value" }
4  }
5  task B {
6    f = file("file.txt")
7    doFirst { f.text = "value" }
8  }
```

There are two tasks (lines 1, 5) that produce the same file with the same contents (lines 3, 7). In this case, there is not an ordering violation. Any execution order of tasks `A` and `B` has the same effect: at the end of the build, we always end up with `file.txt` containing the text `"value"`. Although such benign cases were not observed in practice and usually indicate a flaw in the build scripts (e.g., unnecessary computation), someone can identify them by also checking the contents written to every file.

4.4 Implementation

We have implemented our method as a command-line OCaml program. To trace system operations, we employ `strace` [McDougall et al., 2006]. Note that this can be accomplished by using either other system call tracing utilities (e.g., `DTrace` [Rodriguez, 1986])) or dynamic binary instrumentation [Bruening et al., 2012; Nethercote and Seward, 2007].

Our tool parses the `strace` output and translates it into a BuildFS representation. The implementation supports two modes: *offline*, and *online*. In the offline mode (which is primarily used for debugging purposes), our tool does not monitor builds. Instead, it expects a file containing the `strace` output obtained from previous runs. When in online mode, the tool generates and analyzes a BuildFS program, while monitoring a build command through `strace`. To do so, it creates two processes. The first process runs `strace` on the build command, while the second reads the `strace` output produced by the first process and runs the BuildFS generation and analysis steps in a streaming fashion. Communication is done through pipes which allows processes to run concurrently. Notably, this eliminates the observable time spent on the analysis phase, because running the build is much slower than the analysis of the corresponding BuildFS programs. Therefore, in a multicore architecture, our tool exploits a spare core to perform the analysis as the build runs.

To instrument Gradle scripts, we have implemented a Gradle plugin written in Kotlin that hooks *before* and *after* the execution of every task as shown in Figure 13a (lines 1, 14 – irrelevant code is omitted). The plugin utilizes the Gradle API [Gradle Inc., 2020b] to print the following elements: (1) declared inputs / outputs of every task (lines 3–5, 6–8), (2) declared dependencies of every task (lines 9–11), and (3) execution boundaries of every task (lines 12, 16). This output is identified by our dynamic analysis and converted to BuildFS tasks as explained in 4.1. To apply our plugin to a Gradle project, we modify Gradle scripts by inserting *only* four lines of code.

```
1  // Runs BEFORE the execution of every task.
2  fun processTaskBegin(task: Task) {
3    task.inputs.files.forEach { input ->
4    println("#BuildFS#: ${task.name} input ${input}")
5   }
6   task.outputs.files.forEach { output ->
7    println("#BuildFS#: ${task.name} output ${output")
8   }
9   task.getTaskDependencies().forEach { d ->
10   println("#BuildFS#: ${task.name} after ${d.name}")
11  }
12  println("#BuildFS#: Begin ${task.name}")
13 }
14 // Runs AFTER the execution of every task.
15 fun processTaskEnd(task: Task) {
16  println("#BuildFS#: End ${task.name}")
17 }
```

(a) Fragment of the instrumentation applied to Gradle builds.

```
1  target=$1
2  prereqs=$2
3  taskName=$(pwd):$target
4  echo "#BuildFS#: Begin $taskName"
5  echo "#BuildFS#: $taskName input
       $prereqs"
6  shift 2
7  /bin/bash "$@"
8  echo "#BuildFS#: End $taskName"
```

(b) The `fsmake-shell` that instruments every Make rule.

Fig. 13. The instrumentation implemented for Gradle and Make builds.

For instrumenting Make scripts, we created a shell script (`fsmake-shell`) that *wraps* the execution of every Make rule (Figure 13b). As with Gradle, this script prints the execution boundaries (lines 4, 8) and prerequisites of each task (lines 5). To achieve this, we override Make's built-in variable `$SHELL` to point to our script. After printing the necessary information, our script invokes the underlying shell to eventually execute the requested Make command (line 7). To handle Make dependencies generated at build time (e.g., through `gcc -MD`), we refine the task graph computed during the analysis phase by adding missing edges. To do so, we exploit information stemming from the Make database by running `make -pn` after each build. Note that we do not need to make any changes in the source code of build scripts to enable tracing; we simply build projects by running

```
make "$@" -- SHELL='fsmake-shell '\''$@'\'' '\''$+'\''' '''
```

Applying our method to a new build tool requires little development effort. Our Gradle plugin contains 90 lines of Kotlin code, while `fsmake-shell` consists of *only 8 lines of shell code*.

Limitations. Currently, our tool can trace builds only in a Linux environment. However, extending our implementation to support monitoring in other platforms is straightforward. Also, `strace` introduces a 2× times slowdown on builds, on average (see Section 5.5). Employing a tracing utility that runs in the kernel space to track system operations, may reduce the overhead on build execution [Celik et al., 2017].

5 EVALUATION

We evaluate our approach by answering the following research questions:

RQ1 (Effectiveness) What is the effectiveness of our approach in locating faults in build scripts? (Section 5.2)

RQ2 (Fault Importance) What is the perception of developers regarding the detected faults? (Section 5.3)

RQ3 (Fault Patterns) What are the main fault patterns? (Section 5.4)

RQ4 (Performance) What is the performance of our approach? (Section 5.5)

RQ5 (Comparison with state-of-the-art) How does the proposed approach perform with regards to other tools, i.e., mkcheck? (Section 5.6)

Table 1. Faults detected by our approach. Each table entry indicates the number of faulty projects/the total number of the examined projects (Projects), the average LoC (Avg. LoC), and the average lines of build scripts (Avg. BLoC). The columns MIN, MOUT, and OV indicate the number of projects where missing inputs, missing outputs, and ordering violations appear respectively.

	Project Characteristics			Fault types		
Build System	**Projects**	**Avg. LoC**	**Avg. BLoC**	**MIN**	**MOUT**	**OV**
Gradle	73/312	35,536	589	58	21	25
Make	251/300	74,414	838	249	-	5

5.1 Experimental Setup

We applied our approach to a large set of Gradle and Make projects. To identify interesting Gradle projects, we employed the Github API to search for popular Java, Kotlin, and Groovy repositories that use Gradle. We selected 200 projects for each language (i.e., 600 projects in total) ordered by the number of stars. For every project, we performed the following steps. First, we instrumented the Gradle scripts as described in Section 4.4. Then, we ran the instrumented Gradle scripts through the `gradle build` command, which is the de-facto command for building Gradle projects. Note that this command executes the compilation, assembling and testing tasks as well as other user-defined tasks. For efficiency, we ran our tool in online mode (Section 4.4). In the end, we successfully analyzed and generated reports for 312 projects. The build of the remaining projects failed because it required human intervention, e.g., to set up a specific environment for the build. This was also observed in prior work [Hassan et al., 2017].

For diversity, we discovered Make projects from two sources. First, we used the Github API to collect popular C/C++ projects. Second, to ensure the buildability of the examined projects, we also employed the Ultimate Debian Database (UUD) [Debian, 2020b] to identify widely-used Debian packages based on the *"vote"* metric, which indicates the number of people who regularly use a specific package [Avery Pennarun and Reinholdtsen, 2020]. The build workflow of Debian packages utilizes the `sbuild` utility [Debian, 2020a], which automates the build process of Debian binary packages by creating the necessary build environment (e.g., it installs all build dependencies in an isolated environment) for a particular architecture (e.g., x86-64). `sbuild` allows us to hook over the build phase of its process. In this manner we can monitor each build and perform our own analysis. We built every Debian package using our Make wrapper (Section 4.4) instead of the default Make command. In total, we examined 300 Make projects coming from the Github and Debian ecosystems. Overall, the list of the selected Gradle and Make projects contains popular ones (e.g., the SQLite database, the Spring framework, and more) which involve complex build scripts. The characteristics of projects are summarized in Table 1.

We ran every Gradle and Make build in sequential mode as in the work of Licker and Rice [2019] to make fair comparisons against mkcheck. However, our approach is able to support parallel builds by tracking the thread (and its descendants) where every build task is running. Finally, we ran the builds on Docker containers inside a host machine with an Intel i7 3.6GHz processor with 8 cores and 16GB of RAM.

5.2 RQ1: Fault Detection Results

Table 1 summarizes our fault detection results. Our method identified problematic builds in 73 out of 312 Gradle projects. There are 157 issues related to incremental builds from which 122 faults are missing inputs appearing in 58 projects, while the remaining faults (35) are associated with missing outputs found in 21 projects. Faulty parallel builds are also common in Gradle projects, as we uncovered 80 ordering violations in 25 Gradle repositories. Furthermore, our tool detected

issues in 251 Make projects; it discovered 15,740 Make target rules with missing inputs. Most of them involved missing header dependencies concerning object files. It also reported 14 ordering violations that may lead to race conditions in 5 projects.

The large number of missing inputs (15,740) found in Make projects is justified by the complex structure of the examined builds. Most of the projects use configuration scripts (e.g., GNU autotools, etc.) that auto-generate complicated build rules with many recursive Make definitions. We also observed a common programming pattern that may have caused many missing inputs: developers often declare some header files in a Make variable, and use this variable as prerequisite of different build rules. When developers miss out to specify a header file in this variable, then all the rules that use this header file suffer from a missing input issue.

Regarding missing outputs, we did not detect this kind of fault in Make projects, as it is only relevant to Gradle (Section 2.1). Specifically, we modeled every Make rule as a BuildFS task that was producing $\top$ (any file). Based on the [TOP] rule of Figure 8, the subsumption relation always holds when verifying a task whose declared output is $\top$. Therefore, we never reported missing outputs in tasks producing $\top$ because there was no violation of the subsumption property (Definition 3.4).

To verify that the issues detected by our tool are indeed faults, we worked as follows. For Gradle projects, we examined each fault report, and tried to reproduce it. Specifically, we automatically verified each issue related to incremental builds by checking that re-running Gradle does not trigger the execution of tasks marked with missing inputs/outputs by our tool, even after updating the contents of their dependent files. We followed the same automated approach for the verification of the reported Make faults associated with incremental builds. For ordering violations, we manually verified that executing conflicting build tasks in the erroneous order can affect the outcome of a build, e.g., causing build failures, or producing build targets with incorrect contents.

For Make projects, we also analyzed every build with mkcheck, which is the state-of-the-art tool for Make-based builds [Licker and Rice, 2019]. We then cross-checked the results generated by our tool with those produced by mkcheck (see Section 5.6).

5.3 RQ2: Fault Importance

We provided fixes for 71 Make and Gradle projects that we chose while we were examining their fault detection results, and in turn, we submitted patches to the upstream developers. Patch generation was done manually, and substantial additional work was required to propose a suitable fix that would satisfy the development standards of each project. This was mostly because of the complex structure of the faulty projects, and the peculiar semantics of build systems' DSL (especially that of Gradle). We leave repairing build scripts through automated means as future work.

Table 2 enumerates the faults that are confirmed and fixed. Notably, 237 issues found in 47 out of 71 projects were fixed, while most of the remaining patches are in a pending state. In a small number of projects, the corresponding developers rejected our patches, but they confirmed and finally fixed the reported faults on their own.

The list of projects where our patches were accepted contains popular projects, such as `tinyrenderer` (~8k stars), `caffeine` (> 7k stars), `aeron` (~5k stars), `Cello` (~5k stars), and more. The list also includes projects that are maintained and developed by well-established organizations, such as `conductor` (developed by Netflix), `tsar` (developed by the Alibaba Group). This indicates that the faults we identified do matter to the community.

5.4 RQ3: Fault Patterns

When we manually examined the issues generated by our tool, we recognized five fault patterns, which result in build failures, time-consuming builds, or erroneous build outcomes. We identified three kinds of faults related to incremental builds caused by missing inputs or outputs.

Table 2. Faults confirmed and fixed by the developers.

Project	Build System	Total	MIN	MOUT	OV
junit-reporter	Gradle	10	2	3	5
aeron	Gradle	7	2	2	3
muwire	Gradle	6	0	0	6
xtext-gradle-plugin	Gradle	4	0	0	4
rundeck	Gradle	3	1	1	1
apina	Gradle	3	0	0	3
caffeine	Gradle	2	0	0	2
conductor	Gradle	2	0	1	1
RxAndroidBle	Gradle	2	0	0	2
jmonkeyengine	Gradle	1	0	1	0
tsar	Make	31	31	-	0
Cello	Make	27	27	-	0
webdis	Make	27	27	-	0
density	Make	17	17	-	0
VRP-Tabu	Make	12	12	-	0
pspg	Make	9	9	-	0
janet	Make	8	8	-	0
parcellite	Make	8	8	-	0
reptyr	Make	5	5	-	0
hdparam	Make	4	4	-	0
Others[1]	Make,Gradle	49	40	0	9
	Total	**237**	**193**	**8**	**36**

[1] **Others:** gps-sdr-sim, cscout, proxychains, gradle-scripts, p2rank, fzy, groocss, ShellExec, anna, fulibGradle, nf-tower, alfresco-gradle-sdk, Gradle-RIO, helios, kscript, coveralls-gradle-plugin, gradle-test-logger-plugin, joystick, cqmetrics, JenkinsPipelineUnit, swagger-gradle-plugin, gradle-swagger-generator-plugin, gradle-sora, tinyrenderer, libcs50, LuaJIT, greatest

Test resources. To ensure the correctness of their programs, developers typically specify dedicated build rules for performing different forms of testing (e.g., unit and functional testing) during build. Running tests is a time-consuming task [Gligoric et al., 2015], so the build rules associated with tests are triggered only when there are updates to any of the source files that tests rely on. As with source files, changing any of the resources used by tests (e.g., test data or additional helper scripts) must re-run tests to make sure that the change does not break anything. Not running tests is a missed opportunity to identify potential issues and may lead to late identification of bugs.

For example, the Gradle project `kscript` contains a test suite of Kotlin files included in the `test/resources` directory. The tests of `kscript` contains test assertions that rely on the state and contents of the files included in this test suite. However, the developers failed to declare the test suite directory as input of the Gradle task `test`. Our tool detected this fault, and we reported to the developers who fixed it.

Stale artifacts. As already discussed, the main goal of a build is to construct artifacts, such as executables, libraries, documentation accompanying software, and more. The build process must re-generate these artifacts, when any of the files used for their construction is updated since the last build. Failing to do so can lead to stale artifacts, which in turn, can either harm the reliability of applications, (e.g., cause runtime errors), or generate wrong build outputs.

This pattern is particularly common in Make builds where developers do not enumerate the dependencies of object files correctly. As an example of stale artifacts, recall the build script of Figure 1. This example demonstrates that even best practices for tracking dependencies automatically (e.g., through `gcc -MD`) are not sufficient for ensuring the correctness of builds.

Time consuming tasks. The purpose of incremental builds is to reduce build time by running only the build tasks needed to achieve a specific goal. This boosts productivity as it enables

```
1  LIBS := $(addprefix build/lib/, $(LIB_BASE) $(LIB_VERSION))
2  $(LIBS): $(SRC)
3    $(CC) $(CFLAGS) -o $(LIB_VERSION) $(SRC)
4    $(CC) $(CFLAGS) -c -o $(LIB_OBJ) $(SRC)
5    rm -f $(LIB_OBJ)
6    ln -sf $(LIB_VERSION) $(LIB_BASE)
7    mv $(LIB_VERSION) $(LIB_BASE) build/lib
```

Fig. 14. A Make script with conflicting producers.

```
1  sourceSets { main { srcDir "build/generated-sources/" } }
2  task generateNodes(type: JavaExec) {
3    main = "com.github.benmanes.caffeine.cache.NodeFactoryGenerator"
4    args "build/generated-sources/"
5    outputs.dir "build/generated-sources/" }
6  apply plugin: "com.bmuschko.nexus"
```

Fig. 15. A Gradle script manifesting an ordering violation.

developers to get feedback and respond to changes of their codebase much earlier. To avoid unnecessary computation, it is important that time consuming build tasks are incremental.

We identified an instance of this pattern in the Gradle project `junit-reporter`. This project contains some JavaScript source files used to visualize JUnit reports in HTML format. Developers define a Gradle task to bundle JavaScript source files into a single file (`site.js`) that is finally incorporated in the HTML page of test reports. However, this task was not declared as incremental. As a result, Gradle was bundling JavaScript files at every build, causing the subsequent Gradle tasks that were dependent on `site.js` to be executed, as Gradle was creating a newer version of `site.js` each time. Our tool marked `site.js` as a missing output of the bundler task. Based on this, we sent a patch to the upstream developers who integrated it in their codebase. Fixing this issue made builds eight times faster.

Below we discuss two categories of faults related to parallel builds.

Conflicting Producers. We have identified issues associated with tasks that produce the same file or write to the same output directory. Parallel execution of such build tasks is harmful, because race conditions may emerge, when two tasks affecting the same state (i.e., files) run concurrently.

The Gradle script of Figure 2 is an example of conflicting producers. Figure 14 shows another example coming from the `libcs50` project. This Make script defines a rule (line 2) that creates two libraries inside the `build/lib` directory (see variable `$(LIBS)`). The code first compiles the source file into the corresponding object file (line 3) from which a shared library, namely `$(LIB_BASE)`, is constructed (line 4). Then, it creates a symbolic link (`$(LIB_VERSION)`) pointing to the newly-created library (line 6), and finally moves these files to the `/build/lib` directory (line 7). The official documentation of GNU Make states that such a rule definition is incorrect [GNU Make, 2020b], and can result in race conditions. In particular, the rule at line 2 is executed twice (one for every target defined in the `$(LIBS)`) variable). Consequently, the parallel build might crash with the error *"mv: cannot stat 'libcs50.so.10.1.0': No such file or directory"*, as every rule execution races against each other. Specifically, when the second rule invocation attempts to move the libraries, they may have already been moved to `/build/lib` by the first rule. The developers of `libcs50` immediately fixed this problem.

Generated Source Files and Resources. Many projects generate part of their source code or resources at build time. These automatically generated source files and resources are then compiled or used later by other build tasks to form the final artifacts of the build process, e.g., binaries.

Table 3. Time spent on analyzing builds and detecting faults by our approach vs. mkcheck (in seconds).

Phase	Median (seconds)		Average (seconds)	
	BuildFS	mkcheck	BuildFS	mkcheck
Build	4.68	4.91	21.64	22.2
Fault detection	0.01	182.62	0.44	3,368
Overall	4.69	186.75	22.08	3,390

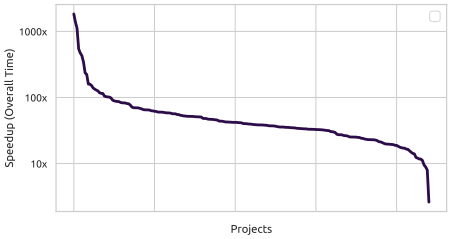


Fig. 16. The speedup for every Make project by our approach over mkcheck (ordered by y-axis). This plot considers the overall times (build time + fault detection time) spent by our approach and mkcheck.

Developers must be careful enough to preserve the correct execution order between the build tasks that are responsible for generating and using these source files and resources. Ordering violations (e.g., compiling code when source files are missing) are the root cause for build failures, or subtle errors detected at a later stage of software lifecycle.

Figure 15 presents a code fragment taken from the popular `caffeine` project. The code specifies that the source files of the project are stored in the `build/generated-sources` directory (line 1). These source files are generated automatically by the Gradle task `generateNodes`. To do so, this task runs the class `NodeFactoryGenerator` with `"build/generated-sources"` as an argument (lines 2–5). Then, this code applies the plugin `"com.bmuschko.nexus"` used for uploading the sources JAR file to a remote repository. To assemble a JAR file containing the source files of the application, this plugin adds the `sourcesJar` task to the project. The problem with this code is that no dependency is declared between the tasks `generateNodes` and `sourcesJar`. Thus, the build process uploads empty artifacts to the remote repository, when Gradle executes `sourcesJar` before `generateNodes`. The developers of `caffeine` confirmed and fixed this ordering issue.

5.5 RQ4: Performance

To measure the performance of our approach we recorded the time spent at each step (recall Figure 10). The generation step, which is responsible for executing and monitoring our instrumented builds, dominates the execution time of our method. In particular, this step slows down both Gradle and Make builds by a factor of around two for the 90th percentile of the examined projects. This is consistent with the recent literature [Licker and Rice, 2019], as the main overhead of this phase stems from the system call tracing utility (i.e., `strace`).

The analysis of BuildFS programs takes around 2.47 and 5.1 seconds on average for Make and Gradle projects respectively, and is linear to the size of programs. This phase is efficient enough to analyze GBs of programs in a reasonable time (e.g., 6.9GB in less than 3 minutes). In online mode, though, the observable time spent on the analysis step is eliminated, as the overall time is bounded to the time needed for a build. As explained in 4.4, this is because the processing of BuildFS programs is faster than the build itself, and thus we take advantage of multicore architectures. Finally, the fault detection step is pretty fast; it takes only 0.11 and 0.45 seconds on average for Gradle and Make projects respectively.

5.6 RQ5: Comparison with state-of-the-art

As a first step, we built and analyzed a number of Gradle projects with mkcheck. After finding that mkcheck produces an overwhelming number of false positives (this is not surprising because

mkcheck is unable to deal with JVM-based builds as explained in Section 3.1), we focused on performing comparisons only for Make projects.

We applied mkcheck to the 300 Make projects of our experimental setup and we recorded the fault reports and the time spent at each phase (i.e., build time and fault detection time). Note that build time includes the time needed for building the project as well as the time taken for generating and analyzing the build trace.

In terms of fault detection, mkcheck produced false positives in three cases due to the granularity of processes. That is, two build tasks were merged into a single task because they were governed by the same system process, leading to imprecision. False positives are also observed in the initial work of Licker and Rice [2019]. On the contrary, our approach did not generate false positives as it can reliably determine all file accesses of each task as explained in Section 4.1.

Regarding efficiency, we compared our approach with mkcheck when (1) building a project, and (2) detecting faults. Starting with build times, our tool and mkcheck spend almost the same amount of time for building and monitoring a project. The minimum speedup is -1.25×, the median is 1.04×, the average is 1.19×, while the maximum speedup is 9×. Although mkcheck uses `ptrace` for tracking system operations, which is 20% times faster than `strace` [Licker and Rice, 2019], our approach benefits from running the generation and analysis steps concurrently. Moving to fault detection times, we observe that our approach outperfoms mkcheck in terms of efficiency: the minimum speedup of our fault detection algorithm is 83×, the median is 25,431×, the average is 56,863×, and the maximum speedup is 2,708,917×. This huge speedup is explained by the fact that mkcheck needs to trigger multiple incremental builds (one for every source file) for verifying correctness. Therefore, incremental builds dominate the time needed for detecting faults in mkcheck. Notably, in projects consisting of a large number of source files, mkcheck required days to detect faults (e.g., it spent 3.3 days for analyzing `ghostscript`). Contrariwise, our fault detection algorithm performs no incremental builds; it just iterates over the file accesses computed by the analysis of the buildfs program, and checks whether the expected subsumption and happens-before relations hold (Section 4.2). Finally, when considering the overall time (i.e., build + fault detection time), our approach is 74× faster than mkcheck, on average. The median speedup is 39×, while the minimum and maximum speedup is 2.6× and 1,837× respectively (see Figure 16).

Furthermore, we provide some absolute performance times on Table 3. Overall, our method required 22 seconds (on average) for analyzing builds and detecting faults (see phase "Overall" on Table 3), while mkcheck spent 3,390 seconds on average for performing the same tasks. The median time is 4.69 and 186.75 seconds for our tool and mkcheck respectively. Notably, mkcheck spent more than ten minutes for reporting faults in the 29% of the inspected Make projects.

Our findings, indicate that our method is superior to the state-of-the-art in terms of both fault detection and performance. Moreover, due to its effectiveness and efficiency, we argue that our method can be used in practice as part of the software testing pipeline.

6 RELATED WORK

Formally-Verified Build Systems. CloudMake [Christakis et al., 2014] is a modern build system developed by Microsoft. It supports both incremental and parallel builds, and exposes a functional interface in JavaScript. Developers can invoke external tools (e.g., compilers, linkers, other arbitrary programs), and define their own build tasks using the `exec` primitive. When calling an external tool via `exec`, developers need to explicitly enumerate the dependencies of this tool. Then, CloudMake internally uses these static dependencies to (1) form a parallel schedule, and (2) verify that each tool accesses only the files provided by the developer.

Christakis et al. [2014] introduce a language and a semantics for builds, and use them to formally verify some of the core algorithms of CloudMake, namely parallel and cached builds (but not

incremental builds). Although CloudMake tackles similar issues, we follow a different approach. New build systems face adoption issues [Gligoric et al., 2014; McIntosh et al., 2015], and migrating existing build scripts to a new (formally-verified) build tool may require significant engineering effort [Gligoric et al., 2014; McIntosh et al., 2015]. Therefore, instead of designing a new build system, our work introduces a generally-applicable method that detects issues in builds written in existing systems without requiring extensive modifications to the underlying build scripts. To demonstrate applicability, we apply our method to two build systems with diverse build specification languages and designs that hold a significant stake in build technology [Hassan et al., 2017; McIntosh et al., 2015].

Testing and Debugging Builds. Testing build scripts is an emerging research area. mkcheck [Licker and Rice, 2019], and bee [Bezemer et al., 2017] are two tools that also detect missing inputs, but they are tailored for Make-based builds. As we pointed out in 3.1, these tools have two important limitations that concern: (1) low precision when applied to JVM build tools, and (2) efficiency & applicability. As we explained earlier, our approach tackles both limitations.

Beyond testing, a number of studies have been developed to identify the root causes of problematic builds, and suggest fixes for them. Al-Kofahi et al. [2014], have designed a tool that given a failed build, it identifies the faulty Make rules that caused the build crash. Their approach performs an instrumentation on a Make build that tracks the execution trace of each Make rule, and records the crash point of the build. Based on a probabilistic model, they assign different scores to every rule, indicating the probability that the rule caused the crash. Subsequent work [Ren et al., 2018; Ren et al., 2019] have focused on locating faults of unreproducible builds. Reproducibility is a property that ensures that a build is deterministic and always results in bitwise-identical targets given the same sources and build environment. The initial work of Ren et al. [2018] analyzes the logs from an unreproducible build, and proposes a ranked list of problematic source files that might contain the fault. Recently, the authors extended their approach [Ren et al., 2019] to locate the specific command that is responsible for the unreproducible build. To do so, they employed a backtracking analysis on system call trace stemming from build execution. Contrary to these approaches, our method does not require a build failure, but it is capable of detecting latent future faults.

Scott et al. [2017] develop `detmake`, a system that uncovers failures in Make builds by enforcing a deterministic execution of parallel builds. To do so, the technique behind `detmake` provides arbitrary system operations (such as file system operations) with deterministic semantics and side-effects, by intercepting system calls and libraries at runtime. Running a Make build through `detmake` always results in a build failure when the corresponding build scripts contain ordering violations. The developers then need to manually find and fix the root cause of the failure to run their builds successfully. `detmake` is not fully compatible with GNU Make as the technique behind this tool comes with several restrictions for the supported builds. Unlike `detmake`, our approach focuses on fault localization: we report the *exact* build rule whose definition is incorrect, and can potentially lead to issues. Further, our method is also capable of identifying *subtle* errors that lead to wrong build outcomes rather than failures. Finally, the work of Scott et al. [2017] does not reason about issues related to incremental builds.

Understanding and Refactoring Builds. There are plenty of tools developed over the past decade to assist developers in understanding and refactoring builds. *Makao* [Adams et al., 2007] is a Make-related framework used for visualizing build dependencies. By extracting knowledge from such dependencies through filtering and querying, Makao provides support for refactoring build scripts via an aspect-oriented approach. *SYMake* [Tamrawi et al., 2012], evaluates Makefiles and produces (1) a symbolic dependency graph, and (2) a symbolic execution trace. Then, it applies different algorithms to the results to detect a number of code smells (e.g., cyclic dependencies), and perform refactoring on Make scripts (e.g., target renaming). Metamorphosis [Gligoric et al., 2014]

is a tool used to migrate existing build scripts to CloudMake [Christakis et al., 2014]. As a starting point, Metamorphosis analyzes the execution trace of a given build and then automatically synthesizes an initial CloudMake script that reflects the behavior of the original script. Then, it optimizes the build script synthesized by the previous step by applying a sequence of transformations and choosing the best possible ones based on a fitness function. Vakilian et al. [2015] propose a new refactoring method, target decomposition, for dealing with underutilized targets; a build-related code smell that causes slower builds, larger binaries, and less modular code.

Trace Analysis. Most of the existing work [Ammons, 2006; Gligoric et al., 2014; Licker and Rice, 2019; Ren et al., 2019; Sotiropoulos et al., 2019; van der Burg et al., 2014] that is relevant to the domain of builds employs techniques for analyzing traces—and especially system call traces. Our work differs from the previous approaches as the proposed model (buildfs), and in turn, its practical realization captures both the dynamic behavior and the static specification of high-level programming constructs (i.e., build tasks). This enables us to verify—contrary to existing approaches—the execution of a build phase with regards to its specification, while monitoring build, making our method more precise, efficient, and generally-applicable.

Regression Test Selection. Recently, there have been advances on dynamic regression test selection techniques (RTS) [Celik et al., 2017; Gligoric et al., 2015; Wang et al., 2018; Zhang, 2018]. Dynamic RTS methods improve the performance of regression testing by running only those tests affected by a specific code change. To do so, they compute test dependencies from previous test runs. Gligoric et al. [2015], and Celik et al. [2017] extract test dependencies by determining the execution boundaries of each test, and tracking all intermediate file accesses. RTS methods and our technique are complementary; they can both used as part of a build to improve efficiency and reliability respectively.

7 CONCLUSION

We developed a generic and practical approach for discovering faults that can cause incremental and parallel build failures. To do so, we proposed a model (buildfs) for arbitrary build executions that captures the static specification and the dynamic behavior of each build task. We then formally defined three types of faults concerning incrementality and parallelism, and presented an approach for exploiting buildfs and detecting such faults in practice. Combining static and dynamic information in a single representation made our method efficient and applicable to any build system.

Our method was able to uncover issues in hundreds of Make and Gradle builds. Notably, our approach tackled the limitations of existing work, and it is the first to deal with JVM-based build tools. We demonstrated the importance of the discovered faults by providing patches to numerous projects. Thanks to our tool, the developers of 47 open-source projects confirmed and fixed 237 issues, in total. Moreover, a comparison between our tool and a state-of-the-art Make-based tool showed that our approach is more effective and significantly more efficient (74× faster on average). We argue that our tool could be part of the software testing pipeline, helping developers to discover defects and inconsistencies in their software artifacts that arise from faulty build definitions.

ACKNOWLEDGMENTS

We thank Vaggelis Atlidakis and the anonymous reviewers for their constructive feedback. This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 825328.

REFERENCES

B. Adams, H. Tromp, K. de Schutter, and W. de Meuter. Design recovery and maintenance of build systems. In *2007 IEEE International Conference on Software Maintenance*, pages 114–123, Oct 2007. doi: 10.1109/ICSM.2007.4362624.

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*, ICSE Companion 2014, pages 600–601, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2768-8. doi: 10.1145/2591062.2591135.

G. Ammons. Grexmk: Speeding up scripted builds. In *Proceedings of the 2006 International Workshop on Dynamic Systems Analysis*, WODA '06, pages 81–87, New York, NY, USA, 2006. ACM. ISBN 1-59593-400-6.

B. A. Avery Pennarun and P. Reinholdtsen. Debian popularity contest. https://popcon.debian.org/, 2020.

Bazel. Build and test software of any size, quickly and reliably. https://bazel.build, 2020.

C.-P. Bezemer, S. McIntosh, B. Adams, D. M. German, and A. E. Hassan. An Empirical Study of Unspecified Dependencies in Make-Based Build Systems. *Empirical Software Engineering*, 22(6):3117–3148, 2017.

D. Bruening, Q. Zhao, and S. Amarasinghe. Transparent dynamic instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, pages 133–144, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1176-2. doi: 10.1145/2151024.2151043.

J. Calcote. *Autotools: A Practitioner's Guide to GNU Autoconf, Automake, and Libtool.* No Starch Press, 2020.

A. Celik, A. Knaust, A. Milicevic, and M. Gligoric. Build system with lazy retrieval for Java projects. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 643–654, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342186. doi: 10.1145/2950290.2950358.

A. Celik, M. Vasic, A. Milicevic, and M. Gligoric. Regression test selection across JVM boundaries. In *Symposium on the Foundations of Software Engineering*, pages 809–820, 2017.

M. Christakis, R. Leino, and W. Schulte. Formalizing and verifying a modern build language. In *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, volume 8442 of *Lecture Notes in Computer Science*, pages 643–657. Springer, May 2014. ISBN 978-3-319-06409-3.

D. Coetzee, A. Bhaskar, and G. Necula. apmake: A reliable parallel build manager. In *2011 USENIX Annual Technical Conference (USENIX)*, 2011.

D. Dashenkov. Gradle task ordering constraints. https://github.com/SpineEventEngine/base/issues/516, 2020.

Debian. sbuild. https://wiki.debian.org/sbuild, 2020a.

Debian. Ultimatedebiandatabase. https://wiki.debian.org/UltimateDebianDatabase/, 2020b.

E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes. Keep me updated: An empirical study of third-party library updatability on Android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2187–2200, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4946-8. doi: 10.1145/3133956.3134059.

S. Erdweg, M. Lichter, and M. Weiel. A sound and optimal incremental build system with dynamic dependencies. *SIGPLAN Not.*, 50(10):89–106, Oct. 2015. ISSN 0362-1340. doi: 10.1145/2858965.2814316.

S. I. Feldman. Make—a program for maintaining computer programs. *Software: Practice & Experience*, 9(4):255–265, 1979.

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*, OOPSLA '14, pages 599–616, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2585-1. doi: 10.1145/2660193.2660239.

M. Gligoric, L. Eloussi, and D. Marinov. Practical regression test selection with dynamic file dependencies. In *International Symposium on Software Testing and Analysis*, pages 211–222, 2015.

GNU Make. Generating prerequisites automatically. https://www.gnu.org/software/make/manual/html_node/Automatic-Prerequisites.html, 2020a.

GNU Make. Handling tools that produce many outputs. https://www.gnu.org/software/automake/manual/html_node/Multiple-Outputs.html, 2020b.

Gradle Inc. Build cache. https://docs.gradle.org/current/userguide/build_cache.html, 2020a.

Gradle Inc. Developing custom Gradle plugins. https://docs.gradle.org/current/userguide/custom_plugins.html, 2020b.

Gradle Inc. Gradle vs Maven: Performance comparison. https://gradle.org/gradle-vs-maven-performance/, 2020c.

Gradle Inc. Gradle - plugins. https://plugins.gradle.org/, 2020d.

S. Greene. Introducing incremental build support. https://blog.gradle.org/introducing-incremental-build-support, 2015.

F. Hassan, S. Mostafa, E. S. L. Lam, and X. Wang. Automatic building of Java projects in software repositories: A study on feasibility and challenges. In *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '17, pages 38–47, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5090-4039-1. doi: 10.1109/ESEM.2017.11.

M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig. Usage, costs, and benefits of Continuous Integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 426–437, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450338455. doi: 10.1145/2970276.2970358.

R. Karanpuria and A. S. Roy. *Kotlin Programming Cookbook: Explore more than 100 recipes that show how to build robust mobile and web applications with Kotlin, Spring Boot, and Android.* Packt Publishing Ltd, 2018.

G. Konat, S. Erdweg, and E. Visser. Scalable incremental building with dynamic task dependencies. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 76–86, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450359375. doi: 10.1145/3238147.3238196.

N. Licker and A. Rice. Detecting incorrect build rules. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 1234–1244, Piscataway, NJ, USA, 2019. IEEE Press.

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

R. McDougall, J. Mauro, and B. Gregg. *Solaris Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris.* Prentice Hall PTR, Upper Saddle River, 2006. ISBN 0131568191.

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*, ICSE '11, pages 141–150, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304450. doi: 10.1145/1985793.1985813.

S. McIntosh, M. Nagappan, B. Adams, A. Mockus, and A. E. Hassan. A large-scale empirical study of the relationship between build technology and build maintenance. *Empirical Software Engineering*, 20(6):1587–1633, Dec 2015. ISSN 1573-7616.

J. D. Morgenthaler, M. Gridnev, R. Sauciuc, and S. Bhansali. Searching for build debt: Experiences managing technical debt at Google. In *2012 Third International Workshop on Managing Technical Debt (MTD)*, pages 1–6, June 2012. doi: 10.1109/MTD.2012.6225994.

N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. doi: 10.1145/1250734.1250746.

K. Pelgrims. *Gradle for Android.* Packt Publishing Ltd, 2015.

Z. Ren, H. Jiang, J. Xuan, and Z. Yang. Automated localization for unreproducible builds. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 71–81, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5638-1. doi: 10.1145/3180155.3180224.

Z. Ren, C. Liu, X. Xiao, H. Jiang, and T. Xie. Root cause localization for unreproducible builds via causality analysis over system call tracing. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 527–538, Nov 2019. doi: 10.1109/ASE.2019.00056.

R. Rodriguez. A system call tracer for UNIX. In *USENIX Conference Proceedings*, pages 72–80, Berkeley, CA, Summer 1986. USENIX Association.

R. G. Scott, O. S. Navarro Leija, J. Devietti, and R. R. Newton. Monadic composition for deterministic, parallel batch processing. *Proc. ACM Program. Lang.*, 1(OOPSLA), Oct. 2017. doi: 10.1145/3133897.

T. Sotiropoulos, D. Mitropoulos, and D. Spinellis. Detecting missing dependencies and notifiers in Puppet programs. *arXiv preprint arXiv:1905.11070*, 2019.

A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. N. Nguyen. SYMake: a build code analysis and refactoring tool for makefiles. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 366–369, Sep. 2012. doi: 10.1145/2351676.2351749.

M. Vakilian, R. Sauciuc, J. D. Morgenthaler, and V. Mirrokni. Automated decomposition of build targets. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 123–133, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4799-1934-5.

S. van der Burg, E. Dolstra, S. McIntosh, J. Davies, D. M. German, and A. Hemel. Tracing software build processes to uncover license compliance inconsistencies. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 731–742, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3013-8.

J. Visser, S. Rigal, G. Wijnholds, and Z. Lubsen. *Building Software Teams: Ten Best Practices for Effective Software Development.* " O'Reilly Media, Inc.", 2016.

K. Wang, C. Zhu, A. Celik, J. Kim, D. Batory, and M. Gligoric. Towards refactoring-aware regression test selection. In *International Conference on Software Engineering*, pages 233–244, 2018.

L. Zhang. Hybrid regression test selection. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 199–209, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356381. doi: 10.1145/3180155.3180198.