Noname manuscript No.
(will be inserted by the editor)

Assessing the Exposure of Software Changes

The DiPiDi Approach

Mehran Meidani · Maxime Lamothe · Shane McIntosh

Author pre-print copy. The final publication is available at Springer via:
https://doi.org/10.1007/s10664-022-10270-y

Abstract Changing a software application with many build-time configuration settings may introduce unexpected side effects. For example, a change intended to be specific to a platform (e.g., Windows) or product configuration (e.g., community editions) might impact other platforms or configurations. Moreover, a change intended to apply to a set of platforms or configurations may be unintentionally limited to a subset. Indeed, understanding the exposure of source code changes is an important risk mitigation step in change-based development approaches. In this paper, we present DiPiDi, a new approach to assess the exposure of source code changes under different build-time configuration settings by statically analyzing build specifications. To evaluate our approach, we produce a prototype implementation of DiPiDi for the CMake build system. We measure the effectiveness and efficiency of developers when performing five tasks in which they must identify the deliverable(s) and conditions under which a source code change will propagate. We assign participants into three groups: without explicit tool support, supported by existing impact analysis tools, and supported by DiPiDi. While our study does not have the statistical power to make generalized quantitative claims, we manually analyze the full distribution of our study's results and show that DiPiDi results in a net benefit for its users. Through our experimental evaluation, we show that DiPiDi results in a 36 average percentage points improvement in

M. Meidani
University of Waterloo
E-mail: mehran.meidani@uwaterloo.ca

M. Lamothe
Polytechnique Montreal
E-mail: maxime.lamothe@polymtl.ca

S. McIntosh
University of Waterloo
E-mail: shane.mcintosh@uwaterloo.ca

F_1-score when identifying impacted deliverables and a reduction of 0.62 units of distance when ranking impacted patches. Furthermore, DiPiDi results in a 42% average task time reduction for our participants when compared to a competing impact analysis approach. DiPiDi's improvements to both effectiveness and efficiency are especially prevalent in complex programs with many compile-time configurations.

Keywords build systems · exposure of a change · build dependency graph

1 Introduction

Complex software programs employ many compile-time configuration settings to build different software products (a.k.a., variants) from the same artifacts (i.e., source files) (Tu and Godfrey, 2001). For example, the Linux kernel has more than 10,000 compile-time configuration settings (Sincero et al, 2007). These software programs have multiple dependency paths to their source files from their *deliverables*, i.e., software artifacts that users can interact with, such as executable files or libraries. Build systems derive default configuration settings by analyzing the execution environment or reading user override settings. Build systems use these settings to reason about whether source files (or conditionally compiled code snippets) should be included or excluded from the produced deliverables. Under some conditions, a source file may play a role in one compiled deliverable without affecting others. For example, in the Linux kernel, the source files written specifically for the ARM architecture will be excluded from the x86 version of the kernel (Nadi and Holt, 2014). In these complex systems, a change in a source file may have unexpected side effects on deliverables outside of the current compilation path. Software systems that support multiple variants can therefore create complex arrangements of effects and side effects, where the deliverables exposed to a code-change can be unclear (Bezemer et al, 2017).

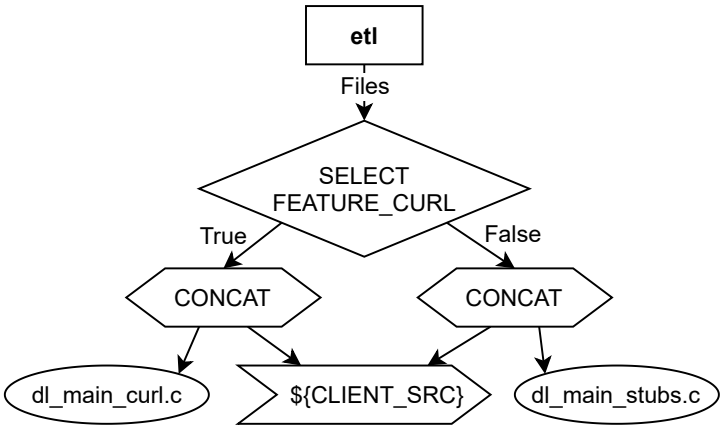
Software engineering practices that assess source code changes, like code review, are expensive and time-consuming (Cohen, 2010; Bosu et al, 2015). Extra time and effort must be spent by developers on activities like finding which deliverables are exposed to a change. In this paper, we define the exposure of a change as the set of deliverables affected by a change, including executables and libraries, as well as the different build-time configuration and environment settings under which the changes propagate. Changes that impact critical deliverables or configurations may require more quality assurance effort than others to mitigate their exposure risk (Wen et al, 2018).

When modifying complex software programs, source code changes may be localized or broad. Figure 1 shows an example of a dependency graph for the ET: Legacy project.[1] A change to the `dl_main_curl.c` file impacts the deliverable `etl` only if the `FEATURE_CURL` option is `ON`. On the other hand, changes to files represented by `$CLIENT_SRC` will always impact the deliverable.

[1] https://github.com/etlegacy/etlegacy

```
if(FEATURE_CURL)
  add_executable(etl ${CLIENT_SRC} dl_main_curl.c)
else()
  add_executable(etl ${CLIENT_SRC} dl_main_stubs.c)
endif()
```

(a) A sample snippet of CMake build script from the ET: Legacy project. In this sample, `etl` is the deliverable, `FEATURE_CURL` is a build configuration, and `CLIENT_SRC` is a variable pointing to the source files.



(b) A build dependency graph generated by DiPiDi. Arrows show dependency relation between a source node to the destination.

Fig. 1: A real-world example of a small section of a CMake build script and its corresponding Build Dependency Graph

A change that only impacts one variant of a system may not be as important as a change that affects all variants. Exposing the effect of a change under different configuration settings can help developers assess the impact of that change.

Despite its importance, assessing which deliverables are impacted by a change, and the conditions under which they are impacted, is not well supported by current software tools (Hassan and Wang, 2018). Change Impact Analysis (CIA) is one way to determine the consequences of a change on a software application (Arnold and Bohner, 1993). Many CIA techniques have been proposed (Li et al, 2013; Ahsan and Wotawa, 2010; Gethers and Poshyvanyk, 2010; Tamrawi et al, 2012; Adams et al, 2007; Gyori et al, 2017). However, to the best of our knowledge, none of them consider environment or build-time configuration settings. While build impact analysis has been shown to be effective (Wen et al, 2018; Adams et al, 2007), current techniques rely on a dynamic analysis of build execution, which cannot expose the impact of a change on different environmental and configuration settings.

Therefore, we propose DiPiDi, an approach to assess the exposure of changes to the source code of systems using the build system specification files. One of the key roles of the build system is finding and selecting files based on build scripts, build-time configurations, and environmental variables (Zhou et al, 2014; Seo et al, 2014; Al-Kofahi et al, 2012). By statically analyzing the build scripts and constructing the *Build Dependency Graph* (BDG), we can assess the exposure of a change on all software variants.

To evaluate the proposed approach, we conduct an experiment to assess the effect of DiPiDi on the effectiveness and efficiency of determining the exposure of source code changes on projects that are using CMake build system.[2] To that end, we form three participant groups – those with no tool assistance, those with the assistance of a CIA tool, and those with the assistance of DiPiDi – and compare their efficiency and effectiveness on prescribed tasks. The participants are asked to identify the impacted deliverables and variants for given source code changes while we monitor their performance. A tool that could significantly improve effectiveness and efficiency for these tasks could be useful in many applications both for researchers who design experiments based on source code change (e.g., mutation testing) (Rovegård et al, 2008) and practitioners in the allocation of quality assurance resources.

Result: Our results indicate that without tool support, identifying impacted deliverables is a difficult task, even for experienced developers. Members of the No Tool group obtained the lowest F_1-score in Task Type A and the highest rank distance in Task Type B despite having more experienced developers and professional CMake users than other groups. Moreover, our results suggest that DiPiDi helps developers to identify impacted deliverables more effectively than current solutions. Indeed, the identified impacted deliverables by the members of the DiPiDi group are 32, 40, 36 average percentage points better in terms of precision, recall, and F_1-score over the members of the Existing Tool group. Moreover, we find that developers using our approach identify impacted targets more efficiently than others. DiPiDi results in 42% average task time reduction when compared to the approach used in the positive control group.

The remainder of this paper is organized as follows. We first describe our research questions in Section 2. In Section 3, we present and describe our approach called DiPiDi and its prototype implementation. In Section 4, we describe the design of the experiment that we use to evaluate DiPiDi. In Section 5, we present the results of our experiments. We situate our work with respect to the literature in Section 6 and then Section 7 discloses the threats to the validity of our approach and experiments. Finally, Section 8 concludes the paper.

The data that support the findings of this study are available on request from the corresponding author Meidani, M. The data are not publicly available due to them containing information that could compromise research par-

[2] This study has been reviewed and received ethics clearance through the University of Waterloo Research Ethics Committee (ORE# 43727)

ticipant privacy and ethical constraints. Nonetheless, we share the technical artifacts and questions in our repository.

2 Research Questions

In this study, we aim to determine whether a static analysis of build systems can improve the effectiveness and efficiency of software developers striving to assess the exposure of a source code change.

Despite the importance of understanding exposure, we conjecture that it is difficult to assess without tool support. To this end, we propose DiPiDi to improve awareness of the exposure of changes. We hypothesize that DiPiDi will allow developers to more efficiently and effectively determine the exposure of source code changes.

A source code change, or patch, that impacts an application under a specific and rare configuration would likely not merit as much developer attention as a source code change that always impacts the application. A change that impacts more deliverables and/or configurations (high-exposure) has a broader "surface area" and a greater potential to impact users, should a defect be introduced, than a change with low-exposure. Therefore, we believe that knowing which deliverables are affected by a source code change or a patch can allow developers to make more informed decisions when making source code changes. To investigate whether DiPiDi approach help developers to identify the impacted deliverables, we formulate the following research question:

RQ1: Does DiPiDi help developers assess the exposure of source code changes more effectively?

While finding all of the deliverables impacted by a change is important, it also is time-consuming because it requires project-wide knowledge, an understanding of the relations between the files and the build system. Developers attempting this task must identify the modified source code throughout the project and trace them through the build dependency graph, while taking care to consider build-time configuration settings. Some of these configurations may be related to the environment of the user, like the operating system. So, a change may have a side-effect on one machine without appearing on others. On the other hand, build-scripts may use wildcard addressing, like **.cpp*, for the source files, making it challenging to follow a complete compilation path from a deliverable to the changed source file. Therefore, developers may rely on heuristics (e.g., directory structure), or worse, ignore this important step in assessing the risk of a change. We pose the following research question to explore the efficiency of developers while using DiPiDi:

RQ2: Does DiPiDi help developers assess the exposure of source code changes more efficiently?

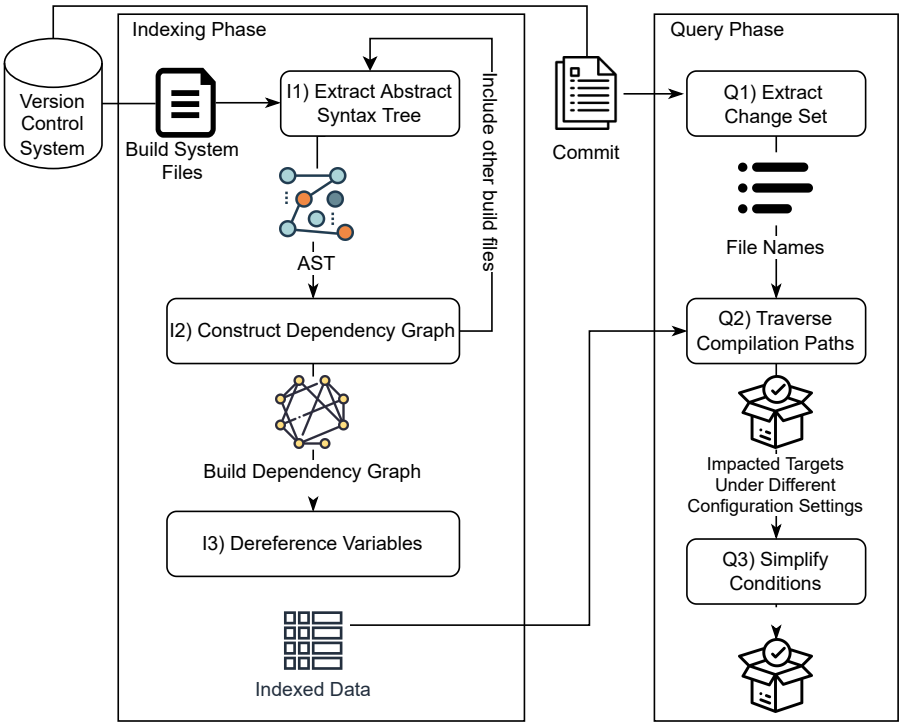


Fig. 2: An overview of the DiPiDi approach

3 DiPiDi

An overview of the DiPiDi approach can be found in Figure 2. The approach has two main phases, the *Indexing Phase* and the *Query Phase*. The purpose of the *Indexing Phase* is to construct an internal representation of the build system. This internal representation includes all the possible compilation paths from each deliverable to the source files. This data can be stored and used later in the *Query Phase*. The purpose of the *Query Phase* is to allow DiPiDi to leverage the data constructed by the *Index Phase* and to output the impacted deliverables under different configuration settings given a set of changed file.

Implementation: In order to conduct our study, we produce a prototype implementation of DiPiDi for the CMake build system. CMake is a cross-platform build system that builds deliverables from artifacts, like source files (Kitware, 2020). CMake has two distinct phases. First, it generates platform-based low-level build specifications (e.g., Makefiles, Visual Studio #.sln files, or Ninja files) (Martin and Hoffman, 2010). Then, CMake invokes the low-

level build tool like `make` to build the project. Our implementation is available online on our public GitHub repository.[3]

We explain each step of the approach presented in Figure 2 in more detail below.

3.1 Indexing Phase

We explain our approach for each step in the *Indexing Phase* in more detail below. Some steps may require an implementation tailored to the build system being used. In those cases, we also explain our implementation for the prototype of DiPiDi.

I1) Extract Abstract Syntax Tree

Every build system has its own entry file to start building the project. For example, GNU Make looks for a file named `Makefile` in the root of the project. The entry file describes how to build the project using the build system specific language. The project can contain helper build files in other folders or split the entry file and relocate it into multiple folders. All of those files should be addressed and included in the entry point file. To capture the content of the build files, we parse the entry file and build an Abstract Syntax Tree (AST) using a parser that understands the build system grammar. The output of this stage is an AST for one build system related file.

Implementation: Projects using CMake should contain `CMakeLists.txt` in their root directory as the entry file for CMake. Other helper files which have `.cmake` extensions can be in other folders. The tool first parses the CMake specifications starting with the `CMakeLists.txt` file in the project root directory. We use ANTLR (Parr and Quong, 1995) to parse and build the *Abstract Syntax Tree (AST)* from the CMake file. The grammar for CMake is straightforward since CMake commands follow the same structure which can be captured by the following parser rule:

```
command_invocation
  : Identifier '(' (single_argument|compound_argument)* ')'
  ;
```

I2) Construct Dependency Graph

Next, we traverse the AST to construct the *Build Dependency Graph*, which represents the relationship between the deliverables, source files, and the conditions in each compilation path from deliverables to source files. Table 1 shows the different node types used in DiPiDi to construct the Build Dependency Graph from the AST. In this step, DiPiDi also creates a lookup table for

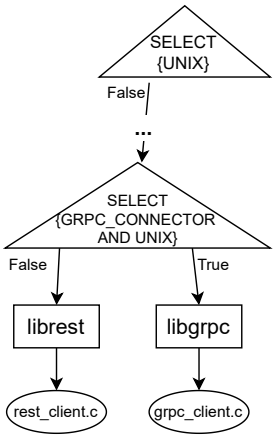
[3] https://github.com/software-rebels/cmake-inspector

Table 1: Type of nodes in Build Dependency Graph generated by DiPiDi after traversing the AST

Type	Description	Example Command
TargetNode	Represents a target or deliverables in the project. This node may depend on other nodes to show dependency between a deliverable on libraries, variables, or a list of source files.	add_executable
RefNode	Shows explicitly defined or environmental variables. This node often depends on another node such as a ConcatNode to represent a list or a LiteralNode to show the value of the variable	set
OptionNode	Shows the user-defined build-time configurations in the project.	option
LiteralNode	Represents literal strings or numbers. RefNodes or TargetNodes may point to these nodes to show the value of a variable or source files for a target.	“foo.cc”
SelectNode	Shows conditional paths which have three properties: a condition, a `True` path, and a `False` path.	if
ConcatNode	Represents multiple possible values for a node which should be concatenated together and it points to two or more other nodes	list
CustomCommandNode	All other commands in CMake are represented by this node which can point to an arbitrary number of nodes showing different arguments for a command	find

each of the variables and targets found while traversing the AST. Some build systems like CMake support scoping for the variables, while others like GNU Make do not. To enable scoping, the lookup table dynamically changes as we parse other files or functions.

As we reach each AST node, based on the name of the command, we select a corresponding node from Table 1 and use the lookup table to find the variables and other nodes that this node may depend on. In this step, we cannot assign values to the variables since they might have different values based on the paths we took to reach to them. As an example, consider a variable called `srcs` holding a list of source files. Based on the operating system, the build system may append some additional files, like `foo_arm.cc`, to that variable. Thus, we only keep the nodes and their dependencies. At this level, we may need to include and parse other build-related files found while traversing the AST by repeating the previous step.



(a) Part of a dependency graph from the iSulad project.[4]

```
def flatten(node, cond):
  if instance(node, SelectNode):
    if satisfiable(cond + node.cond):
      result += flatten(node.trueNode, cond +
    node.cond)
    if satisfiable(cond + Not(node.cond)):
      result += flatten(node.falseNode, cond +
     Not(node.cond))
  elif instance(node, ConcatNode):
    result += flatten(child) for child in node
    .getChildren()
  elif instance(node, LiterlNode):
    result += node.getValue(), cond
  elif ...
    pass
  return result
```

(b) Flattening algorithm for the SelectNode

Fig. 3: When flattening the second SelectNode in (a), the approach should remember the `UNIX=False` assumption from the first SelectNode, prune the `True` path and only consider the `False` path.

At the end of this step, DiPiDi has a graph and a lookup table representing the whole project under analysis, variables, source files, conditions, and targets.

I3) Dereference Variables

Often in large software applications, there are build-time configuration and environmental settings that help the build system to reason about different variants of the system (Liebig et al, 2010; Hochstein and Jiao, 2011). These settings create different dependency paths from the deliverable to the source files. In the generated *Build Dependency Graph*, the target nodes which represent the deliverables reside at the top and the leaves are source files represented by LiteralNodes. Using this graph and starting from a target node, we traverse the graph down to the leaves and resolve variables to their values under different build-time configuration settings (i.e., flatten the variables).

By flattening the variables, we obtain all of the possible values for each variable for all configuration settings. This information is then saved and can be accessed through an API when attempting to determine the exposure of a source code change.

To evaluate the expressions and conditions while flattening the variables, we used Z3 (Moura and Bjørner, 2008), a library that determines whether a formula is satisfiable, developed by Microsoft Research. Z3 supports formu-

[4] `https://gitee.com/openeuler/iSulad`

```
{
"dl_main_curl.c": {
        "FEATURE_CURL": ["etl"]
    },
"dl_main_stubs.c":{
        "NOT FEATURE_CURL": ["etl"]
    },
"common.c": {
        "": ["etl"]
    }
}
```

Fig. 4: An example of the output of the tool based on the given graph in Figure 1. Each key in the root dictionary is a source file in the project. For each source file, another dictionary with conditions as keys and targets as values represents the impacted target given a change which includes the source file.

las involving Boolean, numbers, and strings. We keep track of the evaluation of each condition along the path to prune the build dependency graph while reaching each SelectNode. Figure 3(a) shows two SelectNodes in one compilation path. The algorithm does not have any assumption about the variables when it reaches the first SelectNode, which has a condition on the `UNIX` variable. Thus, it expands both paths and calls the algorithm to flatten each path with different assumptions, `UNIX=True` for the True path and `UNIX=False` for the False path. Given the assumption for the False path, when the algorithm reaches the second SelectNode, which has a condition on `UNIX And GRPC_CONNECTOR`, it does not expand the True path because it is not satisfiable, as `UNIX` is False. The output of this phase is all the compilation paths from each target down to the source files with the conditions that are being held as True during the path. An example of the output is shown in Figure 4.

3.2 Query Phase

This phase uses the index data generated from the previous phase to find the impacted deliverables. The steps of this phase are described below:

Q1) Extract Change Set

A commit in the version control systems contains a list of changed files. Since DiPiDi operates at the file level, which is the same granularity as the build system, we need the changed file names to start the impact analysis. The output of this step is a list of changed source files in a commit.

Q2) Traverse Compilation Paths

Given a list of file names and the output of the *Index Phase*, we can search all of the compilation paths that include the changed file and create a list of exposed targets. In this step, the user can optionally add some assumptions on the configuration settings. Since we store the required conditions for each path, we can use the Z3 library to filter out the paths which are not reachable given the conditions set by the user. The output of this step is a list of exposed targets and the conditions under which each target will be impacted by the change.

Q3) Simplify Conditions

Since we add each condition to our assumptions while traversing a compilation path, the list of conditions generated by the previous step may contain duplicates and can be simplified. In Z3, we can pass functions to the reasoning engine for custom processing steps. These functions are also known as tactics.[4] We use the following tactics to simplify the assumptions:

- `propagate-values`: This tactic propagates the value of each variable between assumptions. For example, if we have an assumption that `a = 0` and `b > a`, we can simplify the second assumption to `b > 0`.
- `propagate-ineqs`: We then propagate the inequalities and remove the subsumed ones. For example, if we have `a > 10` and `a > 2` in our assumptions, we can remove the second one since it is always True if `a` is greater than 10.
- `ctx-solver-simplify`: Finally, we remove the assumptions that are always True. As an example, if we have `a, b, a AND b` in our assumptions, we can remove the third one as it is True.

The output of this step is the same as the previous one with some simplification on the condition list.

4 Research Protocol

To test our hypotheses, we conduct randomized controlled experiments with the three groups defined. Study participants are asked to perform a set of prescribed tasks with their usual development setup without additional help (control group), with a baseline change impact analysis tool (positive control group), and with DiPiDi (treatment group). We measure the effectiveness of our tool by comparing the responses of the participants with an established ground truth. We measure the efficiency of our participants by comparing the duration of each task across the groups.

[4] https://www.philipzucker.com/z3-rise4fun/strategies.html

4.1 Variables

This section presents an overview of the study variables, which are further described below.

4.1.1 Independent Variable

In our study design, the tool support provided to the participants varies (*No Tool*, *With Existing Tool*, and *With DiPiDi*) which is represented by the *Tooling Level* independent variable. All tooling levels have access to the same information and interface. The only difference in access is the additional output of the Existing Tool/DiPiDi for the relevant groups. More specifically, each group is defined as follows:

1. **No Tool**. This group has access to the code change and other files in the project, including the build specifications. They can use their preferred development environment to perform the tasks. This group is a control group and represents the current practices used by software developers attempting to determine which deliverables are affected by a source code change.
2. **Existing Tool**. This group has access to the same environment as the *No Tool* group, as well as the output of the change impact analysis generated using the Understand Tool (SciTools, 2022). This group is a positive control group and represents the current approaches used by software engineering research to aid software developers attempting to determine which deliverables are affected by a source code change.
3. **DiPiDi**: This group – the treatment group – has access to the same environment as the *No Tool* group, as well as DiPiDi. The participant can interact with the tool using the Query Interface as described in Section 3. Our tool can print the impacted deliverables at the file level. Although the file granularity may overestimate the true impact of a change, it is the granularity at which the build system operates.

Participants in all groups may use any external tool that they feel may be helpful. Thus, even the results from the No Tool group can be viewed as a baseline set of current approaches used by the developers. We collect the names of the tools that our participants used and report them in Section 5.

4.1.2 Dependent Variables

Our dependent variables are outlined in Table 2. We discuss our reasoning for these variables below.

1. **Exposure analysis effectiveness**: The score from each task indicates how close the answers of the participants are to the ground truth. We could alternatively determine if a participant provides fully correct answers for each task and consider the ratio of correct answers to total tasks. However,

Table 2: The dependent variables of the study

Name	Description	Scale	Operationalization
Number of correctly identified deliverables	Ratio of the impacted deliverables correctly identified by the participants under a specific build-time configuration over the known impacted deliverables (RQ1)	ratio	Computed at the end using the harmonic mean (F-measure) for task types A & C. See Sections 4.3 & 4.6.2
Relative rate of correctly identified deliverables	Normalized pairwise disagreements between participant rankings of patches in terms of the number of impacted deliverables, and known correct rankings (RQ1)	ratio	Calculated at the end for tasks of type B. See Section 4.6.2
Exposure analysis effectiveness	The sum of the number of correctly identified deliverables and relative rate of correctly identified deliverables (RQ1)	ratio	Computed at the end using the number of correctly identified deliverables and the relative rate of correctly identified deliverables.
Task time	The time needed for each participant to complete a task subtracting pauses (RQ2)	ratio	Measured by our web application. The participant can pause a task and resume manually. See Section 4.2.4
Exposure analysis efficiency	Ratio of the total score of the participant over the sum of all Task times (RQ2)	ratio	Total score is the sum of the scores of all of the individual tasks. See Section 4.6.2

we believe that our approach, which indicates how close participants are to fully correct answers, allows us to obtain a finer-grained insight into how participants complete their tasks. Thus, we consider our task scores (i.e., *Number of correctly identified deliverables* & *Relative rate of correctly identified deliverables*) to be good proxies for exposure analysis effectiveness.

2. **Exposure analysis efficiency**: We define exposure analysis efficiency as the duration from the initiation to completion of each task in seconds. As a result, completing tasks more rapidly will result in higher efficiency. This way, we consider both the fully correct answers and the partial ones, especially in the rank-based tasks (see Section 4.3).

4.1.3 Confounding Variables

Because different code changes might affect the results of our participants, we control the code changes made available to them. The confounding variables that we consider for our study are presented in Table 3. We present patches from three different projects to ensure our results are not biased towards any single project. We also control build-time configuration settings to

Table 3: The confounding variables of the study

Name	Description	Scale	Operationalization
CMake experience	Participant's experience in working with CMake build system	ordinal	Measured: 3-point scale ("none", "tried", "used in professional development"); questionnaire
Code changes	Changed code in diff format along with the other source files of the project	nominal	Design: each participant gets patches from three real-world projects
Configuration settings	Environmental and build configuration settings of the build system: default configuration, custom	nominal	Design: for applicable tasks, each participant gets two configurations for build settings.
Current programming practice	How often the participant currently programs	ordinal	Measured: 3-point scale ("not", "sometimes", "often"); questionnaire
Development experience	Participant's software development experience in years	ordinal	Measured: 5-point scale ("less than a year" ... "10 years or more"); questionnaire
Fitness	Physical fitness of the participant, like tiredness, during the experiment	ordinal	Measured: 5-point scale ("very tired" ... "very fit"); questionnaire
Perceived task difficulty	Participant's overall perception of the task provided during the experiment	ordinal	Measured: 3-point scale ("easy", "average", "hard"); questionnaire at the end
Project-specific experience	Participant's past experience with the provided project and patch	ordinal	Measured: 3-point scale ("none", "user", "contributor"); questionnaire at the end

evaluate tooling levels with multiple build configurations without introducing confounding factors. We gather demographic information like the *Development experience* in order to control their correlation with the dependent variables. We also use these variables to inform our data preprocessing (e.g., provide context to determine why a participant might not have finished a task) and for further analysis. We use this data to augment the statistical analysis and make decisions about whether a participant is suitable for a task.

4.2 Materials

In this section, we describe the materials that we use in this study.

4.2.1 DiPiDi

We developed a prototype implementation of DiPiDi to reveal the exposure of a change in a structured manner. In a nutshell, our tool processes build specifications statically to produce a *Build Dependency Graph (BDG)*, which we traverse to assess exposure. Before conducting the experiments, we perform the Indexing Phase on the projects that are being presented to our participants and save the output. Participants in the DiPiDi tooling level of the experiment use tool's querying features to perform the assigned tasks.

4.2.2 Existing tool

To assess whether the improvements in the DiPiDi tooling level (treatment) group are related to the approach implemented by our tool, we select a recent and available impact analysis tool to employ in the *Existing tool* (positive control) group.

Unfortunately, most of the proposed impact analysis tools are prototypes (Li et al, 2013). Additionally, due to our project selection and since our implementation of the DiPiDi approach supports CMake build specifications, the impact analysis tool must support the C++ programming language. For example, we had originally selected Frama-C; a tool proposed by Kirchner et al (2015). Frama-C is an industrial-grade static analysis tool, which can perform impact analysis on C and C++ projects. However, Frama-C only works on C++ projects with the help of an early access plugin, which has limited support called Frama-Clang, which converts C++ code to plain C code before running other analysis in Frama-C. This plugin is in its early stage of development and has known issues, as mentioned on the official Frama-C website.[5] While we originally believed that this plugin would allow us to complete comparisons with DiPiDi, in our case, Frama-C could not parse or convert any of the projects that we analyzed in the study. This appears to be due to new syntax introduced in C++17 which is currently not supported by the Frama-Clang plugin.

Therefore, as a replacement, we decided to use Understand, a commercial tool developed by SciTools (2022) and used in previous studies (Orrú et al, 2015; Fontana et al, 2011). Understand is a comprehensive static analysis tool with more than 100 features. However, acquiring a license, installing, and applying the Understand to each of the studied projects would be unwieldy for our participants; thus, it is not applicable to use in our study as is. Fortunately, Understand's features are also available through a Python API. Therefore, we develop a presentation layer for Understand's impact analysis API and

[5] `https://frama-c.com/fc-plugins/frama-clang.html`

represent the result in a web application for the participants. This allows our participants to access useful Understand functionality without the burden of installing and applying it. More specifically, for each project:

1. We extract the list of function-level dependencies from Understand.
2. We keep track of where functions are defined in each file as reported by the Understand tool.
3. We persist the result in a structured format (i.e., JSON) that can be consumed by our web application.

Later during the study, the participants can paste a commit ID into the web application to produce Understand-based impact result for the changed program elements. The web application extracts a list of changed functions from the commit and identifies impacted files by traversing the dependencies that Understand computes. Note that the existing tool provided to the participants is simply a presentation layer for the Understand tool—all of the results presented to the participants are therefore calculated by Understand.

The difference between Understand and Frama-C is that Understand operates at the function level, while Frama-C can analyze impacts at the statement level. However, when using Frama-C, the user should install the tool, import the project, and manually select the statements that changed in the commit. By leveraging the API of Understand, however, we make the results of the existing tool accessible through a web interface. The script we use to persist the structured function-level data is available in our repository.[6]

While DiPiDi identifies the impacted deliverables by statically analyzing the build code and considering all build-time configurations, Understand (and other impact analysis tools available today) identify the impacted files by analyzing the source code of the project, without considering build configurations. Then, it is the responsibility of the developer to find the impacted deliverable by matching the file names with the build code.

4.2.3 Studied Projects

Table 4: Summary of the selected projects

Name	Line of Codes	Commits
ET: Legacy	3,706,703	11,047
libuv	113,414	4,928
Box2D	128,474	1,282

We select three projects using GitHub search. We first select projects that mentioned CMake in their README file, and then sort by the number of stars for each project. A summary of the selected projects is presented in Table 4.

[6] `https://github.com/software-rebels/cmake-inspector/blob/master/UNDGraph.py`

As explained in Section 4.3, participants are asked to rank three patches based on their impact on the project, e.g., a patch that impacts three platform-specific versions of the project has higher rank than a patch that impacts one platform-specific version. Participants are also asked to identify the configuration settings in which the changes in the patch propagate to the project deliverables. Thus, for each studied project, we iterate over patches in reverse chronological order, selecting patches that impact a different number of deliverables under different configuration settings until three patches have been selected (nine patches in total). To identify the impacted deliverables, we manually inspect the source files and find the deliverables that are impacted by the changed code. We use this as our ground truth. While DiPiDi reports changes at the file-level, in this study, participants are asked to report impacted deliverables at the code level, a subset of reported deliverables by the tool.

4.2.4 Experiment UI

Figure 5 shows the web interface for our prototype. As soon as DiPiDi completes the indexing phase, the web interface connects to the tool using a Remote Procedure Call. In the first section, the user can either choose a changed file or select a commit. In the second section, the user can add the build configuration settings, which can be Boolean, string, or arithmetic conditions. Although more complex types of expressions are possible, we leave their evaluation for future work, since simple expressions are already pushing the limits of what our control group can handle. Additionally, we only support the `equal` operator in the web interface; however, DiPiDi supports other operators (e.g., `>, <, etc.`). The web application issues the request to a backend service, which processes the DiPiDi query. The results are then communicated to the frontend, and the impacted deliverables are presented in the third section. On the backend side, DiPiDi first iterates over the indexed data to identify the targets that are impacted by the changed files. Then, DiPiDi applies the specified conditions (if any were provided) using the Z3 library. If the conditions are still specifiable, DiPiDi adds the target to the impacted list and returns the final list to the web application. This application is available in our repository.[7]

We additionally developed an interactive Web based application to allow us to conduct our experiment with a diverse range of participants and allow our participants to rely on their own development environments. The application retains a log of answers and the duration of each task. The experiment UI randomly assigns each participant to a tooling level group and randomly assigns tasks to the participants, all the while logging which project and tasks are assigned to whom. Participant information was only made available to the researchers after all the results had been scored to reduce experimenter bias (Rosenthal, 1976). The interactive UI is also available in our repository.[8]

[7] https://github.com/software-rebels/dipidi-experiment-ui

[8] https://github.com/software-rebels/dipidi-participants-ui

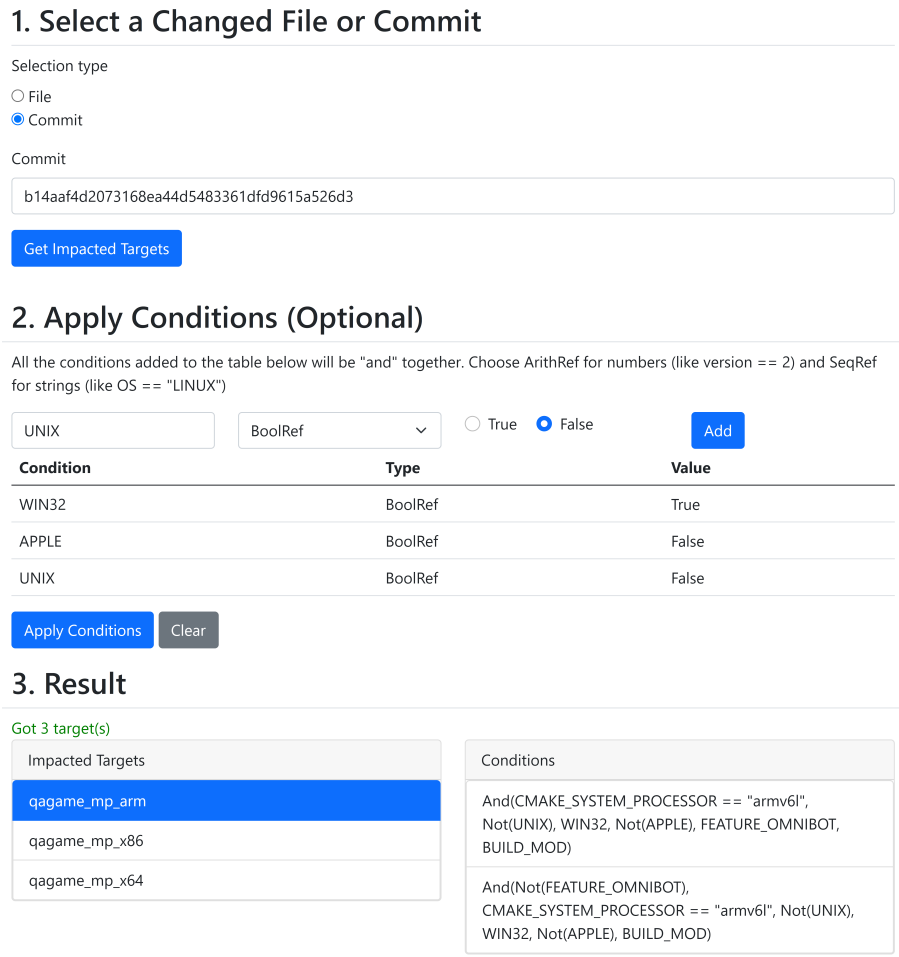


Fig. 5: DiPiDi Web Query Interface

4.3 Tasks

We ask our participants to complete five tasks, one Type A task, two Type B tasks, and two Type C tasks. After a participant initiates our experiment through our experiment UI, they are randomly assigned to a tooling level and the tasks are randomly ordered and logged. The order of the tasks is randomized to account for learning effects that could occur if developers improve by learning from previous tasks. Furthermore, we construct each task using three different open-source projects, and randomly assign each task to each participant. Therefore, participants cannot share answers with each other, and tasks are less biased towards a specific project or task. Participants must obtain the

data and files required to complete each task through our experiment UI and must also provide their answers through it.

Our tasks are constructed to answer both RQ1 and RQ2. The results obtained for each task can be used to answer our first research question (i.e., RQ1), while the duration of the tasks can be compared for each group to answer RQ2. The three task types are as follows.

Task Type A: The purpose of this task is to compare the exposure assessment effectiveness and efficiency of the participants in different tooling levels. The participant is provided with the names of changed files and a set of build specifications. The participant is then asked to list impacted deliverables (without having the source code). The experiment UI provides a text input field for the participant to identify those deliverables.

Task Type B: The purpose of these tasks is to determine the effect of presenting exposure reports on the effectiveness and efficiency of developers assessing the relative exposure of patches. The participant is assigned three patches and a set of build specifications. We ask the participant to rank the patches listed in the experiment UI based on (a) the number of impacted deliverables; and (b) the number of impacted application variants (e.g., number of affected OS). We ensure that the patches do not affect the same number of deliverables and application variants. Furthermore, the patches are sampled from a different project than the ones studied for other tasks.

Task Type C: The purpose of these tasks is to determine the impact of DiPiDi when participants are particularly interested in the exposure in a given setting. Participants are presented with three patches and asked to identify those that (a) affect a specified set of deliverables; (b) affect a specific variant of the software; and (c) identify the configuration settings under which the changes will propagate. For this task type, we use a different project than for tasks of types A and B to make sure that all of the participants see examples from each of the three projects that we selected for this study.

4.4 Participants

Since our tasks are centred around specific software engineering practices, our participants should have the programming experience necessary to allow them to find the deliverables impacted by a source code change. We therefore populate our pool of participants with software developers, or individuals with programming experience.

We solicited participation from CMake user web forums, the developer mailing lists of large projects that are implemented in CMake (e.g., KDE, Qt), at a user summit of a code reviewing platform, via our personal contacts on social media, and a local group of graduate students, all of whom have developed software in a practical setting. A total of 72 participants enrolled in the study. We piloted the experiment UI and tasks with two participants. The pilot results are not included in our reported findings below. The remaining 70 participants were invited to participate in the study. Out of those, 34 par-

Table 5: Demographic information about the participants

Tooling Level	ID	Programming Experience	CMake Familiarity	Current Programming Practice
DiPiDi	P1	five years or more	Tried it at least once	More than once per week
	P2	five years or more	Used in professional development	More than once per week
	P3	five years or more	Tried it at least once	More than once per week
	P4	two to five years	Tried it at least once	More than once per week
	P5	five years or more	None	More than once per week
	P6	five years or more	None	More than once per week
	P7	two to five years	Tried it at least once	Sometimes
	P8	two to five years	Tried it at least once	More than once per week
	P9	five years or more	Tried it at least once	More than once per week
	P10	a year to two years	None	Sometimes
	P11	two to five years	Tried it at least once	Sometimes
Existing Tool	P12	two to five years	Tried it at least once	More than once per week
	P13	five years or more	Tried it at least once	More than once per week
	P14	two to five years	Tried it at least once	Sometimes
	P15	five years or more	Tried it at least once	More than once per week
	P16	two to five years	Tried it at least once	More than once per week
	P17	a year to two years	Tried it at least once	More than once per week
	P18	two to five years	None	More than once per week
No Tool	P19	five years or more	Used in professional development	More than once per week
	P20	two to five years	Used in professional development	More than once per week
	P21	five years or more	Used in professional development	More than once per week
	P22	five years or more	Used in professional development	More than once per week
	P23	five years or more	Tried it at least once	More than once per week
	P24	two to five years	None	More than once per week
	P25	five years or more	Tried it at least once	More than once per week
	P26	two to five years	Tried it at least once	More than once per week
	P27	two to five years	Tried it at least once	More than once per week
	P28	two to five years	None	More than once per week
	P29	five years or more	Tried it at least once	More than once per week
	P30	five years or more	Tried it at least once	More than once per week
	P31	two to five years	Tried it at least once	More than once per week
	P32	five years or more	Used in professional development	More than once per week

ticipants completed the set of tasks. Of those who finished, two participants skipped at least 3 tasks, so we exclude them from further analysis. In the end, 32 participants remain – eleven in the DiPiDi group, seven in the Existing Tool group, and fourteen in the No Tool group. Table 5 shows an overview of the profiles of the participants in this study.

4.5 Execution Plan

We provided our participants with access to our web application in batches of three. This staged approach allowed us to fix any potential problems without

invalidating too large of a subset of our participant data. Based on the feedback that we received, we clarified the task descriptions with additional detail, but the tasks themselves remained the same. We enhanced the experiment UI to indicate when the backend is processing the issued query, as processing queries took on the order of five to ten seconds, and users would mistakenly submit multiple requests. The application has the following procedure for each participant:

4.5.1 Welcome Page

Participants are first presented with an outline of the tasks and an estimate of the time required to complete the tasks. In addition, we request the consent of participants to participate in the experiment. The participants are asked to refrain from sharing task information with other participants. For ethical compliance reasons, participants are also informed that they may stop the experiment at any time for any reason.

4.5.2 Onboarding

After obtaining consent from the participants, we provide a more detailed explanation of the specific set of tasks to be completed during the experiment. Based on the tooling level assigned to the participant, we explain the steps required to prepare the environment and the tool (if applicable). We inform participants that they may use their preferred development tools (e.g., CLI tools, IDE). Participants are also informed that each task is timed, that their responses will remain anonymous unless they explicitly request otherwise, and that they may skip individual tasks.

4.5.3 Tasks

We present our participants with the tasks outlined in Section 4.3 in a random order. For each task, our application provides a hyperlink to download the source code. A timer begins as soon as the task page is loaded. We also record when checkpoints are reached during the experiment. Before showing the description of the task, we provide the download link and the necessary steps to prepare the environment. The participants must click on the "ready" button to initiate the experiment. We also log the moments that the participants begin to enter their responses. The page describes the task and shows the configuration settings that the participant should consider. We present the results of the tools in the experiment UI for participants in the 'Existing Tool' and 'DiPiDi' tooling levels in an interactive way through a Web interface. The application provides input spaces for the participant to enter their responses. The application logs the time that the participant spent on each task. The participant may click a pause button to pause the timer if a distraction of any kind interrupts their focus. A skip button allows the participant to move on if

they feel that they cannot complete a task. A sample of each task is provided in appendices C to H.

4.5.4 Questionnaire

Prior to the start of the experiment, the participants are asked demographic questions about their background and programming experience. The questionnaire is included in Appendix A. After a participant completes their five tasks, we follow up with a questionnaire which is included in Appendix B. The purpose of the post-study questionnaire is to collect tool usage questions about the CLI tools, IDEs, and/or other tools that were used to complete the tasks. Additionally, we ask whether the participants found the provided tool useful. We also ask participants to comment on any problems that they may have encountered during the experiment. Finally, we thank the participants and invite them to provide other feedback if they desire. The results of the post-questionnaire are presented in Table 9.

4.6 Analysis Plan

In this section, we describe the analysis plan we use in this study.

4.6.1 Data Cleaning

We assign each participant five tasks to complete. However, it is possible for a participant to exit the application before completing all of their assigned tasks. Since the experiment UI accepts input from participants in any text format, we manually check that answers are acceptable before analyzing them. Next, we review the participant's questionnaire submission and feedback for mentions of problems that may (partially) invalidate their submission, removing their invalid answers when appropriate. Additionally, we use outlier detection approaches, i.e., Tukey's fences (Tukey et al, 1977) and box plots, which do not require regression models. If there are outliers, we analyze them by hand to gain insight into them. Finally, we remove those data if we find enough evidence to do so after both outlier detection and manual evaluation.

4.6.2 Measuring Effectiveness

For rank-based tasks, i.e., *task type B*, we use Kendall's tau ranking distance formula (Kendall, 1938) to compute the distance between participant answers and the ground truth. Kendall's tau ranking is defined as:

$$K_d(\tau_1, \tau_2) = \sum_{\{i,j\} \in P, i<j} \bar{K}_{i,j}(\tau_1, \tau_2)$$

where P is the pairwise set of elements in τ_1 and τ_2, $\bar{K}_{i,j}(\tau_1, \tau_2)$ is 0 if i and j are in the same order in τ_1 and τ_2 otherwise it is 1. For example, the

Kendall's tau distance between `2,1,3` and `1,2,3` is one because pair $\{2, 1\}$ are in different order. We report the distance as a number between zero and three for those tasks.

For list-based tasks, i.e., *task types A and C*, like previous studies, we compute precision and recall. As discussed, the goal of this study is to expose the change under different configuration settings and help developers to identify impacted deliverables for a specific configuration setting. To compute the correctness and completeness of the participant's Estimated Impacted Deliverables (EID), we compare them to Actual Impacted Deliverables (AID) using the following precision (correctness) and recall (completeness) formulas:

$$Precision = \frac{EID \cap AID}{EID}; Recall = \frac{EID \cap AID}{AID}$$

Due to the natural trade-off between precision and recall, we calculate the F_1-score (i.e., the harmonic mean of the precision and recall) to get an overall impression of task effectiveness.

4.7 Deviations From the Registered Report

This paper is the stage two submission of a registered report accepted at MSR 2021 registered report track (Meidani et al, 2021). To complete the study, we had to deviate from our original registered report protocol during the course of this submission. In this section, we summarise the deviations from the original protocol.

4.7.1 Replacing the Frama-C Tool

While the Frama-C tool was our original choice to compare DiPiDi to an existing tool, we could not make use of it as discussed in Section 4.2.2. We decided to make use of another existing tool capable of analysing code impacted by a code change. The Understand tool has features that allow developers to trace a change and find the parts of a program that it impacts. However, installing the tool, and learning to use it, was not feasible for the participants given the constraints of the study (time and computing environment). Thus, we developed a UI tool that consumes the output of the Understand API, and represents the result in a web application for the participants. Thus, the tool that we develop is a presentation layer for the Understand results. We therefore switched Frama-C for Understand.

Unfortunately, our initial tool selection could not analyze the studied projects; however, we believe that switching from Frama-C to Understand will not substantially impact the performance of the positive control group because (1) both tools are commercial grade and (2) both tools can perform similar styles of change impact analysis via source code analysis. While our original choice may have been easier to use for our participants, we believe that our presentation layer wrapper bridges that gap.

4.7.2 Change of Studied Projects

Originally, we wanted to conduct the study on projects from the KDE and QT communities. However, we found that projects in those communities use customized CMake commands to maximize reuse and productivity among the projects.[9]

Developing support for this set of commands required additional engineering effort for DiPiDi. Unfortunately, we did not have sufficient time to invest the engineering time to implement these supports for custom commands within the Stage 2 registered report submission time frame. Therefore, we systematically selected alternative projects that use the 'vanilla' version of CMake specifications. To identify candidate projects, we sorted repositories that are hosted on GitHub and use CMake, by the number of stars, which we believe is a good proxy of the popularity of a project. We believe that improvements that can work on popular projects are more likely to benefit a larger number of developers. From that list of projects, we selected three projects of varying size and domain for our experiment. Table 4 provides an overview of the studied projects.

4.7.3 Number of Participants

In our registered report, we set out to conduct our study with 66 participants to be able to compare the groups with large effect sizes using one-way ANOVA. Since participants are required to be developers who are familiar with build systems, we faced difficulties recruiting such a large number of developers for this study. We recruited participants using a variety of communication channels, such as social media (Twitter, LinkedIn, Reddit), mailing lists of open-source projects, developer forums, and developer conferences. After leveraging those channels, we ended up with 72 candidates who signed up to participate in the study. Of those, 32 completed at least 4 of the 5 tasks, 11 in the DiPiDi group, 7 in the Existing Tool group, and 14 in the No Tool group.

Due to the limited number of participants, we could not conduct our planned ANOVA analysis. Therefore, we follow our contingency plan and conduct a preliminary analysis of our results instead. The details of our analysis can be found in Section 5.

5 Results

In this section, we present the results of our experiment with respect to our two research questions.

[9] https://linux.die.net/man/1/kdecmake

5.1 **RQ1:** Does DiPiDi help developers assess the exposure of source code changes more effectively?

The participants in the DiPiDi tooling level outperformed the other two groups in terms of their accuracy in identifying the impacted deliverables and assessing the magnitude of the impact. As shown in Table 7, the DiPiDi group outperforms the Existing Tool group by 42 and 31 percentage points in terms of F_1-score for Task Type A and Task Type C, respectively. Moreover, the DiPiDi group outperforms the Existing Tool group by 0.62 units of distance in the impact ranking task (Task Type B).

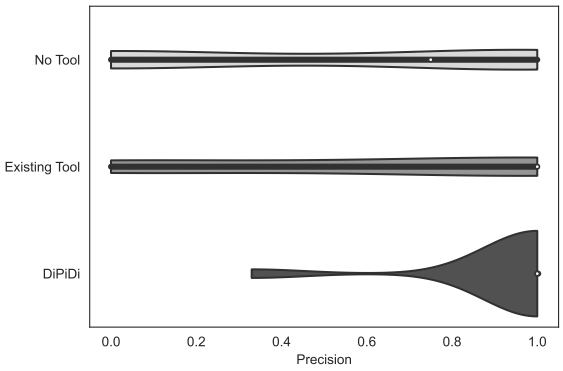
As described in Section 4.3, we assign one Task Type A out of three, two Task Type B out of six and two Task Type C out of nine to each participant. To calculate the metrics shown in Table 7, we compute the average (mean) performance measure across participants in each group to aggregate the measures to the granularity of group comparison. For Task Type B, the distance represents the number of pairwise ranking swaps required to change the order of the participant's answer to match the ground truth. Since we asked the participants to order exactly three commits in Task Type B, the upper bound for this number is three, meaning the order is reversed.

The effectiveness of DiPiDi in Task Type A is also illustrated in Figure 6, which shows the distribution of the Precision, Recall, and F_1-score for Task Type A and each tooling level. In the DiPiDi group, 10 out of 11 participants perform better than the Existing Tool and No Tool groups, achieving an F_1-score of 1 as shown in Figure 6c. However, it also shows a tail extending to 0.33 (P1) in the DiPiDi group. We reached out to P1 to understand if there was any problem with the tasks. P1's experience and familiarity with CMake were limited to a classroom setting. P1 reported that it was difficult to understand the tasks, but despite P1's lack of experience, DiPiDi did help P1 to complete the tasks to a certain degree.

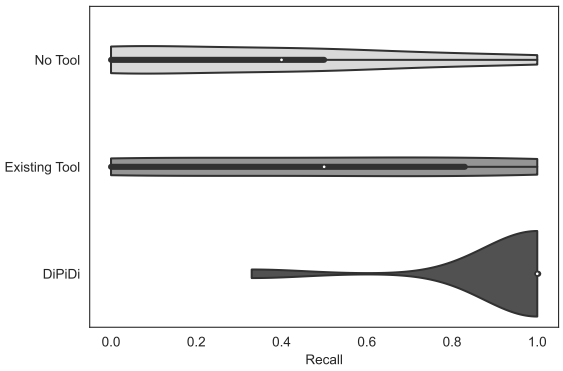
For Task Type B, Figure 7 shows that DiPiDi is effective in identifying the most impactful commits. We believe that accuracy in assessing the riskiness of changes relative to each other can help reviewers and quality assurance teams to manage their resources. Current impact analysis approaches, including the one used for the Existing Tool tooling level, do not consider the build-time configuration settings and, therefore, report the impacted file or statements for a single set of configuration settings (often, the default settings).

Finally, in Task Type C, Figure 8 shows that participants in the DiPiDi tooling level outperform others. The F_1-score for 10 out of 11 participants is greater than 0.9 in the DiPiDi group. Surprisingly, the No Tool group outperforms the Existing Tool group. Since in Task Type C, participants are asked to identify the patches that impact the deliverables under a specific set of configuration settings, a tool that does not consider all the build-time configurations, like the existing tool, may have misled the participants.

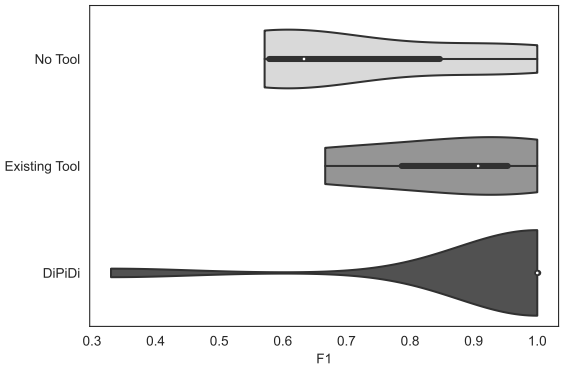
Table 6 shows the participants' expertise in each group. We consider participants to be experienced developers if they have more than five years of programming experience. Additionally, we identify the participants who use



(a) Participant precision for Task A



(b) Participant recall for Task A



(c) Participant F1 Score for Task A

Fig. 6: Participants in the DiPiDi group outperform two other groups in all the three metrics. While the Existing Tool group performs better than the No Tool group, the difference is not negligible.

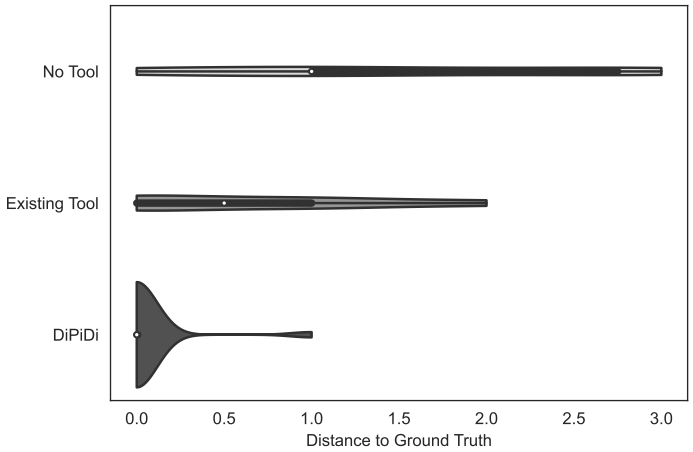
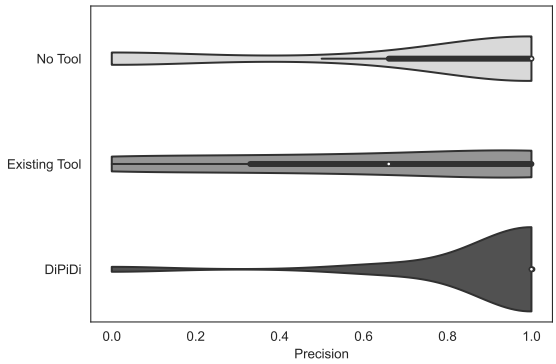


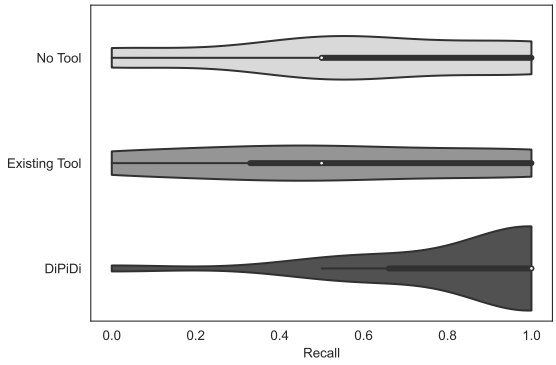
Fig. 7: Distance between the participant's responses to the ground truth calculated using Kendall tau rank distance formula. The larger the distance, the more dissimilar the responses and the ground truth.

CMake in a professional setting. Participants can be neither experienced developers nor professional CMake users if they have two to five years of programming experience and tried CMake at least once. As shown in Tables 6 and 7, assessing the exposure without any tooling support is difficult, even for those participants with extensive professional experience and those who use CMake in a professional setting. Table 9 shows an overview of the post-study questionnaire results. In general, participants in the DiPiDi group find the tool useful and find the tasks less difficult in comparison to other groups. Although we do not draw any firm conclusions about this, the fact that fewer participants find the study difficult suggests that performing with build-related tasks without tool support is daunting.

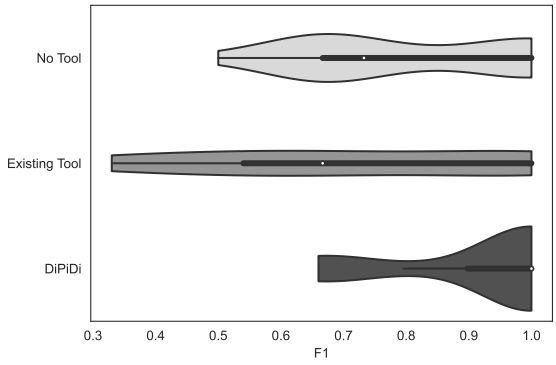
The DiPiDi approach helps practitioners and researchers to identify the impacted deliverables given a change under different build-time configuration settings. In an experimental evaluation, our prototype implementation of DiPiDi outperforms a current impact analysis tool by 36 average percentage points in F_1-score when identifying impacted deliverables. More importantly, participants in the DiPiDi group could assess the riskiness of changes relative to each other with less error.



(a) Participant precision for Task C

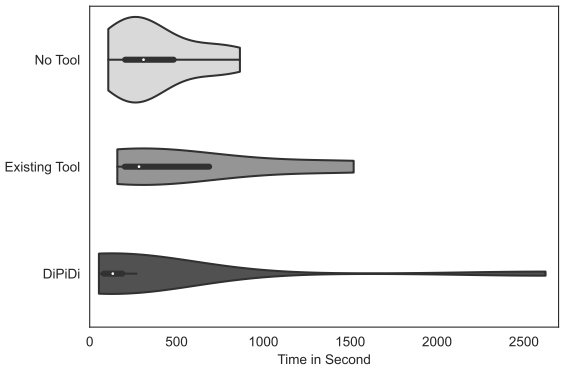


(b) Participant recall for Task C

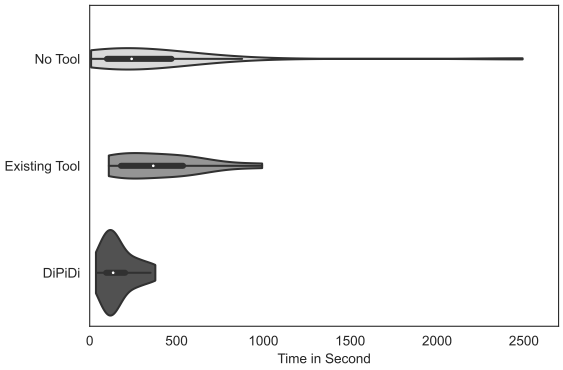


(c) Participant F1 Score for Task C

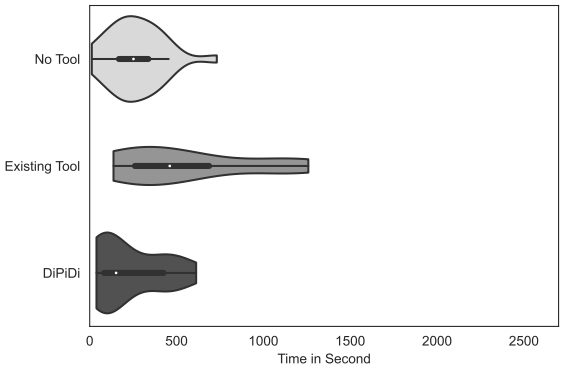
Fig. 8: Participants in the DiPiDi group outperform two other groups in all the three metrics in Task C. Interestingly, the No Tool group outperforms the Existing Tool.



(a) Duration for Task A in Seconds



(b) Duration for Task B in Seconds



(c) Duration for Task C in Seconds

Fig. 9: Participants in the DiPiDi group finish the tasks faster compared to other groups. While the No Tool group performs more efficiently than the Existing Tool, they are not necessarily more effective.

Table 6: Participants' expertise based on the demographic questions. Participants can be in more than one experience category.

Tooling Level	Total	Experienced Developers	Professional CMake Users
No Tool	14	8	5
Existing Tool	7	2	0
DiPiDi	11	6	1

Table 7: Summary of the result for each task per tooling level

	A			B	C		
Tooling Level	Precision	Recall	F_1	Distance	Precision	Recall	F_1
No Tool	0.52	0.33	0.40	1.5	0.77	0.58	0.66
Existing Tool	0.60	0.47	0.52	0.67	0.61	0.52	0.57
DiPiDi	0.94	0.94	0.94	0.05	0.92	0.85	0.88

5.2 **RQ2**: Does DiPiDi help developers to assess the exposure of source code changes more efficiently?

The DiPiDi approach helps developers to assess the exposure of changes more efficiently than baseline approaches. Table 8 shows that the DiPiDi group spent on average 137, 219, 151 fewer seconds on tasks A, B, C, respectively. Figure 9 shows the distribution of duration for each task type in each tooling group. As shown, the majority of the participants in the DiPiDi group complete the tasks more quickly than the participants in the other groups. More specifically, 70%, 81%, and 80% of the participants in other groups performs slower than DiPiDi group in Task Type A, B, and C respectively.

However, P6 took 20 times longer to complete the Task A than other participants in the DiPiDi group. Indeed, P6 completed Task A in 2625 seconds, while the average of the other participants in this group is 127.5 seconds. P6 reported in the feedback form that 'It was hard to understand what to look at in the beginning. My first task is probably affected by that'. P6 performs very well in other tasks and, based on their feedback, believes the tool is very useful. Similarly, P25 in the No Tool group took 3371 seconds to complete Task Type B, which like the above case, was their first task. The average for the No Tool group in Task Type B without P25 is 469 seconds. P25 reported that the experiment was 'Time-consuming and difficult'. Removing those two cases reduce the standard deviation for Task Type A and Task Type B to 67 in DiPiDi group and 256 in No Tool group, respectively and makes all standard deviations less than equal to the average.

Interestingly, except for Task Type B, the No Tool group performs more efficiently than the Existing Tool group. While the Existing Tool group achieves slightly better correctness scores than the No Tool group as shown in Figures 6 and 7, we suspect that the additional information provided to the participants by the existing tool reduced their efficiency.

Table 8: Time it takes for each group of participants to do the tasks

	A			B			C		
Tooling Level	Skips	Time	S.D	Skips	Time	S.D	Skips	Time	S.D
No Tool	3	383	248	2	364	787	3	269	261
Existing Tool	2	570	510	2	401	299	1	552	486
DiPiDi	0	354	720	1	163	133	2	229	236

Still, even for the tasks that require more creative ways of interacting with the tool (e.g., Task Type C), considerable efficiency improvements are detected. For Task Type B, the average time taken for the No Tool and Existing Tool groups is 1.5 standard deviations larger than that of the DiPiDi group. However, for Task Type A and C the difference between the means is less than one standard deviation. We suspect that the differences are not as pronounced because DiPiDi reports many possible compilation paths and participants confirm their answer with the code and the build script in addition to the output of the tool, a time consuming affair.

The DiPiDi approach increases the efficiency of identifying impacted deliverables under different build-time configuration settings. We show that our prototype implementation reduces the time required to assess the exposure of changes by 42% on average. More notably, participants in the DiPiDi group assess the riskiness of changes relative to each other with 92% less error in 59% of the time with 1.7 standard deviations difference over the existing approaches.

5.3 Discussion

In this section, we discuss the results of the study, including the pre- and post-questionnaire data.

Three of the 32 reported that they had difficulty understanding what to do in their first task. However, since we shuffle the order of tasks before assigning them to the participants, the effect of this difficulty is distributed throughout the tasks. For example, Figures 9a and 9b show two tails in the DiPiDi and No Tool groups. In both cases, the participant's first task takes longer than the average because they are struggling to understand the tasks.

Some participants with strong programming experience familiar with build systems performed well even without tool support. For example, P32, who worked with C and build systems during their time researching operating systems in grad school, achieved Precision of 1 and Recall of 0.83 in Task Type A without tool support.

Table 9 shows the results of our post-experiment questionnaire for each participant. We assign numbers to the fitness level, i.e., 1 = very tired, and 5 = very energetic. The average fitness level for DiPiDi, Existing Tool, and No

Table 9: Post Questionnaire Result

Tooling Level	ID	Another Tool	Tool Usefulness	Fitness	Difficulty	Experience
DiPiDi	P1		Somehow	Tired	Hard	None
	P2			Energetic	Easy	None
	P3		Very Useful	Neutral	Average	None
	P4	Intellij IDE	Somehow	Neutral	Average	None
	P5	VSCode	Very Useful	Very Energetic	Average	None
	P6	VSCode	Very Useful	Very Tired	Hard	None
	P7		Very Useful	Energetic	Average	User
	P8		Very Useful	Neutral	Average	User
	P9			Neutral	Easy	None
	P10		Very Useful	Neutral	Easy	None
	P11	VS Code	Very Useful	Tired	Average	None
Existing Tool	P12	VSCode	Somehow	Neutral	Average	None
	P13	VSCode	Very Useful	Neutral	Hard	None
	P14	VSCode	Somehow	Very Tired	Hard	None
	P15	Github Online	Very Useful	Energetic	Average	User
	P16			Very Tired	Average	None
	P17	Clion	Not Useful	Very Tired	Hard	None
	P18		Somehow	Tired	Hard	None
No Tool	P19	Sublime		Tired	Average	User
	P20	Bash		Tired	Average	User
	P21			Neutral	Easy	User
	P22	Nvim, ripgrep, fzf		Neutral	Average	User
	P23	Git		Tired	Hard	None
	P24			Tired	Hard	None
	P25			Very Tired	Hard	
	P26			Very Tired	Average	None
	P27			Neutral	Hard	None
	P28			Neutral	Hard	None
	P29			Neutral	Hard	None
	P30			Very Tired	Hard	None
	P31			Very Tired	Hard	None
	P32			Very Tired	Hard	None

Tool tooling groups are 3, 2.14, and 2 respectively. This shows that participants assigned to the DiPiDi group felt slightly less fatigued after the study.

Interestingly, some participants who were in the No Tool group and found the experiment difficult, suggested that having a tool that can track the dependencies would be very useful. For example, P30, said 'The tracking of configurations and conditions was almost infeasible. Maybe a visualization tool where user can navigate dependencies and targets can help'. P14 also reported that 'Looking for variables, targets, and file names at the same time was exhausting'. Feedback like this provides more motivation for the need for build-aware tools, such as DiPiDi.

6 Related Work

In this section, we situate our study and its results with respect to the literature on the (6.1) Co-evolution of Source and Build Code, (6.2) Static Analysis of Build Code, and (6.3) Build Dependency Graph Applications.

6.1 Co-evolution of Source and Build Code

There are plenty of empirical studies on the relationship between source code and its corresponding build code. These studies have shown that changes to source code files may lead to changes in the build files that are required to build software programs successfully. McIntosh et al (2011) showed this relationship and concluded that, like source files, build code evolves and may have defects. Hochstein and Jiao (2011) found that 19%–58% of commits change build files only, and 37%–65% of them touch at least one build file. Robles et al (2006) found that many commits mainly involve a build file, showing frequent changes to the build procedure. Also, studies have shown the relationship between the complexity of source and build code (Adams et al, 2008; McIntosh et al, 2010). However, to the best of our knowledge, no prior work has studied the relationship between source code changes and their exposure under different configuration settings.

6.2 Static Analysis of Build Code

Build description files are often quite complex, making it difficult for any developer to fully grasp all of their intricacies. Thus, it is often challenging to both identify bad design practices within build files, and to improve them through refactoring efforts. To remedy this situation, tools like SYMake Tamrawi et al (2012), MAKAO Adams et al (2007) and HireBuild Hassan and Wang (2018) have been proposed in prior works. SYMake is a tool that can discover smells within build-system files and help developers to refactor these files by building a symbolic dependency graph from a GNU makefile. MAKAO is a tool developed by Adams et al (2007) which focuses on visualizing makefiles to aid in refactoring them using an aspect-oriented approach. Hassan and Wang (2018) developed a tool called HireBuild, which automatically fixes buggy build files using a history-driven approach. These studies used properties of build description files to analyze the build files themselves. In this study, we use build dependency graphs to analyze the impact of the changes on the software programs and source files.

6.3 Build Dependency Graph Applications

Impact analysis of changes has applications both for researchers and practitioners. Wen et al (2018) introduced an approach to integrate impact analysis

with code review. They showed that changes that impact critical deliverables may require more reviewing efforts than others. In another study, Cao et al (2017) proposed a tool that can estimate the duration of an incremental build using the build dependency graph, history of the builds, and changed files. They created the graph using the output messages generated by GNU make. However, this method constructs the graph based on the environment the build is currently running on and the build-time configurations passed to the build system for that specific run. Thus, the generated graph does not include the files and the dependencies for other configurations. The graph generated using DiPiDi approach considers all the possible outcomes of the build system, and produce a more global analysis result.

7 Threats to Validity

In this section, we discuss the threats to the validity of our study.

7.1 Threats to internal validity

Participants may vary in their capacity to estimate exposure. Due to the challenges associated with recruiting a large sample of software developers, participant characteristics that may interact with or confound our dependant variables, (e.g., experience), could not be controlled to a statically significant degree. Nevertheless, we strove to mitigate this threat by randomly assigning tasks to participants and by recruiting participants with varying levels of experience. Additionally, due to the Hawthorne effect, our participants were likely to behave differently in our experimental setting because they were aware that they were being monitored. We attempted to mitigate this threat by giving developers realistic tasks, letting them work on their own computers at a time and place of their choosing. Furthermore, we did not discuss the hypotheses of the study with the participants until after they completed their tasks.

We observed differences in the self reported fitness levels in each tooling groups. There are two potential reasons for these differences. First, the tooling provided by DiPiDi may reduce the cognitive load on the participants in that group. Or second, it is possible that this is simply a random occurrence due to the participants being randomly assigned to a group. We suspect this is the former because we observed a trend in the fitness level based on the tooling group with No Tool presenting the least fit participants.

7.2 Threats to external validity

Although we believe that the DiPiDi approach is general enough to apply to most build systems, our prototype implementation only supports CMake. Therefore, our findings might be limited in scope to the CMake context. On the other hand, CMake shares several concepts with other build systems, especially

those based on a platform abstraction layer. For example, GNU Autotools also uses a target-based representation and generates low-level build code (i.e., Makefiles) from higher abstractions and contextual information from the build execution environment. While we believe the results are likely to generalize, replication of the study in the context of other build systems may be fruitful.

7.3 Threats to construct validity

Our selected measurements may not fully capture the phenomena that we set out to measure (i.e., effectiveness and efficiency of assessing patch exposure). Nonetheless, we selected a broad range of measurements and tasks that we believe to be meaningfully representative of the underlying phenomena of interest.

8 Conclusion

Large-scale software systems often produce different variants that execute on different hardware and software platforms, or that restrict access to features. The build system, which is responsible for orchestrating the preprocessing, compilation, testing, and assembly of applications, manages this complex set of variants within its specifications. In these highly-configurable software projects, a change in the source code may impact a subset of the variants of the system, while others remain unchanged. To assess the risk of a change, it is important to identify the set of deliverables and configurations that are impacted.

In this paper, we introduced DiPiDi, an approach that we developed to assess the impact of changes by statically analyzing the build system specification files. To evaluate our approach, we implemented a prototype of our approach and designed an experiment to evaluate whether DiPiDi is associated with improvements to the effectiveness and efficiency of developers performing impact assessment tasks. The result of that experiment suggests that (1) DiPiDi approach helps practitioners and researchers to identify the impacted deliverables given a change under different build-time configuration settings. Our prototype implementation of DiPiDi outperforms current impact analysis tool by 36 average percentage points in F_1-score when identifying impacted deliverables. More importantly, participants in the DiPiDi group could assess the riskiness of changes relative to each other with fewer errors; and (2) the DiPiDi approach increases the efficiency of identifying impacted deliverables under different build-time configuration settings. We show that our prototype implementation reduces the time required to assess the exposure of changes by 42% on average. More notably, participants in the DiPiDi group assess the riskiness of changes relative to each other with 0.05 units of distance in 53% of the time with 1.5 standard deviations difference over the existing approaches.

References

Adams B, Tromp H, De Schutter K, De Meuter W (2007) Design recovery and maintenance of build systems. In: 2007 IEEE International Conference on Software Maintenance, IEEE, pp 114–123

Adams B, De Schutter K, Tromp H, De Meuter W (2008) The evolution of the linux build system. Electronic Communications of the EASST 8

Ahsan SN, Wotawa F (2010) Impact analysis of scrs using single and multi-label machine learning classification. In: Proceedings of the 2010 ACM-IEEE international symposium on empirical software engineering and measurement, pp 1–4

Al-Kofahi JM, Nguyen HV, Nguyen AT, Nguyen TT, Nguyen TN (2012) Detecting semantic changes in makefile build code. In: 2012 28th IEEE International Conference on Software Maintenance (ICSM), pp 150–159, DOI 10.1109/ICSM.2012.6405266

Arnold RS, Bohner SA (1993) Impact analysis-towards a framework for comparison. In: 1993 Conference on Software Maintenance, IEEE, pp 292–301

Bezemer CP, McIntosh S, Adams B, German DM, Hassan AE (2017) An empirical study of unspecified dependencies in make-based build systems. Empirical Software Engineering 22(6):3117–3148

Bosu A, Greiler M, Bird C (2015) Characteristics of useful code reviews: An empirical study at microsoft. In: Proceedings of the 12th Working Conference on Mining Software Repositories, IEEE Press, MSR '15, p 146–156

Cao Q, Wen R, McIntosh S (2017) Forecasting the duration of incremental build jobs. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 524–528

Cohen J (2010) Modern code review. Making Software: What Really Works, and Why We Believe It pp 329–336

Fontana FA, Mariani E, Mornioli A, Sormani R, Tonello A (2011) An experience report on using code smells detection tools. In: 2011 IEEE fourth international conference on software testing, verification and validation workshops, IEEE, pp 450–457

Gethers M, Poshyvanyk D (2010) Using relational topic models to capture coupling among classes in object-oriented software systems. In: 2010 IEEE International Conference on Software Maintenance, IEEE, pp 1–10

Gyori A, Lahiri SK, Partush N (2017) Refining interprocedural change-impact analysis using equivalence relations. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA, ISSTA 2017, p 318–328, DOI 10.1145/3092703.3092719, URL `https://doi.org/10.1145/3092703.3092719`

Hassan F, Wang X (2018) Hirebuild: An automatic approach to history-driven repair of build scripts. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), IEEE, pp 1078–1089

Hochstein L, Jiao Y (2011) The cost of the build tax in scientific software. In: 2011 International Symposium on Empirical Software Engineering and

Measurement, IEEE, pp 384–387
Kendall MG (1938) A new measure of rank correlation. Biometrika 30(1/2):81–93
Kirchner F, Kosmatov N, Prevosto V, Signoles J, Yakobowski B (2015) Frama-c: A software analysis perspective. Formal Aspects of Computing 27(3):573–609
Kitware (2020) CMake. Https://cmake.org
Li B, Sun X, Leung H, Zhang S (2013) A survey of code-based change impact analysis techniques. Software Testing, Verification and Reliability 23(8):613–646
Liebig J, Apel S, Lengauer C, Kästner C, Schulze M (2010) An analysis of the variability in forty preprocessor-based software product lines. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1, pp 105–114
Martin K, Hoffman B (2010) Mastering CMake: a cross-platform build system. Kitware
McIntosh S, Adams B, Hassan AE (2010) The evolution of ant build systems. In: 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010), IEEE, pp 42–51
McIntosh S, Adams B, Nguyen TH, Kamei Y, Hassan AE (2011) An empirical study of build maintenance effort. In: 2011 33rd International Conference on Software Engineering (ICSE), IEEE, pp 141–150
Meidani M, Lamothe M, McIntosh S (2021) Assessing the exposure of software changes: The dipidi approach. arXiv preprint arXiv:210400725
Moura Ld, Bjørner N (2008) Z3: An efficient smt solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, pp 337–340
Nadi S, Holt R (2014) The linux kernel: A case study of build system variability. Journal of Software: Evolution and Process 26(8):730–746
Orrú M, Tempero E, Marchesi M, Tonelli R, Destefanis G (2015) A curated benchmark collection of python systems for empirical studies on software engineering. In: Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering, pp 1–4
Parr TJ, Quong RW (1995) Antlr: A predicated-ll (k) parser generator. Software: Practice and Experience 25(7):789–810
Robles G, Gonzalez-Barahona JM, Merelo JJ (2006) Beyond source code: the importance of other artifacts in software development (a case study). Journal of Systems and Software 79(9):1233–1248
Rosenthal R (1976) Experimenter effects in behavioral research. Irvington
Rovegård P, Angelis L, Wohlin C (2008) An empirical study on views of importance of change impact analysis issues. IEEE Transactions on Software Engineering 34(4):516–530
SciTools (2022) Understand. URL `https://scitools.com/`
Seo H, Sadowski C, Elbaum S, Aftandilian E, Bowdidge R (2014) Programmers' build errors: A case study (at google). In: Proceedings of the 36th International Conference on Software Engineering, Association for Comput-

ing Machinery, New York, NY, USA, ICSE 2014, p 724–734, DOI 10.1145/2568225.2568255, URL https://doi.org/10.1145/2568225.2568255

Sincero J, Schirmeier H, Schröder-Preikschat W, Spinczyk O (2007) Is the linux kernel a software product line. In: Proc. SPLC Workshop on Open Source Software and Product Lines

Tamrawi A, Nguyen HA, Nguyen HV, Nguyen TN (2012) Build code analysis with symbolic evaluation. In: 2012 34th International Conference on Software Engineering (ICSE), IEEE, pp 650–660

Tu Q, Godfrey MW (2001) The build-time software architecture view. In: Proceedings IEEE International Conference on Software Maintenance. ICSM 2001, IEEE, pp 398–407

Tukey JW, et al (1977) Exploratory data analysis, vol 2. Reading, Mass.

Wen R, Gilbert D, Roche MG, McIntosh S (2018) Blimp tracer: integrating build impact analysis with code review. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 685–694

Zhou B, Xia X, Lo D, Wang X (2014) Build predictor: More accurate missed dependency prediction in build configuration files. In: 2014 IEEE 38th Annual Computer Software and Applications Conference, pp 53–58, DOI 10.1109/COMPSAC.2014.12

Appendix A Demographic Questions

1. How much experience do you have in programming?

- None
- Less than a year
- a year to two years
- two to five years
- five years or more

2. How much are you familiar with CMake?

- None
- Tried it at least once
- Used in professional development

3. How often do you currently program?

- Never
- Sometimes
- More than once per week

Appendix B Post Study Questionnaire

1. If you used any other tool(s) (CLI/IDE) please name it here:
2. If we provided a tool for you to use, how useful it was?
3. How do you feel? (1=Very Tired, 2=Tired, 3=Neutral, 4=Energetic, 5=Very Energetic)
4. How difficult were the tasks? (1=easy, 2=average, 3=hard)
5. How much experience did you have with the projects provided to you?
6. Did you encounter any problem during the experiment?
7. Any feedback about the experiment?
8. Can we contact you for a follow up interview?

Appendix C Task A

You will be provided with the names of changed files and a set of build specifications. Your task is to list impacted deliverables (targets). Deliverables are defined in CMake files (CMakeLists.txt or .cmake files) using `add_library` or `add_executable` commands. You can find these files in the project repository. These commands take a target name and a list of files which impact the target. Some files may be excluded under different configuration. As an example, a code file related to ARM processor may not be included in the deliverable for Intel CPUs. Read more at `https://cmake.org/cmake/help/latest/manual/cmake-buildsystem.7.html#binary-targets`. The experiment UI provides text inputs for you to list those deliverables.

Follow the steps below to prepare for the task. Once you completed the steps, click on ready and the task will begin.

1. Access DiPiDi tool at ...
2. Clone the repository from `https://github.com/libuv/libuv`

Given the following commit id and the build time configuration, please find the impacted targets (deliverables). There maybe more input fields than necessary to complete the task.

1. Change Commit ID: cdced3a3ad1b3e4287f92c9d434b543a9e509938
2. Build Configuration: APPLE==False

Input1: ...
Input2: ...

Appendix D Task B - (Impacted Deliverables)

You will be shown three commits and a set of build specifications which are the conditions passed to the build system and may change the build process. These conditions are defined using option or if commands. Read more at `https://cmake.org/cmake/help/latest/command/if.html`. We ask you to rank the commits listed in the experiment UI based on the number of impacted deliverables. Rank the commits in an ascending order (1=Most Impact, 3=Less Impact)

Follow the steps below to prepare for the task. Once you completed the steps, click on ready and the task will begin.

1. Access DiPiDi tool at ...
2. Clone the repository from `https://github.com/libuv/libuv`

Given the following build time configurations, please rank the commits based on the given criteria.

1. Build Configurations: MAKE_SYSTEM_NAME==APPLE
2. Criteria: Impacted Deliverables

1. e89abc80ea43065a726ade191b810af53ec6158a: ...
2. 953f901dd2330a9979838cd43ff04eacde71b25a: ...
3. e43eb667b5e0cace1eef4b6f5898de83cde262c6: ...

Appendix E Task B - (Impacted Variants)

You will be shown three commits and a set of build specifications which are the conditions passed to the build system and may change the build process. These conditions are defined using option or if commands. Read more at `https://cmake.org/cmake/help/latest/command/if.html`. We ask you to rank the commits listed in the experiment UI based on the number of impacted application variants (e.g., number of affected OS). Rank the commits in an ascending order (1=Most Impact, 3=Less Impact)

Follow the steps below to prepare for the task. Once you completed the steps, click on ready and the task will begin.

1. Access DiPiDi tool at ...
2. Clone the repository from `https://github.com/libuv/libuv`

Given the following build time configurations, please rank the commits based on the given criteria.

1. Build Configurations: LIBUV_BUILD_TESTS==False
2. Impacted Application Variants (Operating systems)

1. e89abc80ea43065a726ade191b810af53ec6158a: ...
2. 953f901dd2330a9979838cd43ff04eacde71b25a: ...
3. e43eb667b5e0cace1eef4b6f5898de83cde262c6: ...

Appendix F Task C - (Identify Commits Affect Deliverables)

You will be shown three commits and asked to identify the commits that affect a specified set of deliverables.

Follow the steps below to prepare for the task. Once you completed the steps, click on ready and the task will begin.

1. Access DiPiDi tool at ...
2. Clone the repository from `https://github.com/libuv/libuv`

Identify the commits which affect these deliverables: [‘uv’]

1. e89abc80ea43065a726ade191b810af53ec6158a: ?
2. 953f901dd2330a9979838cd43ff04eacde71b25a: ?
3. e43eb667b5e0cace1eef4b6f5898de83cde262c6: ?

Appendix G Task C - (Identify Commits Affect Variant)

You will be shown three commits and asked to identify the commits that affect a specific variant of the software.

Follow the steps below to prepare for the task. Once you completed the steps, click on ready and the task will begin.

1. Access DiPiDi tool at ...
2. Clone the repository from `https://github.com/libuv/libuv`

Identify the commits which affect this variant: BSD Operating System

1. e89abc80ea43065a726ade191b810af53ec6158a: ?
2. 953f901dd2330a9979838cd43ff04eacde71b25a: ?
3. e43eb667b5e0cace1eef4b6f5898de83cde262c6: ?

Appendix H Task C - (Configuration Setting)

You will be shown three commits and asked to identify the configuration settings under which the changes will affect at least one target. The build configurations may exclude or include a file in the build process for an specific target using conditional commands in the CMake files. Read more at `https://cmake.org/cmake/help/latest/command/if.html` CMake website.

Follow the steps below to prepare for the task. Once you completed the steps, click on ready and the task will begin.

1. Access DiPiDi tool at ...
2. Clone the repository from `https://github.com/libuv/libuv`

For each of the given commits, identify at least one configuration setting under which the change will propagate to at least one deliverable(target). If the change will propagate irrespective of the conditional settings, enter the term "ALL".

1. e89abc80ea43065a726ade191b810af53ec6158a: ...
2. 953f901dd2330a9979838cd43ff04eacde71b25a: ...
3. e43eb667b5e0cace1eef4b6f5898de83cde262c6: ...