Assessing the Exposure of Software Changes: The DiPiDi Approach

Mehran Meidani
University of Waterloo
Waterloo, Canada
mehran.meidani@uwaterloo.ca

Maxime Lamothe
University of Waterloo
Waterloo, Canada
maxime.lamothe@uwaterloo.ca

Shane McIntosh
University of Waterloo
Waterloo, Canada
shane.mcintosh@uwaterloo.ca

***Abstract*—*Context*: 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.**

***Objective*: In this experiment, we seek to evaluate DiPiDi, a prototype implementation of our approach to assess the exposure of source code changes by statically analyzing build specifications. We focus our evaluation on the effectiveness and efficiency of developers when assessing the exposure of source code changes.**

***Method*: We will measure the effectiveness and efficiency of developers when performing five tasks in which they must identify the deliverable(s) and conditions under which a change will propagate. We will assign participants into three groups: without explicit tool support, supported by existing impact analysis tools, and supported by DiPiDi.**

***Index Terms*—build systems, exposure of a change, build dependency graph**

I. 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) [1]. For example, the Linux kernel has more than 10,000 compile-time configuration settings [2]. These systems have multiple dependency paths from their deliverables (i.e., executables and libraries) to their source files. 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 [3]. 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 [4].

Software engineering practices that assess source code changes, like code review, are expensive and time-consuming [5], [6]. 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 [7].

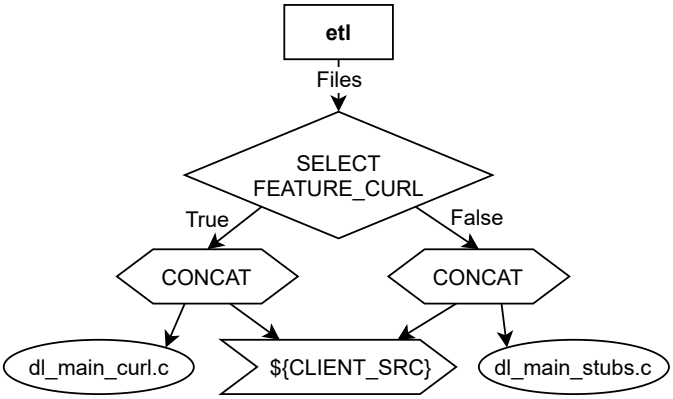

Fig. 1: A real-world example of a build dependency graph

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` if the `FEATURE_CURL` option is `ON`. On the other hand, changes to files represented by `$CLIENT_SRC` will always impact the deliverable. 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 [8]. Change Impact Analysis (CIA) is one way to determine the consequences of a change on a software application [9]. Many CIA techniques have been proposed [10]–[15]. However, to the best of our knowledge, none of them consider environment or build-time configuration settings. While build impact analysis

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

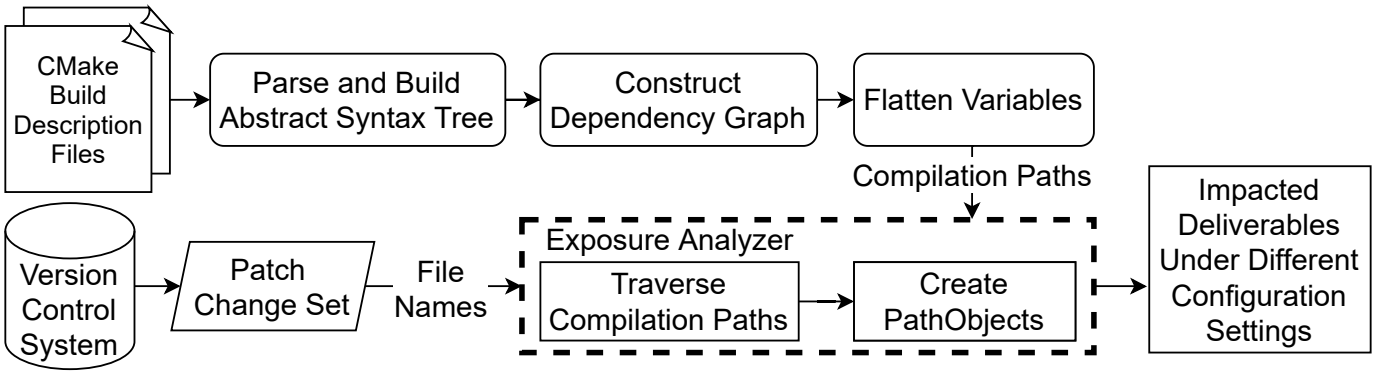

Fig. 2: An overview of the DiPiDi approach

has been shown to be effective [7], [14], 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 that are built using CMake. One of the key roles of the build system is finding and selecting files based on build scripts, build-time configurations, and environmental variables [16]–[18]. 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.

This paper presents our plan to experimentally evaluate the effect of DiPiDi on the effectiveness and efficiency of determining the exposure of source code changes. To that end, we form three participant groups – those with no tool assistance, those with the assistance of a commercial 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) [19] and practitioners in the allocation of quality assurance resources.

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

To test our hypothesis, we pose the following research questions (RQs):

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

We address RQ1 by testing the following hypotheses:
$H_{1.1}$: DiPiDi significantly affects the effectiveness of developers in assessing the exposure of a patch.
$H0_{1.1}$: DiPiDi **does not** significantly affect the effectiveness of developers in assessing the exposure of a patch.

Additionally, we ask:

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

We formalize RQ2 in the following hypotheses:
$H_{2.1}$: DiPiDi significantly affects the efficiency of developers in assessing the exposure of a patch.
$H0_{2.2}$: DiPiDi **does not** significantly affect the efficiency of developers in assessing the exposure of a patch.

III. DiPiDi

Our proposed solution to raise developer awareness of the exposure of changes, DiPiDi, works on projects that use the CMake build system. CMake is a cross-platform build system that builds deliverables from artifacts, like source files [20]. CMake has two distinct phases. First, it generates platform-based low-level build specifications (e.g., Makefiles, Visual Studio #.sln files, or Ninja files [21]). Then, CMake invokes the low-level build tool to build the project.

An overview of the approach used by DiPiDi can be found in Figure 2. DiPiDi first parses the CMake specifications starting with the `CMakeLists.txt` file in the project root directory (i.e., the entry point for the CMake build system). We use ANTLR [22] to parse and build the *Abstract Syntax Tree (AST)* from the CMake file. At this level, we may need

to include and parse other CMake files as instructed in the `CMakeLists.txt` file.

Next, we traverse the AST to create the *Build Dependency Graph*, which represents the relationship between the deliverables, source files, and the conditions in each path. Using the graph, we 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.

Given a list of changed file names, the flattened variables can be used to traverse dependency paths to create a list of exposed `PathObjects`, the output of DiPiDi. A `PathObject` contains all the possible dependency paths from deliverables to the changed source files. 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 [23], [24]. These settings create different dependency paths from the deliverable to the source files. An example of the output of DiPiDi is shown in Figure 3. This output can then be used by developers to identify which deliverables and variants are exposed by source-code changes.

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

Fig. 3: An example of output of the tool based on the given graph in Figure 1

IV. Research Protocol

To test our hypotheses, we will conduct randomized controlled experiments with three groups. Study participants will be 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 will measure the efficiency of our participants by comparing the duration of each task across the groups.

A. Variables

Table I provides an overview of the study variables, which we describe below.

1) Independent Variable: In our study design, the tool support provided to the participants varies (*No Tool*, *With Existing Tool*, and *With DiPiDi*). We use the following groups *(Tooling level)* to evaluate our hypotheses:

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.

Existing Tool. This group has access to the same environment as the *No Tool* group, as well as a state-of-the-art change impact analysis tool [25]. 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.

DiPiDi: This group–the treatment group–will have access to DiPiDi. For each changed file, the tool will print a `PathObject`. Our tool will 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.

2) Dependent Variables: Our dependent variables are outlined in Table I. We discuss our reasoning for these variables below.

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, 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.

Exposure analysis efficiency: We define exposure analysis efficiency as the scores the participant get for tasks for a given time (in minutes) spent for the tasks. As a result, getting higher score in a shorter time will result in a higher efficiency. This way, we consider both the fully correct answers and the partial ones, especially in the rank based tasks.

3) Confounding Variables: Because different code changes might affect the results of our participants, we control the code changes made available to them. 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 evaluate *Tooling level* with multiple build configurations without introducing confounding factors. We gather some 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., get some context to determine why a participant might not have finished a task) and for further analysis.

TABLE I: The variables of the study

Name	Description	Scale	Operationalization
Independent variables:			
Tooling level	The tools available to the participants: no tool, existing tool, DiPiDi	nominal	See Section IV-A; randomized.
Dependent variables:			
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 IV-C & IV-F2
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 IV-F2
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-based application. The participant can pause a task and resume manually. see Section IV-B4
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 IV-F2
Confounding/Measured variables:			
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
Understandability	Participant's overall understanding of the code provided during the experiment	ordinal	Measured: 3-point scale ("nothing", "somewhat", "fully understand"); questionnaire at the end

B. Materials

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

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

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 [10]. 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. We have selected Frama-C; a tool proposed by Kirchner et al. [25]. Frama-C is an industrial grade static analysis tool, which can perform impact analysis on C and C++ projects. Moreover, Frama-C is open-source and can therefore be customized if needed.

3) Studied projects: To allow for realistic evaluations, all of the patches that form the basis for the tasks in this experiment are sampled from real-world projects. Since our tool currently supports the CMake build system, we limit our selection to large and successful projects that use CMake. We select three projects from the Qt and KDE open-source communities as the scope from which to sample tasks to conduct our study. KDE is a collection of projects comprising an open-source desktop environment. Qt is an open-source toolkit for creating Graphical User Interfaces (GUI). Both of these communities use C/C++ as their programming languages and CMake as their build technology.

To select our projects, we first start with all of the projects available on the GitHub organization pages for KDE and Qt and filter out projects that do not use CMake. We then filter out projects that have not had commits in the last 6 months to guarantee that we are looking at active projects. Finally we select the top three projects by number of forks, a proxy that allows us to gauge developer interest. With these criteria in mind, we select Kdenlive, Qt Base, and Krita.

For each studied project, we will select three patches that impact a different number of deliverables under different

configuration settings. 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 file level, in this study we ask participants to report impacted deliverables at the code level, a sub-set of reported deliverables by the tool.

4) Experiment UI: To conduct our experiment with a diverse range of participants and allow our participants to rely on their own development environments, we develop a Web based application with which our participants will interact. The application will retain a log of answers and the duration of each task. The logic behind the experiment UI will randomly assign each participant to a *Tooling level* group and randomly assign tasks to the participants, all the while logging which project and tasks are assigned to whom. Participant information will only be made available to the researchers after all results have been scored to reduce experimenter bias [26].

C. 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.

D. Participants

Since our tasks are centered 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 seek to populate our pool of participants with software developers, or individuals with programming experience.

We calculated the required size of our pool of participants using the standard settings for uncovering a medium effect size (0.25) when applying a one-way ANOVA (i.e., $\alpha = 0.05$, $\beta = 0.8$, three levels) [27]. The results require us to recruit 159 participants. Since recruiting such a large pool of participants is unlikely, we relax our effect size target to large effect size (0.40), giving a more achievable pool of 66 participants.

We will recruit our participants for our study from the development teams of our industrial partners, which include large multinationals like Huawei and Dell EMC, as well as start-ups like YourBase. We strive to recruit at least 50 professional software developers from these organizations.

We will also seek to recruit software developers through personal and professional contacts, e.g., on social media platforms like LinkedIn and Twitter. We expect to recruit at least ten software developers from these sources. Finally, we will also post open calls for participants in various schools of computer science and software engineering. We expect to recruit at least ten more participants through these sources.

E. Execution Plan

We will provide our participants with access to our web-based application in batches of five. This staged approach will allow us to fix any potential problems without invalidating too large of a subset of our participant data. The application will have the following procedure for each participant:

1) Welcome Page: We first provide our participants with an outline of the tasks and an estimate of the time required to complete the tasks. In addition, we will request the requisite 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.

2) Onboarding: After obtaining consent from the participants, we provide an explanation of the tasks to be completed during the experiment. 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 they may skip individual tasks.

3) Tasks: We present our participants with the tasks outlined in Section IV-C in a random order. For each task, our application will provide a hyperlink to download the source code. A timer will begin as soon as the task page is loaded. The page will describe the task, and show 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 level*, in a form that emulates supplementary code review information that would be available if the tools were part of a CI/CD pipeline. The application will provide input spaces for the participant to enter their responses. The application will log 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.

4) Questionnaire: After a participant completes their five tasks, we will follow up with a questionnaire, which collects demographics questions about their background and programming experience, as well as tool usage questions about the CLI tools, IDEs, and/or other tools that used to complete the tasks. We also ask participants to comment on any problems that they may have encountered during the experiment. Finally, we will thank the participants and invite them to provide other feedback if they desire. This questionnaire will take fewer than five minutes to complete.

F. Analysis Plan

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 will manually check that answers are sane before analyzing them. Finally, we will review the participant's questionnaire submission and feedback for mentions of problems that may (partially) invalidate their submission, removing their invalid answers when appropriate.

2) Measuring Effectiveness: For rank-based tasks, i.e., *task type B*, we will use Kendall's tau ranking distance formula [28] to compute the distance between participant answers and the ground-truth. We report that number as the score between zero and one for those tasks. For list-based tasks, i.e., *task types A and C*, like previous studies, we compute precision and recall [29]. 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-measure (i.e, the harmonic mean of precision and recall) to get an overall impression of task effectiveness.

3) Descriptive Statistics: For each group, we will provide the mean, standard deviation, and relevant quantile values for our dependent variables and participant demographics. We will also include Spearman's ρ pairwise correlation values to measure the strength of relations between the variables.

4) Inferential Statistics: We will first use the Shapiro–Wilk test, along with a visual analysis, to determine if our data is normally distributed. If our data follows non-normal distributions, we will use non-parametric statistical tests to answer RQ1 & RQ2 because they impose fewer constraints on the distributions of analyzed data. In this case, we will use the Kruskal–Wallis test (i.e., One-way ANOVA on ranks) statistical hypothesis testing technique to identify whether the difference between the results in treatment group and the control groups are statistically significant. We will then apply the Cliff's Delta non-parametric effect size measure to assess the magnitude of the difference between each pair of groups. If our data does follow a normal distribution, we will use a one-way ANOVA technique to compare our treatment and control groups, and Cohen's d for effect size calculations.

V. Threats to Validity

Threats to internal validity: Participants may vary in their capacity to estimate exposure. We strive to mitigate this by randomly assigning tasks to participants and by recruiting participants with varying levels of experience. Due to the challenges associated with obtaining a large sample of software developers, this study will focus on a statistically valid, but non-maximal number of participants. The pool of participants will retain enough statistical power to reject our null hypotheses. We are aware that the statistical power of the effect sizes of our findings is dependent on our final sample size and shall therefore endeavor to recruit as many participants as possible.

Threats to external validity: We anticipate that most of our participants will volunteer from our three partner organizations. As such, they will likely have similar technological backgrounds. This might reduce the generalizability of our findings. To mitigate this effect, we also endeavor to obtain participants from other backgrounds.

Threats to construct validity: Due to the Hawthorne effect, our participants are likely to behave differently in our experimental setting because they are aware that they are being monitored. We attempt 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 will not discuss the hypotheses of the study with the participants until after they completed their tasks. We are aware that our selected measurements do not fully capture the phenomena that we set out to measure (i.e., effectiveness and efficiency of assessing patch exposure). Nonetheless, we select a broad range of measurements and tasks that we believe to be meaningfully representative of the underlying phenomena of interest.

REFERENCES

[1] Q. Tu and M. W. Godfrey, "The build-time software architecture view," in *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001.* IEEE, 2001, pp. 398–407.

[2] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk, "Is the linux kernel a software product line," in *Proc. SPLC Workshop on Open Source Software and Product Lines*, 2007.

[3] S. Nadi and R. Holt, "The linux kernel: A case study of build system variability," *Journal of Software: Evolution and Process*, vol. 26, no. 8, pp. 730–746, 2014.

[4] C.-P. Bezemer, S. McIntosh, B. Adams, D. M. German, and A. E. Hassan, "An empirical study of unspecified dependencies in make-based build systems," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3117–3148, 2017.

[5] J. Cohen, "Modern code review," *Making Software: What Really Works, and Why We Believe It*, pp. 329–336, 2010.

[6] A. Bosu, M. Greiler, and C. Bird, "Characteristics of useful code reviews: An empirical study at microsoft," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15. IEEE Press, 2015, p. 146–156.

[7] R. Wen, D. Gilbert, M. G. Roche, and S. McIntosh, "Blimp tracer: integrating build impact analysis with code review," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 685–694.

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

[9] R. S. Arnold and S. A. Bohner, "Impact analysis-towards a framework for comparison," in *1993 Conference on Software Maintenance*. IEEE, 1993, pp. 292–301.

[10] B. Li, X. Sun, H. Leung, and S. Zhang, "A survey of code-based change impact analysis techniques," *Software Testing, Verification and Reliability*, vol. 23, no. 8, pp. 613–646, 2013.

[11] S. N. Ahsan and F. Wotawa, "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*, 2010, pp. 1–4.

[12] M. Gethers and D. Poshyvanyk, "Using relational topic models to capture coupling among classes in object-oriented software systems," in *2010 IEEE International Conference on Software Maintenance*. IEEE, 2010, pp. 1–10.

[13] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. N. Nguyen, "Build code analysis with symbolic evaluation," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 650–660.

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

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

[15] A. Gyori, S. K. Lahiri, and N. Partush, "Refining interprocedural change-impact analysis using equivalence relations," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 318–328. [Online]. Available: https://doi.org/10.1145/3092703.3092719

[16] B. Zhou, X. Xia, D. Lo, and X. Wang, "Build predictor: More accurate missed dependency prediction in build configuration files," in *2014 IEEE 38th Annual Computer Software and Applications Conference*, 2014, pp. 53–58.

[17] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, "Programmers' build errors: A case study (at google)," ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 724–734. [Online]. Available: https://doi.org/10.1145/2568225.2568255

[18] J. M. Al-Kofahi, H. V. Nguyen, A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Detecting semantic changes in makefile build code," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012, pp. 150–159.

[19] P. Rovegård, L. Angelis, and C. Wohlin, "An empirical study on views of importance of change impact analysis issues," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 516–530, 2008.

[20] Kitware, *CMake*, 2020, https://cmake.org.

[22] T. J. Parr and R. W. Quong, "Antlr: A predicated-ll (k) parser generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.

[23] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "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*, 2010, pp. 105–114.

[24] L. Hochstein and Y. Jiao, "The cost of the build tax in scientific software," in *2011 International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2011, pp. 384–387.

[25] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c: A software analysis perspective," *Formal Aspects of Computing*, vol. 27, no. 3, pp. 573–609, 2015.

[26] R. Rosenthal, "Experimenter effects in behavioral research," 1976.

[27] J. Cohen, "Statistical power analysis," *Current directions in psychological science*, vol. 1, no. 3, pp. 98–101, 1992.

[28] M. G. Kendall, "A new measure of rank correlation," *Biometrika*, vol. 30, no. 1/2, pp. 81–93, 1938.

[29] L. Hattori, D. Guerrero, J. Figueiredo, J. Brunet, and J. Damásio, "On the precision and accuracy of impact analysis techniques," in *Seventh IEEE/ACIS International Conference on Computer and Information Science (icis 2008)*. IEEE, 2008, pp. 513–518.