BLIMP Tracer: Integrating Build Impact Analysis with Code Review

Ruiyin Wen*, David Gilbert†, Michael G. Roche†, Shane McIntosh*
*Software REBELs, McGill University, Canada; ruiyin.wen@mail.mcgill.ca, shane.mcintosh@mcgill.ca
†Dell EMC Corporation, Burlington, Canada; david.gilbert@dell.com, mike.roche@dell.com

***Abstract*—Code review is an integral part of modern software development, where patch authors invite fellow developers to inspect code changes. While code review boasts technical and non-technical benefits, it is a costly use of developer time, who need to switch contexts away from their current development tasks. Since a careful code review requires even more time, developers often make intuition-based decisions about the patches that they will invest effort in carefully reviewing.**

Our key intuition in this paper is that patches that impact mission-critical project deliverables or deliverables that cover a broad set of products may require more reviewing effort than others. To help developers identify such patches, we introduce BLIMP Tracer, a build impact analysis system that we developed and integrated with the code review platform used by a globally distributed product team at Dell EMC, a large multinational corporation. BLIMP Tracer operates on a Build Dependency Graph (BDG) that describes how each file in the system is processed to produce the set of intermediate and output deliverables. For a given patch, BLIMP Tracer then traverses the BDG to identify the deliverables that are impacted by the change. Finally, the results are reported directly within the code review interface.

To evaluate BLIMP Tracer, we conducted a qualitative study with 45 developers, observing that BLIMP Tracer not only improves the speed and accuracy of identifying the set of deliverables that are impacted by a patch, but also helps the community to better understand the project architecture.

I. Introduction

Code review refers to the practice where fellow developers inspect code changes and provide feedback to the author. Dedicated code review tools that manage the modern code review process have become a commonplace in practice. These tools allow developers to post patches and select relevant reviewers to inspect their patches. The review process itself is a valuable practice that development teams use to ensure software quality [17], improve team communication [5], and leverage team problem-solving capacity [24].

However, the mere existence of a code review does not improve code quality. To truly improve the quality of a patch, reviewers must consider the potential implications of the patch and engage in a discussion with the author. Prior work shows that a lack of reviewer participation is correlated with a drop in software release quality [18], [29] and a drop in design quality [20].

On the other hand, rigorous code review introduces overhead on developers, whose time is a limited and valuable commodity. Bosu and Carver [8] find that developers spend an average of six hours per week reviewing code. The time spent reviewing code is an expensive context switch from other important development tasks (e.g. repairing and improving code). Making matters worse, patch authors at Microsoft report that an average of 35% of code review comments are not useful [9], suggesting that a large proportion of reviewing time may be misspent generating feedback that is not valuable.

Since some changes are of greater risk than others, some patches will require a more rigorous review than others. Czerwonka et al. [13] argue that spending an equal amount of reviewing effort on all code patches is a suboptimal use of development resources. Currently, to reduce waste in the reviewing process, developers use their intuition and their past experience to decide which patches require detailed feedback. However, knowing which patches require more reviewing attention than others is a difficult problem for code authors and reviewers alike.

An understanding of the impact that a patch has on the entire software system may help stakeholders to focus on reviewing effort on patches that have a broader impact on the project. More specifically, we believe that: Patches that impact mission-critical project deliverables or deliverables that cover a broad set of products should involve more reviewing investment than others. However, such information is missing from modern code reviewing interfaces.

To understand the impact that a patch will have on a system, Change Impact Analysis (CIA) techniques have been proposed [4]. However, recent work suggests that CIA techniques are rarely adopted in practice [16]. To understand the state of CIA within the studied product team at Dell EMC, we conducted a preliminary survey of 45 developers. In the survey, we ask developers how they assess the potential impact of a patch. The results indicate that, despite their tendency to produce fault prone and incomplete results, developers choose to use command line tools, such as `grep` and `find` to estimate the impact of changes (likely due to their ubiquity and flexibility), to complement their intuition-based on prior knowledge of the modified files. Indeed, as Li et al. [16] reported, dedicated and commercialized use of CIA tools is rare. Use of ad hoc and intuition based approaches may lead to false positives (i.e. patches that did not need to be reviewed rigorously, but were) or false negatives (i.e. patches that should have been reviewed rigorously, but were not).

To help reviewers make deliverable-based decisions of reviewing where to invest reviewing effort, we developed Build Impact (BLIMP) Tracer, an impact analysis system that we integrated with the code review platform of the studied

product team at Dell EMC. Unlike traditional change impact analysis [4], BLIMP Tracer exposes the impact that a change has on project deliverables rather than other areas of the source code. This difference is of key importance in the context of the studied Dell EMC team because the subject system is comprised of several deliverables that belong to several customer-facing products. In a nutshell, BLIMP Tracer produces impact analysis reports by first extracting the Build Dependency Graph (BDG) of the system, then recursively traverses the graph starting from the changed files to identify the (set of) impacted product deliverables.

To evaluate BLIMP Tracer, we deployed it within the production reviewing environment at the studied Dell EMC team. We conducted a comparative user study with five developers. More specifically, we solicit feedback from participants during their use of BLIMP Tracer, and compare it with current style of conducting impact analysis. In all five cases, BLIMP Tracer improves the speed and accuracy of locating impacted software components of a code patch. In addition, we find that BLIMP Tracer provides developers with a clearer understanding of the project build-time architecture [30], which will likely help when onboarding newcomers to the project.

A. Paper Organization

The remainder of this paper is organized as follows. Section II describes the code review process at the studied Dell EMC team and shows a motivational example. Section III describes the design and results of the preliminary survey distributed to the developers. Section IV describes the design of BLIMP Tracer. Section V discusses the design of our user study. Section VI presents the results with respect to our user study. Section VII discusses the background of this study and surveys related work. Section VIII describes the threats to the validity of our study, and finally, Section IX draws conclusions.

II. Background

In this section, we present an overview of the code reviewing process of the studied product team at Dell EMC. In addition, we present a motivating example to demonstrate the value of BLIMP Tracer.

A. Code Review at Dell EMC

Code review is widely considered a best practice for software quality assurance. The modern code review process–a lightweight, tool-supported variant of the traditional code inspection process–allows developers to post patches for review.

The studied Dell EMC product team uses Review Board[1]—a web-based code review platform that tightly integrates with several version control systems. Figure 1 provides an overview of how Review Board-supported code review process is implemented at Dell EMC. We describe each step below.

1) **Upload patch:** When a developer has completed a patch, they upload it to Review Board. During the upload, reviewers are recommended based on the areas of the code base that were modified; however, Review Board prompts the author to confirm or replace those reviewers.
2) **Execute sanity checks:** When a new patch (or revision) appears on Review Board, it is scanned for simple errors by a continuous integration bot. This bot automatically checks whether the patch causes build breakage or has blatant issues such as incorrectly formatted code.
3) **Notify relevant reviewers:** The selected reviewers receive notification and start reviewing the patch. At the same time, they can decide to add other developers in the code reviewing process.
4) **Reviewers assess the patch:** The reviewers provide feedback to the author by commenting on the uploaded patch. The team then renders a decision to either accept or reject the code patch.
5) **Integrate into VCS:** If the patch receives 'Ship it!' labels from two members of the core team, it will be approved for integration into the main project repository.

If any of the steps 2–4 are not passed, the patch returns to the author, who may revise the patch by addressing the feedback, and then repeat the code review process.

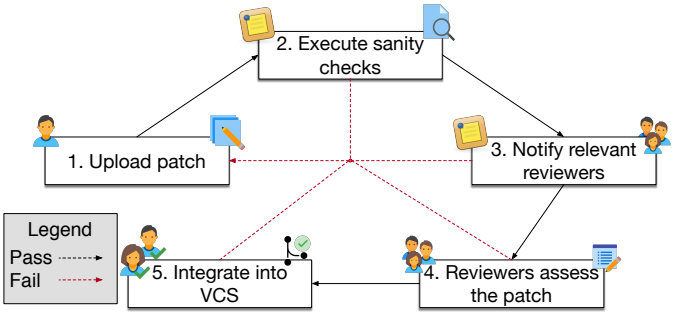

Fig. 1: The code review process of the studied product team at Dell EMC.

B. A Motivational Example

Adam, a new developer who has just started working at Dell EMC one week ago, is submitting his first patch on Review Board. As is part of the Dell EMC code reviewing practice, Adam has to indicate which team members to invite to review his patch. Being a new team member, Adam does not yet know which members of the team have the necessary expertise in the areas of the code base that he modified. Thus, he relies on the Review Board recommendations to select his team lead, Becky, as a reviewer.

Being a team lead, Becky receives plenty of review requests. In order to have time to complete her other tasks, she needs to be selective where she focuses her reviewing effort. She needs to decide whether to apply her full effort to the review (high time cost) or perform a quick review (low time cost).

At the time when Becky is notified of Adam's review request, she already has a backlog of ten review requests. All ten of the review requests are associated with issue reports of equal severity and priority. Based on her intuition,

[1]https://www.reviewboard.org/

Becky decides to prioritize the patches that are larger in size because she believes that larger patches are inherently more risky. However, Becky's decision may not be optimal because Adam's patch involves a change to broadly adopted in-house libraries. Changes to those libraries may be inherently more risky than large changes because they impact a large amount of customer-facing functionality, ending up being linked with several project deliverables.

BLIMP Tracer is designed to provide decision support for Becky. She can consult the results of BLIMP Tracer for Adam's patch, and determine which deliverables may be impacted by his patch. By exposing the impacted deliverables of a patch, Becky can reason about the impact of a change on customer-facing products and functionality. Knowing which products are impacted by a patch helps Becky make a more informed decision about how to prioritize her backlog of patches for reviewing.

III. Preliminary Survey

In this section, we present the design of a developer survey that we conducted at Dell EMC. Through the survey, we aim to gain a better understanding of how developers conduct impact analysis. The questions in the survey were intentionally open ended to allow for developers to explain their current practices in language that is natural to them.

A. Survey Design

At a high level, the survey is composed of three parts. The first part focuses on information about the developer, with demographic questions about their general software development experience and their experience working on the product suite at Dell EMC. The second part asks developers about their prior knowledge in software change impact analysis. The third part asks developers to reflect on their day-to-day procedures for assessing the impact of patches. We also invite developers who are interested in further discussing their experiences to participate in follow-up interviews afterwards.

The survey was broadcasted to 45 on-site developers, 12 of whom responded (27% response rate). The survey was also broadcasted to off-site developers, four of whom responded.

B. Demographic Information

The 16 respondents of this survey work in development offices in Canada and India. Figure 2 shows that the respondents have a broad range of experience with the Dell EMC product suite, ranging from one to 18 years, with a median of 6.5 years. Similarly, the respondents' experience on the current product varies from zero to 18 years, with a median of 6.5 years. On the other hand, the majority of the respondents are senior software engineers, with general software development experience ranging from one to 29 years, with a median of 14 years. Indeed, twelve of the respondents have more than five years of experience. The wide span of experience of the responding developers within Dell EMC assures that the observed results apply to a broad range of development backgrounds. The large number of senior developers ensures

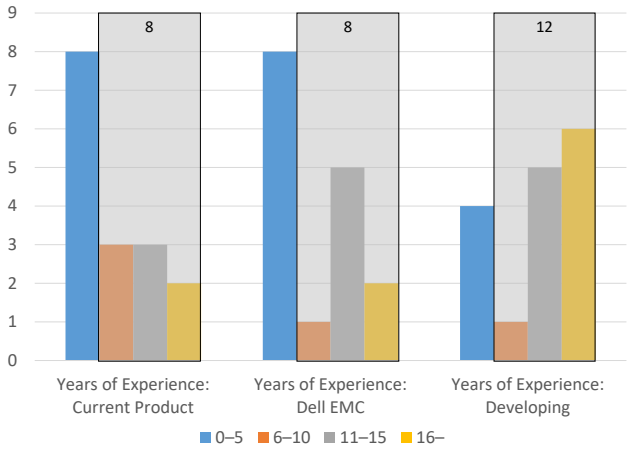

Fig. 2: The survey respondents' experience in software development in multiple contexts. The veterans (with more than five years of experience) are shadowed in gray.

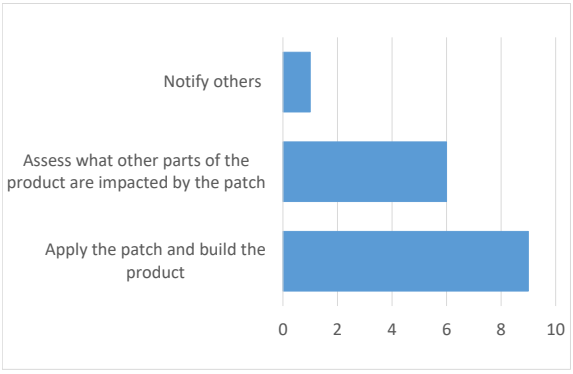

Fig. 3: The number of developers on what they would first do when they start reviewing patches.

that our results are not biased towards new developers who may not have formed impact analysis habits.

C. Change Impact Awareness

Figure 3 shows that although most (nine of the 16) respondents claim they would first apply the code patch under review and execute a project build job, build impact assessment is also the first thing code reviewers do when they start reviewing a code patch. The process of applying code patches under review to software systems and building them can be automated. In fact, the studied product team at Dell EMC employs an automated tool that applies code patches, builds the project, and reports whether the build breaks after applying the patch (recall Step 2 from Section II-A). This tool is integrated with the code review platform and posts a comment indicating whether the patch introduces build problems once the result

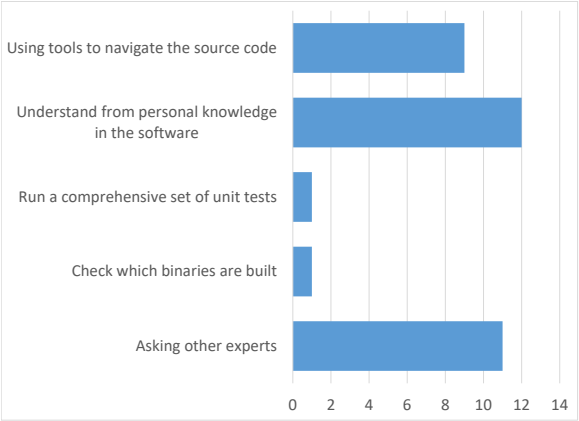

Fig. 4: How developers analyze the impact of a patch.

is generated. However, this tool does not provide feedback on the build impact of the code change, the other important step that developers take into consideration before performing a code review.

Meanwhile, we find that six of the 16 respondents acknowledge that they would check what part of the software project the code patch impacts as the first step of code review. Indeed, one developer stated that understanding the impact of a code patch is so important that 70–80% of his reviewing time is spent performing impact analysis. He explained that *"The small chunk of code that an engineer author in a change should already be examined before submitting (to Review Board), so there should not be a problem there. The real problems come with the files that depend on the change."*

Although a number of community members recognize the importance of impact analysis, they do not use any specialized tools for it and often resort back to using command line tools to investigate the impact of a patch. In fact, only two out of 16 survey respondents declare that they have used impact analysis tools. Without dedicated impact analysis tools, using other methods for finding the impact of code patches are relatively time-consuming.

Developers on the studied Dell EMC product team are aware of the importance of impact analysis prior to code reviewing, yet few of them use dedicated impact analysis tools.

D. Change Impact Practice

Developers tend to use command line tools and their background knowledge to understand the impact of a code change. We asked developers to describe how they determine what other parts of the system is impacted by a code patch. As shown in Figure 4, a majority (ten of the 16 respondents) claim they use tools, such as `grep` or `find` to navigate source code to find the impact. In addition, twelve developers say that the analysis is based on their understanding from prior experience developing the software system. The respondents also described some other ways that they would use to help perform impact analysis, including asking other experts (eleven of the 16), checking which binaries are built (one of the 16), and running unit tests (one of the 16).

Although impact analysis is one of the most crucial tasks in code review, the developers on the studied Dell EMC product team often use general-purpose command line tools to investigate the impact of patches.

IV. BLIMP Tracer Design

To help developers conduct impact analysis accurately and efficiently, we propose BLIMP Tracer, an impact analysis tool that focuses on project deliverables. We focus on deliverable-level impact because we believe it is the most useful granularity for the review understanding task that we aim to support. In this section, we describe how BLIMP Tracer performs impact analysis for the changes posted on Dell EMC's code review platform. Figure 5 provides an overview of the approach, which contains changed file detection, build dependency graph extraction, graph traversal and filtering, and results presentation steps.

A. Changed File Detection

Using the Review Board API, we poll for newly posted reviews and revisions periodically. For every new change, we extract the names of the modified source code files. Those files serve as the query that we will use in later steps to trace through in order to detect the impacted deliverables.

B. Build Dependency Graph Extraction

The build system of the studied product team is implemented using non-recursive make [19]. In make-based build systems, developers specify targets, dependencies, and rules. Targets specify an intermediate or a deliverable file. Dependencies list other targets that must exist or be updated before the target can be updated. Rules explain what command needs to be run to create the targets. For example, in Listing 1, line 6 specifies that all `.o` files (targets) depend on their corresponding `.c` file, as well as the `DEPS` variable that expands to the header file `example.h` (dependencies). Line 7 shows that to update the dependencies in line 6, a C compiler (in this case, `gcc`) will run to create the targets. Similar patterns can be seen in the following lines that specify rules for the deliverables.

The targets and their dependencies form what is called a Build Dependency Graph (BDG). This is a directed acyclic graph that is at the heart of the incremental build—a commodity feature of the modern build system. After executing a full build that executes all of the necessary build commands to produce project deliverables, an incremental build will only execute a subset of the full build commands that are required by activities that have taken place after the previous full build. For example, Figure 6 shows the BDG that corresponds to the Makefile snippet from Listing 1. After executing a full build, if a developer were only to update eg3.c, the build process would only re-execute the rules to eg3.o and deliverable_b.

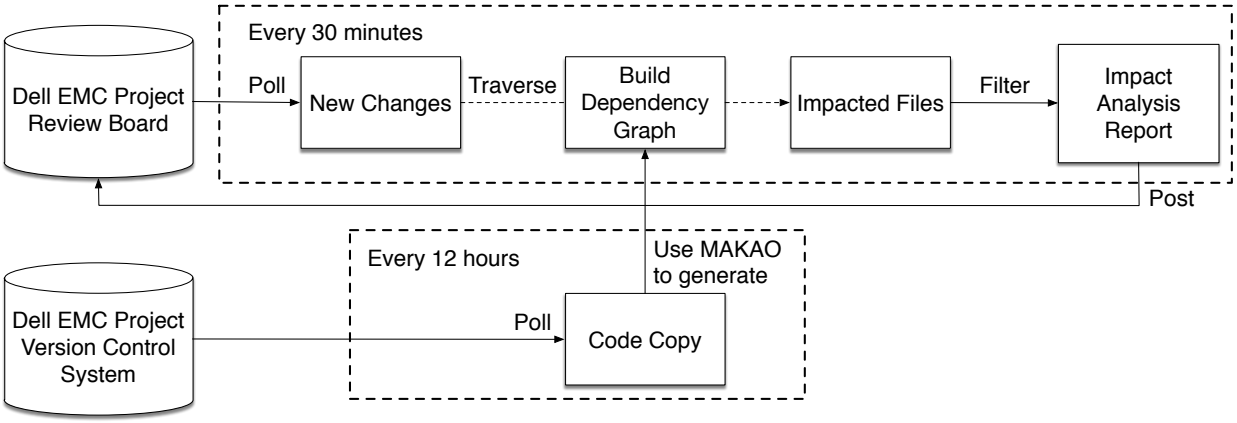

Fig. 5: An illustration of the design of BLIMP Tracer.

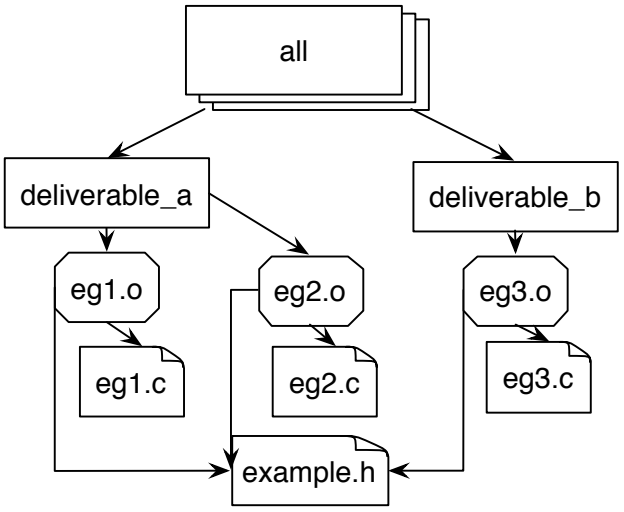

Fig. 6: A sample build dependency graph.

Listing 1: A sample Makefile

```
1  CC=gcc
2  DEPS=example.h
3
4  all: deliverable_a deliverable_b
5
6  %.o: %.c $(DEPS)
7    $(CC) -c -o $@ $< -I.
8
9  deliverable_a: eg1.o eg2.o
10   $(CC) -o $@ $^
11
12 deliverable_b: eg3.o
13   $(CC) -o $@ $^
```

In this paper, we query the BDG to understand which deliverables are impacted by a set of modified files. Extraction of the BDG from a make-based build system is a non-trivial task.

We use MAKAO [1] to extract the build dependency graphs from the Dell EMC project. Since MAKAO constructs the BDG by parsing build trace logs, we fully build the project in one of the supporting Linux platforms with GNU make tracing enabled and extract the trace log.

The trace log contains a listing of commands and their corresponding files that were executed during the build job, which can be parsed by MAKAO to generate a BDG.

Extracting a BDG for the Dell EMC project requires building the software in full. At the same time, the Dell EMC project we study is large and complex, and undergoes rapid development. Indeed, conducting a full, tracing-enabled build for each uploaded review revision is impractical. Moreover, similar to prior work [12], we find that changes that modify the BDG structure are relatively rare in practice. In order to have an up-to-date BDG for accurate impact analysis results while maintaining a low build load, BLIMP Tracer updates the BDG twice daily. We discuss the potential ramifications of this decision in Section VIII.

C. Graph Traversal and Filtering

Once we obtain a list of files within a change, we traverse the BDG for each of the file in the list of changed files and record every target that is impacted by the change. To keep the report page readable, we filter out binary and archive files that are directly impacted by a changed file,since the impact of such files is already clear for the developers to recognize.

In addition, we identify whether the list of changed files contains build specification files (e.g. `*.mk` or `Makefile`), or newly added files that had never been shown in the repository. Since a change in the build system or adding new files may change the build dependency graph we extracted previously, a traversal on the old BDG may not provide accurate impact analysis information. In such cases, BLIMP Tracer will print a warning message in the impact analysis report page, indicating that the analysis may be incomplete. Again, in practice, BDG-changing commits are rare. Additional computing power could be used to solve this problem, but given the scarcity of the

issues, the warning message was deemed sufficient for the time being.

D. Results Presentation

We generate a summary of the change impact analysis report, and post it as a code review comment to the code review platform as shown in Figure 7. Using an internal document that maps the (non-intermediate) build targets to the names of the deliverables of the studied project, we show the deliverables that the code patch impacts on. The names of the deliverables also correspond to internal teams that are responsible for them. Therefore, the patch authors will know who may be interested before deciding to which team this code review should be assigned.

Moreover, the summary includes how broad the code change will affect the system by displaying the number of high-level deliverables each changed file impacts. If the impact on the system is broad or affects a key customer-facing deliverable, the report can alert reviewers to more carefully assess the patch. To assist users in knowing what exactly is impacted, the summary contains a URL that points to a full impact analysis report page. The report page contains the full paths of the files and components that each changed file in the patch impacts.

Once the comment containing the summary is posted, developers and code reviewers who are designated to assess the code patch will receive a notification from the code review platform, indicating that BLIMP Tracer has finished analyzing and publishing the impact analysis of the patch. Developers and reviewers can plan their code assessment activities accordingly using the information provided by BLIMP Tracer. BLIMP Tracer typically publishes reports for newly uploaded patches within 30 minutes. Note that the speed of BLIMP Tracer is not of concern because developers rarely react to a new review request more quickly than that.

V. User Study Design

We now discuss the design and execution of our user study with developers at Dell EMC. More specifically, we describe how we conducted semi-structured interviews and plan for future improvements.

A. An Overview of the Industrial System

BLIMP Tracer is deployed within a large, multinational product team of Dell EMC. The product suite that this team produces provides solutions for enterprise data backup and recovery. This product itself dates back to early 1990s, and has over ten million lines of code. Despite being implemented in a variety of programming languages (C, C++, Java, and C# to name a few), the product suite is highly portable, supporting product variants that run on Windows, as well as various flavours of UNIX, Linux, and macOS.

The complexity of the system and its build dependency graph make it an ideal subject system to pilot BLIMP Tracer. Indeed, understanding the large and complex build dependency graph of this product suite is difficult, even for senior Dell EMC developers.

TABLE I: An overview of the interviewed developers.

Name	Dell EMC Exp. (Yrs)	Studied Proj. Exp. (Yrs)
Developer A	2	0
Developer B	13	8
Developer C	11	11
Developer D	2	2
Developer E	11	11

B. Interview Design

Using the information that we gathered during the survey, we designed semi-structured interviews [26] for developers who expressed their interest in further discussing our proposed solution and how it could be improved. Semi-structured interviews contain planned open and close-ended questions, but the order is not necessarily the same as planned. In addition, during a semi-structured interview, interviewers are free to explore new findings and improvise the questions. We choose to perform semi-structured interviews instead of a more rigidly structured interview to allow for ideas that emerge during the interview to be explored to some extent. The interview sessions were recorded, transcribed, and coded.

In total, we conducted one-on-one interviews with five developers. Table I provides an overview of the participant experience levels. The participants have two to 13 years of experience working at Dell EMC (median of eleven years), and zero to eleven years of experience working on the subject product suite (median of eight years).

The purpose of the interviews was to discern whether BLIMP Tracer helps developers to perform impact analysis on patches. During the interview, we asked participants to show on screen how they assess the impacted deliverables of a code patch (without the help of BLIMP Tracer) Developers were asked to follow a think aloud protocol to enable us to gather data about why they are performing the tasks that they are performing. At the same time, we record how much time they spend. We then invite the interviewee to use BLIMP Tracer to assess the impact of the same patch, and determine if the impacted deliverables computed by BLIMP Tracer match the expected result.

VI. User Study Results

In this section, we present the results of the semi-structured interviews with respect to effectiveness and additional benefits.

A. Effectiveness of BLIMP Tracer

One of the primary goals of BLIMP Tracer is to help developers to better understand the impact of patches. During the one-to-one interviews, we observe how developers currently conduct impact investigations for a given patch. More specifically, we selected patches that include regular `.c` files, as well as header `.h` files, to cover files that have both small and large potential impact. We invite participants to estimate the number of components and deliverables that are impacted by the patch and name some of them, using methods that are most comfortable to them. After that, we introduce BLIMP Tracer and ask developers how they would use it during

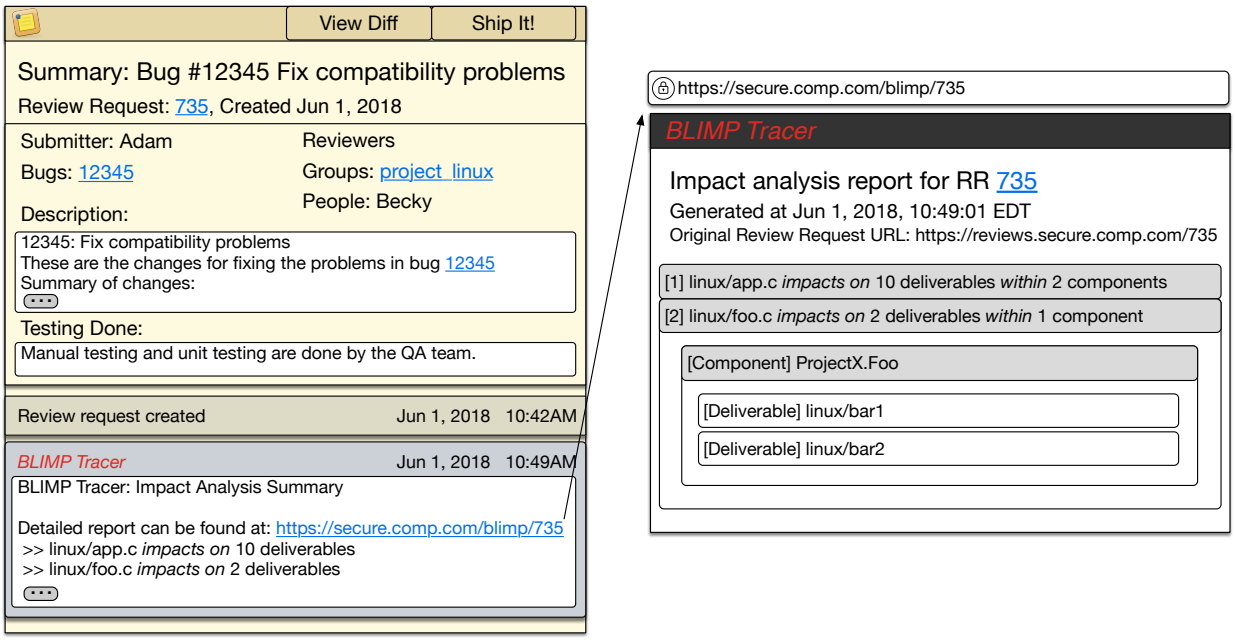

Fig. 7: An illustration of the BLIMP Tracer interface, integrated with the code review platform. The bottom of the left image resembles a sample comment posted by BLIMP Tracer.

code review. We record the interviews, transcribed and coded the developers response. By analyzing the transcribed and coded interview data, we identify three aspects that BLIMP Tracer can improve in the impact analysis procedures for the developers.

1) Using only command line tools for impact analysis is inefficient: Echoing the responses from the survey, all participants stated that when they investigate the impact of a patch for code review purposes, they usually use command line tools. However, using command line tools often does not give a full picture of the deliverables that a patch impacts. Developers B, C, and E state that if no dedicated tool for impact analysis is provided, they would only check the components that are immediately impacted by the changed file. In other words, the manual impact analysis that they conduct does not trace beyond one layer of impacted deliverables. Since these first-layer deliverables often have several transitive dependencies, the impact is likely being underestimated. Moreover, the developers agree the impact analysis process using `grep` and `find` is "*slow*". Developer A commented that for a relatively large file used by several components, "*it is impossible to do [impact analysis] by hand*".

2) BLIMP Tracer provides access to impact information at the right time for developers: The participants agree that integrating an impact analysis system with the code review platform is beneficial for a more well-rounded understanding of a code patch in a timely manner. Indeed, Developer C commented that although there are tools that analyze the static dependency structure of a system, he does not run the tool every time when he is asked to review a patch. Integrating a build-based impact analysis directly into the code reviewing process shows plenty promise. Indeed, Developer D explains that since doing an impact investigation of a patch takes time, he only checks the impact for the patch that he authors. The introduction of BLIMP Tracer will change the reviewing and developing behaviour. Developer C stated that BLIMP Tracer would likely help him to assess "*a patch that has a lot of impacted [deliverables]*".

3) BLIMP Tracer improves developers' awareness of the impact that patches have on system architecture: Participants agree that integrating BLIMP Tracer with the code review platform will accelerate and improve their workflows (Developers C, D and E). Since BLIMP Tracer automatically displays impact analysis reports directly, no manual input is required for developers to see the impacted deliverables. While Developer D stated that he had used a dedicated impact assessment tool during development, the others rely on command line tools to get an approximate sense of the impact of a patch. Indeed, Developer C commented that having BLIMP Tracer in the code reviewing platform can help reviewers and patch authors to "*make sure the impacted deliverables [have been tested]*".

Developers on the studied product team at Dell EMC agree that in addition to improving development workflow, BLIMP Tracer has the potential to improve the breadth and depth of impact analysis, as well as save developer time and effort.

B. Additional Benefits of BLIMP Tracer

In addition to making impact analysis easier and faster for the developers, we wish to know how BLIMP Tracer would benefit other aspects of development. In order to do so, we asked survey participants open-ended questions after they had the opportunity to try BLIMP Tracer. More specif-

ically, following the nature of semi-structured interview, the questions are based on the participant's comments on BLIMP Tracer during the trials. For example, if the participant made comments about the accuracy of the results of BLIMP Tracer, we would follow up with questions asking what deliverables were surprisingly included in or excluded from the impact list. The follow-up questions unveil two types of benefits that BLIMP Tracer may provide to improve understanding of the dependency structure of the studied system.

1) BLIMP Tracer provides knowledge to developers to reduce unnecessary dependencies in a system: First, using BLIMP Tracer, developers are able to identify deliverables that do not have a surface-visible, direct relationship with the changed files. This may expose problematic or unnecessary dependencies in the build dependency graph [7]. Removal of these unnecessary dependencies may speed up incremental builds. One theme that Developer C and E raised after examining the BLIMP Tracer report was that there were some unexpected deliverables appearing in the report. For example, Developer E commented that he found it "*odd*" that a component may be impacted by some change in a function in some other module. However, after some contemplation, he commented that it is "*understandable how the function is used in that deliverable, but I need some time to understand the logic behind it*". Knowing this dependency, he can decide whether to note this dependency, or to notify the module owners to remove the potentially unnecessary dependencies.

2) BLIMP Tracer helps accelerating newcomers' onboarding process: Moreover, BLIMP Tracer provides a resource that can help new developers to improve their understanding of the system architecture. A solid understanding of the project structure will reduce the risk of "shotgun surgery" [22], which would degrade the system architecture. Indeed, Developer D mentioned that when he was a newcomer to the project, the learning curve was steep to understand the connections between different project components. Therefore, he said: "*if I were a newcomer, I would use BLIMP Tracer to learn the dependency of files*".

BLIMP Tracer can help the community and developers in improving and understanding the system architecture.

VII. Related Work

In this section, we situate our work with respect to the related work on code review and impact analysis.

A. Code Review

Defect hunting is not the only outcome of the code review practice. Code review also serves as a platform for knowledge transfer and building team awareness [5]. Rigby and Storey [25] examined five open-source software projects and found that developers discuss not only code defects, but also project design, architecture, project scope, and process issues. Beller et al. [6] found that for each functional issue in review discussion, three maintainability issues are fixed. BLIMP Tracer aims to provide more information on the impact of code patches under review, so that developers can have a clearer understanding of the impact that patches have on the set of (customer-facing) deliverables and products.

Understanding the architecture of the software system is crucial as software defects may often be related to incorrect dependencies. Seo et al. [27] studied 26.6 million builds at Google and observe that most of the build failures are associated with dependencies (i.e., design or architectural-level component interactions). Indeed, Paixao et al. [21] found that developers are generally not aware of architectural changes. They analyzed code review data from four open source systems in conjunction with their commits, and found that only 38% of time do developers discuss the impact of their changes on the architectural structure. To aid in exposing developers to the higher level impact of their changes, we propose BLIMP Tracer, a build impact analysis tool that plugs into the code reviewing interface. The long term vision of BLIMP Tracer is to improve software quality by more clearly explaining to developers what the impact of their patches are. Armed with that clearer understanding, reviewers and testers can focus their effort more effectively.

B. Build Impact Analysis

We derive the definition of build impact analysis from that of change impact analysis. Change Impact Analysis (CIA) refers to the efforts to identify the potential consequences of a change to a software system [4]. Similarly, build impact analysis finds the consequence of a change with regard to build system inputs (source code, data files) and outputs (project deliverables, products).

Researchers have explored ways to conduct CIA for software in different languages and at various granularity levels. Ren et al. [23] designed *Chianti*, which uses the interdependent changes' history to determine change impact for Java programs. Apiwattanapong et al. [3] introduced an algorithm that uses a small amount of dynamic information to efficiently analyze change impact at the level of methods. Gyori et al. [14] proposed an algorithm that uses equivalence relations to discover change impact at the level of statements. Li et al. [16] surveyed 30 academic publications and found that although CIA is increasingly crucial in software maintenance, most of the proposed tools in academia are yet to be applied in industry. To bridge the gap, we introduce an impact analysis tool (BLIMP Tracer) that we developed and integrated with a production code reviewing environment in industry.

Researchers have proposed techniques to analyze data in previous studies with respect to impact analysis and build system analysis. Breech et al. [10] used static analysis to estimate the influence of a change by considering scoping, function signatures, and global variable accesses. Canfora and Cerulo [11] used information retrieval algorithms to link the text-based change request description and the code entities impacted by the change. Jashki et al. [15] proposed an impact analysis technique that creates clusters of closely associated files by mining their co-modification history in version control systems. Tamrawi et al. [28] proposed SYMake, an infrastruc-

ture and tool that evaluates Makefiles symbolically, and used it to detect code smells and errors. Al-Kofahi et al. [2] developed MkDiff to detect changes to a Makefile at the semantic level. Adams et al. [1] designed MAKAO, a tool for visualizing, querying, refactoring, and validating build dependency graphs through parsing build logs. BLIMP Tracer combines impact and build system analyses by providing build impact analysis report based on information retrieved from build data in the past.

VIII. Threats to Validity

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

A. External Validity

Threats to external validity have to do with the generalizability of our results to other subject systems. BLIMP Tracer is only integrated with the code review platform of the studied product team at Dell EMC. The focus on one project may affect the generalizability of the results. Although we have reached out to developers with various years of development and project experience, future study on other subjects may be needed to arrive at more general conclusions.

Similar to other software engineering studies, we have a low response rate with our surveys and interviews at Dell EMC. We recognize that the general population of our studied projects might have different characteristics and opinions than the ones that we present. Nevertheless, the purpose of our survey and interview is not to achieve generalizability, but rather to gather feedback and insights from practitioners who will interact with BLIMP Tracer on a daily basis.

B. Internal Validity

Threats to internal validity have to do with whether other plausible hypotheses could explain our results. We conclude that BLIMP Tracer shows promise due to our survey and interview studies. It may be that the participants were biased towards providing positive feedback to us due to social pressure. To combat this, we explained to all participants that their frank and honest feedback was what we needed to collect in order to improve BLIMP Tracer. Nevertheless, the response may still have been biased towards the positive.

C. Construct Validity

Threats to construct validity have to do with the alignment of our choice of indicators with what we set out to measure. Since we generate the build impact analysis report based on the traversal of build dependency graphs, changes that modify the BDG may result in inaccurate report. However, in the studied system, changes that have the potential to change the BDG are infrequent. In order to provide the most accurate BDG for generating the reports, we extract the BDG frequently (twice every day). In addition, BLIMP Tracer prints a warning message when a change to a known build specification is detected in the code patch under analysis.

BLIMP Tracer calculates the result of build impact analysis using files as a unit. Because of that, some of the 'impacted modules' in the report page may not be a direct result from the code change. Rather, the 'impacted modules' could be directly related to some other functions in the file. Our analysis is at the file-level because this is the granularity at which the Make build technology operates. Nonetheless, if a finer-grained build dependency graph were to become available, the impact analysis results would likely be even more useful for developers.

IX. Conclusion

Code review is widely used in modern development process to ensure software quality. However, the mere existence of code review does not promise improvement in software quality. A key concern in code review is data-driven discussions of patch implications. To accurately assign reviewers to assess code patches and to pinpoint the potential issues that affect the system, stakeholders need to know what areas of the software will be impacted by the changes. We introduce BLIMP Tracer, a build impact analysis system that integrates with the Review Board code reviewing environment of a product team at Dell EMC. We evaluate the effectiveness of BLIMP Tracer by conducting a qualitative study. Through a study that involves semi-structured interviews with Dell EMC developers, we make the following conclusions:

- Before the introduction of BLIMP Tracer, developers often use general-purpose command line tools to analyze the build impact of a code patch.
- BLIMP Tracer not only made build impact analysis on code patches faster, but also vastly improves the depth and breath of impact analysis when compared to traditional methods.
- BLIMP Tracer can help to onboard new developers by helping them to better understand the system architecture.

We acknowledge that the Dell EMC project is unique as it uses a single build system (`make`). Because of that, future work is needed to tackle more complex problems in the presence of multiple build tools or a fractured build graph. To serve as a transition between the existing technologies that enable local dependency checks to BLIMP Tracer, it may also be helpful to offer a service to query for dependencies that offers information on the degree of dependency of an impacted deliverable. In addition, future data on whether the developers use other dimensions of code patches to determine the priority of review requests may also be a useful extension to this work.

Acknowledgements

This research project was supported by Mitacs Canada.

References

[1] B. Adams, H. Tromp, K. De Schutter, and W. De Meuter, "Design recovery and maintenance of build systems," in *Proceedings of the 23rd International Conference on Software Maintenance (ICSM)*. IEEE, 2007, pp. 114–123.

[2] 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 *Proceedings of the 28th International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 150–159.

[3] T. Apiwattanapong, A. Orso, and M. J. Harrold, "Efficient and precise dynamic impact analysis using execute-after sequences," in *Proceedings of the 27th International Conference on Software Engineering*. ACM, 2005, pp. 432–441.

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

[5] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. IEEE Press, 2013, pp. 712–721.

[6] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern code reviews in open-source projects: Which problems do they fix?" in *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*. ACM, 2014, pp. 202–211.

[7] 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.

[8] A. Bosu and J. C. Carver, "Impact of developer reputation on code review outcomes in oss projects: An empirical investigation," in *Proceedings of the 8th International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM, 2014, p. 33.

[9] 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 (MSR)*. IEEE, 2015, pp. 146–156.

[10] B. Breech, M. Tegtmeyer, and L. Pollock, "Integrating influence mechanisms into impact analysis for increased precision," in *Proceedings of the 22nd International Conference on Software Maintenance (ICSM)*. IEEE, 2006, pp. 55–65.

[11] G. Canfora and L. Cerulo, "Fine grained indexing of software repositories to support impact analysis," in *Proceedings of the 3rd International Workshop on Mining Software Repositories (MSR)*. ACM, 2006, pp. 105–111.

[12] Q. Cao, R. Wen, and S. McIntosh, "Forecasting the duration of incremental build jobs," in *Proceedings of the 33rd International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 524–528.

[13] J. Czerwonka, M. Greiler, and J. Tilford, "Code reviews do not find bugs: how the current code review best practice slows us down," in *Proceedings of the 37th International Conference on Software Engineering (ICSE)*. IEEE Press, 2015, pp. 27–28.

[14] A. Gyori, S. K. Lahiri, and N. Partush, "Refining interprocedural change-impact analysis using equivalence relations," in *Proceedings of the 26th International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2017, pp. 318–328.

[15] M.-A. Jashki, R. Zafarani, and E. Bagheri, "Towards a more efficient static software change impact analysis method," in *Proceedings of the 8th Workshop on Program Analysis for Software Tools and Engineering*. ACM, 2008, pp. 84–90.

[16] 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.

[17] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects," in *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*. ACM, 2014, pp. 192–201.

[18] ——, "An empirical study of the impact of modern code review practices on software quality," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2146–2189, 2016.

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

[20] R. Morales, S. McIntosh, and F. Khomh, "Do code review practices impact design quality? a case study of the qt, vtk, and itk projects," in *Proceedings of the 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2015, pp. 171–180.

[21] M. Paixao, J. Krinke, D. Han, C. Ragkhitwetsagul, and M. Harman, "Are developers aware of the architectural impact of their changes?" in *Proceedings of the 32nd International Conference on Automated Software Engineering (ICSE)*. IEEE Press, 2017, pp. 95–105.

[22] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[23] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: a tool for change impact analysis of java programs," in *ACM Sigplan Notices*, vol. 39, no. 10. ACM, 2004, pp. 432–448.

[24] P. C. Rigby and C. Bird, "Convergent contemporary software peer review practices," in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (FSE)*. ACM, 2013, pp. 202–212.

[25] P. C. Rigby and M.-A. Storey, "Understanding broadcast based peer review on open source software projects," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*. ACM, 2011, pp. 541–550.

[26] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, p. 131, 2009.

[27] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, "Programmers' build errors: a case study (at google)," in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 724–734.

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

[29] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *Proceedings of the 37th International Conference on Software Engineering (ICSE)*. IEEE Press, 2015, pp. 99–108.

[30] Q. Tu and M. W. Godfrey, "An integrated approach for studying architectural evolution," in *Proceedings of the 10th International Workshop on Program Comprehension*. IEEE, 2002, pp. 127–136.