Hypothesizer: A Hypothesis-Based Debugger to Find and Test Debugging Hypotheses

Abdulaziz Alaboudi
George Mason University
Fairfax, VA, USA
aalaboud@gmu.edu

Thomas D. LaToza
George Mason University
Fairfax, VA, USA
tlatoza@gmu.edu

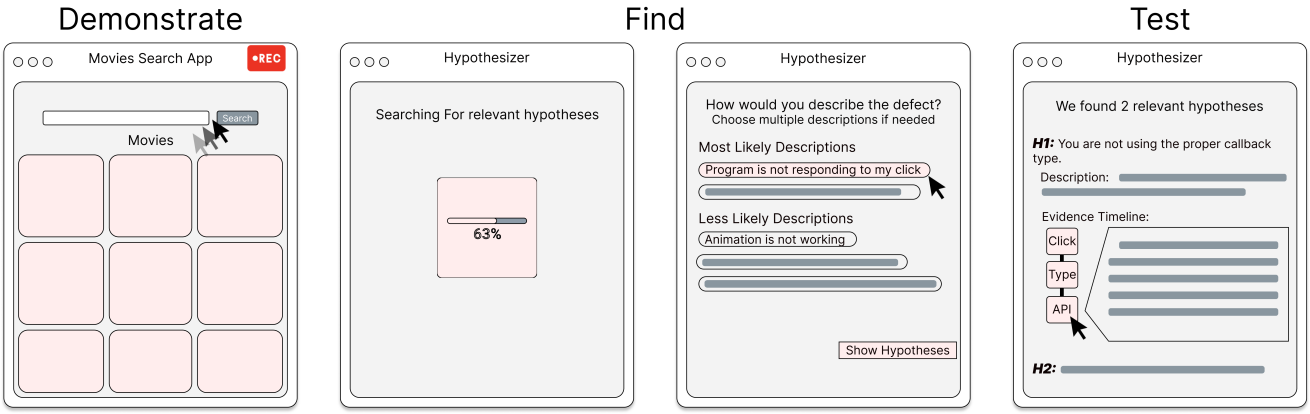

Figure 1: In Hypothesizer, a developer first *demonstrates* a defect while Hypothesizer records the program behavior. The developer then works collaboratively with Hypothesizer to *find* relevant hypotheses, answering questions to clarify the defect's symptoms. The developer *tests* a hypothesis by investigating behavior using a timeline of key events and follows step-by-step instructions to implement a fix.

ABSTRACT

When software defects occur, developers begin the debugging process by formulating hypotheses to explain the cause. These hypotheses guide the investigation process, determining which evidence developers gather to accept or reject the hypothesis, such as parts of the code and program state developers examine. However, existing debugging techniques do not offer support in finding relevant hypotheses, leading to wasted time testing hypotheses and examining code that ultimately does not lead to a fix. To address this issue, we introduce a new type of debugging tool, the hypothesis-based debugger, and an implementation of this tool in Hypothesizer. Hypothesis-based debuggers support developers from the beginning of the debugging process by finding relevant hypotheses until the defect is fixed. To debug using Hypothesizer, developers first demonstrate the defect, generating a recording of the program behavior with code execution, user interface events, network communications, and user interface changes. Based on this information and the developer's descriptions of the symptoms, Hypothesizer finds relevant hypotheses, analyzes the code to identify relevant evidence to test the hypothesis, and generates an investigation plan through a timeline view. This summarizes all evidence items related to the hypothesis, indicates whether the hypothesis is likely to be true by showing which evidence items were confirmed in the recording, and enables the developer to quickly check evidence in the recording by viewing code snippets for each evidence item. A randomized controlled experiment with 16 professional developers found that, compared to traditional debugging tools and techniques such as breakpoint debuggers and Stack Overflow, Hypothesizer dramatically improved the success rate of fixing defects by a factor of five and decreased the time to debug by a factor of three.

KEYWORDS

debugging, debugging hypotheses, debugging tools

ACM Reference Format:
Abdulaziz Alaboudi and Thomas D. LaToza. 2023. Hypothesizer: A Hypothesis-Based Debugger to Find and Test Debugging Hypotheses. In *The 36th Annual ACM Symposium on User Interface Software and Technology (UIST '23), October 29–November 01, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3586183.3606781

UIST '23, October 29–November 01, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0132-0/23/10.
https://doi.org/10.1145/3586183.3606781

1 INTRODUCTION

When a program behaves unexpectedly, developers often ask themselves, "Why is the program not working as expected?" They then begin to formulate hypotheses to explain the cause of the defect (e.g., "I think I may not have properly parsed the data coming from the server") [26, 33, 45]. These hypotheses serve as a guide for the

rest of the debugging process, during which developers gather evidence to accept or reject the hypothesis. Evidence may be found in many places, such as source code, program state as observed through the debugger or logs, or network communications [4, 26]. If a developer has a hypothesis that correctly explains the cause of the defect, their chances to fix the defect increase significantly [2].

However, identifying a correct hypothesis can be a daunting task for developers. Instead, developers often formulate many incorrect hypotheses, leading to time-consuming investigation gathering evidence to test hypotheses that do not yield any progress towards a fix [2, 8, 20]. For example, when a program fails to respond to a user's click, developers may struggle to determine the correct hypothesis due to the numerous possible causes of such a defect [36]. A button may not work as expected for many reasons, ranging from an improperly implemented click event handler to broken asynchronous communication between servers triggered after the click. Investigating each hypothesis they formulate, developers may waste valuable time and resources investigating hypotheses that do not advance the debugging process. In some cases, developers may become stuck, unable to identify any further hypotheses, making it even more challenging to fix the defect.

While traditional debuggers assist developers in more easily observing runtime values or in localizing the defect to a specific line of code, these tools do not explicitly help to find a hypothesis. Developers must first still rely on their knowledge to formulate a hypothesis and then use these tools to test it. Without a hypothesis, developers may end up stepping through the entire program without progressing toward a fix [22]. The most effective tool for developers to find the correct hypothesis is to seek help from experienced developers [19, 26]. Hypotheses that explained a defect in one program may be applied to other programs with the same underlying issue. Expert developers who have encountered defects before leverage this experience to formulate a hypothesis. However, these hypotheses reside in experts' heads, and current methods for sharing and searching for them such as Stack Overflow often do not work effectively [7, 30, 37].

This raises an important question: **How can a debugger directly assist in finding and testing hypotheses?** To address this question, we propose the concept of hypothesis-based debuggers, which support developers in working with hypotheses throughout the debugging process. Hypothesis-based debuggers work collaboratively with the developer throughout the debugging process to find and test relevant hypotheses until the defect is fixed. We have implemented this approach in *Hypothesizer* (Figure 1).

To begin debugging with Hypothesizer, a developer *demonstrates* a defect by interacting with the program. Hypothesizer records program behavior during this interaction, encompassing code execution, user interface events, network communications, and user interface changes. Hypothesizer then *finds* relevant hypotheses by loading a dataset of hypotheses and querying the recording for evidence which confirms or contradicts each hypothesis. Each hypothesis consists of a description (e.g., "You may not have parsed the data coming from the server correctly"), a set of labels describing the defect's symptoms (e.g., "clicking a button does nothing"), and a list of conditions (e.g., a user clicks on a button, a click event is triggered, data is loaded from the server, but the parsing API is not used). When a program behavior satisfies a condition, it provides evidence that the hypothesis is relevant. Hypothesizer reports relevant hypotheses matching at least 50% of its conditions within the recorded behavior. To further narrow down relevant hypotheses, the developer may select labels describing defect symptoms. Hypothesizer then presents an investigation plan to *test* a hypothesis in the form of a timeline view. This summarizes evidence identified in the recording which confirms or contradicts each condition and allows the developer to quickly examine evidence by viewing code snippets for each evidence item. The investigation plan includes potential code locations for modifications to fix the defect.

Hypothesizer's ability to find relevant hypotheses stems from its access to and use of a dataset of hypotheses capturing the knowledge of experienced developers. For this study, the first author populated a dataset with ten hypotheses based on discussions of defects on Stack Overflow. To evaluate Hypothesizer's ability to utilize the hypotheses dataset, we introduced ten defects relevant to these hypotheses into separate open-source web applications. We found that Hypothesizer successfully identified the most relevant hypothesis for all of the defects in multiple programs.

In a randomized controlled experiment with 16 professional developers, we examined the effectiveness of Hypothesizer in assisting developers with debugging. The findings indicate that when developers employed Hypothesizer, they were significantly more successful in fixing defects than when they relied on traditional tools such as breakpoints debugger and Stack Overflow. Developers with Hypothesizer fixed the defect within eight minutes. In contrast, without Hypothesizer, developers struggled, with only 19% of developers successfully fixing the defect, taking an average time of 21 minutes. Hypothesizer helped developers to identify relevant hypotheses and fix the defect with fewer code navigation and program reruns. Even with limited knowledge of the program being debugged, the timeline view and step-by-step plan provided by Hypothesizer enabled developers to investigate relevant hypotheses and fix defects more efficiently.

The contributions of this paper are :

- The concept of hypothesis-based debuggers, and an initial prototype for web debugging in Hypothesizer. Hypothesis-based debugging supports the developer throughout the debugging process to demonstrate the defect, find relevant hypotheses, gather evidence to test hypotheses, and implement a fix.
- Evidence from a technical evaluation demonstrating the ability to successfully identify relevant hypotheses for defects in open source applications within a few seconds.
- Evidence from a randomized controlled experiment on the impact of hypothesis-based debugging on the debugging process.

2 MOTIVATING EXAMPLE

Sara is a software engineer on a team building a movie search app using modern web technologies. Her current task involves implementing a filtering feature for the search interface. While she has managed to implement the basic filtering functionalities and UI elements, she notices that clicking the search button after typing any movie in the search input does not show any results.

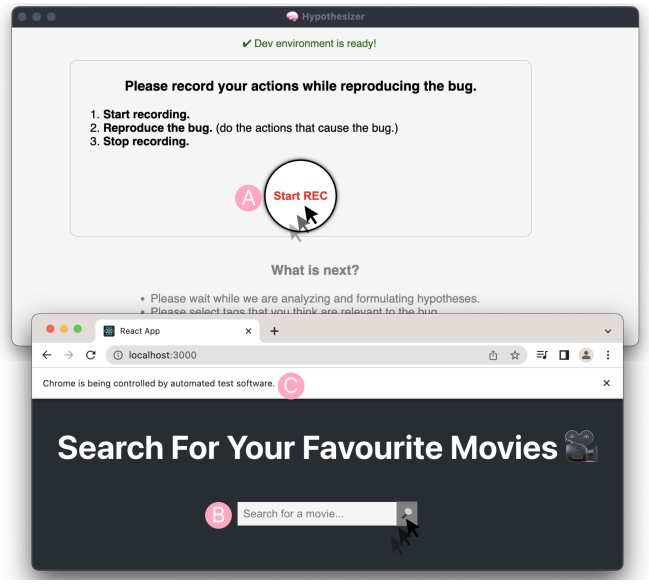

Figure 2: A Developers click the "Start REC" button to start the recording, B demonstrate the defect within a fully instrumented Chrome web browser, and C generate a recording until ending the session.

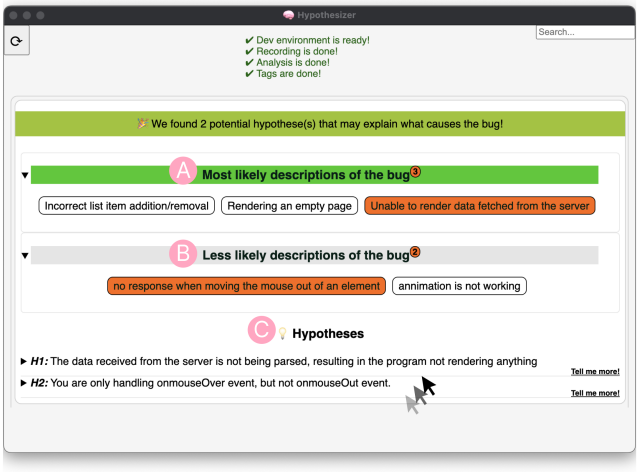

Figure 3: Developers clarify their intent by selecting one or more symptom labels. Hypothesizer categorizes labels into most likely (A) and less likely descriptions (B). Hypothesizer then lists relevant hypotheses (C).

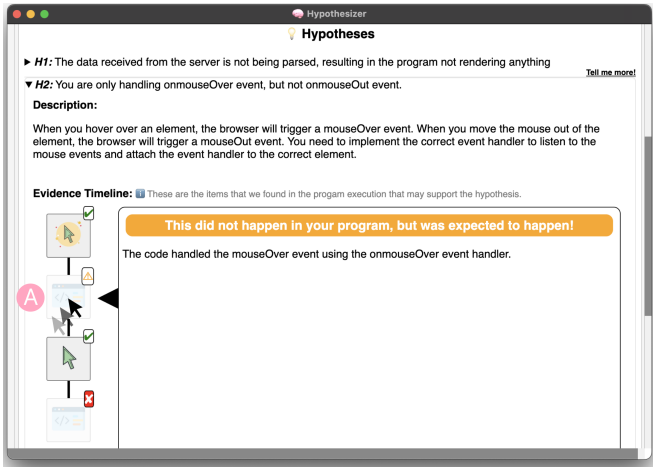

Figure 4: Evidence contradicting the hypothesis is indicated with a warning indicator (⚠) in the timeline (A).

"Why does clicking the search button not show any movies?", Sara asks. Rather than formulate and investigate hypotheses with a traditional debugger, Sara decides to start with Hypothesizer.

When Sara opens Hypothesizer, two panels immediately appear (Figure 2). The first panel is an instrumented Chrome browser running the movies search app, as shown in Figure 2 (C). The second panel is the Hypothesizer recording interface, shown in Figure 2 (A). Sara begins debugging with Hypothesizer by demonstrating the defect. She clicks the "Start REC" button, moves her cursor to the movie app and types "Superman", and clicks the search button. After demonstrating that the movies search button does not show any movie, she clicks "Stop REC" inside Hypothesizer.

Hypothesizer prompts Sara to wait while identifying relevant hypotheses. After a few seconds, Hypothesizer asks Sara to select one or more labels describing the symptoms of the defect. These labels are grouped into most likely and less likely descriptions. Sara reads through the labels and selects two. The first states that the defect is "Unable to render data fetched from the server" (Figure 3 A), which she picked because the movie search app communicates with a server to load the movie's data. The second states, "No response when moving the mouse out of an element" (Figure 3 B). Although Sara is unsure about the relation between the defect and this symptom, she selects it as she thinks the program may need to respond when the mouse moves from the search button.

Based on these selections, Hypothesizer offers Sara two relevant hypotheses that may explain the defect (Figure 3 C). Sara begins her investigation by selecting the hypothesis "You are only handling the onmouseOver event, but not the onmouseOut event." She reviews its description and examines the timeline view summarizing the evidence supporting it. Sara notices that this hypothesis has a contradictory evidence item, indicated with a warning indicator (⚠) (Figure 4). The hypothesis expects a callback handler of a mouseOver event within the recorded program behavior. However, Hypothesizer did not find this in the program behavior, suggesting the hypothesis may be incorrect. Sara quickly moves on to a different hypothesis.

Sara continues debugging by selecting a hypothesis which suggests that "The data received from the server is not being parsed, resulting in the program not rendering anything." (Figure 5 A). She quickly skims the timeline view of the evidence and notices no contradictory evidence. She proceeds to review each item in

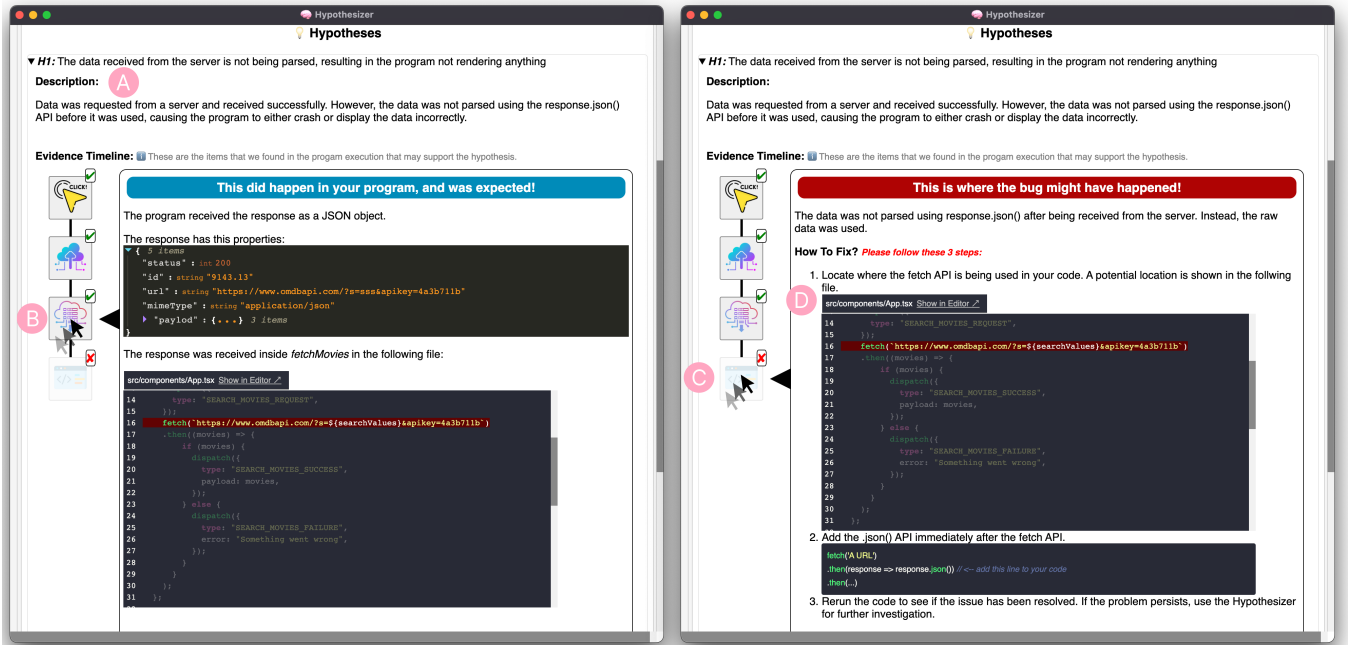

Figure 5: Developers test a hypothesis by expanding it, revealing an extended description(A) and timeline view summarizing evidence (B) confirming (indicated with a ✔) or contradicting the hypothesis. Evidence items may also indicate that they contain a potential starting point for a fix (indicated with a ✘)(C), with step-by-step instructions for implementing a fix (D).

the timeline, clicking and reading each item's description and examining the associated details, such as source code, runtime state, network activities, and events (Figure 5 B).

As she investigates the evidence, Sara sees that most items in the timeline are marked with a checkmark symbol (✔), indicating that these evidence items confirm the relevance of the hypothesis. However, one item is marked with a cross symbol (✘), which prompts Sara to wonder what it means. She clicks on it, and discovers that it indicates a potential starting point to fix the defect (Figure 5 C). Hypothesizer provides step-by-step instructions for fixing the defect. Based on her investigation so far, Sara believes this hypothesis is correct and decides to follow the plan to fix the defect. She opens the suggested code location by clicking "Show in Editor" (Figure 5 D). After making the suggested changes, Sara reruns the app and verifies that the defect is fixed.

3 RELATED WORK

Hypothesizer builds on decades of work examining the process by which developers debug as well as prior debugging tools that help developers debug more effectively.

Early research on code comprehension and debugging demonstrated the critical role of hypotheses in the debugging process [9, 16, 27, 29, 39]. Brooks [9] introduced the idea of "global hypotheses" that drive developers to search for evidence and refine their understanding of a program. Letovsky [27] observed professional developers as they asked questions and formulated hypotheses in the form of "conjectures" to guide their debugging efforts. A study by Jeffries [16] found that experts were more likely to form correct hypotheses and effectively find relevant evidence. These early studies provide insight into the mental processes involved in debugging and suggest that the ability to form and test hypotheses is important to effective debugging.

Other research has focused on the specific steps that developers take during the debugging process. Developers ask questions and form hypotheses about the cause of incorrect output and then test these hypotheses by examining code and inspecting program state [18, 19, 33]. However, most hypotheses that developers form are incorrect [2, 8, 19, 20], leading them to inspect irrelevant code and prolonging debugging. Information Foraging Theory has been used to explain how developers navigate through code while debugging by focusing on the scent of relevant code and avoiding explicit reference to developers' hypotheses [25].

When developers are unable to form hypotheses about the cause of a fault, they may turn to external resources for help. This often involves searching the internet for possible explanations or asking questions in online communities such as Stack Overflow [38, 44]. However, the success of this approach depends on the developer's ability to ask precise and well-informed questions and formulate high-quality search queries [7, 30, 37]. Another source of hypotheses is expert colleagues or coworkers [15]. In teams working on a large codebase, developers often communicate with each other to share knowledge and expertise. However, asking a teammate for help can introduce a context switch that wastes time and disrupts the debugging process, especially when the teammate is not immediately available [24]. Rather than ask a teammate each time a challenge occurs, developers may also capture and externalize

this knowledge for later use. In particular, developers may share strategic knowledge, but face challenges in communicating this knowledge effectively [5]. Tools may offer better support for this process, offering ways to share, find, and request programming strategies [6].

Many fully automated debugging tools do not present their results in a way that provides enough information for developers to understand the problem, why it is a problem, and what to do differently [17]. Fault localization tools and techniques have been built to help developers search for fault locations. For example, program slicing tools [40] often display to the user a ranked list of potentially faulty statements [12, 43, 46], shrinking the search space of potentially faulty statements developers must presumably consider. However, studies have found that developers often struggle to form correct hypotheses to explain the underlying cause of the fault and that fault locations alone do not support this [3, 32, 42]. Developers raise concerns about the usability of the output produced by such tools, reporting that false positives and a large volume of warnings make it difficult to effectively use the tool and understand the defect [17]. Automated analysis tools may detect code syntax or style errors but may be unable to identify defects caused by the absence of certain code patterns [41]. In these cases, the absence of code patterns may be crucial evidence in formulating a hypothesis, but static analysis tools are not designed to identify this type of evidence.

A variety of interactive techniques have been proposed to improve debugging. The Whyline [21] allows developers to ask questions about program output, enabling them to interactively trace dependencies backwards and locate the source of the fault without making any assumptions about the incorrect behavior. REACHER [23] provides an interactive call graph visualization that encodes various properties to assist developers in answering questions related to causality, ordering, type membership, repetition, choice, and other relationships, helping developers to stay oriented while navigating. Timelapse [10] is an omniscient [28] debugging tool designed for rapidly recording, reproducing, and debugging interactive behaviors in web applications, allowing developers to browse, visualize, and navigate within recorded program executions using familiar debugging tools. These tools primarily focus on answering questions about program behavior, and do not directly address the task of finding hypotheses that explain the defect behavior.

While forming and testing hypotheses is essential to debugging, current debugging tools lack support for this critical process. Our paper seeks to bridge this gap by introducing a new type of debugging tool known as hypothesis-based debuggers which aid developers in identifying and testing hypotheses about the cause of a defect.

4 HYPOTHESIS-BASED DEBUGGERS

In this paper, we introduce hypothesis-based debuggers, which offer assistance throughout the debugging process by helping the developer demonstrate a defect, clarify the symptoms of the defect, identify a debugging hypothesis, and quickly test a hypothesis by identifying relevant evidence to examine.

4.1 Design Goals

Hypothesis-based debuggers aim to address three key design goals:

(1) **D1. Help developers find relevant hypotheses early in the debugging process.** Defect behavior can be complex, requiring reasoning about and connecting together user input, program state, code execution, network activity, API calls, and user interface changes, before developers may formulate hypotheses. Hypothesis-based debuggers assist developers in finding relevant hypotheses early in the debugging process. This helps in two ways. First, ensuring that developers focus on the most relevant hypotheses from the beginning saves time and effort by reducing the need to explore unrelated hypotheses. Second, having hypotheses to investigate from the beginning prevents developers from feeling stuck or directionless, providing a clear starting point for the debugging process. Hypothesizer addresses this design goal by letting developers demonstrate a defect, clarify the expected behavior, and then view hypotheses with supporting evidence found in the recording (Section 4.2.1).
(2) **D2. Help developers gather evidence to test hypotheses.** Developers test hypotheses by gathering a wide variety of evidence, such as by adding log statements or using breakpoint debuggers to manually set of breakpoints, step through code, and inspect variables [3, 33]. This process can be time-consuming, particularly when developers must stitch together information from multiple tools for their investigation. Hypothesis-based debuggers gather together and display the necessary information for developers to test a hypothesis. Hypothesizer achieves this by providing an investigation plan, shown through an interactive timeline view of the evidence for each relevant hypothesis (Section 4.2.2).
(3) **D3. Help developers fix the defect.** Possessing a hypothesis without a clear understanding of how to fix the defect hinders a developer's progress. For instance, developers may be aware that incorrect data parsing causes the defect, but may not know where or how to parse the data correctly. Hypothesis-based debuggers offer developers step-by-step instructions to implement a fix, suggesting potential code locations to start and a general description of what a fix might entail. Hypothesizer addresses this design goal by providing step-by-step instructions that guide developers toward fixing the defect. These instructions outline specific evidence items and their connection to the fix, indicating a potential location of the fix based on a related evidence item. We discussed this further in Section 4.2.3.

4.2 Hypothesizer

We developed Hypothesizer, a prototype hypothesis-based debugger which helps developers to find and test hypotheses for web applications. Hypothesizer is implemented as a stand-alone debugger utilizing the Chrome DevTools Protocol [11].

4.2.1 Finding Relevant Hypotheses (D1). Hypothesizer draws inspiration from the process developers use to formulate a hypothesis. Developers gather information from various sources to comprehend program behavior and formulate hypotheses [26]. As in omniscient

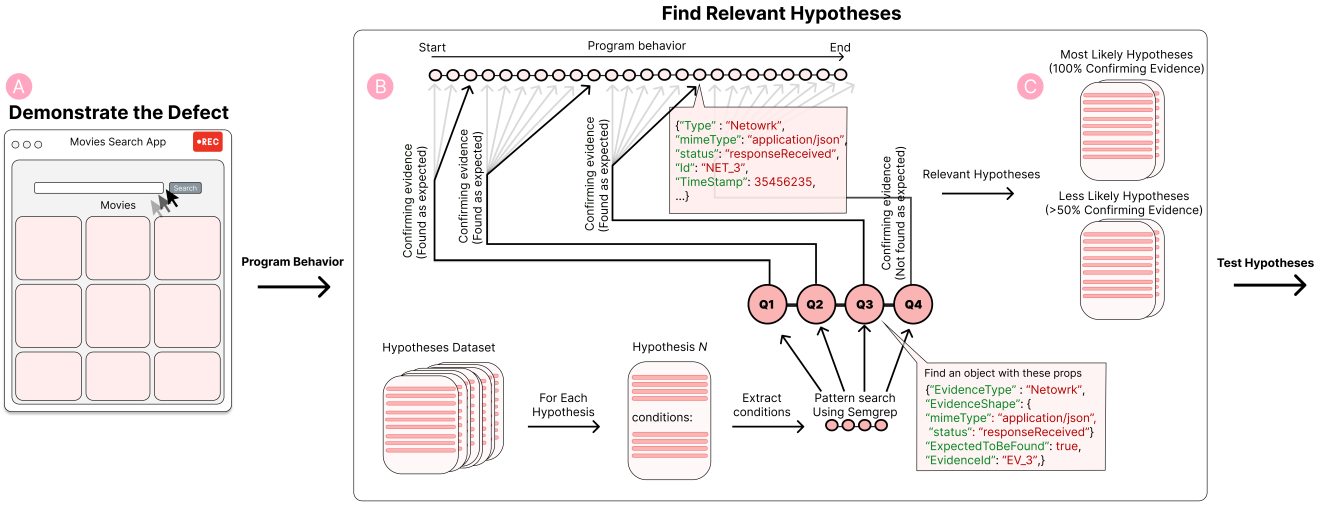

Figure 6: (A) As the developer demonstrates the defect, Hypothesizer records the program behavior. (B) For each hypothesis in the dataset, Hypothesizer then queries the recorded program behavior for the hypothesis's conditions. (C) Hypothesizer reports hypotheses with all conditions marked as confirming evidence as most likely and less likely if more than half but not all conditions marked as confirming evidence.

debuggers [28], Hypothesizer simplifies this process by asking the developer to demonstrate the defect a single time, capturing a trace recording all of the entire program behavior, and then letting the developer gather evidence by inspecting the recorded program behavior. After beginning the recording, developers interact with the program within a fully instrumented version of the Chrome browser, which Hypothesizer manages. Hypothesizer constructs a recording with timestamps of all user interface events, code execution, network communications, and user interface changes, until the developer terminates the recording session. Figure 2 illustrates the recording interface.

The recording captures the program's behavior, including the demonstration of the defect. Hypothesizer analyzes the recording, using a dataset of hypotheses to identify relevant hypotheses. This dataset embodies the knowledge possessed by expert developers, who use this knowledge to formulate hypotheses more effectively [13]. Each hypothesis contains a list of conditions that describe the program's behavior that would make the hypothesis relevant. When a recorded behavior of a program satisfies a condition, it provides supporting evidence for the hypothesis.

The conditions list of a hypothesis comprises descriptions of specific instances of program behavior and the order in which they must be satisfied in the recording. Incorporating an order for conditions enables Hypothesizer to determine a hypothesis's relevance more precisely. To identify relevant hypotheses, Hypothesizer iterates through all hypotheses in the dataset and searches for their conditions within the recorded program. Hypothesizer employs Semgrep [35], an open source pattern-matching engine, to search for condition items within the program behavior. For each condition item in a hypothesis, Hypothesizer generates a Semgrep pattern using the open source library to represent an instance of the program behavior and queries it against the recorded behavior. Semgrep's robust pattern-matching capabilities allow Hypothesizer to effectively search for a wide variety of patterns representing conditions in the program's behavior. For instance, consider a hypothesis with a condition item involving a submit button click. This condition can be described as follows:

```
{
"EvidenceType": "click",
"evidenceShape": {
"inputType": "submit",
"target": "BUTTON"
},
"shouldBeFound": "true"
}
```

Hypothesizer creates a Semgrep pattern to search for an entry in the program behavior that corresponds to a submit button click event. Figure 6 provides an overview of the process involved in identifying relevant hypotheses.

Hypothesizer categorizes each item in a hypothesis' conditions list as either confirming or contradictory evidence. A condition item is considered confirming evidence if it meets one of two criteria: it must either be present in the program behavior and expected to be found, or not present in the program behavior and not expected to be found. If a condition item does not meet these criteria, it is marked as contradictory evidence. Hypothesizer identifies hypotheses as relevant if 50% or more of the conditions list contains confirming evidence. Hypothesizer then presents relevant hypotheses to developers in two categories: most likely, if all of the hypothesis's conditions are marked as confirming evidence, and less likely, if more than half but not all conditions are marked as confirming evidence in the recorded program behavior.

After identifying relevant hypotheses, Hypothesizer further narrows down potentially relevant hypotheses by letting developers

Table 1: Evidence Types in Recording and Timeline View

Evidence Type	Icon	Information Displayed
keyboard event		Pressed keys, associated code snippets, and their corresponding locations.
Click/Submit event		Click count, associated code snippets, code location
MouseOver event		Mouseover element count, associated code snippets, code location
OnmouseOut event		OnmouseOut element count, associated code snippets, code location
Code pattern/API call		API calls count, associated code snippets, code location
Network request		# network requests, request bodies, associated code snippets, code location
Network response		# network responses, their response bodies, associated code snippets, code location
UI changes		# UI changes, types of changes, removed and added code snippets, code location

describe the symptoms of the defect. Developers can choose one or more symptom labels, where each label corresponds to at least one relevant hypothesis Hypothesizer has identified. Clarifying the defect's symptoms helps recover the developers' intention, which may be difficult to deduce solely from the program's behavior. For example, suppose a developer intends a part of the interface to animate after clicking a button. If the animation does not work as expected, the developer may then select a label stating "animation is not working ". With this additional information about the developer's intent, Hypothesizer can focus on hypotheses related to animation. This level of specificity is challenging to achieve from the recorded program behavior alone.

4.2.2 Testing a Hypothesis with an Investigation Plan (D2). After identifying relevant hypotheses, Hypothesizer helps developers quickly skim the list and decide which to investigate first. Hypothesizer displays those with the most confirming evidence first, showing a collapsible list of relevant hypotheses with a heading summarizing the hypothesis (Figure 3). Clicking on a hypothesis expands the hypothesis detail pane, offering the developer an investigation plan to test the hypothesis through an interactive timeline view of evidence items (Figure 5).

The interactive timeline summarizes how evidence found in the program's behavior confirms the hypothesis. By clicking on each item in the timeline, developers can examine the evidence and view related code snippets from the source code. To further investigate a specific code snippet, a developer can click on it, opening the corresponding source code within the developer's IDE. Evidence items include user interface events, network activity properties, method calls in the execution trace, code patterns, or user interface changes. Each evidence item displays information based on its type. For example, network communication items display the request and response payloads, and UI changes show the type of change and what was added or removed (Table 1).

To enable developers to quickly skim the evidence items in the timeline view, the type of each evidence item is indicated through an icon. Whether the evidence item was found in the recorded behavior is indicated by its opacity. Evidence marked with a (✔) indicates confirming evidence, meaning the evidence satisfied the condition of the hypothesis. Evidence marked with a (⚠) indicates contradictory evidence, where the condition was not identified in the recorded behavior, making the hypothesis less likely to be true.

4.2.3 Explaining How To Fix The Defect (D3). Hypothesizer identifies at least one evidence item as an initial step to fix the defect, indicated with a cross mark (✘) on the left corner of its icon. Clicking on this item opens the evidence information as well as a "How to Fix?" section offering step-by-step instructions to fix the defect (Figure 5 D). These instructions are built from a template in the hypothesis. These step-by-step instructions serve two primary purposes. First, they clarify the relationship between specific evidence items and a potential fix. Second, they suggest a location and plan to implement a fix, which developers can follow by using the link to open the corresponding code in their IDE.

4.2.4 Authoring Hypotheses. Experienced developers can share their expertise in identifying the root cause of defects by contributing to the dataset of hypotheses used by Hypothesizer. To author a hypothesis, a developer writes text explaining the hypothesis and the conditions that determines when it will apply. Developers first write text that captures how the hypothesis explains the a cause of a defect. Developers then specify the hypothesis conditions through a two-step process. Developers first record the program while demonstrating the defect. This recording is then exported by Hypothesizer as a JSON file, which contains an itemized representation of the execution where each event is represented as an object and its properties. This data serves as the foundation for identifying the conditions of the hypothesis. Developers can use this to craft conditions, deciding how specific or general the hypothesis should be based on their understanding of the defect (e.g., narrowing to a click event from a submit button rather than any click event). These conditions are then combined to form a hypothesis. Each condition includes a description of its relevance and importance to the hypothesis and a pattern describing how it appears in the execution, which can be used to construct a Semgrep pattern. An example of a hypothesis is shown in Appendix A.

4.2.5 System Scope and Limitations. While the scope of our Hypothesizer prototype is web applications, Hypothesizer is open source and may be adapted to support other types of applications. Hypothesizer uses the Chrome DevTools Protocol [11] to instrument the Chrome browser. To extend Hypothesizer to application types that are incompatible with the Chrome web browser, users may either extend the Chrome DevTools Protocol for custom runtime instrumentation or employ alternative methods of instrumentation.

Table 2: A summary of the hypotheses dataset used in the evaluation of Hypothesizer.

ID	Defect Symptom(s)	Hypothesis	Conditions
$H1$	Incorrect add/remove item of a list of items, Unexpected reload of the page	You are not using "preventDefault" API to prevent the default behavior of the submit button.	keyboard event, submit event, no prevent API call, network communication, UI reload
$H2$	Rendering an empty page, Unable to render data fetched from the server	The data received from the server is not being parsed, resulting in the program not rendering anything.	click event, Get network request, network response with JSON, no response.json() API call
$H3$	No response when pressing keys on the keyboard	You are not attaching the event listener for keyboard events to the correct element.	click event, keyboard event, no onKeydown callback
$H4$	No response when moving the mouse out of an element	You only handle onmouseOver event, not onmouseOut event.	mouseover event, mouseover callback, onmouseOut event, no onmouseOut callback
$H5$	Animation is not working or sluggish	You are not applying the animation inside setTimeout.	click event, setState API call, UI change, no setTimeout API call, transform animation API call
$H6$	Clicking on a button does something unexpected	You are not explicitly passing the props to the onClick event handler.	click event, onClick callback, no props passed to onClick callback
$H7$	Clicking on a button does nothing	You are not updating the state of the element(s) after the onClick event handler is called.	click event, onClick callback, no setState API call
$H8$	Clicking on a button does something unexpected	You are not assigning a type to the button element.	click event, onClick callback, no button with type property = button
$H9$	Clicking on a radio button selects the wrong choice	You are not placing the label element for the radio elements correctly	radio elements UI rendering, click on a radio element event, no label on the radio element
$H10$	Clicking on a button does nothing	You are not using currentTarget API to get the button value.	click event, onClick callback, pass the button value inside the callback, no currentTarget API call, error handling API call

Hypothesizer is designed to process codebases which generate tens of thousands of recorded program behavior entries. Hypothesizer is multi-threaded, significantly reducing analysis time. Hypothesizer's scalability is primarily influenced by the quantity of conditions to be evaluated rather than the number of hypotheses. Our observations revealed that many hypotheses share identical conditions (as shown in Table 3). Based on this insight, Hypothesizer evaluates unique conditions once and caches the results.

For defect recordings which are longer or contain numerous user interactions, processing time will increase. Several additional techniques may offer the potential for further improving scaling behavior. The analysis of defect recordings may be offloaded from developers' local machines to more scalable cloud servers. This approach may introduce security and privacy risks by exposing information about the source code to servers not owned by the developer. But emerging commercial tools such as Replay [34] suggest that it is possible to offload computations to third-party servers while ensuring privacy.

5 TECHNICAL EVALUATION

To examine the ability of Hypothesizer to identify the cause of defects in real-world open source applications, we conducted a technical evaluation. The goal of this evaluation was to examine how effectively hypotheses represent a cause of a defect by investigating the extent to which conditions for hypotheses generalize effectively across programs. We applied each hypothesis to different programs, rather than new defects. We examined the size of the program behavior recording, performance of Hypothesizer in working with these recordings, and overall effectiveness in finding the correct hypothesis. We inserted defects into each application and then examined the performance of Hypothesizer across these applications and defects. The study materials as well as the Hypothesizer source code are publicly available [1].

5.1 Method

We aimed to select prevalent defects in web applications that pose challenges for developers. Modern web apps incorporate extensive interactive features, encompassing button clicks, keyboard inputs, mouse events, and server communications. We chose ten defects representing a broad spectrum of evidence types, adapted from Stack Overflow questions. Each defect was introduced into two open source web applications: a To-Do app and one of six additional applications, including applications for Movie Search, Tetris, Excalidraw, an Interactive Timeline, a Simple Survey, or Online Pizza Order.

We assembled a dataset of ten hypotheses through a five-step process. First, we began by investigating popular posts on Stack

Table 3: The result of the technical evaluation of Hypothesizer across multiple defects and web applications. For each defect (H1 to H10), the table lists the program name, time taken to demonstrate the defect, size of the program behavior recording (expressed in files, LOC, events, and API calls, with files and LOC presented as a percentage of the total), time taken to find relevant hypotheses, and the most likely and less likely hypotheses identified. The final row displays the mean values of these metrics.

			Size of Program Behavior Recording (% of total)				Relevant Hypotheses		
ID	Program Name	Time (s)	Files	LOC	Events	API Calls	Time (s)	Most Likely	Less Likely
*H*1	To-Do	5	18 (72%)	487 (71%)	37	5263	9	*H*1	*H*2, *H*7
	Movies Search	6	5 (50%)	193 (62%)	109	5974	10	*H*1	*H*7
*H*2	To-Do	6	4 (16%)	121 (17%)	29	4934	8	*H*2	*H*7
	Movies Search	6	4 (40%)	158 (51%)	37	4743	8	*H*2	*H*7
*H*3	To-Do	3	3 (12%)	103 (14%)	12	648	6	*H*3	*H*5
	Tetris Game	3	1 (4%)	166 (4%)	10	598	6	*H*3	*H*5
*H*4	To-Do	4	15 (60%)	495 (65%)	16	1155	7	*H*4	*H*5, *H*7, *H*6
	Movies Search	4	5 (50%)	208 (63%)	72	1577	8	*H*4	*H*5, *H*7, *H*6
*H*5	To-Do	3	2 (8%)	61 (9%)	7	728	6	*H*5	*H*3
	Excalidraw	3	34 (8%)	16060 (12%)	9	1001	9	*H*5	*H*6
*H*6	To-Do	7	17 (68%)	483 (69%)	41	5965	9	*H*6	*H*5, *H*7
	Movies Search	7	5 (50%)	197 (62%)	102	6109	9	*H*6	*H*5, *H*7
*H*7	To-Do	4	17 (68%)	498 (68%)	45	5028	9	*H*7	-
	Interactive TimeLine	5	1 (20%)	88 (56%)	30	2323	8	*H*7	*H*3, *H*5
*H*8	To-Do	3	15 (60%)	387 (55%)	12	1160	7	*H*8	*H*7
	Simple Survey	3	1 (50%)	102 (94%)	12	528	6	*H*8	*H*3, *H*5, *H*6
*H*9	To-Do	3	5 (20%)	240 (33%)	12	805	6	*H*9	*H*3, *H*5
	Online Pizza Order	3	2 (4%)	106 (1%)	11	708	6	*H*9	*H*3, *H*4, *H*5, *H*7
*H*10	To-Do	3	5 (20%)	243 (33%)	12	714	6	*H*10	*H*3, *H*5
	Online Pizza Order	3	1 (2%)	74 (1%)	11	1421	8	*H*10	*H*3, *H*4, *H*5
	Mean	4	8 (34%)	1024 (42%)	31	2569	8		

Overflow. We aimed to understand common issues that web developers frequently encounter. From our investigation, we selected posts that we believed to represent ten common defects, paying particular attention to the symptoms and causes discussed within these posts. Second, we locally reproduced these defects. Third, we utilized Hypothesizer's recording functionality to capture program behavior associated with each defect and extract relevant evidence items. We then constructed a hypothesis for each defect. The complete dataset includes 38 conditions across ten hypotheses. Finally, we validated the hypotheses through peer review with experienced developers. A summary of the hypotheses dataset can be found in Table 2.

We instrumented Hypothesizer to collect additional information for the technical evaluation. Specifically, we logged the time to record the defect demonstration steps and find relevant hypotheses. Additionally, we collected information regarding the program behavior recorded, including the number of files, lines of code, and API calls. We conducted our technical evaluation on a MacBook Pro with an Apple M1 Pro processor with 32GB of RAM running macOS 13.3.

5.2 Results

Across all ten defects in each of the two applications we examined, Hypothesizer was able to correctly identify the correct hypothesis as the single most likely hypothesis. On average, Hypothesizer identified two less likely hypotheses for each defect, providing other possible explanations for developers to investigate. Table 3 summarizes the results.

Hypothesizer's recording feature efficiently captured program behavior in brief recordings, lasting just a few seconds. On average, each recording comprised 31 events and 2,569 API calls. Hypothesizer extracted a smaller portion of the code, averaging at less than half of the codebase. As the program size increased, Hypothesizer demonstrated greater precision. In one of the largest programs in the study (Online Pizza Order) with 14,000 lines of code, Hypothesizer managed to narrow down the collected lines of code to a mere 74 lines. On average, Hypothesizer's analysis to find relevant hypotheses took 8 seconds.

6 EVALUATION

We conducted a controlled experiment with 16 professional developers to assess the influence of Hypothesizer on the debugging process. The primary aim of this evaluation was to investigate the system's efficacy in supporting debugging tasks rather than examine the breadth of its applicability. In particular, we focused on the case where there exists a hypothesis which explains the defect, in addition to other unrelated hypotheses, and did not examine the case where there is no such hypothesis.

Table 4: Participant background, including years of professional and web development experience.

ID	Occupation	Pro. Exp.	Web Dev. Exp.
D1	Software Engineer	2	2
D2	Software Engineer	1.5	0.25
D3	Software Developer	2	1.2
D4	Software Engineer	2	1
D5	Data Scientist	2	2
D6	Mobile Application Engineer	3	3
D7	Graduate Student (MS)	3	3
D8	Graduate Student (PhD)	3	3
D9	Graduate Student (PhD)	1	10
D10	Front-end Developer	7	4
D11	Software Engineer	2	2
D12	Software Engineer	5	3
D13	Software Engineer	1.5	1.5
D14	Software Engineer	2	3
D15	Web Developer	1	1
D16	Software Engineer	1	2
	Mean	2.5	2.5

6.1 Method

The experiment was conducted online through a video conference, and participants were given access to the experimenter's computer where Hypothesizer was installed. Participants were asked to debug two defects in open source web applications. Participants were randomly asked to use Hypothesizer as the only debugging tool on one debugging task. For the other task, participants were free to use any debugging tools or web resources, including Stack Overflow.

6.1.1 Participants. After obtaining IRB approval, we started recruiting participants on social media platforms such as Twitter, Slack, mailing lists, and personal contacts. We utilized the snowball sampling technique to recruit additional participants through the existing participants' contacts. We included participants with at least one year of professional experience and who were familiar with web development. A total of 16 professional developers were recruited, with an average of 2.5 years of professional experience in the industry and web application development. Table 4 summarizes participants' occupations and years of experience.

6.1.2 Tasks. The goal of the debugging tasks in this study was to fix the defects successfully. Participants were given 30 minutes for each task, and the task ended when they demonstrated that the defect was fixed and the program functioned correctly. Participants in the control group were allowed to use any debugging tools, including the debugger, DOM viewer, logging, network analyzer, and any online resources. Half of the participants were assigned to debug the first task using Hypothesizer (D4, D5, D6, D8, D10, D11, D14, D16), while the other half were asked to use Hypothesizer for the second task (D1, D2, D3, D7, D9, D12, D13, D15).

Both tasks were based on a defect from the technical evaluation. In the first task, participants were asked to fix a defect that caused a movie search app not to render the search results. In the second task, participants were asked to fix a defect in a Tetris game that prevented the game from correctly responding to keyboard input to move and rotate blocks. The correct hypotheses for these defects are listed in Table 2 as $H2$ and $H3$.

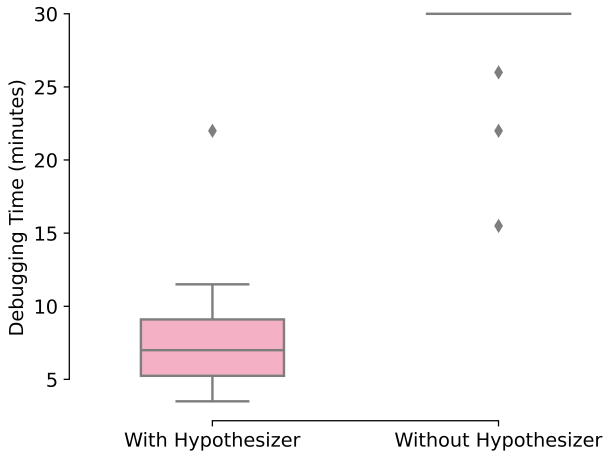

Figure 7: Time spent debugging for participants with and without Hypothesizer. The maximum debugging time per task is 30 minutes.

Table 5: Debugging work with and without Hypothesizer.

	Control	Experimental
Tools Used	Logs, Breakpoints, DOM inspector, Network inspector, React devtools, Online resources	Hypothesizer
% Success	19%	100%
Avg. Time to Fix	21 minutes	8 minutes
Avg. Files Viewed	4	1
Avg. Program Reruns	11	3

6.1.3 Post-task interview. Following each debugging task, the first author conducted informal, open-ended interviews with the participants. These interviews aimed to gain insights into their experience with the debugging process, including the most challenging aspects of the task and any tools or techniques that were particularly helpful. Once participants had completed both tasks, we asked them to reflect on the differences they noticed in their debugging process when using Hypothesizer compared to debugging without it.

6.2 Results

Participants employing Hypothesizer managed to fix defects within an average debugging time of eight minutes. In contrast, only three participants (D6, D7, and D12) were successful without Hypothesizer, taking an average of 21 minutes to fix the defects. These results highlight the substantial impact of Hypothesizer when compared to existing tools such as breakpoint debuggers and Stack Overflow (two-sided Wilcoxon signed-rank test, p-value < 0.001). Figure 7 and table 5 summarize the differences in debugging between the two groups.

6.2.1 Hypothesizer enabled developers to more efficiently identify relevant hypotheses with less effort. Participants without Hypothesizer spent more time examining the program behavior. On average, they navigated four times the number of code files and reproduced the defect four times more often. These participants relied on a broad range of tools, such as console logs (D1-D16), breakpoints (D1, D4, D6, D7, D10), the DOM inspector (D5, D10, D13), the network inspector (D1, D3, D9, D12-D15), and the React devtools (D1, D4, D9, D11). All, except for D1 and D3, searched the web for documentation, Stack Overflow posts, and code snippets.

One of the main difficulties that participants encountered was that the defects did not generate an explicit error message, which made the defect behavior "hidden". Participant D2 expressed, "When working without Hypothesizer, the issue remained hidden. The program did not provide any feedback or error messages, and although Stack Overflow was somewhat helpful, I felt like something was missing that prevented me from making progress. I appreciate how Hypothesizer addresses this by requiring me to replicate the bug."

A shared sentiment among the participants was that Hypothesizer streamlined the process of observing the defect behavior. D5 commented, "Hypothesizer summarizes what happened, so I didn't need to set up breakpoints. All I had to do was reproduce the bug." D4 mentioned that Hypothesizer required less effort than usual to observe the defect and come up with a hypothesis, stating, "I didn't have to think deeply into the problem or create a description of the defect. I simply recorded my actions, and the tool handled the rest. Describing the defect behavior is often a problem for me, but Hypothesizer eliminated that requirement."

D1 and D3, in particular, did not seek assistance from online resources such as Stack Overflow to search for relevant hypotheses. They thought that the defect was "specific" to their program, making it impossible to find any helpful information on the internet. D3 explained, "I did not use Stack Overflow because the defect was linked to many specific implementations you cannot express to Google or Stack Overflow." Nonetheless, both defects encountered in our study were typical in web applications.

6.2.2 The Timeline View aided developers in testing hypotheses, despite limited knowledge of the codebase. Participants found the Hypothesizer timeline view valuable in quickly focusing on related program behavior, replacing the traditional method of exploring the codebase while debugging. D5 reported, "The way I often debug is by setting up breakpoints and stepping through the program. Hypothesizer did not require that as it presents a timeline and made me check different things quickly." D9 reported that "having the timeline right here and getting the information immediately" helped them test the relevant hypotheses faster.

Participants appreciated the new experience Hypothesizer offered in supporting testing hypotheses. D7 noted that "using Stack Overflow was like guessing my way through it," In contrast, Hypothesizer offered "a complete thesis, explaining what was causing the problem and why," which helped them test relevant hypotheses more quickly. D8 and D12 explained how Hypothesizer made debugging more interactive, with D8 stating, "The step-by-step process is very interactive and helpful for me. Without Hypothesizer, I would first have to understand the codebase and inspect other stuff, and I might end up looking in places with no issues." D12 reported that they would typically test many incorrect hypotheses first, but with Hypothesizer they were able to skip this guessing process.

Participants (D5, D6, D8-D10, and D12) found that Hypothesizer not only guided them toward the relevant hypotheses but also helped dismiss initial, incorrect ones they might have considered without its support. D10 highlighted that sorting relevant hypotheses by evidence derived from defect behavior allowed them to "focus on the more likely first, which avoided wasting time testing into other hypotheses." D6 appreciated how Hypothesizer visualized relevant evidence items in the program behavior, enabling them to quickly discard initial incorrect hypotheses: "I like how Hypothesizer showed me that the event handler and networking worked as expected in the timeline. In the real world, these are the first things I would check if I face a similar bug, but manually."

6.2.3 The step-by-step instructions facilitated fixing the defect contributed to a deeper understanding of the defect. All participants were successful in fixing the defect while using Hypothesizer. They utilized the step-by-step "how to fix?" instructions to understand and fix the defect. Participants reported that Hypothesizer helped them learn while debugging, improving their ability to fix future defects. D8 said, "I will be able to fix future similar defects more effectively because Hypothesizer helped me learn how to look at different evidence."

Participants D11 and D13 observed that Hypothesizer was similar to working with an experienced developer who knows the codebase, facilitating a learning experience similar to pair programming activities. D11 commented, "Hypothesizer would really help in working on a codebase that belongs to a teammate. It will make it easier for me to iterate faster with a limited program understanding without physically consulting that expert teammate." D13 added that "Hypothesizer seems to work as we work with another more expert developer. You need to show it the defect by recording the reproduction step, and then it starts thinking. Then it offers a list of hypotheses and a plan to fix the defect. This is much different than Stack Overflow, which requires you to give detailed questions."

7 DISCUSSION AND FUTURE WORK

Effectively formulating and testing hypotheses has long been found to be crucial to successful debugging. However, current debugging tools lack direct support for this activity. In this paper, we introduce the concept of hypothesis-based debuggers, which collaboratively work with developers throughout the debugging process from identifying relevant hypotheses to fixing the defect. We found that a hypothesis-based debugger can substantially increase success fixing defects by a factor of five and reduce debugging time by a factor of three compared to traditional breakpoints debuggers and Stack Overflow. These promising results indicate the potential to dramatically decrease the time developers devote to debugging. Further work is necessary to further scale hypothesis-based debugging as well as apply it outside the context of web applications.

To effectively use hypothesis-based debuggers in the real-world, it is essential to facilitate the creation of an extensive database of hypotheses. Tools should aid developers in the process of identifying the conditions necessary to test a hypothesis. While Stack Overflow

provides a valuable resource for developers to learn from the experiences of others, one challenge in using it as a starting point for populating datasets for hypothesis-based debuggers is that posts are often specific to a particular problem or context. Hypotheses, on the other hand, need to be more general. Future research could explore methods to curate and cluster similar Stack Overflow posts into generalized hypotheses. Large language models such as ChatGPT by OpenAI [31] might aid this process by recognizing patterns and generating initial hypotheses. However, due to the potential for error in these techniques, it is likely to remain important to have a human in the loop to review and improve generated hypotheses.

Hypothesis-based debuggers might also fit into new or existing communities such as StackOverflow. Developers might post questions, and answers might include a hypothesis, helping generalize the issue and making it much easier for developers to find this content.

While the current prototype was implemented as a standalone debugging tool, it included clickable links to view code snippets within the developer's IDE. However, participants suggested that deeper integration with the IDE would be helpful to reduce context-switching between tools. During our study, we observed that some participants made typos while moving back and forth between Hypothesizer and the IDE, which hindered their ability to test hypotheses effectively.

Several participants suggested that Hypothesizer might be a valuable tool for novice developers learning to debug and discover defects, particularly when using new APIs. Instructors might use hypothesis-based debuggers by populating the database of hypotheses with those related to defects that students frequently make, such as in tools like HelpMeOut [14]. Integration between a hypothesis-based debugger into the learning process might give further visibility into the common challenges their students face, where instructors might use log data to identify areas of concern where further resources are needed.

Hypothesis-based debuggers might also support the process of onboarding developers onto new software projects. When developers join a new project or start working on an unfamiliar codebase, they often face challenges in understanding why common defects occur. Hypothesis-based debuggers might ease this learning curve by providing relevant hypotheses for common defects within the project.

More broadly, hypothesis-based debuggers might facilitate collaboration within software teams. Team members might identify common challenges and offer solutions as debugging hypotheses, working to collectively improve their team's debugging skills.

ACKNOWLEDGMENTS

We would like to express our gratitude to Saigautam Bonam, Henry Zheng, and Madhav Shroff for their contributions to the early prototype of Hypothesizer. We are also deeply thankful to Anas alhumud and Sajed Jalil for their invaluable assistance in recruiting participants for our study. This research was made possible through the support of the National Science Foundation under Grant No. 1845508, and the scholarship from King Saud University.

REFERENCES

[1] Alaboudi. 2023. Replication package. https://archive.org/details/replication_Hypothesizer_UIST
[2] Abdulaziz Alaboudi and Thomas D LaToza. 2020. Using hypotheses as a debugging aid. In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Dunedin, New Zealand, 1–9.
[3] Abdulaziz Alaboudi and Thomas D. LaToza. 2021. Edit-Run Behavior in Programming and Debugging. In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Los Alamitos, CA, USA, 1–10.
[4] Abdulaziz Alaboudi and Thomas D LaToza. 2023. What Constitutes Debugging? An Exploratory Study of Debugging Episodes". *Empirical Software Engineering (EMSE)* Forthcoming, Forthcoming (2023), To be published.
[5] Maryam Arab, Thomas D. LaToza, Jenny Liang, and Amy J. Ko. 2022. An exploratory study of sharing strategic programming knowledge. In *Conference on Human Factors in Computing Systems (CHI)*. ACM, New Orleans, LA, USA, 15 pages.
[6] Maryam Arab, Jenny Liang, Yang Kyu Yoo, Amy J. Ko, and Thomas D. LaToza. 2021. HowToo: a platform for sharing, finding, and using programming strategies. In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, St Louis, MO, USA, 1–9.
[7] Muhammad Asaduzzaman, Ahmed Shah Mashiyat, Chanchal K. Roy, and Kevin A. Schneider. 2013. Answering questions about unanswered questions of Stack Overflow. In *International Conference on Mining Software Repositories (MSR)*. ACM, Francisco, CA, USA, 97–100.
[8] Marcel Böhme, Ezekiel O. Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. 2017. Where is the bug and how is it fixed? an experiment with practitioners. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, Paderborn, Germany, 117–128.
[9] Ruven Brooks. 1983. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies* 18, 6 (1983), 543–554.
[10] Brian Burg, Richard Bailey, Amy J Ko, and Michael D Ernst. 2013. Interactive record/replay for web application debugging. In *Symposium on User Interface Software and Technology (UIST)*. ACM, St. Andrews, Scotland, United Kingdom, 473–484.
[11] Chromedevtools. 2022. Chrome DevTools Protocol. https://chromedevtools.github.io/devtools-protocol/
[12] Richard A. DeMillo, Hsin Pan, Eugene H. Spafford, Richard A. DeMillo, Hsin Pan, and Eugene H. Spafford. 1996. Critical slicing for software fault localization. In *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, San Diego, California, USA, 121–134.
[13] L. Gugerty and G. Olson. 1986. Debugging by Skilled and Novice Programmers. In *Conference on Human Factors in Computing Systems*. ACM, Boston, Massachusetts, USA, 171–174.
[14] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R Klemmer. 2010. What would other programmers do: suggesting solutions to error messages. In *Conference on Human Factors in Computing Systems (CHI)*. ACM, Atlanta, GA, USA, 1019–1028.
[15] Morten Hertzum and Annelise Mark Pejtersen. 2000. Information-seeking practices of engineers: Searching for documents as well as for people. *Information Processing and Management* 36, 5 (Sep 2000), 761–778.
[16] Robin Jeffries. 1982. A comparison of the debugging behavior of expert and novice programmers. *Proceedings of AERA annual meeting* 10, 5 (1982), 1–7.
[17] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *International Conference on Software Engineering (ICSE)*. ACM/IEEE, San Francisco, CA, USA, 672–681.
[18] Amy J Ko. 2006. Debugging by asking questions about program output. In *International Conference on Software Engineering (ICSE)*. ACM/IEEE, Shanghai, China, 989–992.
[19] Amy J Ko, Robert DeLine, and Gina Venolia. 2007. Information needs in collocated software development teams. In *International Conference on Software Engineering (ICSE)*. ACM/IEEE, Minneapolis, MN, USA, 344–353.
[20] Amy J Ko and Brad A Myers. 2004. Designing the whyline: a debugging interface for asking questions about program behavior. In *Conference on Human Factors in Computing Systems (CHI)*. ACM, Vienna, Austria, 151–158.
[21] Amy J Ko and Brad A Myers. 2008. Debugging reinvented: asking and answering why and why not questions about program behavior. In *International Conference on Software Engineering (ICSE)*. ACM/IEEE, Leipzig, Germany, 301–310.
[22] Amy J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. 2006. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on software engineering (TSE)* 32, 12 (2006), 971–987.
[23] Thomas D LaToza and Brad A Myers. 2011. Visualizing call graphs. In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Pittsburgh, USA, 117–124.

[24] Thomas D. LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining Mental Models: A Study of Developer Work Habits. In *International Conference on Software Engineering (ICSE)*. ACM/IEEE, Shanghai, China, 492–501.
[25] Joseph Lawrance, Christopher Bogart, Margaret Burnett, Rachel Bellamy, Kyle Rector, and Scott D. Fleming. 2013. How programmers debug, revisited: An information foraging theory perspective. *IEEE Transactions on Software Engineering (TSE)* 39, 2 (2013), 197–215.
[26] L. Layman, M. Diep, M. Nagappan, J. Singer, R. Deline, and G. Venolia. 2013. Debugging Revisited: Toward Understanding the Debugging Needs of Contemporary Software Developers. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM/IEEE, Baltimore, Maryland, USA, 383–392.
[27] Stanley Letovsky. 1987. Cognitive processes in program comprehension. *Journal of Systems and Software (JSS)* 7, 4 (1987), 325–339.
[28] Bil Lewis. 2003. Debugging backwards in time. arXiv preprint cs/0310016.
[29] David C. Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. 1987. Mental models and software maintenance. *Journal of Systems and Software (JSS)* 7, 4 (1987), 341–355.
[30] Lena Mamykina, Bella Manoim, Manas Mittal, George Hripcsak, and Björn Hartmann. 2011. Design Lessons from the Fastest Q&A Site in the West. In *Conference on Human Factors in Computing Systems (CHI)*. ACM, Vancouver, BC, Canada, 2857–2866.
[31] OpenAI. 2023. OpenAI API. https://openai.com/ Accessed: March 2023.
[32] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, Toronto, Ontario, Canada, 199–209.
[33] Michael Perscheid, Benjamin Siegmund, Marcel Taeumel, and Robert Hirschfeld. 2017. Studying the advancement in debugging practice of professional software developers. *Software Quality Journal* 25, 1 (2017), 83–110.
[34] Replay. 2022. replay. https://www.replay.io/
[35] Semgrep. 2023. Semgrep: A Code Analysis Tool. https://semgrep.dev/ Accessed: March 2023.
[36] Stackoverflow. 2022. reactBugs. https://stackoverflow.com/search?q=clicking+a+button+does+not+work+%5Breact%5D Accessed: 2023-03-06.
[37] Christoph Treude, Ohad Barzilay, and Margaret-Anne Storey. 2011. How do programmers ask and answer questions on the web?. In *International Conference on Software Engineering (ICSE)*. ACM/IEEE, Waikiki, Honolulu, HI, USA, 804–807.
[38] Bogdan Vasilescu, Alexander Serebrenik, Prem Devanbu, and Vladimir Filkov. 2014. How social Q&A sites are changing knowledge sharing in open source software communities. In *Conference on Computer Supported Cooperative Work (CSCW)*. ACM, Baltimore, Maryland, USA, 342–354.
[39] A. Von Mayrhauser and A.M. Vans. 1996. On the role of hypotheses during opportunistic understanding while porting large scale code. In *International Workshop on Program Comprehension (ICPC)*. IEEE, Berlin, Germany, 68–77.
[40] Mark Weiser. 1984. Program slicing. In *International Conference on Software Engineering (ICSE)*. ACM/IEEE, Washington, D.C., USA, 439–449.
[41] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering (TSE)* 42, 8 (2016), 707–740.
[42] X. Xia, L. Bao, D. Lo, and S. Li. 2016. "Automated Debugging Considered Harmful" Considered Harmful: A User Study Revisiting the Usefulness of Spectra-Based Fault Localization Techniques with Professionals Using Real Bugs from Large Systems. In *International Conference on Software Maintenance and Evolution (ICSME*. IEEE, Raleigh, North Carolina, USA., 267–278.
[43] Xiangyu Zhang, R. Gupta, and Youtao Zhang. 2003. Precise dynamic slicing algorithms. In *International Conference on Software Engineering (ICSE)*. ACM/IEEE, Portland, Oregon, USA, 319–329.
[44] Jie Yang, Claudia Hauff, Alessandro Bozzon, and Geert-Jan Houben. 2014. Asking the Right Question in Collaborative Q&a Systems. In *Proceedings of the 25th ACM Conference on Hypertext and Social Media*. ACM, Santiago, Chile, 179–189.
[45] Andreas. Zeller. 2005. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, Burlington, MA, USA. 480 pages.
[46] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006. Pruning dynamic slices with confidence. In *Conference on Programming Language Design and Implementation (PLDI)*. ACM, Ottawa, Ontario, Canada, 169–180.

A APPENDIX

A complete example of a hypothesis (*H1*).


```
"hypotheses": [{
"id": "HYPOTHESIS_1",
"hypothesis": "You are not using 'preventDefault'
    API to prevent the default behavior of the
    submit button.",
"description": "When you use the submit button
    inside a form, the default behavior is to send
     the form data to the server and load a new
    page. However, this may not be the desired
    behavior, especially in React apps where you
    want to partially update the page.",
"tags": [
"incorrect add/remove item of a list of items",
"unexpected reload of the page"
],
"evidence": [
{
"id": "EVENT_2",
"pattern":
{
"objectShape": {
"InputType": "text",
"type": "keydown"
}
"description": "The first thing you did was typing
     in the input box.",
"isFound": true,
"DoesContainTheDefect": false
},
{
"rule": "EVENT_1",
"patterns": [
"objectShape": {
"target": "BUTTON",
"type": "click",
"InputType": "submit",
}],
"description": "You clicked on the submit button,
    which triggered a submit event.",
"isFound": true,
"DoesContainTheDefect": false
},
{
"rule": "API_11",
"patterns": [
{
"codePattern": "const $Y = (...) => {...}"
},
{
"codePattern-inside": "const $X = (...) => {...
    return (<form onSubmit={$Y}>...</form>);...}"
},
{
"codePattern-not": "const $Y = ($T) => {... $T.
    preventDefault(); ...}"
}],
"description": "The submit event was handled by a
    callback triggered after the submit event.",
"isFound": true,
"DoesContainTheDefect": false
},
{
```

```
"rule": "API_1",
"patterns": [
{
"functionName": "preventDefault",
},
{
"codePattern": "$E.preventDefault()"
}
],
"description": "However, you did not use the
    preventDefault API to prevent the default
    behavior of the submit button. This often
    happen inside onSubmit event handler inside
    the form.",
"isFound": false,
"DoesContainTheDefect": true,
"HowToFix": {
"steps": [
{
"description": "Start by searching for the
    onSubmit event handler inside the form inside
    this file.",
"relatedEvidenceLocation": {
"rule": "API_11",
"exactLocation": true
}},
{
"description": "Inside the onSubmit event handler,
     add the preventDefault API to prevent the
    default behavior of the submit button.",
"codeExample": "const XXXXX = (e)=>{\n e.
    preventDefault() // <-- use this API inside
    the onSubmit callback\n}"
},
{
"description": "Now, when you click on the submit
    button, the page will not be reloaded."
}],}},
{
"rule": "NETWORK_1",
"patterns": [
"objectShape": {
"type": "responseReceived",
"mimeType": "text/html"
},
"id": "NETWORK_1",
]
"description": "The browser reload since the
    submit behaior was not prevented.",
"isFound": true,
"DoesContainTheDefect": false
}]]
```