UC San Diego
UC San Diego Electronic Theses and Dissertations

Title
Detecting Bugs and Security Issues by Identifying Developers' Blind Spots

Permalink
https://escholarship.org/uc/item/4dc059n1

Author
Zhong, Li

Publication Date
2024

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA SAN DIEGO

Detecting Bugs and Security Issues by Identifying Developers' Blind Spots

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy

in

Computer Science

by

Li Zhong

Committee in charge:

Professor Yuanyuan Zhou, Chair
Professor Amy Ousterhout
Professor Aaron Schulman
Professor Geoff Voelker
Professor Xinyu Zhang

2024

Copyright

Li Zhong, 2024

All rights reserved.

The Dissertation of Li Zhong is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2024

TABLE OF CONTENTS

Dissertation Approval Page iii

Table of Contents iv

List of Figures vi

List of Tables viii

Acknowledgements ix

Vita xi

Abstract of the Dissertation xii

Chapter 1 Introduction 1
- 1.1 Motivation 1
- 1.2 Contribution 4
- 1.3 VALUECHECK: Effective Bug Detection with Unused Definitions 5
- 1.4 FENCEHOPPER: Detect Vulnerabilities by Client-Side Code Mutation 6

Chapter 2 VALUECHECK: Effective Bug Detection with Unused Definitions 7
- 2.1 Motivation 7
- 2.2 Background 11
  - 2.2.1 Liveness Analysis 11
  - 2.2.2 Unused Definitions 12
- 2.3 Design Overview 12
  - 2.3.1 Detection Scope 12
  - 2.3.2 Framework Overview 14
- 2.4 Detecting Cross-Scope Unused Definitions 15
  - 2.4.1 Detect Local Unused Definitions 15
  - 2.4.2 Authorship Lookup 17
- 2.5 Pruning 20
  - 2.5.1 Configuration Dependency 20
  - 2.5.2 Cursor 20
  - 2.5.3 Unused Hints 21
  - 2.5.4 Peer Definition Pruning 21
- 2.6 Ranking based on Code Familiarity 22
- 2.7 Implementation 23
- 2.8 Evaluation 24
  - 2.8.1 Experiment Setup 24
  - 2.8.2 Detect New Bugs 25
  - 2.8.3 Accuracy of VALUECHECK 28
  - 2.8.4 Comparison with Existing Tools 30

2.8.5 Authorship and Code Familiarity Effectiveness 34
2.8.6 Scalability of VALUECHECK 36
2.9 Limitations and Discussion 36
2.9.1 Limitation 36
2.9.2 Alternatives of the DOK model 37
2.10 Related Work 37
2.11 Acknowledgments 38

Chapter 3 FENCEHOPPER: Detect Vulnerabilities by Client-Side Code Mutation 39
3.1 Introduction 39
3.1.1 Problem: Over-relying on Security Checks in Web Client 39
3.1.2 Existing Solutions 41
3.1.3 Our Approach 42
3.2 Client-side Code Mutation 44
3.2.1 Identify Security Checks in Client-Side 45
3.2.2 Mutate Security Checks in Client-Side 48
3.2.3 Generate Data Input Based on Checks 50
3.2.4 Fix Missing Data after Code Mutation 52
3.3 Implementation 53
3.4 Evaluation 54
3.4.1 Detect New Vulnerabilities 54
3.4.2 Comparison to Other Tools 56
3.4.3 Detailed Evaluation of Each Component 58
3.4.4 Coverage of FENCEHOPPER 60
3.4.5 Manual Effort and Time 62
3.5 Security Impact Analysis 62
3.5.1 Systematic Analysis 62
3.5.2 Case Study 64
3.6 Limitations and Discussion 65
3.7 Related Work 66
3.8 Case Study: Blind Spots in Third-Party Services 68
3.9 Acknowledgments 69

Chapter 4 Conclusion 70
4.1 Summary of this Dissertation 70
4.2 Lessons Learned 71

Bibliography 73

LIST OF FIGURES

Figure 2.1. **Two real-world bugs underlying unused definitions detected by VALUECHECK** with severe consequences like security issues and configuration bugs. Both bugs cross the author scope. 8

Figure 2.2. **Overview of VALUECHECK.** VALUECHECK consists of cross-scope unused definition detection, false positive pruning, and familiarity ranking. 14

Figure 2.3. **An Example of Define Set.** The first definition of v is overwritten by other developers on all the successor paths. 17

Figure 2.4. **Cross-Scope Unused Definition Detection.** 18

Figure 2.5. **Example of Cursors.** 21

Figure 2.6. **Examples of New Bugs Detected by VALUECHECK.** 27

Figure 2.7. **Bug Categorized by Component Distribution, Security Severity and Days before Detected.** All evaluated software undergo thorough testing. Despite this, VALUECHECK uncovers high-severity bugs in critical components that have previously gone undetected for a long time. 28

Figure 2.8. **A bug detected by VALUECHECK but not detected by fb-infer, Smatch-unused and Coverity-unused.** The developer forgot to handle the return error status from get_permset, which would cause access control error if the acl entry is invalid. 34

Figure 2.9. **Precision of bug detection with different cutoffs after familarity ranking.** VALUECHECK has a precision of 97.5% when reporting the top 10 detected unused definitions with the lowest familiarity from each applications. 35

Figure 3.1. **Overview of FENCEHOPPER.** FENCEHOPPER mutates the client-side code to bypass client-side checks to detect if such checks are missed by servers—a common overlook by developers due to false sense of security from client-side checks. 42

Figure 3.2. **Overall Workflow of FENCEHOPPER.** 44

Figure 3.3. **Identify and Mutate Security Checks.** 46

Figure 3.4. **Identify Intra and Inter-Procedural Security Checks.** Functions in heavy solid line boxes are extracted directly from the stack traces, which are the ‘stem’ of our analysis. We expand the relevant function set by checking the callees of these functions, which are in the dotted line boxes. 47

Figure 3.5. **Vulnerability Categories.** 55

Figure 3.6. **FENCEHOPPER Supplements Existing Tools.** It complements the vulnerability category of 'missing server-side checks'. The bugs detected by FENCEHOPPER are non-trivial and orthogonal to other vulnerabilities in this category such as cross-site scripting (XSS) and SQL injection. 57

Figure 3.7. **Impact of Threshold Changes on Bug Detection and False Positive Rates.** 60

Figure 3.8. **Email Verification Code Bypass.** The verification code sent to email can be bypassed by mutating client-side code and fill in attacker's email address in missing data objects. Name and UI of the victim website is anonymized. 64

LIST OF TABLES

Table 2.1. **Meaning of notations in the algorithms (Figure 2.4).** 16

Table 2.2. **Summary of pruning patterns in VALUECHECK.** 20

Table 2.3. **The number of bugs newly detected by VALUECHECK.** Among the 210 bugs detected, 154 bugs are confirmed by developers. 25

Table 2.4. **Bug examples detected by VALUECHECK**. Generally, VALUECHECK detects two categories of bugs: missing check bugs and semantic bugs. Of 154 bugs confirmed, 134 are missing check bugs, 20 are semantic bugs. . . . 26

Table 2.5. **Pruning breakdown and sampled false negative rate in VALUECHECK.** The false negative rate of pruning is less than 10% based on sampling with 95% confidence. 28

Table 2.6. **Unused Definition Bugs Detected by Clang, Infer, Smatch, Coverity and VALUECHECK.** VALUECHECK in total detects more bugs with lower false positives than other tools. *Report errors during analysis. 33

Table 2.7. **Effect of authorship and the DOK model in VALUECHECK.** VALUECHECK detects a higher total number of bugs compared to other groups. 34

Table 2.8. **Scalability of VALUECHECK.** 36

Table 3.1. Missing server-side check vulnerabilities affect billions of users in popular websites. 40

Table 3.2. **Branches on the Client Side of Popular Websites.** 44

Table 3.3. **Comparison to Other Tools.** FENCEHOPPER detects vulnerabilities that other state-of-the-art tools fail to detect. 57

Table 3.4. **Effectiveness of Missing Data Fix in FENCEHOPPER.** Missing data fix enables detection of 9 additional vulnerabilities. 59

Table 3.5. **14 existing missing server-side check vulnerabilities.** FENCEHOPPER detects 10 but misses 4 vulnerabilities (in gray cells). 61

ACKNOWLEDGEMENTS

First, I express my sincere gratitude to my advisor, Professor Yuanyuan (YY) Zhou, for her support throughout my whole PhD study and research. It is not possible for me to complete this hard journey without YY's insightful guidance and inspiring suggestions. YY showed me the real essence of science and always pushed me to think deeper, try bravely, and pursue what I truly believed in. The guidance is not only in research but also in life. YY also taught me great lessons to confront the hardships in life and always be tough and resilient. Her spirit encourages me to continue in this field as a minority and keep fighting. She is my role model and will always be! Thank you, YY! It is really my luck to become your student and get the chance to work with you!

I would like to thank Professor Geoffrey M. Voelker. Geoff organizes SysLunch and CSE294, which brings me a window to the most state-of-the-art system research. Every time I turn to him for questions, he always gives generous help and suggestions! The SysNet hiking and end-of-year celebration organized by Geoff are among my most interesting memories in UCSD SysNet.

I would also like to thank my thesis committee members, Professor Aaron Schulman, Professor Amy Ousterhout, Professor Xinyu Zhang, and Professor Geoffrey M. Voelker, for always being supportive. They give me a lot of help in scheduling each presentation and provide insightful feedback on this dissertation.

I was very lucky to have the chance to work with my academic brothers, Chengcheng Xiang, Bingyu Shen, Haochen Huang, and Eric Mugnier. They showed a great example to me of how to drive a research project and provided many hands-on experiences. They keep helping me to revise my research draft even after graduation. It is lucky to have you all on my journey.

I would like to thank Cindy Moore, who provided experimental data for my past projects. Her experience as a real sysadmin provides important insights into our research. Thank Bokai Zhang who helped me conduct the experiments in my second project with high quality.

I would also like to thank all the faculties, staff, and students in the SysNet group and

the whole CSE department. In this place, I learned the dedication to research and the inspiring attitude to life and work. Thanks to Junda Chen, Julie Conner, Tierra Terell, Tyler Potyondy, Vector Li, Shelby Thomas, Stewart Grant, Lixiang Ao, Nishant Bhaskar, Alex Liu, Zesen Zhang, Alisha Ukani, Han Zhao, Tianyi Shan, Mingyao Shen, Yudong Wu for their help during my stay in this department.

I learned a lot during my internships at Google working with Snehasish Kumar and Sotiris Apostolakis, at Meta working with Hongtao Yu, Wenlei He, and Lei Wang, and at Apple working with Cecile Foret and Zheng Li. They gave me the best guidance in industry projects and helped me gain valuable experience. This invaluable experience gave me more insight into what problems the industry focuses on. I would also thank my friends Xinyu Tang, Huiqi Ni, Yu Pei, Yuanhang Zhang, Wuyue Lu, Lianke Qin, and other people that help me along the road.

Last but not least, I must thank my family. My PhD study went through the unprecedented global pandemic and international travel restrictions. For many years, I became 'the mysterious family member in the US that nobody knows what she is doing'. But their remote care and love never stopped. I am also deeply thankful to my boyfriend, Zilong Wang, whose warm hugs and emotional support are always there whenever I need. I can never overstate my gratitude to him.

Chapter 2, in full, are reprints of the material as it appears in Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys'24). Zhong, Li; Xiang, Chengcheng; Huang, Haochen; Shen, Bingyu; Mugnier, Eric; Zhou, Yuanyuan. The dissertation author was the primary investigator and author of this paper. Chapter 3, in full, are reprints of the material under submission. Zhong, Li; Zhang, Bokai; Zhou, Yuanyuan. The dissertation author was the primary investigator and author of this paper.

VITA

2015-2019 Bachelor of Engineering, University of Science and Technology of China

2024 M.S. in Computer Science, University of California San Diego

2024 Ph.D. in Computer Science, University of California San Diego

PUBLICATIONS

"Effective Bug Detection with Unused Definitions". **Li Zhong**, Chengcheng Xiang, Haochen Huang, Bingyu Shen, Eric Mugnier, Yuanyuan Zhou. In Nineteenth European Conference on Computer Systems (EuroSys '24), April 22–25, 2024.

"Don't Trust Your Code! Detect Vulnerabilities by Client-Side Code Mutation in Web Services". **Li Zhong**, Bokai Zhang, Yuanyuan Zhou. Under submission.

"PYLIVE: On-the-Fly Code Change for Python-based Online Services". Haochen Huang*, Chengcheng Xiang* (co-first), **Li Zhong**, Yuanyuan Zhou. 2021 USENIX Annual Technical Conference (ATC'21), July 2021.

"Protecting Data Integrity of Web Applications with Database Constraints Inferred from Application Code". Haochen Huang, Bingyu Shen, **Li Zhong**, Yuanyuan Zhou. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23), March 25–29, 2023.

ABSTRACT OF THE DISSERTATION

Detecting Bugs and Security Issues by Identifying Developers' Blind Spots

by

Li Zhong

Doctor of Philosophy in Computer Science

University of California San Diego, 2024

Professor Yuanyuan Zhou, Chair

Bugs and security issues are primary concerns for software developers. Existing research has continuously focused on addressing these problems. However, the evolution of software engineering leads to increasingly complex software systems that are more susceptible to bugs. The rise of third-party services, cross-vendor libraries, and collaborative development introduces significant challenges for developers, making it difficult for them to have a comprehensive understanding of the entire codebase. Under the pressure of agile development timelines, developers often work with incomplete knowledge, leading to potential blind spots in software development. These blind spots can result in developers being unaware of certain constraints or security implications imposed by other components or authors, causing serious issues in access

control, memory management, I/O operations, and business logic.

This dissertation investigates two aspects of these challenges. The first aspect focuses on cross-authorship blind spots. This part of the study identifies a specific pattern of bug-proneness, namely cross-authorship unused definitions. To address this, we introduce syntactic and semantic patterns that help identify such issues while filtering out false positives. Additionally, to accommodate the time pressures faced by developers, we use a code familiarity model to prioritize bug validation. Our implementation, named VALUECHECK, has been evaluated on large-scale systems including Linux, MySQL, OpenSSL, and NFS-ganesha, successfully detecting 210 unknown bugs, with 154 confirmed. In comparisons with the state-of-the-art tools like Infer and Coverity, VALUECHECK demonstrates greater effectiveness and lower false positive rates.

The second part studies cross-component blind spots. It focuses on blind spots in web applications with a client-server architecture, where client-side code is exposed. Relying solely on client-side security checks for authorization, identity verification, and user input validation is insufficient due to potential user manipulation. To address this, we propose a novel technique that enhances existing methods by altering client-side code to assess server-side security. This approach improves testing efficiency and detects complex vulnerabilities related to business logic, token-based defenses, and data preprocessing. Our testing tool, FENCEHOPPER, identified 48 vulnerabilities in the top 300 websites from the Tranco dataset, including critical access control flaws affecting over 20 million user accounts.

Chapter 1

Introduction

1.1 Motivation

Bugs and security issues have posed significant challenges for developers since the invention of computer programs [32]. Researchers and practitioners have been actively addressing this issue through extensive research on bug detection methods using program analysis and testing. Also, industry companies contribute a range of bug detection tools [8, 4, 7, 158, 130] to eliminate this problem, as bugs and security issues could cause potential financial loss in production services. However, despite ongoing efforts, the battle against bugs and security issues persists and grows more complex with the development of software engineering.

Several factors contribute to this complexity. Firstly, the evolving landscape of software development practices, particularly the prevalence of collaborative development in open-source communities and industry settings [24], presents both benefits and challenges. While collaborative development enhances software delivery efficiency, it also introduces risks as developers may interact with code from other authors without full comprehension [38]. Additionally, reliance on third-party services offering abstract APIs without detailed implementation information further compounds the challenge [21].

Traditional software development practices, characterized by formal documentation, extensive communication, and pre-development discussions [22], have evolved with methodologies like agile development, where shorter project timelines and changing requirements prevail, lead-

ing to compressed documentation efforts [95]. High turnover rate becomes a norm in software industry. In a study conducted in 2015, they found that over half of the studied project have at least 30% turnover rate per year, even in popular open-source projects like Angular.JS [44]. According to the 2021 Bureau of Labor statistics report [5], the average turnover rate of a software developers is 57.3%. Under the high turnover rate, it is hard to guarantee the sharing of developers' knowledge due to large number of newcomers and leaving members [111].

Besides, due to the abundance of open source projects and the popularity of cloud computing and SaaS, developers are using many external APIs written by other developers or by third-parties [171]. Under time pressure, developers may not be able to fully understand the usage of external APIs, so they reuse code from online forums and tutorial examples, which however cause bugs in their programs without their awareness [178, 166]. As reported in 2021, 90% of businesses experienced API security vulnerabilities in 2020 [6].

Under these new practices, developers often encounter blind spots, areas within the software where their understanding is limited or incomplete, yet they must deliver code under development pressures [55]. For example, they may not be familiar with external APIs [135], or code and components written by other developers [114]. But still, due to the requirements of agile development, they need to take action and deliver code even with blind spots in their mind. If we resemble the process of software development as car driving, then the developers are drivers in each car. They control their car, namely, the code they are responsible for. But they may not be clear about what other developers are doing, like a driver cannot see some direction – we call the missing piece of developers' knowledge as blind spots in software development.

Bugs and security issues are often introduced due to developers' blind spots. In 2020, Zoom was found to use a substandard encryption API in its implementation due to the developer's insufficient knowledge of encryption APIs [40]. In 2023, hackers found security vulnerabilities in Facebook Instant Games when this feature interacted with Facebook authentication [106]. Even within the same organization, since developers on the game module have blind spots on the other component, the authentication APIs. They introduce security holes into the software.

While the definition of blind spots is clear, how to effectively apply knowledge of blind spots remains a problem. Existing works has been proposed to detect bugs and vulnerabilities [183, 150] with formal method [87, 36, 152], static analysis [93, 172, 88, 23, 69, 185, 184] and automatic testing [83, 139, 151]. Some work directly infer rules from source code and picks up deviant outliers, which could be potential bugs [159, 100, 180, 94, 41, 70, 175]. Compared to them, this thesis is the first one to introduce the concept of 'blind spot', which shows a different aspect of software bugs and vulnerabilities from the view of software creators.

To categorize, blind spots happen in two categories:

- Cross-author blind spots. The developers work on the same piece of software but due to a lack of communication or knowledge sharing, they may not know clearly about other developers' intentions of writing some code snippets, even if they need to modify those code snippets.

- Cross-component blind spots. In this category, developers work on different parts of the software, which may require different skill sets or domain knowledge and cause blind spots in other developers' work. However, the different components of the system need to have consistent design or security requirements, which cause bugs in the blind spots.

Although the code under blind spots has a high possibility of being bugs, blind spots are not equivalent to bugs. There are several challenges to detecting real bugs from developers' blind spots: (1) How to pick up the potential bug candidates from the blind spots caused by cross-developer collaboration? Developers collaborate widely across the software. It's impossible to mark all of these code snippets as bugs. (2) How to help developers be aware of their blind spots in other components? It's unreasonable to blame developers for not understanding thoroughly code they don't have ownership of. However, they need to know the missing pieces in their code that are caused by cross-component blind spots. Therefore, it is demanding to develop bug detection tools that can help resolve these challenges and identify bugs, including catching signs of bugs like special code patterns that indicate potential problems from developers' blind

spots, and automatically extracting important information like security assumptions from other components and report the potential violations to developers.

1.2 Contribution

This dissertation presents the corresponding methods of tackling blind spots to help developers detect software bugs and security issues. First, we study a special type of bug pattern that goes unnoticed in existing program analysis and testing research, *cross-authorship unused definitions*. This pattern happens when developers work on code that interacts with code written by other developers, while they might not fully understand other developers' code. As a typical example of cross-authorship blind spots, it could indicate non-trivial bugs like security issues or data corruption, which calls for more attention from developers. Although there are existing techniques to detect unused definitions, it is still challenging to detect critical bugs from unused definitions because only a small proportion of unused definitions are real bugs. We present VALUECHECK to detect this specific pattern based on authorship analysis and code familiarity ranking, which helps to detect bugs from cross-authorship blind spots.

Second, the architecture of the software itself could internally bring blind spots, which we call *cross-component blind spots*. This problem happens when end-to-end testing cannot guarantee the security of each component and validate the coordination between components. We study a typical representative, the web application, which is a special type of client-server application that sends its client source code to the users for local execution. Due to the client-server architecture, web developers should enforce the security checks on the server side to avoid missing server-side check vulnerabilities, which could be cross-component blind spots. We propose a method to detect missing server-side checks by client-side code mutation. It provides several advantages over the existing bypass testing solutions and allows automatic detection of more complicated vulnerabilities that contain business logic constraints, token-based defense, and data preprocessing.

Thesis Statement: *By automatically capturing cross-author and cross-component blind spots of developers, we can provide tooling support for developers to detect bugs and security vulnerabilities.*

1.3 VALUECHECK: Effective Bug Detection with Unused Definitions

Chapter 2 delves into the concept of unused definitions, which are values assigned to variables but never utilized in the code. Traditionally regarded as inconsequential, these unused definitions have been treated as minor issues, often overlooked by developers. However, this chapter sheds light on a reevaluation of their significance, revealing that certain instances of unused definitions could signal more serious issues such as security vulnerabilities or data corruption, demanding heightened attention from developers.

While existing techniques exist to identify unused definitions, discerning critical bugs from this pool remains a challenge due to the overwhelming majority being benign. To address this, the chapter introduces VALUECHECK, a static analysis framework designed to tackle this problem. Firstly, it identifies that unused definitions situated at the periphery of developers' interactions often indicate bugs. Secondly, it outlines syntactic and semantic patterns where unused definitions are intentionally placed, thereby excluding them from bug consideration. Finally, it employs code familiarity metrics borrowed from software engineering to prioritize detected bugs, aiding developers in focusing their attention effectively.

The efficacy of VALUECHECK is evaluated across various large-scale system software and libraries, including Linux, MySQL, OpenSSL, and NFS-ganesha. Impressively, VALUECHECK uncovers 210 previously unknown bugs within these applications, with 154 bugs being confirmed by developers. Notably, compared to existing tools, VALUECHECK demonstrates a marked ability to detect bugs with minimal false positives, highlighting its effectiveness in enhancing software quality.

1.4 FENCEHOPPER: Detect Vulnerabilities by Client-Side Code Mutation

In Chapter 3, we delve into the unique dynamics of web applications, particularly their client-server architecture, where the client's source code is sent to users for local execution. Given this setup, web developers must fortify security measures on the server side to prevent vulnerabilities stemming from inadequate server-side checks. While existing literature primarily addresses server vulnerabilities through techniques like request forgery detection, these methods, while effective for straightforward web services, fall short in identifying bugs in complex web applications, even with client-side code analysis. The limitations lie in their lack of awareness regarding parameter semantics, interactions with third-party services, and business logic constraints, leaving the industry to rely on white hat hackers for detecting missing server-side checks.

In response to these challenges, this chapter introduces a novel approach capitalizing on the distinctive nature of client-side code. Recognizing that client-side code is both accessible and modifiable by users, and is executed locally, the paper proposes a method to identify missing server-side checks through client-side code mutation. This approach offers several advantages over existing bypass testing solutions, enabling automatic detection of more intricate vulnerabilities encompassing business logic constraints, token-based defense, and data preprocessing.

To showcase the effectiveness of this technique, a testing framework named FENCEHOPPER is implemented. This framework minimizes human effort and security expertise requirements while delivering robust testing capabilities. Evaluation conducted on the login and signup pages of the top 300 websites from the Tranco dataset demonstrates the efficacy of FENCEHOPPER, successfully uncovering 48 missing server-side check vulnerabilities, including critical access control vulnerabilities that could compromise the security of over 20 million user accounts.

Chapter 2

VALUECHECK: Effective Bug Detection with Unused Definitions

2.1 Motivation

Unused definitions have been long regarded as redundant code in C/C++ programs. To be specific, a definition of variable *v* in programs refers to an occurrence of *v* on the left-hand side of an assignment statement, and a use indicates an occurrence of *v* on the right-hand side. If a variable is assigned with a value but the value is not used, that is an unused definition. Since unused definitions do not directly cause severe consequences, they are mostly regarded as bad code practices that do not require much attention. However, unused definitions could indicate deeper problems. When developers write an assignment, intuitively they should use that value from the assignment somewhere afterward. When the value assigned is not used, it violates this intuition, thus reflecting the inconsistency of developers' behavior [48, 15].

Unused definitions could reveal underlying critical bugs. Figure 2.1 gives two real-world examples from a widely-used open-source software, NFS-ganesha [9]. In the first example from Figure 2.1a, `attr` is assigned with the return value of `next_attr_from_bitmap`. However, this definition is soon overwritten and not used. This skips the first attribute of the source bitmap list without copying it to the destination bitmap list, causing a severe problem — important file attributes such as ownership are not copied properly to the destination bitmap, which is a security issue that can further lead to privilege escalation. Another example in Figure 2.1b shows a bug

where the function argument bufsz in function logfile_mod_open is unused and overwritten, which is an implicit unused definition. The unused definition serves as a symptom indicating the code possibly contains a problem — In this example, bufsz originates from the configuration value ‘logging buffer size’. If developers set the logging buffer size to zero, they would expect logs to be output immediately. However, since the value is subsequently overwritten with 1400 within the function, the configuration has no effect. The actual access log buffer size is set to 1400 regardless of how the developer set it, resulting in unexpected memory consumption and wrong buffering behaviors.

```
int bitmap4_to_attrmask_t(bitmap4 *bm, attrmask_t
*mask)
{                    This definition is unused
  int attr = next_attr_from_bitmap(bm); [Author1]
  ...
  for (attr = next_attr_from_bitmap(bm) [Author2];
       attr != -1; attr = next_attr_from_bitmap(bm))
  {...}
}
```

(a) A Severe Bug Underlying Unused Definitions. Function bitmap4_to_attrmask_t() converts NFSv4 attributes mask to the FSAL attribute mask. However, the first attribute returned by next_attr_from_bitmap() is unused and overwritten by another definition.Thus the first attribute doesn't get copied to the FSAL attribute mask, which can result in security issues related to file permissions.

```
headerslog = log_mod_open("headers.log", 0); [Author1]
------------------------------------------------------
int logfile_mod_open(char *path, size_t bufsz)[Author2]
{ // implicit: bufsz = 0 <- This definition is unused
  bufsz = 1400; [Author2]
  if (bufsz > 0) {...}
}
```

(b) A Configuration Bug Underlying Unused Definitions. The developer of logfile_mod_open does not use value of bufsz but directly overwrite it with 1400, thus the value 0 assigned to bufsz is unused. This bug causes unexpected memory consumption and wrong buffering behaviors.

Figure 2.1. Two real-world bugs underlying unused definitions detected by VALUECHECK with severe consequences like security issues and configuration bugs. Both bugs cross the author scope.

It is challenging to detect these bugs from unused definitions, although existing tools can detect unused definitions:

(1) Existing techniques only detect which definitions are unused but do not differentiate non-trivial bugs from simply redundant code. This is critical because non-trivial bugs require developers to diagnose more carefully than simply removing the redundant code. Without differentiation, detecting bugs from a large number of unused definitions will be impractical. Besides, existing techniques fail to detect unused definitions precisely and miss unused definitions that could be bugs. (2) Existing techniques do not consider the semantics of unused definitions. Some of the unused definitions could still express special meanings in the programs. They are written by developers intentionally and thus are not bug indications. (3) Existing techniques provide no priority ranking of the detected unused definitions. Since unused definitions are not directly the root cause of a bug but just bug indications, developers could spend more effort but detect fewer bugs without a way to distill bugs. As a result, existing tools report hundreds of unused definitions from a project with high false positives, which requires overwhelming effort from developers to check. We observe that a subset of developers choose to disable unused definition warnings in compilers. In the top 40 GitHub C++ repositories with the most stars, nearly half of them do not use these options in their default compiler configurations[1]. In search results of 'gcc unused' from StackOverflow [13], the largest online community for developers, the top-voted question looks for suggestions on how to silence unused definitions warnings.

In this paper, we design an effective and practical approach to detect, distill, and rank real bugs from a large number of unused definition candidates. Our approach addresses the above challenges based on three insights.

First, we make a unique observation that the unused definitions caused by code written by multiple developers, which we call *cross-scope unused definitions* in this paper, are more bug-prone. We randomly sample 42 unused definitions bugs from the history commits of 4 open source projects from 2019-2021 (c.f. § 2.3.1) and observe that majority of these bugs involve code written by two developers. The definition is written by one developer but is ignored/over-

[1]We collect this data on April 9, 2022. We regard compilation configuration with '-Wall', '-Wunused-but-set-variable', '-Wunused-value' but without '-Wno-unused-*' options as detecting unused definitions.

written by another developer. We show two examples of such unused definitions. In Figure 2.1a, `attr` is written by author `Author1` but the definition is overwritten by author `Author2`. In Figure 2.1b, the function call is written by author `Author1` but the function is implemented by author `Author2`, overwriting the value provided by `Author1`. The data flow from definitions to uses in programs embeds the assumptions of developers on the programs. However, this implicit knowledge sometimes cannot be fully shared due to the lack of documentation and communication. Therefore, the unused definition on the boundary potentially indicates inconsistency between developers and a higher chance of being bugs. We take advantage of this observation to distill bug-prone unused definitions.

Second, we summarize several patterns of unused definitions that are intentionally written by developers in programs. We observe that not all unused definitions are redundant code in programs, some of which have semantics like moving a cursor or are intentionally written to keep compatibility. This kind of unused definition is not a true bug. Also, by mining the use pattern of definitions in similar contexts, which we call *peer definitions* in this paper, we can avoid reporting unused definitions that are not necessarily used. For example, return values of `printf()` usually require no checks and get ignored but do not harm. With these summarized patterns, we prune thousands of irrelevant unused definitions for detecting bugs, requiring no additional input from developers.

Third, we make the first attempt to apply code familiarity models in bug detection to prioritize the unused definitions that have a higher chance of being bugs. Since unused definitions are not directly linked to bug root causes but just indications of potential bugs, the false positive rate would be an internal drawback when applying them to detect bugs. Therefore, we hope to figure out a way to maximize the output with low developer efforts. The intuition behind the ranking algorithm is that unused definitions reflect the inconsistency in developers' coordination. A developer with lower expertise/familiarity with code may easily ignore the assumption and intercept the original data flow [37]. Research on code familiarity and expertise are applied to guide defect prediction and expert recommendation, etc. [104, 145, 47]. Existing code familiarity

models take authorship [104], commit history [145], and other code editing activities [47] as metrics, which are obtainable from version control systems [72, 140]. However, few bug detection tools ever take advantage of these explorations from the software engineering field. We propose to prioritize code review on unused definitions that have a higher chance of bugs [27] with code familiarity models, instead of requiring the developers to check all of them under time pressure.

We implement a static analysis framework VALUECHECK based on these insights, which detects cross-scope unused definitions, prunes false positives considering the semantics and developers' intention, then applies the code familiarity model to rank and prioritize detected unused definitions. The analysis is flow-sensitive, field-sensitive, alias-aware, and involves inter-procedural authorship analysis. We evaluate VALUECHECK with four systems and libraries: Linux, MySQL, OpenSSL, and NFS-ganesha. VALUECHECK helps detect 210 new bugs, among which **154 are confirmed and fixed by developers**, with a false positive rate of 26%. Compared to existing tools, VALUECHECK demonstrates that it can practically help developers in detecting real bugs from unused definitions.

2.2 Background

2.2.1 Liveness Analysis

In this paper, we adopt the terminology defined in [16] to describe our algorithm.

Definition and Use. A definition of variable *v* in programs refers to an occurrence of *v* on the left-hand-side of an assignment statement[2]. A use indicates an occurrence of *v* on the right-hand-side.

Live Variable. A live variable is a variable that is assigned a value that is used in the future.

Liveness Analysis. Liveness analysis is a type of def-use analysis that identify whether a

[2]This is different from C++ concept of 'definition' where a definition provides a unique description of an entity.

variable is live at certain points. It is a backward data flow analysis. It computes a live variable set from successors of this point and checks whether the variable has a use in this live variable set. If not, the variable is an unused definition. To formalize it as a dataflow problem, for each statement node n, $gen[n]$ is the set of variables used in n. $kill[n]$ is the variable defined in n. $in[n]$ and $out[n]$ are the live variable set before and after n. It has:

$$in[n] = out[n] - kill[n] \cup gen[n]$$
$$out[n] = \bigcup_{s \in succ(n)} in[s]$$

By computing the $in[n]$ and $out[n]$ iteratively based on the worklist algorithm, these sets will converge to a fixed point.

2.2.2 Unused Definitions

Detecting unused definitions has been regarded as compiler optimization [76, 73, 34, 144] for a long time and is extensively discussed in topics of dead variables [85, 167, 58, 30] and liveness analysis [132, 109]. This kind of optimization is even merged into modern compilers [115, 86]. The technique of detecting unused definitions is fully developed and merged into mainstream compilers [11, 1] to eliminate redundant computation in the generated assembly code and release the allocated registers. Besides, compilers also provide warning options at the source code level like '-Wunused-value', '-Wunused-variable', '-Wunused-but-set-parameter', etc. In this paper, we do not regard them as useless code that needs to be removed but as 'useful symptoms' that indicate bugs.

2.3 Design Overview

2.3.1 Detection Scope

Detecting critical bugs from a large number of unused definitions is non-trivial. To achieve this, we make a unique observation that unused definitions involving multiple developers

are likely to be bugs. Therefore, we define the concept *cross-scope unused definitions* to help us pinpoint bugs, which includes the following scenarios: (1) Ignored/unused return value, where the developer that implements this function is different from the developer call this function. (2) Overwritten function argument value, where inside the function the argument value is overwritten but the function call is invoked by another developer. (3) Overwritten definitions, where the code of the old definition is written by one developer, and the code overwrites the old definition is implemented by other developers on all successor paths of the overwritten definition. The common characteristic of them is that the value of the definition is generated by one developer, but ignored/overwritten by another developer, causing an unused definition in the code. In all these scenarios, we only consider local variables in a function. Therefore, we do not consider concurrent access to shared variables since shared variables are global variables.

Our design is inspired by a preliminary experiment on collecting existing bugs related to cross-function definitions: We implement original liveness analysis and apply it to the snapshots of MySQL, NFS-ganesha, OpenSSL, and Linux on the first commit of 2019 and 2021 separately. We collect unused definitions that were present in the 2019 version but were subsequently removed in the 2021 version. If a bug fix commit removes this unused definition, we investigate how this bug relates to the unused definitions. We identified a total of 325 unused definitions through differential comparison. To investigate their impact on bugs, we randomly selected 60 of these unused definitions by assigning them serial numbers and generating random numbers between 1-325. We manually checked the commit messages to ensure that the developers had addressed these unused definitions as part of the bug fixes, and we found 42 bug-related unused definitions. Notably, 39 out of the 42 instances crossed author scopes, indicating that these defects were located at the boundaries of developers' interaction. Building on this insight, we focused our attention on detecting cross-scope unused definitions in VALUECHECK and explored their effectiveness in detecting bugs.

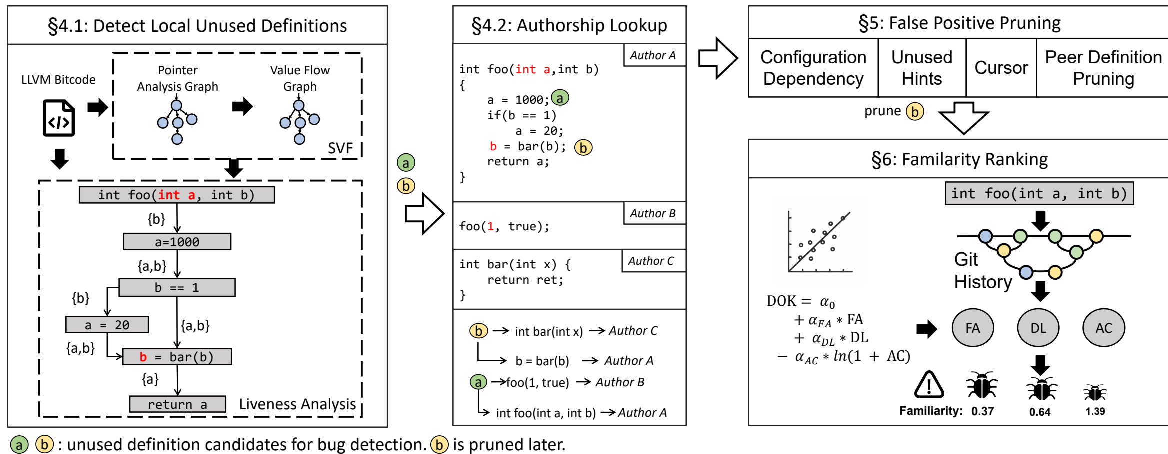


Figure 2.2. Overview of ValueCheck. ValueCheck consists of cross-scope unused definition detection, false positive pruning, and familiarity ranking.

2.3.2 Framework Overview

ValueCheck detects cross-scope unused definitions in source code, prunes the false positives, and then ranks them by code familiarity to help detect bugs. Its workflow is shown in Figure 2.2. The generated ranked unused definitions can be checked by developers with given ranking as priority. ValueCheck faces three unique challenges to achieve its goal:

(1) How to completely and precisely detect cross-scope unused definitions in all scenarios? (Section 2.4) To overcome this challenge, we performed a more precise analysis to achieve better coverage than the state of art compilers [11, 1]). First, we conduct flow-sensitive liveness analysis to achieve a higher precision; Second, we propose a inter-procedural authorship analysis to identify cross-scope unused definitions which have constraints on the authorship of relevant code snippets within and across different functions; Third, we extend the detection scope to unused definitions of field variables (a field in a struct or class) by extending the detection algorithm of local unused definitions to be field-sensitive. Besides, we take advantage of the existing pointer analysis and def-use analysis framework to precisely obtain value flow information in the analysis.

(2) How to prune false positives from the large number of detected unused definitions? (Section 2.5) The main issue with existing solutions is the high false positive rate. This

places a heavy burden on developers to manually check them to detect underlying non-trivial bugs. Therefore, VALUECHECK needs effective approaches to prune false positives that are misreported by the analysis. We propose pruning approaches which trims false positives that have special meanings in the program, or are intentionally written by developers. The pruning requires no additional annotations from developers.

(3) How to rank up unused definitions that are more likely to be bugs? (Section 2.6) To reduce the effort from developers, VALUECHECK adopts code familiarity models to rank the detected unused definitions. Our intuition is that for the cross-scope unused definitions, one of the developers introduces unused definitions into the code because they are not fully aware of the data flow in the program they touch. Therefore, for the developers with low familiarity, we rank the unused definitions introduced by them with high priority.

2.4 Detecting Cross-Scope Unused Definitions

2.4.1 Detect Local Unused Definitions

Liveness Analysis. Existing detection of unused definitions in compilers mostly relies on AST walking, which only reports a definition as unused when the variable is not referenced at all. However, the order of defines and uses could decide whether a definition is unused, which is flow-sensitive. Therefore, we detect unused definitions with a flow-sensitive liveness analysis in VALUECHECK. It conducts analysis on the control flow graph of the function, starts from the end of the function, traverses each basic block backward and updates the live variable set based on the memory operations (load and store) on variables. To deal with loops in the function, we iterate the liveness analysis for several times until it reaches the fix point. The live variable set only records the existence of a use on the variable and cleans all the uses when traversing a definition. After the live variable set converges, we check each definitions in this function to see whether a use of this variable is in the live variable set. If not, we detect an unused definition. In this way, we can also check at the entry of a function whether a parameter is in the use set. If not,

Table 2.1. **Meaning of notations in the algorithms (Figure 2.4).**

Notation Table	
`getVar(v)`	Get variable names of a value
`getRetAuthor(F)`	Get authors of all return statements in function `F`
`checkAuthor(A, List, LiveSet)`	Check whether `A` is different from items from
	`List` before adding the unused definition to `LiveSet`.
`updateDef(v, A, DefSet)`	Update the author of variable `v` to `A` in `DefSet`
`getCallSite(F)`	Get all call sites of function `F`
`getLine(A)`	Get the source code line of `A`
`reverse()`	Traverse the basic blocks or instructions reversely
`checkAlias()`	Check the point-to graph and the value flow graph for the variable aliases

this parameter value is not used in this function, thus also an unused definition.

Indirect Function Call. In cases where the unused definition is generated by a function call, VALUECHECK extracts the source code location of the called function to enable querying in the authorship lookup phase. When handling function pointers, VALUECHECK checks the points-to set of the pointer to look up the corresponding functions. The pointee functions are treated as direct function calls in authorship lookup phase. Since VALUECHECK focuses only on local variables, it does not need to delve into the callees in the analysis phase. Thus, there is no need for us to handle the recursive calls differently.

Pointer and Alias. To detect indirect access via pointers, we utilize pointer analysis and examine the point-to graph to determine whether the definition variable is included in the pointer-to sets of other variables. If the definition is referenced by pointers, it is considered possibly used through indirect reference and is therefore not marked as an unused definition. We conduct field-sensitive Andersen's analysis [17] because its better scalability compared to flow-sensitive pointer analysis, while providing a small difference in help detecting unused definitions according to previous work [66]. To handle aliases of variables, we check the value-flow graph generated based on the point-to graph to see whether this definition is used somewhere else. If it has other use, this definition is not an unused definition.

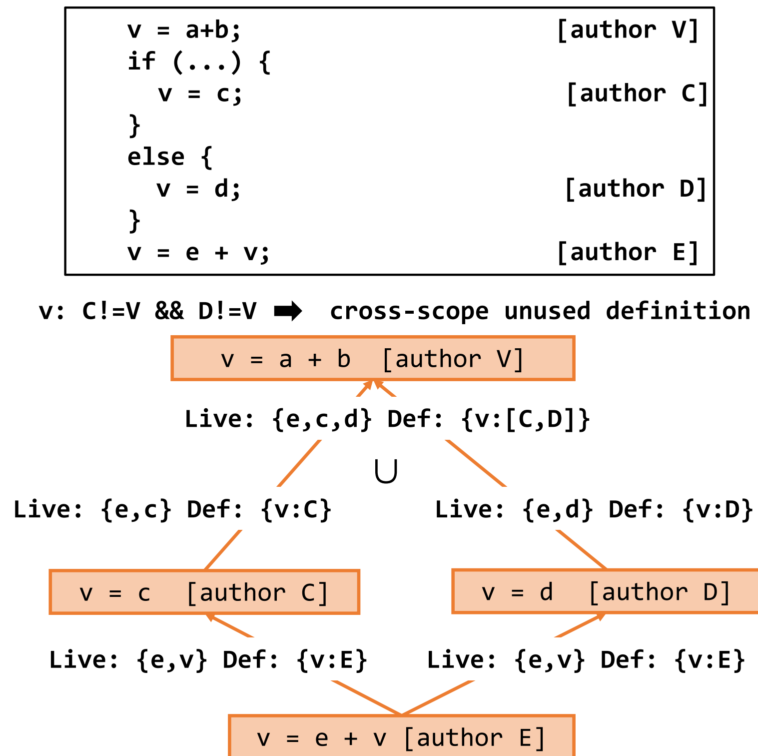


Figure 2.3. An Example of Define Set. The first definition of v is overwritten by other developers on all the successor paths.

2.4.2 Authorship Lookup

To decide whether an unused definition is a cross-scope unused definition, VALUECHECK looks up the authorship information of the unused definitions it detects based on the three scenarios we discussed in Section 2.3.1:

(1) For the unused return value, we get the author of the call site first, assuming it is author D. Then we search the source code file of the callee and look up the authorship of the line that returns this value. However, a function may have multiple locations of returning a value, of which the authors are B_1, B_2, B_3... In this case, we got all these locations. If author B_1, B_2, B_3... are all different from D, we regard this as a cross-scope unused definition. If the callee is a library call not included in this project, we regard the author is different from D.

(2) For the unused function argument, we get the author of the call site, which we assume

```
function Load::Update(v1 = load v2, LiveSet, DefSet)
1: LiveSet.In.add(getVar(v2))
```

```
function Store::Update(I = store v1 v2, LiveSet, DefSet)
1: author1 = getAuthor(getLine(I))
2: if LiveSet.In.find(getVar(v2)) then
3:     LiveSet.In.remove(getVar(v2))
4: else
5:     checkAuthor(author1, DefSet[v2].author, LiveSet.Unused)
6:     if isRetVal(v1) then                  ▷ Check unused return value
7:         author2 = getRetAuthor(getCall(v1))
8:         checkAuthor(author1, author2, LiveSet.Unused)
9: updateDef(getVar(v2), CurrAuthor, DefSet)
```

```
function computeCrossDef(F)
1: while Change do Change = False
2:     for BB in F.reverse() do                    ▷ Traverse basic blocks
3:         OrigLiveSet = LiveSet[BB]
4:         LiveSet[BB].In = ⋃_{s∈BB.getSucc()} LiveSet[s].Out;
5:         DefSet[BB] = ⋃_{s∈BB.getSucc()} DefSet[s];
6:         for I in BB.reverse() do
7:             I.Update(I, LiveSet[BB], DefSet[BB]);
8:         if OrigLiveSet != LiveSet[BB] then
9:             Change = True                       ▷ Iterate to handle loops
10: for A in F.args() do  ▷ Check unused definition of parameters
11:     EntryBB = F.getEntryBB()
12:     if !LiveSet[EntryBB].In.find(A) then
13:         for CS in getCallSite(F) do
14:             author1 = getAuthor(getLine(A))
15:             author2 = getAuthor(getLine(CS))
16:             checkAuthor(author1, author2, LiveSet.Unused)
17: if !LiveSet.Unused.empty() then
18:     checkAlias(LiveSet.Unused)
19: reportCrossUnused(F, LiveSet.Unused)
```

Figure 2.4. Cross-Scope Unused Definition Detection.

is author C, then look up the author B that defines the parameters of function F. If C and B are different, it is a cross-scope unused definition. If the parameter is overwritten inside the function F by developer D, we compare D to author C.

(3) To decide whether an unused definition is overwritten by other developers, VALUECHECK needs to record additional information on whether another definition overwrites it in the successors of this definition. Therefore, we extend the original liveness analysis to maintain another set `define` aside from the live variable set. It records the last definition of variables we traverse and the corresponding authors. The define set follows the same update rule as the live variable set. We show an example in Figure 2.3. In this example, VALUECHECK reversely traverses blocks in this function. Whenever there is a definition of variable `v`, it updates the author of `v` in the define set. For every block, it unions the define set of its successors. When an unused definition is detected, it checks its author against the authors in the define set to decide whether this is a cross-scope unused definition. Therefore, in this example, the first definition of `v` written by author V is a cross-scope unused definition.

Algorithm Details. Table 2.1 and Figure 2.4 show our unused definition detection algorithm. To handle field-sensitive analysis, we check the value inside `getVar()`. If this value is loaded from a field of a struct variable v with offset n, we create a new variable name as the v_n to refer to this field. In this way, we can treat the field definitions similarly to other definitions. In this unified framework, `computeCrossDef()` computes the cross-scope unused definitions for each function. It traverses each basic block reversely. For each load instruction, it adds a use to the live variable set. For each store instruction, it removes all the use of this variable and if there is none, it detects an unused definition. In this case, it checks with `checkAuthor()` to see whether this is a definition overwritten by other developers or a return value written by other developers by `getRetAuthor()`. Besides, it will update `DefSet` by `updateDef()`. By iterating repeatedly on this function, the live variable set and the define set are guaranteed to converge. The cross-scope unused definitions in this function are reported.

Table 2.2. Summary of pruning patterns in VALUECHECK.

Name	Code/IR Pattern

Configuration Dependency	/* Variable host is used when the config USE_ICMP is enabled. */
	char **host**[10] = ”127.0.0.1”
	#if USE_ICMP
	n = netdbLookupHost(**host**);
	#endif

Cursor	/* The definition expresses the semantic of moving cursor. */ (**buf**++) = ‘a’;
Unused Hints	/* The unused definition is marked by developers with aware.*/ int do_flush_info(const bool **force** [[maybe_unused]]) {...}
Peer Definition Pruning	/* The unused definition is intentionally ignored by developers. */ printf(”%d\n”, num); // An implicit definition **[tmp]** = printf()

2.5 Pruning

Not all cross-scope unused definitions are bugs. We observed that there are many cases where an unused definition is intentionally left in programs. Reporting them as bugs can introduce a high false positive rate. Based on our observation, we summarize four patterns for pruning as shown in Table 2.2.

2.5.1 Configuration Dependency

The use statements of some definitions could be controlled by preprocessor directives (e.g., `#if`), which may be disabled by the compilation configurations. In this case, static analysis may regard these definitions as unused because their uses are not compiled into IR. Therefore, VALUECHECK looks into the corresponding source code of each definition and checks if there is any use of this definition enclosed by `#if`, `#ifdef`, `#ifndef` and `#endif` directives in the same function. If so, we prune this definition.

2.5.2 Cursor

As shown in Figure 2.5, after assigning a value to the memory region that the cursor `o` points to in Line 259, the code increments `o`. This definition serves as program semantics

```
237  static void dashes_to_underscores(...)
238  {
239      char *o = output;
254      if (c == '-')
255          *o++ = '_';
259      *o++ = '\0';
260  }
```
This unused definition is a cursor.

Figure 2.5. Example of Cursors.

"moving cursor" intentionally. Therefore, we regard these unused definitions are not bugs and prune them to reduce false positives based on the uses of a variable in the value flow graph. If a variable is incremented repeatedly by the same constant, VALUECHECK considers it as a cursor and prunes it.

2.5.3 Unused Hints

In some cases, developers keep a definition unused for intended reasons. To hint these definitions are unused, the developers could add an unused attributes to them. We exclude them by matching the keyword 'unused' in the source code of these unused definitions.

2.5.4 Peer Definition Pruning

Sometimes, function calls are guaranteed to be successful or developers just don't care whether the call succeeds or not, resulting in the return value being unused. To quantify how much the developers 'care about' using the definition, we look at peer definitions of this definition. We define *peer definitions* as (1) For the definition of a function $ret = F()$ return value, peer definitions of ret are return values of other call sites of F. (2) For n_{th} parameter of function F, its peer definitions are the n_{th} parameter from functions with the same signatures. If the occurrences are over ten and over half of the peer definitions are not used, we will not report it.

2.6 Ranking based on Code Familiarity

We adopt the code familiarity model to help developers prioritize their effort in checking unused definitions that are more likely to be bugs. Even with rigorous pruning strategies, there are still unused definitions that require checks by developers, which could impose a workload on developers.

To deal with this, we propose to integrate the code familiarity model into VALUECHECK, which helps to rank the unused definitions that are more likely to be bugs. Our intuition is that if the developer is not familiar with a certain snippet of code, he/she is more likely to cause some inconsistent behaviors in code. To measure the developers' code familiarity, the software engineering area has explored the code familiarity models for years. These models extract the familiarity metrics from the code contribution history to measure the developers' expertise. VALUECHECK selects one representative work from the code familiarity area, the degree-of-knowledge (DOK) model [47], to measure code familiarity. The DOK model used in VALUECHECK is:

$$\mathbf{DOK} = \alpha_0 + \alpha_{FA} * \mathbf{FA} + \alpha_{DL} * \mathbf{DL} - \alpha_{AC} * ln(1 + \mathbf{AC})$$

With this model, we select the author of each code line and compute this author's familiarity with the current file. **FA**, **DL**, and **AC** respectively represent first authorship, the number of deliveries from a developer, and deliveries to this file that are not authored by this developer. We count the commit numbers instead of committed lines because it is less resource-intensive and time-consuming. Based on prior literature [108], there is a strong correlation between commit numbers and commit line numbers. To obtain the weight in this model, we follow the steps of the original paper [47] to sample 40 source code lines from each application and ask the developers to self-rate their code familiarity (from 1-5) on these lines. Then we fit the linear model and get the weights, which are $\alpha_0 = 3.1$, $\alpha_{FA} = 1.2$, $\alpha_{DL} = 0.2$, $\alpha_{AC} = 0.5$. We apply this linear model

in VALUECHECK to compute the code familiarity.

The DOK model is chosen for two reasons. First, the DOK model is the most recent and representative model of code familiarity. It considers common factors that are mostly accessible in real-world software development. Second, the DOK model has generality to different applications. The three factors in DOK model, which are first authorship (FA), deliveries from a developer (DL), and deliveries to a code element that are not authored by this developer (AC), are language-independent and obtainable from most open source projects.

2.7 Implementation

Based on the design and techniques presented in Section 2.3, Section 2.4, Section 2.5 and Section 2.6, we implement the framework VALUECHECK with LLVM-13.0.0 [10], SVF-2.6 [156] and python. It consists of four components:

Code analysis. The clang compiler compiles the source code into LLVM [86] bitcode. Then VALUECHECK obtains point-to graph and sparse value flow graph based on SVF [156]. Since SVF takes a long time to analyze a whole large scale program, we apply VALUECHECK on separate bitcode files generated by each single program file and only call SVF APIs to generates value flow graphs and the point-to graphs when we identify unused definitions within this bitcode file and want to further check aliases and indirect calls. It conducts analysis based on control flow graphs of each function, and accesses the def-use information of variables based on the sparse value flow graph and the point-to graph.

Authorship Lookup. This part is implemented in Python. It reads the meta information such as file names, function names, and line numbers of each unused definition. Then it reads the git meta files to look up authorship of unused definitions based on `GitPython` [2]. Then it compares the authorship and outputs cross-scope unused definitions.

False Positive Pruning. This component consists of two parts in LLVM and Python respectively. It checks the corresponding source code for each definition to see whether it should be pruned

based on the pruning strategies. To match the unused hints and configuration dependency, we use `re` [12] library to do regex matching on code. For cursor and peer definition pruning, we collect all uses of variables and functions by LLVM API `getNumUses()`.

Familiarity Ranking. After pruning, VALUECHECK computes FA, DL, and AC values for each unused definition by traversing the file commit log in git repositories with `GitPython` [2]. Lastly, VALUECHECK outputs the unused definition report ranked by code familiarity.

2.8 Evaluation

In the evaluation, we answer the following questions:

1. How effective is VALUECHECK in applying cross-scope unused definitions to help detect bugs?
2. What is the accuracy of detecting bugs in VALUECHECK?
3. How does VALUECHECK compare with existing tools?
4. What is the contribution of each component in VALUECHECK?
5. How scalable is VALUECHECK on a large code base?

2.8.1 Experiment Setup

Evaluated Applications. We mainly evaluate VALUECHECK with four widely-used open-source system software and libraries, Linux-5.19, MySQL-8.0.21, OpenSSL-3.0.0, NFS-ganesha-4.46. These applications are selected with three criteria. First, they are popular real-world system projects of various types, which reflect how generally VALUECHECK can be applied to real-world applications. Second, the source code of these applications is well-maintained and tested. The detected bugs are not from an immature program. Third, these applications have abundant version histories.

Evaluation Environment. All the experiments are conducted on a machine with 3GHz 6-core Intel i5-9500 CPU, 9216 KB cache, 16GB memory, and a 480GB SSD, which runs Ubuntu 18.04 with kernel 4.15.0. All applications are compiled with `-O0` and `-fno-inline` by clang-12 to retain source-level information. We compile each source object into separate bitcode files then perform analysis on these individual bitcodes. This helps reduce overhead of SVF from the inter-procedural analyses but does not affect the detection results since our detection target is local unused definitions.

2.8.2 Detect New Bugs

Table 2.3. The number of bugs newly detected by VALUECHECK. Among the 210 bugs detected, 154 bugs are confirmed by developers.

Application	#Detected Bugs	#Confirmed Bugs
Linux	63	44
NFS-ganesha	22	18
MySQL	99	74
OpenSSL	26	18
Total	**210**	**154**

Overall Results

VALUECHECK detects 210 new bugs from cross-scope unused definitions in applications, among which 154 are confirmed by developers. We apply VALUECHECK to the recent versions of the evaluated applications listed in Section 2.8.1. We report bugs detected by VALUECHECK to developers and the result is shown in Table 2.3.

Ethics and Responsible Disclosure

We take ethics in the highest standard regarding the new bugs we detect. We report all the bugs we detect to the developers though their official bug mailing list, clarify the potential impact of the bugs and help with the patches. We do not reveal the details of the bugs to any unofficial channels unless they are already fixed. In this paper, we anonymize all developers'

Table 2.4. Bug examples detected by VALUECHECK. Generally, VALUECHECK detects two categories of bugs: missing check bugs and semantic bugs. Of 154 bugs confirmed, 134 are missing check bugs, 20 are semantic bugs.

Bug Type	App.	Bug Description
Missing Check (134)	NFS-g	Unhandled ACL error
	MySQL	Missing sanity check
	MySQL	Unhandled error code
	OpenSSL	Malloc a negative size
	Linux	Fail to check device status
Semantic Bugs (20)	NFS-g	Ignore first bitmap attribute
	OpenSSL	Use the wrong master secret in TLS

names and identifiers and hide irrelevant details of the bug code that could be used to trace the authors.

Bug Case Study

Table 2.4 shows several bugs we detected with VALUECHECK. These bugs vary in terms of types and root causes. We categorize them into two types: 1) Missing check bugs — bugs that fail to check on function return values, parameters, or other variables, as the example shown in Figure 2.6a. This will make the following execution take the wrong assumption on completeness of certain operations and even cause corrupted data to be used by the program silently; 2) Semantic bugs — bugs that break specific program semantics. This will cause no runtime crash but the logic of the programs is wrong, as shown in Figure 2.6b. Some of them are hard to detect with existing solutions. For example, for the bug in Figure 2.6a, VALUECHECK detects latent errors which could cause invisible symptoms and affect further execution but do not crash the programs immediately, which is hard for developers to detect pre-release by testing. The error in initializing the recovery mechanism of the page archiver could cause failure in future execution, which demonstrates the effectiveness of VALUECHECK in detecting real-world non-trivial bugs.

```
dberr_t Arch_Page_Sys::recover() {
    err = arch_recv.init(); ← err is unused
    ... // No reference to err
    err = arch_recv.fill_info(this);
    if (err != DB_SUCCESS) {
        return (DB_OUT_OF_MEMORY);
    }
}
```

(a) A Missing Check Bug Detected by VALUECHECK. The error code returned from `init()` function is unchecked, which could result in the crash of page archiver recovery. It is a latent error that will corrupt data and cause failure in future execution, which is missed by the test suite.

```
void update_sctx() {
    const char *to_host;  ...
    if (!to_host) to_host = "";        [author1]
    sctx->assign_host(to_user_ptr->host.str,
to_user_ptr->host.length);             [author2]
    ...
}                        Need use 'to_host' here
```

(b) A Semantic Bug Detected by VALUECHECK. `to_host` is assigned a value but not used. It should have been used as the first parameter of `assign_host()`. Otherwise, the incorrect host address could corrupt the security context `sctx`.

Figure 2.6. Examples of New Bugs Detected by VALUECHECK.

Bug Categorization

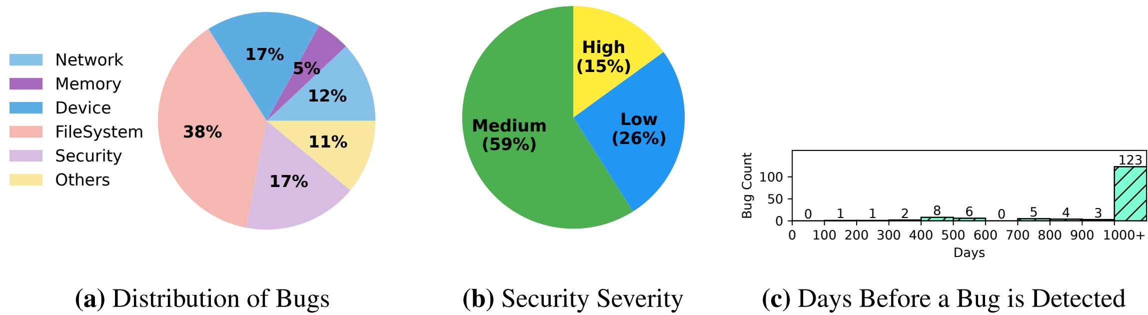
To investigate when the 154 new bugs detected by VALUECHECK arise and how they affect the applications, we classify them based on their distribution across software components, security severity, and the number of days it took to detect them. The results are illustrated in Figure 3.5.

(1) Distribution. 38% of the cross-scope unused definition bugs we detected are related to file system, and 17% of the bugs are located in security modules such as authentication modules (Figure 3.5a).

(2) Security Severity. We categorize the severity levels assigned by developers to the bug reports. In cases that the severity level is not provided, we refer to the corresponding CWE. As shown in Figure 3.5b, 15% of bugs are of high severity and 59% are of medium severity, indicating that the bugs detected by VALUECHECK can point to severe security issues like broken

access control, data leak, etc.

(3) Days before Detected. From Figure 2.7c, more than 80% of the bugs had persisted in the code base for over 1000 days before we reported them and get confirmed, indicating a significant challenge in diagnosing and detecting these bugs. This suggests that VALUECHECK is effective in detecting long-standing bugs that have gone undetected in the code base.



(a) Distribution of Bugs **(b)** Security Severity **(c)** Days Before a Bug is Detected

Figure 2.7. Bug Categorized by Component Distribution, Security Severity and Days before Detected. All evaluated software undergo thorough testing. Despite this, VALUECHECK uncovers high-severity bugs in critical components that have previously gone undetected for a long time.

2.8.3 Accuracy of VALUECHECK

Table 2.5. Pruning breakdown and sampled false negative rate in VALUECHECK. The false negative rate of pruning is less than 10% based on sampling with 95% confidence.

App.	#Original	#Pruned (%Prune Rate)					#Detected After Pruned	% Prune False Negative (sampled)
		Config Dependency	Cursor	Unused Hints	Peer Definition	Total		
Linux	259	1 (0.39%)	22 (8.49%)	46 (17.76%)	127(49.03%)	196 (75.68%)	63	2%
NFS-g	898	7 (0.78%)	7 (0.78%)	839 (93.43%)	23 (2.56%)	876 (97.55%)	22	1%
MySQL	7743	37 (0.48%)	83 (1.07%)	3031 (39.15%)	4493 (58.03%)	7644 (98.72%)	99	3%
OpenSSL	642	18 (2.82%)	74 (11.60%)	322 (50.47%)	202 (31.66%)	616 (96.55%)	26	1%

False Positives

VALUECHECK has a low false positive rate (18%-30%) for detecting bugs if we only consider the bugs already confirmed by developers. These false positives come from three sources: (1) 51 false positives are still admitted by developers as minor defects in programs but not serious bugs. Sometimes the developers do not mark a definition as unused even though

they know clearly it is no longer used. When we report them, some developers add the unused markers to avoid confusing other developers and the markers will help improve code readability. However, some developers just ignore them. For example, some return error codes are unused because the developers know the function call will not fail in this context. (2) 5 false positives are from debugging code and deprecated code but are not included in the release version. We detect them and report them as bugs since we compile all applications in debug mode and conduct the analysis on all functions. However, the developers do not put a high priority on these unused definitions.

False Negatives

We apply VALUECHECK to detect the 39 existing bugs we collect in the preliminary experiments. VALUECHECK successfully detects 37 existing bugs from them, namely 92.3% recall. 2 bugs are missed from the detection due to the peer definition pruning, which prunes the bugs when most of their peer definitions are unchecked.

Prune Rate

To help understand the effectiveness of our pruning strategies, we present the breakdown of each pruning strategy in Table 2.5. Overall, VALUECHECK's pruning strategies largely reduce the number of candidates for detecting bugs. For all the evaluated software, VALUECHECK prunes between 75.68% to 98.72% of the cases, significantly reducing the burden of developers in reviewing all the potential bugs. Specifically, the unused hints and peer definition pruning strategies are found to be the most effective in reducing the number of candidates. For example, in MySQL, pruning eliminates over 7000 cases, with 98% due to these two pruning strategies. It's worth noting that the prune numbers are obtained from the pipeline of pruning as we showed in Figure 2.2, which means some false positives may match multiple patterns in our pruning but are pruned by the pruning strategies in the earlier stage. With powerful and aggressive pruning strategies, VALUECHECK ensures that most of the detected cases are truly unused.

False Negatives of Pruning

To further evaluate the precision of the pruning strategy, we sample 100 cases from the unused definitions that get pruned from each application and compute the sampled false negatives, as shown in Table 2.5. We sampled pruned cases by assigning serial numbers to the cases in the order they were detected by VALUECHECK. We generated random numbers within the serial number range and selected the pruned cases corresponding to these numbers. For each application, the false negative rate of pruning is less than 10%, which is statistically significant with 95% confidence. This suggests that the vast majority of cases that were pruned are indeed false positives. Among 7 false negative cases, 2 are due to the configuration dependency pruning, for which developers mark the variables with `(void)` to silent the unused warnings. However, this is not good practice of dealing with unused definitions. 5 are due to peer definitions pruning. Despite the potential false negatives, we still regard these pruning strategies as necessary because they effectively reduce the false positive rate to an acceptable level. According to experience from [26], developers tolerates false negatives better than false positives.

2.8.4 Comparison with Existing Tools

In this section, we compare VALUECHECK to Clang, fb-infer, Smatch and Coverity on detecting bugs from unused definitions, as shown in Table 2.6.

Comparison with Clang

We compare VALUECHECK to the compiler Clang. Maintainers of the evaluated applications periodically clean up code based on Clang warnings as indicated in their commit history. Therefore, no unused definitions are reported when we compile with the option '-Wunused'. Many unused definitions detected by VALUECHECK but not by Clang is because Clang does not perform a precise analysis to detect unused definitions but just depends on recursive AST walking. It follows gcc as the specification and only detects a variable as unused when it never gets referred to on the right-hand side. Therefore, no bugs newly detected by VALUECHECK are

detected by Clang.

Comparison with fb-infer

FB-infer is a static analysis tool from Facebook. It can detect unused definitions that are referred to as "Dead Store" in its report, which we refer as 'Infer-unused'. Table 2.6 shows fb-infer detects fewer bugs than VALUECHECK because they are incomplete in detecting all types of unused definitions in programs like overwritten/ignored arguments and field unused definitions. Also, fb-infer has a much higher false positive rate than VALUECHECK. The false positives come from the following reasons: (1) fb-infer reports many unused definitions that are not cross-scope. When developers call the function written by themselves, they usually have the sense of when to use the parameter and the return value and when not. Therefore, they typically do not confirm unused definitions that are not cross-scope as bugs. (2) Cursor assignments, which are not excluded from fb-infer results. In our sampling, all the true bugs detected by fb-infer are also detected by VALUECHECK. VALUECHECK detects more bugs from unused definitions with a lower false positive rate compared to fb-infer.

Comparison with Smatch

Smatch [4] is a static analysis tool for Linux based on AST. It reports warnings when bug patterns are matched. It helps kernel developers detect thousands of bugs in kernel [3].

Smatch detects fewer bugs with a higher false positive rate from unused definitions compared to VALUECHECK. Smatch detects one type of unused definitions: the return value of a function is unused. We refer the unused definition bugs detected by Smatch as Smatch-unused. We run Smatch on the evaluated software. However, Smatch-unused reports compilation error on all applications except Linux. Therefore, we only compare the result of Linux: Smatch-unused detects 28 real bugs compared to 154 real bugs detected by VALUECHECK, with a false positive rate of 81%. The bug number is lower and the false positive rate is higher than VALUECHECK due to two reasons: (1) It only detects unused return values among unused definitions. Besides,

due to inlining, some unused return values are inlined as unused assignments, thus are not detected by Smatch-unused. (2) It conducts analysis based on the AST parser instead of control flow analysis, so the analysis is not precise and has high false positives.

Comparison with Coverity Scan

Coverity is a static analysis tool that can detect defects in C/C++ projects, which is a commercial tool. We apply for its basic version Coverity Scan and evaluate it on the four projects. Coverity Scan two types of unused definition bugs: unused value and unchecked return value (unused return value is a subset). We call the bugs detected by Coverity Scan from unused definitions as Coverity-unused. Coverity-unused detects 170 bugs with a total false positive rate 62% from four applications. For Linux, though Coverity-unused detects more bugs, it **misses 35 bugs** detected by VALUECHECK. For the other applications, their commit history shows developers of some evaluated applications previously utilized Coverity and addressed its warnings, which explains why Coverity detects much less bugs. VALUECHECK can detect new bugs with lower false positive rate because: (1) Coverity-unused only detects unused assignment and unused return value, excluding other types of unused definitions (e.g. assigned but unused arguments). Besides, to avoid the huge number of unpruned results, it infers whether function return values need be used based on the percentage of used return values. If the function is only used once, it cannot correctly infer whether the return value should be used. Compared to Coverity-unused, VALUECHECK additionally considers authorship when deciding whether an unused definition should have been used, which is not limited by the number of function invocations. (2) Coverity-unused pruning does not consider any authorship information and code semantics, so it does not prune unused definitions that are intentionally left in the code, resulting in higher total false positives.

Table 2.6. Unused Definition Bugs Detected by Clang, Infer, Smatch, Coverity and VALUECHECK. VALUECHECK in total detects more bugs with lower false positives than other tools. *Report errors during analysis.

Tool	#Found Bugs/#Real Bugs/%Bug False Positive				
	Linux	NFS-g	MySQL	OpenSSL	Total
Clang	0	0	0	0	**0**
Infer -unused	−*	8/2/75%	45/9/80%	13/3/77%	**66/14/79%**
Smatch -unused	147/28/81%	−*	−*	−*	**147/28/81%**
Coverity -unused	157/56/64%	3/3/0%	4/1/75%	6/4/33%	**170/64/62%**
VALUECHECK	63/44/30%	22/18/18%	99/74/25%	26/18/31%	**210/154/26%**

Case Study: a bug detected by VALUECHECK but missed by other tools .

Figure 2.8 shows a bug example that fb-infer, Smatch-unused and Coverity-unused fail to detect. Since the variable `ret` is referred in `if(ret)`, all definitions of `ret` are regarded as used by fb-infer and Smatch-unused due to their inaccurate analysis. Besides, Coverity-unused does not report it as a bug because it fails to infer that the return value of `get_permset` should be checked since it is only invoked once. However, this actually is a real bug acknowledged by developers with security concerns of broken access control when invalid permission set is read. In fact, the definition in line 237 was previously used. But after author2 committed line 239, it became an unused definition. When an unused definition spans multiple authors, it indicates such bugs, which can be identified using authorship information.

It is worth noting that VALUECHECK is not a replacement of other tools but a complement. It focuses on precisely detecting bugs from cross-scope unused definitions. However, it does not detect unused definition bugs introduced by the same developers due to carelessness or other reasons. Existing tools report this type of bugs but with high false positives, which remains as an open problem to explore in the future.

```
235 acl_t fsal_acl_posix(...)
236 {
237   ret = get_permset(en, &pset);[Author 1]
238   ...  Unused Definition
239   ret = calc_mask(&allow_acl); [Author 2]
240   if(ret)                      [Author 1]
...}
```

Figure 2.8. A bug detected by VALUECHECK but not detected by fb-infer, Smatch-unused and Coverity-unused. The developer forgot to handle the return error status from `get_permset`, which would cause access control error if the acl entry is invalid.

Table 2.7. Effect of authorship and the DOK model in VALUECHECK. VALUECHECK detects a higher total number of bugs compared to other groups.

App.	#Detected Bugs from Top 20 Bugs					
	VALUECHECK	w/o Authorship	w/o Familiarity	w/o AC	w/o DL	w/o FA
Linux	**20**	14	16	20	19	20
NFS-g	**17**	2	16	17	16	17
MySQL	**20**	10	15	19	18	19
OpenSSL	**17**	2	15	17	16	17
Total	**74**	28 (-62%)	58 (-16%)	73 (-1%)	69 (-7%)	71 (-4%)

2.8.5 Authorship and Code Familiarity Effectiveness

Effectiveness of the Cross-Scope Authorship

To explore how cross-scope authorship can help distill unused definitions that are real bugs from programs, we remove cross-scope filtering from VALUECHECK and preserve all other components (w/o Authorship group). Then we report the top 20 bugs detected by the modified tool. The result is presented in the second column of Table 2.7. Compared to original VALUECHECK, the detected real bugs are much fewer, in total 28 bugs. This is because without cross-scope authorship filtering, the number of detected unused definition is much higher (2259 in total), for which pruning and ranking are insufficient to reduce false positives to an acceptable level.

Effectiveness of the DOK Model

To explore how effective the DOK model prioritizes bugs from detected unused definitions, we set up four groups: 1) (w/o Familarity) Remove the ranking from VALUECHECK. Select the first 20 cross-scope unused definitions detected by VALUECHECK from each application. 2) (w/o AC, w/o DL, w/o FA) Individually removing each factor from the code familiarity model and applying VALUECHECK to get 20 cross-scope unused definitions with the lowest code familiarity. Table 2.7 shows the number of bugs detected by the four groups. In total, VALUECHECK detects 74 existing bugs, 16% more than detecting without the familiarity model. Removing AC and DL factor decreases the total number of detected bugs and the precision of bug detection in the evaluated applications.

Bug Detection Precision of Different Cutoffs

We evaluate the precision of bugs detected by VALUECHECK with different cutoffs on report numbers in Figure 2.9. When VALUECHECK only reports the top 10 unused definitions with the lowest code familiarity from each applications, the precision of confirmed bug is the highest at 97.5%. With the increasing of the reported bug number, the precision of bug detection decreases, which indicates the relevance of code familiarity and the possibility of detecting bugs. From the result, it shows that the code familiarity model is effective in prioritizing real bugs.

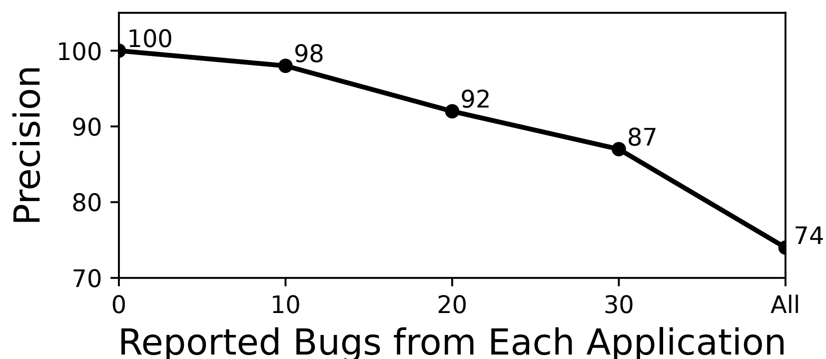


Figure 2.9. Precision of bug detection with different cutoffs after familarity ranking. VALUECHECK has a precision of 97.5% when reporting the top 10 detected unused definitions with the lowest familiarity from each applications.

2.8.6 Scalability of VALUECHECK

As shown in Table 2.8, for each application, the execution time of VALUECHECK on the whole application code base is under 30 min even for Linux with 27.8M LOC (we turn on `allmodconfig` compilation flag). Further, when integrating VALUECHECK into the code testing and analysis process, this overhead could be reduced by running the analysis incrementally, i.e., only on the changed functions and the affected files in a commit. We do the incremental analysis on the first 20 commits after 2022 on each application. The average execution time on each commit is under 5s for all the applications we evaluate. It empirically demonstrates that VALUECHECK can be integrated into the code development with an acceptable time cost for a large-scale code base.

Table 2.8. Scalability of VALUECHECK.

Application	#LOC	Time	Incremental Time
Linux	27.8M	28m12s	4.6s
NFS-ganesha	315K	2m13s	2.2s
MySQL	1.7M	16m32s	2.6s
OpenSSL	1.5M	3m54s	1.9s
Total	**31.3M**	**50m51s**	**11.3s**

2.9 Limitations and Discussion

2.9.1 Limitation

VALUECHECK demonstrates to have better accuracy in detecting bugs from unused definitions compared to the state-of-the-art. However, VALUECHECK still has limitations on having false positives. For example, some unused definitions are just legacy code or debugging, which could be further pruned by analyze the commit history and comments. But this will incur much more overhead so we do not prune this type of false positive. Besides, our exploration of cross-scope unused definitions is not an assertion that unused definitions which do not cross author scopes are not bugs. The assumption we make in the design of VALUECHECK is based on

our preliminary experiments (Section 2.3.1). Whether there are other types of unused definitions that are prone to be bugs is another problem to be explored in the future.

2.9.2 Alternatives of the DOK model

We use the DOK model because it is one of the state-of-the-art model and considers accessible factors from public code repositories. However, some alternative models could be considered, which may be less accurate but do not require the original developers to participate. The EA model [104] models the type of commits made by a developer, such as bug fixes, refactoring, and new functionality and assigns familiarity score to them differently. [112] can automatically infer developer expertise through their time to fix detects in commit histories. Another model [19] considers activities like comment and review which may also increase familiarity with the code. It is possible to replace with alternative familiarity models in VALUECHECK.

2.10 Related Work

Bug Detection. A range of research has been proposed to detect bugs and vulnerabilities [183, 150] with formal method [87, 36, 152], static analysis [93, 172, 88, 23, 69, 185, 184] and automatic testing [83, 139, 151]. Some work directly infer rules from source code and picks up deviant outliers, which could be potential bugs [159, 100, 180, 94, 41, 70, 175, 187]. Compared to them, our work detects a potential bug pattern, cross-scope unused definitions, which previously is regarded as redundant code, and demonstrates that it is possible to detect bugs from the unused definitions with low effort.

Code Familiarity. Previous work measure how familiar a developer is with a code snippet in a project by metrics include change history [96, 104, 68], file dependency [101], authorship [104], and interaction information [161]. [47] proposes the Degree-of-Knowledge (DOK) model to achieve a better measurement. In [168], the authors reveal the relationship between bug fixes and developer familiarity. Our work borrows the existing literature in this field to help reduce developers' efforts under time pressure. Some work rank the reports from static analysis tools

with other methods like AdaBoost [137], which is orthogonal to our ranking method.

Inconsistent Code. Researchers studied the inconsistency in code in some past literature [143, 25, 162]. They focus on detecting unreachable code and removing them to reduce static analysis and coverage analysis effort, but not for bug detection while our work focuses on bug detection. Some previous work illustrates that certain code structures can indicate deeper problems in software design [45, 46, 49, 149], etc. Code smells are also related to the code quality and defect rate [79, 124, 64, 176, 153] of programs, which inspire our work to detect bugs from bad code patterns (unused definitions).

Dead Code Elimination As we already discussed in Section 2.2, eliminating unused definitions of registers has been regarded as a low-level code optimization [76, 73, 34, 144, 85, 167, 58, 30, 113, 136]. Unlike these previous works that simply remove all the redundant code, we propose to treat cross-scope unused definitions as potential bugs.

2.11 Acknowledgments

Chapter 2, in full, is a reprint of the material as it appears in the Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys'24). Zhong, Li; Xiang, Chengcheng; Huang, Haochen; Shen, Bingyu; Mugnier, Eric; Zhou, Yuanyuan [186]. The dissertation author was the primary investigator and author of this paper.

Chapter 3

FENCEHOPPER: Detect Vulnerabilities by Client-Side Code Mutation

3.1 Introduction

3.1.1 Problem: Over-relying on Security Checks in Web Client

Many software deploy a client-server architecture. They provide users with responsive user interfaces in the client and offload the major services on the server side. Among them, one special type of client-server applications is the web application, which normally have the client-side components written in JavaScript, send the client-side *source code* to users, and execute them by JavaScript engine inside a web browser [92]. From a system point of view, this characteristic sets web applications apart from other client-server applications such as Zoom, which typically involve sending executable binary files or other forms of executables to users to install as application clients on laptops or desktops.

To enhance responsiveness and alleviate server loads, developers conduct various checks on user requests within the web client before forwarding them to servers [155, 51]. These checks encompass crucial aspects such as authorization [107], identity verification [71], and validation of user input [147].

Despite the advantages of web client-side checks in terms of saving round-trip time and reducing server load [51], it is imperative to recognize that they cannot serve as a guar-

anteed security measure. This limitation arises from the susceptibility of web client code to manipulation by malicious users, enabling them to circumvent checks and dispatch malicious requests to servers. Regrettably, the misconception of a ’false sense of security’ often arises within development teams which results in a tendency to overlook server-side checks, creating vulnerabilities that malicious attackers can exploit to compromise services and inflict severe damage.

Surprisingly, instances of such blind spots in security practices, attributed to the false sense of security arising from client-side checks, are not infrequent. A notable case in point pertains to users successfully circumventing paywalls on news websites by altering the JavaScript code, thereby gaining access to content without the requisite authorization. Furthermore, Table 3.1 enumerates several examples from prominent web services where the omission of robust server-side checks led to significant consequences.

Table 3.1. Missing server-side check vulnerabilities affect billions of users in popular websites.

Website	Missing Server-Side Check	Consequence
Twitter	Missing password checks on email update [60]	Account Takeover
MTN	Missing passcode checks [62]	Identify Theft
CoinBase	Missing captcha checks [59]	Account Probing
StarBucks	Missing mobile verify checks [61]	Account Takeover
Facebook	Missing user id checks [131]	Account Takeover

Despite the widespread recognition of the importance of server-side checks, missing server-side checks remains prevalent in web applications. There are several possible reasons. First, developers may be inconsistent when implementing checks, especially when they are overwhelmed by the number of checks in a large code base. Second, rapid-evolving front-end frameworks such as Vue [177], React [99], and Angular [52] can give developers a false sense of security, leading them to rely solely on client-side checks. Third, when knowledge is not effectively shared among developers, inconsistencies between client-side checks and server-side checks can also arise. The rapid release cycle also makes it challenging for the developers to ensure comprehensive testing coverage of entire web applications.

3.1.2 Existing Solutions

Given the profound implications of overlooking server-side checks, significant efforts have been dedicated to addressing this issue. Previous research has primarily concentrated on the direct generation of web requests to the server side as a means of identifying and exposing server vulnerabilities. Examples of such efforts include bypass testing [118, 116, 117] and parameter tampering [28, 97]. While effective in simpler web service scenarios, these techniques exhibit limitations when applied to complex web applications characterized by comprehensive client-side computational logic. Within the context of complex web applications, the parameters transmitted to the server side often entail intricate semantics and logic relationships. Complicating matters further, certain web applications also rely on interactions with multiple third-party services before forming and transmitting requests to servers. Unaware of the nuanced semantic and logic relationships inherent in request parameters, these solutions can generate mostly random requests that may easily fail initial server-side checks, thereby cannot expose deeper server vulnerabilities.

Furthermore, due to the exponential nature of the request parameter space, testing potentially unveils tens of thousands of parameter combinations to expose latent server issues. To enhance testing efficiency, recent research endeavors optimize the parameter space exploration by leveraging code analysis of client-side JavaScripts [56, 165]. These approaches prioritize optimizing test efficiency by reducing the number of test cases, rather than concentrating on augmenting effectiveness in uncovering more profound and elusive vulnerabilities beyond simple, initial parameter sanity checks. Moreover, since these approaches rely on *static* analysis, they cannot handle complex web applications where a portion of client-side computation is performed by third-party services, such as Google Analytics, location services, token services, and time zone services. Values obtained from these third-party services may be included as parameters in the request to servers. In our experimental evaluation, the experiments showed that these methods failed to detect many real-world vulnerabilities due to their disadvantages in handling complex requests.

An alternative strategy to thwart bypass attacks involves the implementation of CSRF tokens [123], double-submit cookies [122], and customized headers [121]. These defensive measures effectively mitigate the risk of request forgery. However, their efficacy is compromised in situations where missed server-side checks pertain to initial authentication even before the client obtains the token or cookie, as exemplified in the news website paywall bypass scenario. Additionally, these methods still rely on server-side security checks on those tokens. If servers do not have checks or the checks are not comprehensive, attackers can still exploit such vulnerabilities even with tokens or double cookies.

The common practice in industry relies on white hat hackers to manually detect and report them to bug bounty programs [75, 98, 53, 164], as documented in their blogs [106, 105, 110, 134, 119, 148]. However, manual inspection, while useful, has inherent limitations—it can only cover a finite number of checks in web applications and is also expensive and resource-intensive. As the size of client-side code escalates, sometimes surpassing megabytes [81, 141], relying solely on manual inspection becomes increasingly impractical to ensure comprehensive coverage. This challenge is particularly prohibitive for non-technical organizations, such as schools, hospitals, or non-profit groups, which often manage substantial amounts of sensitive user data.

3.1.3 Our Approach

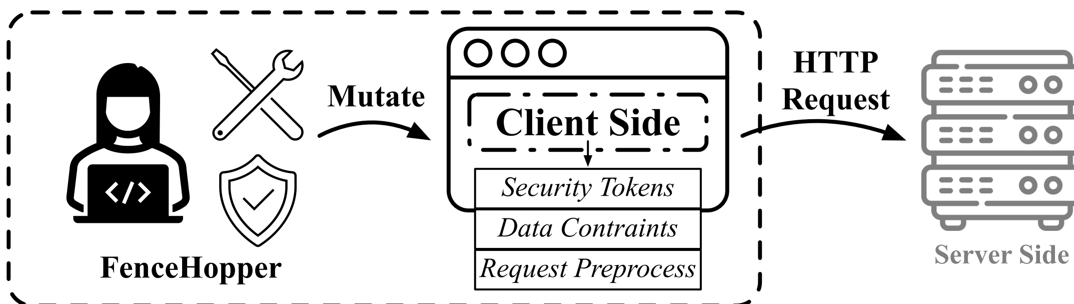


Figure 3.1. Overview of FENCEHOPPER. FENCEHOPPER mutates the client-side code to bypass client-side checks to detect if such checks are missed by servers—a common overlook by developers due to false sense of security from client-side checks.

Drawing inspiration from the observed false sense of security stemming from client-

side checks, this paper introduces a novel technique named FenceHopper[1] designed to autonomously and systematically mutate client-side JavaScript code. The objective is to identify and bypass checks, thereby detecting potential absences of analogous checks on the server side, as depicted in Figure 3.1. As it is too expensive and unnecessary to mutate every branch in client-side JavaScript code, FenceHopper first leverages static code analysis to identify those related to security and access control checks. Moreover, forcefully taking an alternative branch can result in some uninitialized and inconsistent variable values. FenceHopper leverages several techniques to address this challenge to ensure the client-side code does not fall through due to premature failure before sending the request to servers.

Our technique has several complementary advantages over previous solutions that directly manipulate and fuzz requests to servers:

1) FenceHopper leverages existing computation and business logics in client-side code to generate requests that are semantically consistent to expose deeper and harder-to-expose server vulnerabilities, including those shown in Table 3.1.

2) FenceHopper is *dynamic* in terms of executing the mutated client-side code. It can work effectively even if the client-side code requires communication/interactions with web services to maintain server-side states and some parameters were generated remotely by third-party services.

3) FenceHopper operates automatically, and requires little manual effort. It can systematically bypass each security-related check within the client-side JavaScript, assessing whether corresponding server-side checks have been neglected by developers.

We evaluate the effectiveness of FenceHopper through comprehensive experiments conducted on the **top 300 websites** identified within the Tranco dataset. Our framework successfully identifies **48 novel vulnerabilities**, among which 5 are categorized as critical access control vulnerabilities, posing a risk of unauthorized access to accounts of **over 20 million users**. To mea-

[1]Named to metaphorically resemble a hopper (representing the attacker) that leaps into the garden (symbolized by the server) by breaching the fence (symbolized by the client-side code).

sure the coverage of FENCEHOPPER, we manually curate a dataset comprising 14 documented cases of missing server-side checks sourced from HackerOne and Bugcrowd. FENCEHOPPER successfully identifies 10 of these issues. Notably, over 60% of the vulnerabilities detected by FENCEHOPPER are missed by state-of-the-art tools.

3.2 Client-side Code Mutation

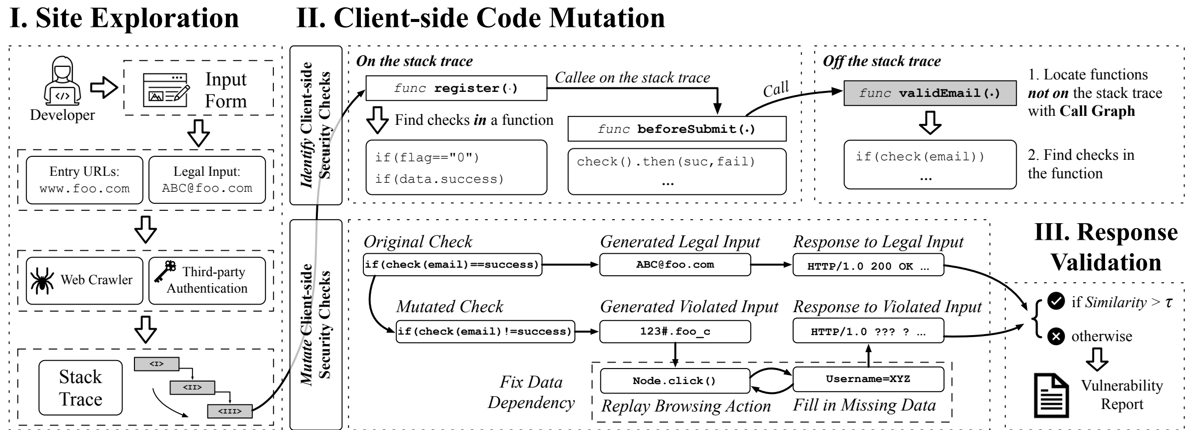


Figure 3.2. Overall Workflow of FENCEHOPPER.

Table 3.2. Branches on the Client Side of Popular Websites.

Website	#Branches in Client-Side Programs
Facebook	2744
Instagram	1434
Amazon	1045
Yelp	838
Youtube	10586

Mutating client-side checks to detect missing server-side checks presents unique challenges. First, we observe many if-else branches in client-side code, many of which were non-security related, from libraries and frameworks (Table 3.2). Mutating all if-else branches unnecessarily incurs high overhead. Also, asking developers to annotate requires a lot of human effort. Therefore, we use static analysis combined with execution traces and dynamic testing to overcome the highly dynamic feature of JavaScript. (Section 3.2.1)

With the extracted security checks, the next challenge is how to generate inputs that can violate the client-side checks efficiently. Previous literature [50, 57] generates test inputs that trigger specific branches. However, due to dynamic language features of JavaScript, end-to-end input generation could suffer from the analysis imprecision. In our solution, rather than mutating the entire end-to-end input, we mutate the intermediate input on demand when they are used in relevant checks. The approach allows us to generate inputs with less cost while guaranteeing that the check is violated. By employing code mutations to assign testing values to condition variables, FENCEHOPPER effectively generates refined inputs corresponding to the negation of check conditions. (Section 3.2.3)

After mutation, FENCEHOPPER needs to ensure crash-free execution so that the client-side code can send the expected requests for testing the server. However, as the execution is forced into another branch even though the data should not pass the check, it can result in missing data objects on the forced execution path under certain circumstances. This can fail client-side programs without sending requests to the server side. For example, the forced branch may attempt to access data that should have been available or select page elements that should have been present on the web page if the conditions were satisfied. To address this issue and allow the execution to proceed until the request is dispatched, FENCEHOPPER fixes the missing data object, which is achieved by assigning possible values to the objects, ensuring that the execution can continue without missing data failures. The evaluation results in Section 3.4.3 validate the necessity of fixing the missing data. (Section 3.2.4)

3.2.1 Identify Security Checks in Client-Side

Identify Security Checks within a Function. To identify security checks, we observe that the majority of the security checks typically diverge into two branches: If the check fails, it would raise errors or alerts. Otherwise, it would proceed to another path that dispatches a request. FENCEHOPPER leverages this observation by analyzing stack traces collected during the website exploration stage (see Section 3.3). These stack traces are generated when a request is

```
Algorithm 1: Identify and Mutate Security Checks
Function MutateCode(stack_trace):
    Input: stack traces collected from website exploration.
    Output: mutants of client-side code to test each check.
    for F ∈ stack_trace
        T = generateAST(F)
        TraverseVisitor.visit(T)
        // Interprocedural Security Checks
        for F_callee ∈ getCallee(F)
            T = generateAST(F_callee)
            TraverseVisitor.visit(T)
        for c ∈ TraverseVisitor.checklist
            mutants.append(MutateVisitor.visit(T, c))
    return mutants
Function TraverseVisitor::visit(node):
    Input: AST node traversed by TraverseVisitor.
    // Add if nodes and promises into the security check list.
    if isIfExpr(node) && (raiseError(node.left)
    || raiseError(node.right))
        self.checklist.add(node)
    end
    if isPromise(node)
        self.checklist.add(node)
    end
Function MutateVisitor::visit(node, c):
    Input: node: node traversed by MutateVisitor.
            c: the security check to mutate and test.
    // Mutate the security check and generate inputs by inserting
    assignments to variables
    if isIfExpr(node)
        if node = c
            if raiseError(node.left)
                input_assignment = solve(node.test)
            end
            else
                input_assignment = solve(¬node.test)
            end
            node.test = ¬node.test
            node.insertBefore(input_assignment)
        end
        // Mutate nested and relevant checks
        if isParent(node.left, c) ||
         raiseError(node.right)
            node.test = true
        end
        if isParent(node.right, c) ||
         raiseError(node.left)
            node.test = false
        end
    end
    else if isPromise(node) & node = c
        switchCallback(node)
    end
```

Figure 3.3. Identify and Mutate Security Checks.

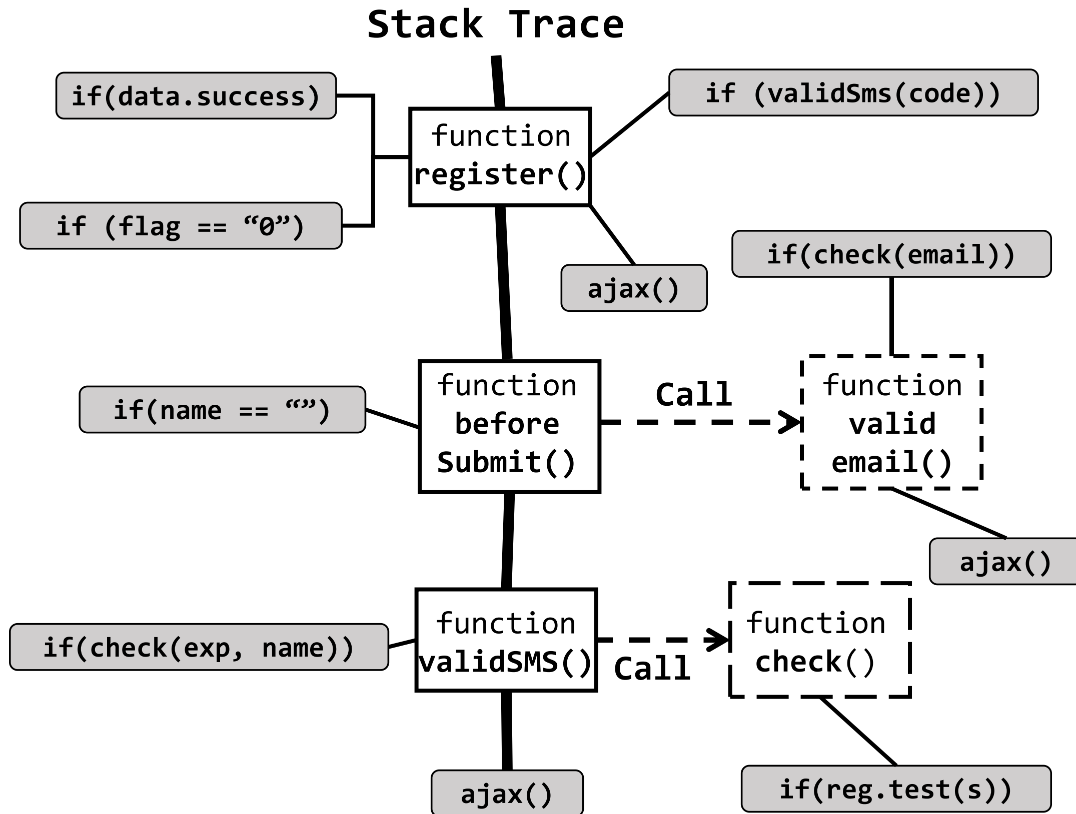


Figure 3.4. Identify Intra and Inter-Procedural Security Checks. Functions in heavy solid line boxes are extracted directly from the stack traces, which are the 'stem' of our analysis. We expand the relevant function set by checking the callees of these functions, which are in the dotted line boxes.

dispatched, allowing us to address the highly dynamic features of client-side code like dynamic code evaluation and dynamic typing. As shown in Algorithm 3.3 and Figure 3.4, FENCEHOPPER starts by extracting the functions from the stack traces. These functions are the 'stem' of our analysis, depicted as heavy solid boxes in Figure 3.4. For each function in stack traces, FENCEHOPPER traverses the abstract syntax tree (AST) of that function to identify two types of security checks: (1) `if` check with one branch specifically designed to raise errors or alerts. (2) Reactions to promises, which can be regarded as condition checks that execute different functions based on whether the promise is fulfilled or rejected. If any of the callbacks raise errors or alerts, FENCEHOPPER considers it as a security check. We add them to the security check set TraverseVisitor.checklist.

In FENCEHOPPER, we include the following patterns as raising errors or alerts: (1) Throw exceptions using the `throw` statement. (2) Raise alerts with the `alert()` and `window.alert()` methods. (3) Invoke other error-raising method with keywords of 'error', 'exception', and 'alert'. For example, modify HTML DOM to show the error messages. We do not include `switch`, `while`, and `for` loops because security checks typically involve an allow-or-deny decision, which is not aligned with the control flow of these statements.

Identify Security Checks Inter-procedurally. Dynamic traces are effective in analyzing client-side JavaScript programs due to its dynamic nature [138]. However, it has limitations in terms of coverage as discussed in existing literature [20]. It can only cover functions that appear on the stack traces. To address this limitation and increase the coverage of FENCEHOPPER, we incorporate static analysis based on the stack traces. In addition to the functions in the traces, FENCEHOPPER collects functions that are called by these functions. These functions are identified by building call graphs of the candidate functions. We similarly identify the security checks within these callee functions.

Handling Obfuscation and Minification. In FENCEHOPPER, we do not specifically handle the cases where the client-side code is obfuscated or minified. However, with deobfuscation techniques on JavaScript [158, 33], FENCEHOPPER can provide more meaningful analysis to assist developers and testing teams in identifying and addressing security vulnerabilities effectively.

3.2.2 Mutate Security Checks in Client-Side

Mutation Strategy. There are numerous possible combinations of check mutations. For a single user operation that triggers N security checks, there are at least 2^N mutation combinations, which is exponentially increasing. To balance time and resource requirements, we propose focusing on one security check at a time and testing whether the corresponding check exists on the server side. This helps pinpoint the missing check on the server side. The limitation of this strategy is discussed in Section 3.6.

Mutate Check Predicates. Once a specific client-side check is selected, FENCEHOPPER mutates the check to allow data that violates it to pass through. One option for mutating checks is to mutate it into a constant boolean so that for any input data, it will proceed. However, we do not adopt this approach because it will still send requests when FENCEHOPPER does not successfully generate an illegal input, which results in false positives. Therefore, for `if` checks, to transform a specific security check node, FENCEHOPPER identifies which branch of this check is the error branch based on the error-raising patterns (`raiseError` function in Algorithm 3.3). Then it mutates the condition node.text of the tested check into its negation ¬node.text. For checks embedded in promises, FENCEHOPPER mutates the promise by reverting the rejected callback with the fulfilled callback (`switchCallback` in Algorithm 3.3). When the promise is rejected, it will execute the original fulfilled callback that assumes the check is successful.

Nested Checks and Relevant Checks. If the check c we test is nested inside branches of other checks, we need to guarantee that the parent checks will proceed to the branch that contains c. Therefore, FENCEHOPPER mutates predicates of checks that dominate c into the boolean value that guarantees the execution on the branch containing c. Another type of checks must be mutated, which could block the generated input from being sent to the server within a request. For example, in checking the format of phone numbers, there are two checks (1), (2):

```
(1) if ( !( mobile.len > 9 ) ) { // The tested check. Its condition is negated
      alert("Too long phone number!");
      return;
  }
  // Relevant check. Need to mutate the condition into false
(2) if (Test("[0-9]{9}", mobile)==false) {
      alert("Invalid phone number format!");
      return;
  }
  sendRequest(mobile);
```

Now assume FenceHopper is testing the first check, which verifies whether the phone number is longer than nine digits. After mutating the check condition as `false`, an empty mobile number `""` can bypass the first check. However, since the second check validates whether the mobile number is a nine-digit string, a mobile number longer than nine digits would not pass this check, which ends up not being able to send the request to the server side. To tackle the scenario where other checks can counteract the effect of the mutated check, FenceHopper mutates predicates of other security checks to make them unconditionally proceed to their success branch and not block the request dispatch. In this above example, the second check condition will be mutated to `false` so that even the mobile is longer than nine digits, it will not fail the second check. As a result, it can be sent to the server for testing whether the server can handle extremely long strings.

Avoid Infinite Loop and Recursion. As a programming practice, developers should not rely on raising errors or exceptions to end loops or terminate recursive functions [146]. In ideal cases, even after predicate mutation, client-side programs would not encounter infinite loops or recursive calls. However, to ensure safe testing in FenceHopper and prevent program crashes due to infinite loops or recursions, a timeout is set to avoid such scenarios during the testing of each mutated check.

3.2.3 Generate Data Input Based on Checks

As shown in Algorithm 3.3, FenceHopper generates test inputs by negating the predicate node.test of the tested check node into ¬node.test. It then solves the resulting constraints using an SMT solver. FenceHopper has several advantages in utilizing an SMT solver in its problem context: (1) Security checks are simpler than functionality checks, involving less complex data structures. (2) The computation of the SMT solver remains limited as we collect constraints from only one execution trace at a time, resulting in efficient constraint solving without significant overhead.

To dispatch requests that violate client-side checks, FenceHopper focuses on mutating

the input data just before the check rather than manipulating the end-to-end input. For instance, when testing the emptiness of a mobile number, FENCEHOPPER doesn't need to mutate the user input in the text box. Instead, it can directly mutate the variable `mobile` that the client-side check accesses. We refer to these mutated variables as *check inputs*. There are four sources of check inputs:

- Mutable objects in the client-side program. FENCEHOPPER mutates the generated test input by instrumentation.
- Immutable objects in client-side programs. FENCEHOPPER mutates them by reassigning the variable to new objects. For example, a `String` variable `username` will be assigned with a new `String` object with a length of 3 so that the constraint `username.length==3` is satisfied.
- Function Call. In the client-side check, a function call can be used to send requests to the server to validate a security requirement. For such function calls as check inputs, FENCEHOPPER replaces them with mutable variables and assigns values to these variables. FENCEHOPPER needs to test the case where the server-side check result is not passed, which occurs when the client side does not perform the check. For example, the security check:

```
// A request is sent to the server in checkCode()
if (checkCode(code)=="true") {...}
```

  is transformed into:

```
checkCode_code_ret = "false"
// Condition is negated
if (checkCode_code_ret!="true") {...}
```

which is a violation of the original client-side check because the check is bypassed without requesting any server-side validations. For promises, we assign the parameters with random strings so that the promises cannot be fulfilled.

3.2.4 Fix Missing Data after Code Mutation

After code mutation, the client-side program is forced to execute a previously infeasible path. This can lead to the problem of missing objects, where the data objects expected along the forced path are not presented or initialized with the provided input. This can result in program crashes and failure to dispatch the testing request. Here is an example of a missing data object:

```
// e = {"error"}
e.valid = false;
if ( ! e.valid) { // Condition is negated
    userSignIn(e.user.userId)
    // Missing object e.user
} else{
    throw new Error("Cannot log in!")
}
```

In this example, the condition `e.valid` is mutated. As a result, the execution is forced to proceed even when the `e.valid` is false, which means the user has entered an invalid email. The path that passes this check then attempts to use `e.user.userId` to log in, which is looked up from the server side based on a valid user email. However, when this path is forced to execution, the user email is invalid and the data object `e` does not contain the `user` object. This leads to the missing data error. Additionally, JavaScript may also read DOM elements from the web page using selectors, which may also be missing or unavailable after code mutation.

To handle missing data objects, FENCEHOPPER fixes the missing data on demand by inserting an assignment statement in the client-side code before the location where the missing data error occurs, ensuring crash-free execution. In JavaScript, where most data objects are mutable and dynamic typing, we directly assign values to undefined objects to resolve the error. For DOM elements, we inject the DOM element based on the provided DOM selector. We apply two strategies to assign value to the newly created data and DOM elements: (1) Complex Fix:

For the created object, we attempt to fill it with meaningful values based on the variable and attribute names. We use several keywords to infer the data type from the variable and attribute names: 'email', 'mobile'/'phone', 'id', 'user', 'number', and 'token'. For example, for 'email', we try all the test emails. For other data objects that do not match any keywords, we apply simple filling. (2) Simple Fix: We assign an empty string to the missing data object. For missing DOM elements, we create them with no inner text and the minimum set of attributes to match the selectors.

These two strategies allow the client-side code to continue executing after the mutation performed by FENCEHOPPER with minimal effort but effectively increases the chance of detecting vulnerabilities. For the code example we provided above, FENCEHOPPER inserts the assignment `e.user={},e.user.userId=12;` before the crash location. The value 12 is a randomly generated number.

3.3 Implementation

FENCEHOPPER is mainly written in `Python` using `Selenium`. The workflow of FENCEHOPPER is divided into three phases:

Site Exploration. FENCEHOPPER navigates target websites, collecting stack traces and inputs for analysis. It employs `WebDriver` for page interactions, executes scripts to load hooks, and logs browser activities. It hooks into functions like `fetch` and `XMLHttpRequest.prototype.send` to log stack traces, performing actions like form submissions and hyperlink clicks. It also identifies third-party login interfaces for post-authentication page detection.

Client-Side Code Mutation. After analyzing stack traces, FENCEHOPPER identifies client-side checks for mutation, generates and traverses the abstract syntax tree (AST) of each function. It creates a call graph [43] and mutates the AST nodes for input generation and missing data fixes. The tool uses `Z3` [182] for input generation to satisfy mutated predicates and logs to validate check execution.

Response Validation. The mutated code is loaded into the browser, replacing the original JavaScript. Actions triggering the mutated checks are replayed to compare responses. Differences in status codes or significant response body disparities measured by Levenshtein distance indicate potential vulnerabilities. FENCEHOPPER then generates reports detailing the actions, mutated checks, and responses before and after mutation for users to verify the vulnerabilities.

3.4 Evaluation

The evaluation aims to answer the following questions:

(1) Can FENCEHOPPER successfully detect missing server-side check vulnerabilities in real-world scenarios? (§ 3.4.1) What bugs does FENCEHOPPER detect that the state-of-the-art tools fail to detect? (§ 3.4.2)

(2) What is the contribution of each component? (§ 3.4.3)

(3) How comprehensive is the coverage of FENCEHOPPER in detecting checks and vulnerabilities? (§ 3.4.4)

(4) What is the time and effort involved in adopting and using FENCEHOPPER? (§ 3.4.5)

(5) What are the impacts of the missing server-side check vulnerabilities detected by FENCEHOPPER? (§ 3.5)

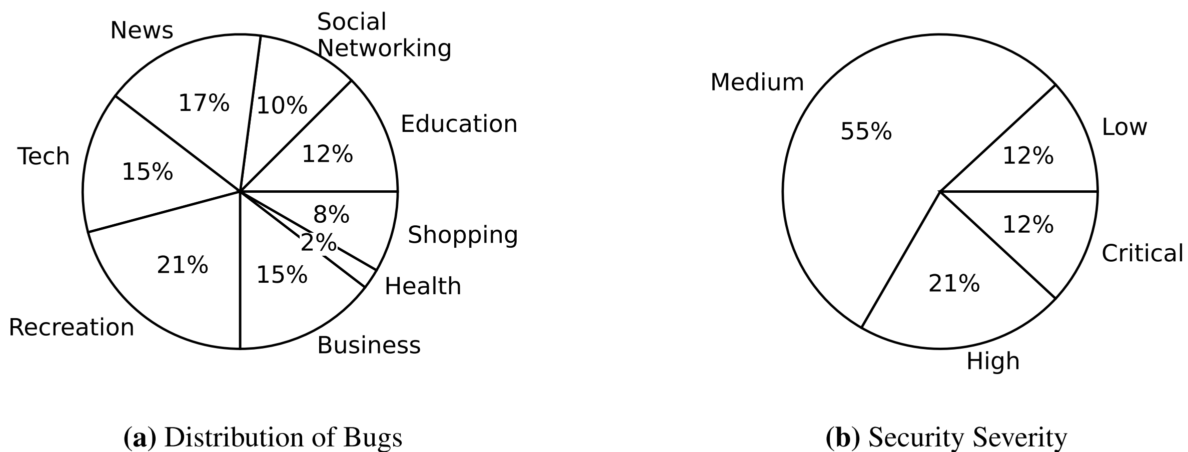
3.4.1 Detect New Vulnerabilities

Methodology. To evaluate the capability of FENCEHOPPER to detect new vulnerabilities, we select the top 300 websites from the Tranco dataset [129] as our evaluation targets. For each website, we manually identify the entry URLs of login, sign up, and user settings, then provide them to FENCEHOPPER as input. Then we run FENCEHOPPER to automatically explore the website starting from the entry URLs and detect potential vulnerabilities. After execution, FENCEHOPPER generates a report including the detected missing server-side checks. We analyze the results and report to the developers.

Results. We spent 42.6 hours testing the 300 websites with 59 missing server-side checks

reported by FENCEHOPPER. Among them, 48 are real vulnerabilities from 38 different websites, which indicate the method is applicable and effective for a wide range of websites. During the detection, FENCEHOPPER analyzes 413,539 lines of client-side code, totaling 494 MB in size. The total users affected by the vulnerabilities exceed 20 million. The detected vulnerabilities include 11 access control vulnerabilities, 13 broken data integrity vulnerabilities, 2 broken business logic vulnerabilities, and 23 legal agreement violation vulnerabilities. A detailed security analysis is provided in Section 3.5, and an example is shown in Section 3.5 (full list in the anonymous repo [18]).

We categorize the vulnerabilities based on their security severity, as shown in Figure 3.5b. Approximately one third (16 vulnerabilities) are categorized as critical or high security severity, which can result in consequences like account takeover, private data leak, and bypassed access control. Over 80% (42 vulnerabilities) are of medium or higher severity. Overall, the vulnerabilities detected by FENCEHOPPER are security issues with a severe impact on a wide range of users. We further categorize the vulnerabilities based on the types of websites in which they were found. Figure 3.5a illustrates that FENCEHOPPER detects most missing server-side check vulnerabilities in recreation, news, and technology websites.



(a) Distribution of Bugs

(b) Security Severity

Figure 3.5. Vulnerability Categories.

False Positives. FENCEHOPPER has a false positive rate of 18.6%. There are two reasons for the 11 false positives: 9 false positive cases are caused by the limitations of response validation,

which can only determine whether the server responds differently, but it cannot differentiate between different failure pages. As a result, FENCEHOPPER may mistakenly identify these cases as missing server-side checks. 2 false positives occurred due to fake success responses from the server side. Although FENCEHOPPER successfully bypassed the client-side checks and received a success page, it failed to perform the intended operation. For example, on a business website, FENCEHOPPER bypasses the client-side check on password requirements and registers an account successfully based on the server's response. However, upon manual verification, it was discovered that the registered username and password could not be used to log into the website.

3.4.2 Comparison to Other Tools

Position of FENCEHOPPER. We aim to delineate the position of FENCEHOPPER in comparison to existing tools. Established tools such as BurpSuite [130] and ZAP [160] already offer vulnerability detection within their functionalities. The vulnerabilities they detect can be categorized as follows: (1) Missing client-side checks, which include DOM-based link manipulation, HTML5 storage manipulation, potential Clickjacking based on frameable responses, etc. (2) Leaked data on the client side, such as exposed email addresses, `robots.txt` files, user agent-dependent responses, and session tokens embedded in URLs. (3) Risky configurations, such as inadequate TLS certificate settings, cacheable HTTPS responses, and cookies lacking HttpOnly flags. (4) Missing server-side checks, such as cross-site scripting (XSS) and SQL injection vulnerabilities. FENCEHOPPER serves as a supplement to the fourth category. However, it adopts a fundamentally different approach to vulnerability detection. Rather than focusing on a specific type of data injection, FENCEHOPPER detects vulnerabilities by mutating client-side checks related to business security logic. Therefore, vulnerabilities detected by FENCEHOPPER are orthogonal to those addressed by existing tools in the fourth category.

Comparison of Detected Vulnerabilities. ffuf [67] is an open-sourced fuzzing tool specialized for web fuzzing with over ten thousand stars on GitHub, which proves its widespread acceptance.

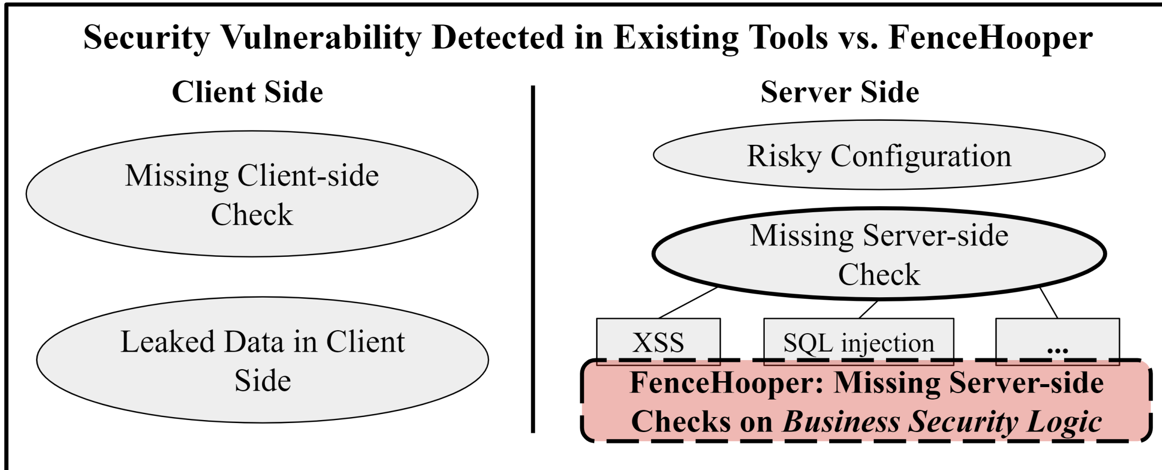


Figure 3.6. FENCEHOPPER Supplements Existing Tools. It complements the vulnerability category of 'missing server-side checks'. The bugs detected by FENCEHOPPER are non-trivial and orthogonal to other vulnerabilities in this category such as cross-site scripting (XSS) and SQL injection.

We applied ffuf on the websites where we detected vulnerabilities with FENCEHOPPER. To make sure the detection accuracy is not affected by the effectiveness of the crawler as observed in previous studies [42], we directly provide the vulnerable endpoints to ffuf and use payloads from dirbuster [163] and jbrofuzz [133] lists. Table 3.3 shows that ffuf can only detect 21% of bugs detected by FENCEHOPPER. Since fuzzing does not consider the parameter structures such as time formats and special strings, the search space could be extremely large and sometimes result in blocking by websites for 'Too Many Requests'. Only testing requests with simple type parameter values could be triggered by plain fuzzing. To evaluate fuzzing enhanced by client-side

Table 3.3. Comparison to Other Tools. FENCEHOPPER detects vulnerabilities that other state-of-the-art tools fail to detect.

Tool	FENCEHOPPER	ffuf [67]	ffuf+graybox	BurpSuite Pro [130]
#Detected Vulnerabilities	48	10 (21%)	16 (33%)	17 (35%)

code analysis, we evaluate gray-box fuzzing. However, the previous work on gray-box web fuzzing either do not provide access to their tools or require instrumentation on the server side, which is not feasible for 'in the wild' testing. Therefore, we combine the test input generation of FENCEHOPPER with ffuf to evaluate gray-box fuzzing. ffuf+graybox detects 33% bugs

detected by FenceHopper. It misses bugs due to: (1) 26 vulnerable endpoints include security tokens in the HTTP requests such as nonce tokens, and the data from third-parties to prevent replay attacks, which are inaccessible to the fuzzing tool. (2) 4 cases have encoded inputs such as appended digest fields in the HTTP body, which require manual effort to craft in fuzzing. (3) 3 vulnerabilities have business logic constraints that fail to be captured by fuzzing tools automatically.

Additionally, we included results from BurpSuite Pro [130] to explore the overlap of other tools in bug detection with FenceHopper. We choose BurpSuite Professional, a popular web penetration testing tool among security testers. With the default settings, it detects 35.4% bugs. While Burp Suite Pro does not directly detect missing server-side checks, it detects untrusted data injection like XSS and SQL injection, insecure direct object references, and broken access control, which is the manifestation of missing server-side checks. The comparison indicates that FenceHopper can detect complex bugs that other tools could miss.

3.4.3 Detailed Evaluation of Each Component

In this section, we provide an in-depth evaluation of each component to demonstrate its effectiveness in FenceHopper.

Code Mutation. The code mutation component of FenceHopper aims to automatically identify security checks in web applications, but it may introduce false positives. To evaluate the false positive rate of code mutation, we randomly sampled 100 checks identified by FenceHopper from the 300 websites as potential security checks. These checks are distributed across 98 websites. Among the sampled checks, 93 are indeed security checks, resulting in a true positive rate of 93%. Based on null hypothesis testing, the false positive rate in code mutation is 15% with 95% confidence.

Fix Missing Data. To evaluate the necessity of data fix post code mutation, we conduct an ablation experiment: We tested two FenceHopper variants on 300 websites: (1) FenceHopper without the complex fix but with simple data fix (FenceHopper-Complex Fix), and (2) Fence-

HOPPER without any data correction, omitting tests after mutation failures (FENCEHOPPER-Data Fix). The results shown in Table 3.4 reveal that the missing data fix in FENCEHOPPER enables 7% more testing requests to be sent to the server side. This increase led to 9 additional detected vulnerabilities, a 19% rise compared to the version lacking data fix. Furthermore, the complex fix identified 6 high-level security vulnerabilities, including one password check bypass and five email one-time passcode bypass cases, compared to the simple data fix.

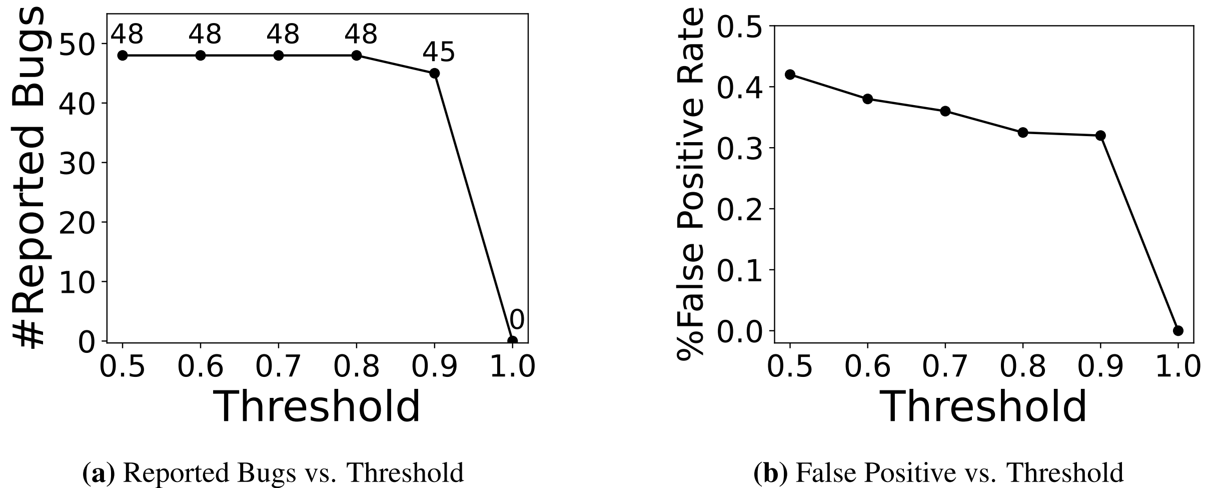
Table 3.4. Effectiveness of Missing Data Fix in FENCEHOPPER. Missing data fix enables detection of 9 additional vulnerabilities.

Tool	FENCEHOPPER	FENCEHOPPER -Complex Fix	FENCEHOPPER -Data Fix
#Detected Vulnerability	48	42 (-13%)	39 (-19%)
#Testing Request	5,605	5,511 (-1.7%)	5,263 (-7%)

Third-party Authentication. Out of the 300 websites evaluated, FENCEHOPPER successfully logs into 183 websites by automated third-party authentication. 4 vulnerabilities are detected from post-authentication web pages. The result is expected: Critical data inputs like registration and login, often with vulnerable checks, occur pre-authentication. Compared to the pre-authentication vulnerabilities, the severity of post-authentication vulnerabilities we detect are generally lower because they require the compromise of one user session, imposing a strong assumption on the attacking.

Validation Threshold. As discussed in § 3.3, FENCEHOPPER conducted a comparative analysis of web responses before and after mutation. This experiment delves into identifying the optimal threshold for validating these responses empirically. We varied the threshold values from 0.5 to 0.9 and evaluate their impact on bug detection and false positive rates. As illustrated in Figure 3.7, increasing the threshold is associated with lower number of bugs and fewer occurrences of false positives. Notably, when the threshold surpasses 0.8, detected bugs notably declines. Conversely, at a threshold of 0.9, the false positive rate only marginally decreases by less than 4% compared

to the 0.8 threshold. Our experiments advocate for an optimal empirical threshold of 0.8.



(a) Reported Bugs vs. Threshold

(b) False Positive vs. Threshold

Figure 3.7. Impact of Threshold Changes on Bug Detection and False Positive Rates.

3.4.4 Coverage of FENCEHOPPER

Coverage of Security Checks. We select 3 websites and manually extract the security checks from their pages to see how many client-side security checks are covered by FENCEHOPPER. These websites are from Entertainment, Education, and Shopping categories, respectively with 132K, 176K, and 124K size of client-side code. In total, we extract 114 client-side security checks, cross-validated by two authors. FENCEHOPPER successfully detects 89 (78%) of them, mutates and tests 84 (74%) of them. It misses 25 checks from the websites because these checks either do not return errors on check failures or use developers' customized error report functions. FENCEHOPPER fails to test 5 checks because they are triggered by one-time operations. Since FENCEHOPPER only replay the data provided by the users, it cannot re-execute one-time operations due to the constraints of business logic.

Detect Existing Vulnerabilities. To evaluate the coverage of FENCEHOPPER in detecting missing server-side checks, we conduct an empirical analysis by applying it to detect existing missing server-side check vulnerabilities. We collect a dataset of existing missing server-side check vulnerabilities by searching public reports from HackerOne [63] and BugCrowd [31] with

Table 3.5. 14 existing missing server-side check vulnerabilities. FENCEHOPPER detects 10 but misses 4 vulnerabilities (in gray cells).

Category	Description of Client-side Check Bypass Issues	Severity
1 Education	Email validation code bypass when verifying user email	Critical
2 Business	Phone validation code bypass in registration	High
3 Government	Email validation bypass	High
4 Technology	Email validation bypass when enabling a feature	High
5 Social	Bypass password authentication when changing password	High
6 Shopping	Phone number validation bypass leads to OTP leak	Critical
7 Business	Bypass Terms of Service	Medium
8 Business	Shop cart limit bypass	Medium
9 Finance	Bypass name input validation	Medium
10 Business	Bypass phone validation in registration	High
11 Technology	Bypass password authentication in password reset	High
12 Finance	Bypass user agreement	Medium
13 Recreation	Email validation is bypassed	High
14 Technology	Username check is bypassed	Medium

keywords 'client side', 'bypass', and 'validation' in titles and descriptions, which end up with 257 matched cases. Then we deduplicate and filter out cases without concrete descriptions, cases that are not related to missing server-side checks for web applications, which ends up with 24 cases. We try to reproduce these 24 cases by looking at the victim website and identify the vulnerable client side interface, among which only 14 cases still preserve the client-side interface that aligns with the vulnerability description. Among these 14 cases, 2 are critical vulnerabilities, 7 are high level, and 5 are medium level. We apply FENCEHOPPER to the websites containing these 14 cases to determine their detection capabilities. Since the disclosed reports are post-mortem, all vulnerabilities we collect have already been fixed. Therefore, we consider FENCEHOPPER successful if it identifies the client-side checks and sends out testing requests.

Out of 14 cases, FENCEHOPPER successfully detects 10 of them as listed in Table 3.5, including 1 critical, 5 high, and 4 medium level vulnerabilities. There are 4 vulnerabilities that FENCEHOPPER fails to detect. 2 false negatives are because of the obfuscation of client-side code, which prevents FENCEHOPPER from analyzing and identifying the client-side checks

effectively. 2 false negative cases are because the client-side checks are in other functions that FENCEHOPPER's inter-procedural analyses fail to cover. Due to the scalability concerns, FENCEHOPPER only extends its inter-procedural analysis for one level. In total, the false negative rate of FENCEHOPPER in detecting existing missing server-side check vulnerabilities is 29%.

3.4.5 Manual Effort and Time

We measure the breakdown of the runtime for each component. For site exploration, the average time is 5.2 minutes. For static analysis, it takes 11.9 seconds on average. For replay and data fix, it takes 3.7 mins on average. In total, it takes 9.1 minutes on average for each website. The site exploration consumes the most time on web page navigation. For adoption efforts in site exploration phases, a PhD student spends less than 10 minutes to record operations on each website with the screenshots and URLs provided by FENCEHOPPER. For the effort of examining vulnerability reports of FENCEHOPPER, it takes 15 minutes on average for the student to check and validate each vulnerability.

3.5 Security Impact Analysis

3.5.1 Systematic Analysis

New vulnerabilities are categorized into four groups: broken authentication (22.9%), broken data integrity (27%), broken business logic (4.2%), and legal agreement violations (45.8%).

Broken Authentication. These vulnerabilities allow attackers to bypass checks in login, registration, and access control on the websites. 5 vulnerabilities allow attackers to bypass email verification during login and gain unauthorized account access. Websites addressed these as critical/high-impact issues upon our reporting. 2 vulnerabilities are missing server-side checks on the password authentication: one enables attackers to log out of other login sessions without the correct password, locking users out of their accounts. The other allows two-factor authentication

disabled without a password, weakening account security.

4 vulnerabilities involve bypassing the rate limit. Missing server-side checks on mobile code rate limits enable attackers to request different mobile code messages at a high frequency, incurring costs for service providers with potential exhaustion of the one-time passcode pools, brute-force attacks, and user harassment. Bypassing captcha verification enables automated scripts to circumvent human verification, leading to denial of service and brute-force attacks.

Risk to Data Integrity. 13 Vulnerabilities in this category enable attackers to input data that does not comply with the checks enforced by developers on the client side. The input data includes usernames, passwords, and emails. For ethics considerations, we do not try any malicious input for further analysis but directly report these to the website owners. However, attackers can potentially try to inject malicious data through these vulnerable input interfaces to perform SQL injection, cross-site scripting, and command injection attacks. They can input excessively long data to flood the application, causing a denial of service. Besides, missing server-side checks on usernames can enable attackers to choose offensive or harmful usernames, making the website a hostile environment for other users.

Legal Agreement Violation. 22 Vulnerabilities in this category enable the attackers to skip agreements related to the terms of service and privacy policy on the websites. By doing so, they can potentially engage in behaviors that violate these agreements and subject web application providers to legal liabilities. This could also lead to legal issues since the attackers can skip agreement on dispute resolution options, which harms the websites' reputation. This category contains *non-trivial* bugs. Existing vulnerability reports [**?**, **?**, **?**] show that these vulnerabilities are of medium to high severity in developers' perspective. Moreover, upon our reporting of these vulnerabilities, 90% of the websites promptly addressed the legal agreement violation bypass rather than dismissing them as inconsequential issues.

Broken Business Logic. 2 vulnerabilities are closely related to the business logic of the websites: missing server-side checks on question correctness and no-cheating agreements, which results in unfair advantages.

3.5.2 Case Study

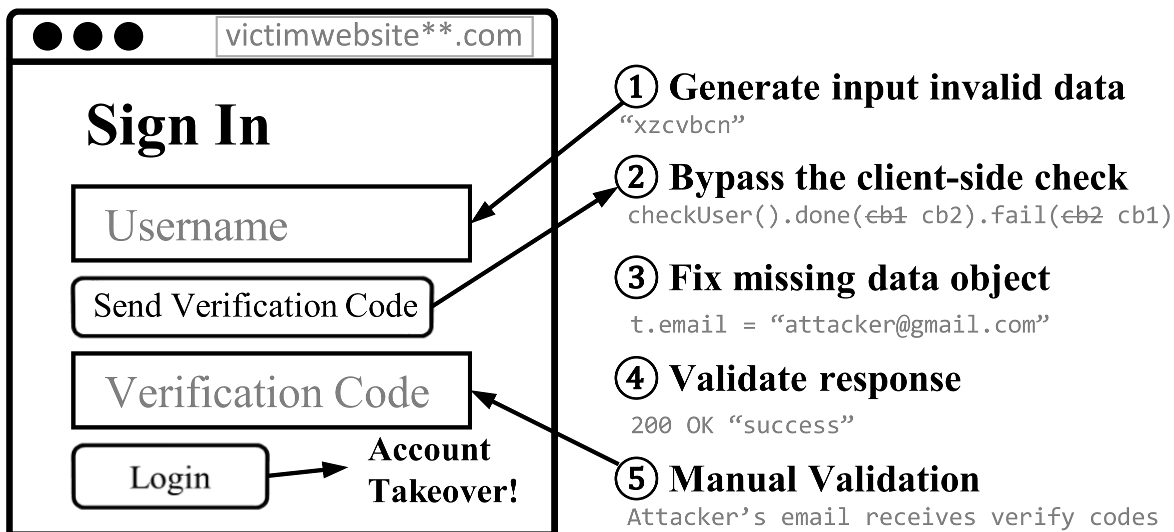


Figure 3.8. Email Verification Code Bypass. The verification code sent to email can be bypassed by mutating client-side code and fill in attacker's email address in missing data objects. Name and UI of the victim website is anonymized.

We illustrate a type of vulnerabilities that FENCEHOPPER effectively identifies, which is found on 5 websites and affects over 20 million users, including a financial website (16 million), a news website (40 million), a real estate information website (1.8 million), an e-magazine website (9 million), and an investment analysis website (358.1 K). This case highlights the importance of client-side code mutation and missing data fixes performed by FENCEHOPPER, which existing tools and even human experts fail to cover. In this case, the victim website allows users to log in by inputting their usernames. The website then sends a one-time passcode to the user's email, with which the users can log in to this website without passwords (Figure 3.8). The client-side code normally checks if the entered username is registered. However, with FENCEHOPPER's mutation, the `passwordlessLogin()` function is triggered regardless of the `checkUser()` result. Attackers can proceed with a passwordless login and trigger a verification code request, even with an unregistered username. FENCEHOPPER automatically fills missing data objects t.email and t.user, leading to a success check bypass. The code snippet is as follows:

```
this.checkUser(user, this.conf.checkUserPath)
```

.done(~~passwordlessLogin(t)~~alert("User not exists!"))

.fail(~~alert("User not exists!")~~ passwordlessLogin(t))

function passwordlessLogin(t) {

post(url, t.user, t.email, ...) // Missing data object

}

Upon further investigation of this reported missing server-side check, we discover that the one-time passcode is received by the address filled into `t.email` by FENCEHOPPER regardless of the data fixed into `t.user`. This means that when `t.email` is the attacker's email and `t.user` is the victim's username, the attacker's email will receive the one-time code, granting them access to the victim's account.

3.6 Limitations and Discussion

Limitations. FENCEHOPPER's effectiveness is partly limited by its reliance on client-side checks to detect missing server-side validations. It operates under the assumption that analogous checks are either fully present or entirely absent on the server, leading to single-instance testing per check. This might overlook checks that are only partially implemented on the server side. Besides, as mentioned in Section 3, our analysis doesn't account for code cloaking, assuming accessibility to uncloaked source code. Moreover, the analysis of FENCEHOPPER is neither complete nor sound due to the known challenges in JavaScript analysis like handling asynchronous events [97, 142], dynamic testing coverage etc. FENCEHOPPER is a supplement to existing security testing tools that target to detect complex vulnerabilities that existing tools fail to detect.

Ethics and Responsible Disclosure. We adhered to stringent ethical standards and responsible disclosure practices. Our testing was conducted using legitimate email and third-party accounts under our control, without accessing private user data. Request generation mimicked normal user behavior, avoiding any form of denial-of-service attacks. Upon identifying vulnerabilities, we promptly informed developers and anonymized sensitive details of the affected websites,

including their identity, interface, and code. For vulnerabilities that potentially affect the data integrity of the website, we report them proactively without injecting any harmful data into the website.

3.7 Related Work

Client-Side Missing Check Detection. Previous works also observe that missing checks in client-side code lead to vulnerabilities, which can be exploited when attackers input illegal data through the web clients and the requests are sent to the servers due to the missing client-side checks. FLAX [142] can detect various types of client-side validation vulnerabilities by analyzing the client-side code and fuzzing the web clients. ZigZag [174] instruments the client-side code so that it can capture the malformed requests deviating from normal control flows and data values in the client-side code. It focuses on detecting client-side validation vulnerabilities in browsers, while FENCEHOPPER aims to find missing server-side security checks. Gelato [65] performs taint analysis on the client source code, traverses the call graph, and generates input values based on the semantics of the source code. These works focus on the client-side flaws and assume that attackers interact with the client and exploit missing checks in the client-side code. However, they cannot prevent attackers from directly sending manipulated requests to the servers, which exploit server-side vulnerabilities.

[81] detects business logic tampering in web application client side like removing the subscription bar above news without memberships, inspired from some Chrome extensions like AdBlock [14] and Bypass Paywalls [74]. Although it also mutates client-side code on the part that displays pop-up windows to block readers from viewing some content, it focuses on detecting client-side vulnerabilities. In comparison, we focus on mutating client-side code to examine if server sides incorrectly neglect some checks (based on the false sense of security stemming from the existence of client-side checks). Therefore, FENCEHOPPER still needs to ensure sending testing requests to the server, while the code mutation in [81] focuses on disabling

client-side rendering.

Server-Side Missing Check Detection. Previous works detect missing checks by inferring invariants from execution traces. BLOCK [90] infers invariants between web messages and session states. LogicScope [91] creates a finite state machine from execution traces and runtime data. BATMAN [89] focuses on database access control violations via SQL query analysis. WAPTEC [29] provides a unified framework for combining the above constraints together. However, these methods have limitations in trace coverage and adaptation to evolving web services. Unlike these, our approach based on mutating client-side code avoids these issues and complements invariant-based methods.

In gray-box methods, client-side code analysis aids payload generation. NoTamper [28] and WARDroid [97] analyze code for constraint violations, while TamperProof [154] focuses on server defense against parameter tampering. These static methods, however, lack comprehensive information. FENCEHOPPER dynamically executes and mutates client code, which generates more representative requests to uncover latent server vulnerabilities, even in complex web services involving third-party services.

Bypass testing, introduced in [118], identifies client-side check patterns for server testing but is limited to specific patterns. FENCEHOPPER offers a broader approach by systematically analyzing and mutating client-side JavaScript checks, thus detecting missing server-side checks more effectively.

Specific Type Vulnerability Detection. Other works [189, 157, 35, 169, 155] detects security vulnerabilities in access control [169, 188], payment [170, 157], and digital content [80] services based on specific work models of these services. For example, [157] detects logic vulnerabilities in e-commerce applications by statically analyzing the invariant violation in checkout processes. [155] infers role-specific security logic in web applications to identify missing authorization checks. AuthScope [189] substitutes HTTP protocol fields to discover unauthorized access. Compared to them, FENCEHOPPER aims to detect missing checks without making any assumptions about the service models. Besides, many previous works focus on detecting input validation

vulnerabilities on the server side, including cross-site scripting [78, 84] and SQL injections [173]. These works focus on specific types or patterns of vulnerability, whereas FENCEHOPPER is more general and systematically explores security checks (regardless of specific types) in client-side code.

Other Related Work. Mutation testing [125, 128, 179, 102, 103] is a technique for evaluating test suite quality by generating a large number of mutants and executing them against the suites. In FENCEHOPPER, we use mutation for testing vulnerabilities in server-side components. Forced execution is essential in FENCEHOPPER after code mutation, inspired by many previous works. [181] forced execution on multiple paths by dynamic instrumentation for debugging the control flow bugs. [126] forced binary execution on different paths to expose hidden malware behaviors with higher coverage. [82] explored forced execution in JavaScript programs to trigger hidden malware scripts. Similar approaches are also used for malware detection in mobile apps [39, 77]. One challenge of forced execution is ensuring crash-free execution as missing data objects can cause execution failures. In FENCEHOPPER, forced execution is used to dispatch requests to the server side while preserving most of the client-side functionality for request processing. It borrows insights from previous work to fix the missing data on demand and proposes fixing strategies based on web scenarios. Code mutation is also applied in fuzzing tools like TFUZZ [127], which mutates tested programs to efficiently search for fuzzing payloads in C++ code. However, FENCEHOPPER diverges from TFUZZ in its objective, generating test inputs based on client-side code rather than employing random fuzzing. Additionally, FENCEHOPPER validates vulnerabilities by inspecting web responses, whereas TFUZZ requires reverting to the unmutated program to confirm the presence of bugs.

3.8 Case Study: Blind Spots in Third-Party Services

In this chapter, we briefly discuss another example of cross-component blind spots as a supplemental case study to this thesis. We study a recent type of web services, AI-related

services, and identify two typical blind spots in OpenAI, a representative platform in this category. Two typical blind spots are: (1) Bring your own key. As illustrated in OpenAI policy, this is not allowed [120]. However, many third-party services are not aware of this but instead ask customers to provide them with the OpenAI API key so that they use customers' quota instead of managing their own OpenAI quota. (2) Client-side key leak. The key leaks happen when the developers are not aware of a fact that the client-side code is accessible to users. If the developers put OpenAI API keys on the client side, by inspecting the client-side code, malicious users can easily obtain these keys and use the keys for their own benefit. To investigate this problem, we conduct a study on the websites and Chrome extensions that could potentially use OpenAI services.

We selected the websites from Google Search resutls. We search with keywords "AI chat", "AI writing", and "AI assistant" and filter out the results before 2023. We choose the first 10 pages of search results for each keywords and manually check each search results to make sure this is a web service that provide AI services. In the end, we get 577 websites. Among them, 80 websites (13.9%) provide 'bring your own key' interface. 2 (0.3%) websites leaked OpenAI API keys in the client-side code. We selected the Chrome extensions from Chrome Web Store [54]. We search with the same keywords as for websites, and filter out the extensions that has less than 10 users. We manually check each search results to make sure this is a Chrome extension that provide AI services. In the end, we get 791 Chrome extensions, with 51 of them (6.4%) have 'bring your own key' interface. 15 extensions (1.9%) leak OpenAI API keys in their client. The study results reveal that a proportion of developers do have these blind spots in the new-emerging services, which could cause severe security and financial consequences.

3.9 Acknowledgments

Chapter 3, in full, is a reprint of the material under submission. Zhong, Li; Zhang, Bokai; Zhou, Yuanyuan. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Conclusion

4.1 Summary of this Dissertation

This dissertation presents two methods for detecting bugs from developers' blind spots. This dissertation presents VALUECHECK to help developers to identify cross-authorship blind spots in the collaborative large-scale code repositories. It discusses the perception of unused definitions in C/C++ programs, initially regarded as redundant code but potentially indicating deeper issues. This dissertation presents multiple real-world examples to illustrate severe bugs caused by unused definitions, such as security vulnerabilities and configuration errors. Existing techniques for detecting unused definitions are insufficient for their inability to differentiate between trivial redundancies and critical bugs, leading to impracticality and high false positives. Furthermore, developers often disable unused definition warnings due to their overwhelming volume. To address these challenges, this dissertation proposes an approach called VALUECHECK, which leverages insights on cross-scope unused definitions, identifies intentional unused definitions, and prioritizes bug detection using code familiarity models. VALUECHECK is evaluated on Linux, MySQL, OpenSSL, and NFS-ganesha, demonstrating its effectiveness in detecting and fixing real bugs with a reduced false positive rate compared to existing tools. This dissertation then presents FENCEHOPPER to detect another type of developers' blind spots, cross-component blind spots. We observe that the internal architecture of web applications cause a false sense of security, which leads to missing server-side checks, a type of cross-component blind spots.

FENCEHOPPER detects missing server-side checks from web applications based on a novel code mutation method, which supplements existing methods when there are complicated constraints or data relationships. We design a check-based code mutation method including code mutation, input generation, and output validation. We conduct experiments with FENCEHOPPER on top 300 websites from Tranco, which proves to be a practical framework. Our evaluation reveals the detection of 48 previously unidentified vulnerabilities by our framework. Within this set, 5 vulnerabilities are classified as critical access control issues, posing a substantial risk of unauthorized account access for more than 20 million users. To evaluate FENCEHOPPER's coverage, we manually compile a dataset containing 14 documented instances of missing server-side checks sourced from HackerOne and Bugcrowd. Impressively, FENCEHOPPER successfully pinpoints 10 of these vulnerabilities. Notably, our findings indicate that over 60% of the vulnerabilities identified by FENCEHOPPER remain undetected by existing state-of-the-art tools.

4.2 Lessons Learned

In this section, I will share insights drawn from my research experiences, providing lessons for future research endeavors and encourage forthcoming investigations.

Firstly, the escalating complexity of software development poses a formidable challenge for developers striving to comprehend the entirety of the development landscape. With the necessity to integrate code from various sources and components, developers face the arduous task of discerning potential risks and understanding underlying assumptions. Moreover, the time constraints in agile development methodologies underscore the need for tools that can help developers in this endeavor, mitigating the limitations in human judgment. Novel approaches are required to analyze developers' interactions with code beyond traditional methods that solely depends on code analysis and testing. Incorporating additional meta information, such as authorship details, could further enable wiser and more reliable decision-making in software development. We provide an illustrative example to demonstrate how modern software

engineering can leverage these underlying factors for enhanced outcomes in VALUECHECK.

Furthermore, other potential factors are worth consideration including: (1) Developers' interactions with Integrated Development Environments (IDEs). (2) The integral processes of code review and documentation within most industry organizations. (3) The reliance on third-party documentation and tutorials, which can sometimes be ambiguous and mislead developers. We believe that future research could gain significant insights by exploring these directions.

Secondly, an expanded range of applications stands to benefit from cross-component blind spot detection. As micro-service architecture becomes more prevalent, the assumptions across the boundaries between different components are not diminishing, but rather proliferating. Within such systems, comprising components from diverse trust zones, ensuring the elimination of cross-component blind spots poses a significant challenge. The methodologies introduced in this dissertation are promising for further application in these contexts. By mutating each component's code strategically, we can effectively transform it into a testing engine for validating other components that interact with it. This approach augments the reliability and security of all components within complex systems, thereby fortifying the integrity of the entire system.

Thirdly, as discussed in one pilot study of this dissertation, the advent of new services introduces fresh challenges within web ecosystems. Recent popular web services such as OpenAI APIs and Stripe APIs exemplify this trend, featuring inadequate access management designs coupled with ineligible usage costs. Such flawed designs leave unwitting developers vulnerable to cross-component blind spots inherently in software development practices. Our preliminary investigation uncovers a concerning prevalence of such practices among developers, leading to significant repercussions. These include 'bring your own key' architectures and client-side key leakage, the ramifications of which cannot be understated. Hence, we advocate for further research to delve into the underlying causes and raise awareness among developers about these risks. Moreover, addressing this issue necessitates collaborative efforts from various stakeholders to promote the adoption of more robust access management protocols in emerging services, even as some vendors may prioritize financial gains over security considerations.

Bibliography

[1] Diagnostic flags in Clang. https://clang.llvm.org/docs/DiagnosticsReference.html#wunused, 2007.

[2] GitPython Documentation — GitPython 3.1.12 documentation. https://gitpython.readthedocs.io/en/stable/index.html, 2015.

[3] Smatch: pluggable static analysis for C. https://lwn.net/Articles/691882/, 2016.

[4] Smatch the Source Matcher. https://smatch.sourceforge.net/, 2020.

[5] Software Developers : Occupational Outlook Handbook: : U.S. Bureau of Labor Statistics. https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm, 2020-09-01.

[6] 90 https://www.itpro.com/development/application-programming-interface-api/358546/nearly-every-company-surveyed-experienced, 2021.

[7] ESLint - Pluggable JavaScript linter. https://eslint.org/, 2021.

[8] Infer Static Analyzer — Infer — Infer. https://fbinfer.com/, 2021.

[9] NFS-ganesha : User Space NFS and 9P File Server. https://nfs-ganesha.github.io/, 2021.

[10] The LLVM Compiler Infrastructure. https://llvm.org/, 2021.

[11] Warning Options (Using the GNU Compiler Collection (GCC)). https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html#Warning-Options, 2021.

[12] re - Regular expression operations. https://docs.python.org/3/library/re.html, 2022.

[13] StackOverflow. https://stackoverflow.com/, 2022.

[14] Inc. AdBlock. Adblock — best ad blocker. https://chromewebstore.google.com/detail/gighmmpiobklfepjocnamgkkbiglidom, 2023.

[15] Mansour Ahmadi, Reza Mirzazade Farkhani, Ryan Williams, and Long Lu. Finding bugs using your own code: detecting functionally-similar yet inconsistent code. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2025–2040, 2021.

[16] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. Compilers, principles, techniques. *Addison wesley*, 7(8):9, 1986.

[17] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, Citeseer, 1994.

[18] Anonymous. Anonymized Repository - Anonymous GitHub. https://anonymous.4open.science/r/FenceHooper-1DBC/, 2023.

[19] John Anvik and Gail C Murphy. Determining implementation expertise from bug reports. In *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*, pages 2–2. IEEE, 2007.

[20] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. A framework for automated testing of javascript web applications. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 571–580, 2011.

[21] Joop Aué, Maurício Aniche, Maikel Lobbezoo, and Arie van Deursen. An exploratory study on faults in web api integration in a large-scale payment company. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pages 13–22, 2018.

[22] MA Awad. A comparison between agile and traditional software development methodologies. *University of Western Australia*, 30:1–69, 2005.

[23] Jia-Ju Bai, Tuo Li, and Shi-Min Hu. {DLOS}: Effective static detection of deadlocks in {OS} kernels. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 367–382, 2022.

[24] Andrew Begel, James D Herbsleb, and Margaret-Anne Storey. The future of collaborative software development. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work Companion*, pages 17–18, 2012.

[25] Cristiano Bertolini, Martin Schäf, and Pascal Schweitzer. Infeasible code detection. In *International Conference on Verified Software: Tools, Theories, Experiments*, pages 310–325. Springer, 2012.

[26] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[27] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code! examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 4–14, 2011.

[28] Prithvi Bisht, Timothy Hinrichs, Nazari Skrupsky, Radoslaw Bobrowicz, and VN Venkatakrishnan. Notamper: automatic blackbox detection of parameter tampering opportunities in web applications. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 607–618, 2010.

[29] Prithvi Bisht, Timothy Hinrichs, Nazari Skrupsky, and VN Venkatakrishnan. Waptec: whitebox analysis of web applications for parameter tampering exploit construction. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 575–586, 2011.

[30] Preston Briggs and Keith D Cooper. Effective partial redundancy elimination. *ACM SIGPLAN Notices*, 29(6):159–170, 1994.

[31] BugCrowd. #1 Crowdsourced Cybersecurity Platform — Bugcrowd. https://www.bugcrowd.com/, 2023.

[32] Martin Campbell-Kelly. The airy tape: An early chapter in the history of debugging. 1990.

[33] CapacitorSet. box-js - a tool for studying javascript malware. https://github.com/CapacitorSet/box-js, 2023.

[34] Gregory J Chaitin. Register allocation & spilling via graph coloring. *ACM Sigplan Notices*, 17(6):98–101, 1982.

[35] Eric Y Chen, Shuo Chen, Shaz Qadeer, and Rui Wang. Securing multiparty online services via certification of symbolic transactions. In *2015 IEEE Symposium on Security and Privacy*, pages 833–849. IEEE, 2015.

[36] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 18–37, 2015.

[37] Lykes Claytor and Francisco Servant. Understanding and leveraging developer inexpertise. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 404–405, 2018.

[38] Cleidson RB De Souza, David Redmiles, and Paul Dourish. " breaking the code", moving between private and public work in collaborative software development. In *Proceedings of the 2003 ACM International Conference on Supporting Group Work*, pages 105–114, 2003.

[39] Zhui Deng, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. iris: Vetting private api abuse in ios applications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 44–56, 2015.

[40] Chris Duckett. Zoom concedes custom encryption is substandard as citizen lab pokes holes in it. https://www.zdnet.com/article/zoom-concedes-custom-encryption-is-sub-standard-as-citizen-lab-pokes-holes-in-it/, 2023.

[41] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. *ACM SIGOPS Operating Systems Review*, 35(5):57–72, 2001.

[42] Benjamin Eriksson, Giancarlo Pellegrino, and Andrei Sabelfeld. Black widow: Blackbox data-driven web scanning. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1125–1142. IEEE, 2021.

[43] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction of approximate call graphs for javascript ide services. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 752–761. IEEE, 2013.

[44] Matthieu Foucault, Marc Palyart, Xavier Blanc, Gail C Murphy, and Jean-Rémy Falleri. Impact of developer turnover on quality in open-source software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 829–841, 2015.

[45] Martin Fowler. Refactoring: Improving the design of existing code. In *11th European Conference. Jyväskylä, Finland*, 1997.

[46] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[47] Thomas Fritz, Jingwen Ou, Gail C Murphy, and Emerson Murphy-Hill. A degree-of-knowledge model to capture source code familiarity. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 385–394, 2010.

[48] Mark Gabel, Junfeng Yang, Yuan Yu, Moises Goldszmidt, and Zhendong Su. Scalable and systematic detection of buggy inconsistencies in source code. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 175–190, 2010.

[49] Emanuel Giger and Harald Gall. Object-oriented design heuristics. 1996.

[50] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[51] David Goltzsche, Colin Wulf, Divya Muthukumaran, Konrad Rieck, Peter Pietzuch, and Rüdiger Kapitza. Trustjs: Trusted client-side execution of javascript. In *Proceedings of the 10th European Workshop on Systems Security*, pages 1–6, 2017.

[52] Google. Angular. https://angular.io/, 2023.

[53] Google. Google bug hunters - google bug hunters. https://bughunters.google.com/, 2023.

[54] Google. Chrome Web Store. https://chromewebstore.google.com/, 2024.

[55] Dan Gopstein, Jake Iannacone, Yu Yan, Lois DeLong, Yanyan Zhuang, Martin K-C Yeh, and Justin Cappos. Understanding misunderstandings in source code. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 129–139, 2017.

[56] Emre Güler, Sergej Schumilo, Moritz Schloegel, Nils Bars, Philipp Görz, Xinyi Xu, Cemal Kaygusuz, and Thorsten Holz. Atropos: Effective fuzzing of web applications for server-side vulnerabilities.

[57] Neelam Gupta, Aditya P Mathur, and Mary Lou Soffa. Generating test data for branch coverage. In *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*, pages 219–227. IEEE, 2000.

[58] Rajiv Gupta, DA Benson, and Jesse Zhixi Fang. Path profile guided partial dead code elimination using predication. In *Proceedings 1997 International Conference on Parallel Architectures and Compilation Techniques*, pages 102–113. IEEE, 1997.

[59] HackerOne. Captcha bypass in coinbase signup form. https://hackerone.com/reports/246801, 2017.

[60] HackerOne. Bypass password authentication for updating email and phone number - security vulnerability. https://hackerone.com/reports/770504, 2020.

[61] HackerOne. India - otp bypass on phone number verification for account creation. https://hackerone.com/reports/762695, 2020.

[62] HackerOne. Otp bypass in verifying nin. https://hackerone.com/reports/1314172, 2021.

[63] Hackerone. HackerOne — #1 Trusted Security Platform and Hacker Program. https://www.hackerone.com/, 2023.

[64] Tracy Hall, Min Zhang, David Bowes, and Yi Sun. Some code smells have a significant but small effect on faults. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):1–39, 2014.

[65] Behnaz Hassanshahi, Hyunjun Lee, and Paddy Krishnan. Gelato: Feedback-driven and guided security analysis of client-side web applications. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 618–629. IEEE, 2022.

[66] Michael Hind and Anthony Pioli. Which pointer analysis should i use? In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 113–123, 2000.

[67] Joona Hoikkala. ffuf/ffuf: Fast web fuzzer written in go. https://github.com/ffuf/ffuf, 2023.

[68] Hao Hu, Hongyu Zhang, Jifeng Xuan, and Weigang Sun. Effective bug triage based on historical bug-fix information. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 122–132. IEEE, 2014.

[69] Haochen Huang, Bingyu Shen, Li Zhong, and Yuanyuan Zhou. Protecting data integrity of web applications with database constraints inferred from application code. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 632–645, 2023.

[70] Haochen Huang, Chengcheng Xiang, Li Zhong, and Yuanyuan Zhou. {PYLIVE}:{On-the-Fly} code change for python-based online services. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 349–363, 2021.

[71] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, pages 40–52, 2004.

[72] Yuan Huang, Qiaoyang Zheng, Xiangping Chen, Yingfei Xiong, Zhiyong Liu, and Xiaonan Luo. Mining version control system for automatically generating commit comment. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 414–423. IEEE, 2017.

[73] Yuan-Shin Hwang and Joel Saltz. Identifying def/use information of statements that construct and traverse dynamic recursive data structures. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 131–145. Springer, 1997.

[74] iamadamdev. iamadamdev/bypass-paywalls-chrome: Bypass paywalls web browser extension for chrome and firefox. https://github.com/iamadamdev/bypass-paywalls-chrome, 2023.

[75] Instagram. Instagram — vulnerability disclosure policy — hackerone. https://hackerone.com/instagram?type=team, 2023.

[76] Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 78–89, 1993.

[77] Ryan Johnson and Angelos Stavrou. Forced-path execution for android applications on x86 platforms. In *2013 IEEE Seventh International Conference on Software Security and Reliability Companion*, pages 188–197. IEEE, 2013.

[78] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 6–pp. IEEE, 2006.

[79] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering*, 17(3):243–275, 2012.

[80] I Luk Kim, Weihang Wang, Yonghwi Kwon, and Xiangyu Zhang. Bftdetector: Automatic detection of business flow tampering for digital content service.

[81] I Luk Kim, Yunhui Zheng, Hogun Park, Weihang Wang, Wei You, Yousra Aafer, and Xiangyu Zhang. Finding client-side business flow tampering vulnerabilities. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 222–233, 2020.

[82] Kyungtae Kim, I Luk Kim, Chung Hwan Kim, Yonghwi Kwon, Yunhui Zheng, Xiangyu Zhang, and Dongyan Xu. J-force: Forced execution on javascript. In *Proceedings of the 26th international conference on World Wide Web*, pages 897–906, 2017.

[83] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 147–161, 2019.

[84] Lukas Knittel, Christian Mainka, Marcus Niemietz, Dominik Trevor Noß, and Jörg Schwenk. Xsinator. com: From a formal model to the automatic evaluation of cross-site leaks in web browsers. In *CCS*, pages 1771–1788, 2021.

[85] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Partial dead code elimination. *ACM SIGPLAN Notices*, 29(6):147–158, 1994.

[86] Chris Lattner. Llvm and clang: Next generation compiler technology. In *The BSD conference*, volume 5, 2008.

[87] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. Samc: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 399–414, USA, 2014. USENIX Association.

[88] Tuo Li, Jia-Ju Bai, Yulei Sui, and Shi-Min Hu. Path-sensitive and alias-aware typestate analysis for detecting os bugs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 859–872, 2022.

[89] Xiaowei Li, Xujie Si, and Yuan Xue. Automated black-box detection of access control vulnerabilities in web applications. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, pages 49–60, 2014.

[90] Xiaowei Li and Yuan Xue. Block: a black-box approach for detection of state violation attacks towards web applications. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 247–256, 2011.

[91] Xiaowei Li and Yuan Xue. Logicscope: Automatic discovery of logic vulnerabilities within web applications. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 481–486, 2013.

[92] Xiaowei Li and Yuan Xue. A survey on server-side approaches to securing web applications. *ACM Computing Surveys (CSUR)*, 46(4):1–29, 2014.

[93] Kangjie Lu, Aditya Pakki, and Qiushi Wu. Automatically identifying security checks for detecting kernel semantic bugs. In *Computer Security–ESORICS 2019: 24th European Symposium on Research in Computer Security, Luxembourg, September 23–27, 2019, Proceedings, Part II 24*, pages 3–25. Springer, 2019.

[94] Kangjie Lu Lu, Aditya Pakki, and Qiushi Wu. Detecting missing-check bugs via semantic- and context-aware criticalness and constraints inferences. In *Proceedings of the 28th USENIX Conference on Security Symposium*, 2019.

[95] Gurpreet Singh Matharu, Anju Mishra, Harmeet Singh, and Priyanka Upadhyay. Empirical study of agile software development methodologies: A comparative analysis. *ACM SIGSOFT Software Engineering Notes*, 40(1):1–6, 2015.

[96] David W McDonald and Mark S Ackerman. Expertise recommender: a flexible recommendation system and architecture. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, pages 231–240, 2000.

[97] Abner Mendoza and Guofei Gu. Mobile application web api reconnaissance: Web-to-mobile inconsistencies & vulnerabilities. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 756–769. IEEE, 2018.

[98] Meta. Meta bug bounty — facebook. https://www.facebook.com/BugBounty/, 2023.

[99] Meta. React - the library for web and native user interfaces. https://react.dev/, 2023.

[100] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 361–377, 2015.

[101] Shawn Minto and Gail C Murphy. Recommending emergent teams. In *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*, pages 5–5. IEEE, 2007.

[102] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. Efficient javascript mutation testing. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 74–83. IEEE, 2013.

[103] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. Guided mutation testing for javascript web applications. *IEEE Transactions on Software Engineering*, 41(5):429–444, 2014.

[104] Audris Mockus and James D Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 503–512. IEEE, 2002.

[105] Youssef A. Mohamed. More secure Facebook Canvas : Tale of $126k worth of bugs that lead to Facebook Account Takeovers. https://ysamm.com/?p=708, 2022.

[106] Youssef A. Mohamed. DOM-XSS in Instant Games due to improper verification of supplied URLs. https://ysamm.com/?p=779, 2023.

[107] Maliheh Monshizadeh, Prasad Naldurg, and VN Venkatakrishnan. Mace: Detecting privilege escalation vulnerabilities in web applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 690–701, 2014.

[108] John C Munson and Sebastian G Elbaum. Code churn: A measure for estimating the impact of code change. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 24–31. IEEE, 1998.

[109] Robert Muth. Register liveness analysis of executable code. *Manuscript, Dept. of Computer Science, The University of Arizona, Dec*, 1998.

[110] Gtm Mänôz. Two factor authentication bypass on facebook — by gtm mänôz — jan, 2023 — pentester nepal. https://medium.com/pentesternepal/two-factor-authentication-bypass-on-facebook-3f4ac3ea139c, 2023.

[111] Mathieu Nassif and Martin P Robillard. Revisiting turnover-induced knowledge loss in software projects. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 261–272. IEEE, 2017.

[112] Tung Thanh Nguyen, Tien N Nguyen, Evelyn Duesterwald, Tim Klinger, and Peter Santhanam. Inferring developer expertise through defect analysis. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1297–1300. IEEE, 2012.

[113] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer Science & Business Media, 2004.

[114] John Noll, Sarah Beecham, and Ita Richardson. Global software development and collaboration: barriers and solutions. *ACM inroads*, 1(3):66–78, 2011.

[115] Diego Novillo. Gcc an architectural overview, current status, and future directions. In *Proceedings of the Linux Symposium*, volume 2, page 185, 2006.

[116] Jeff Offutt, Qingxiang Wang, and Joann Ordille. An industrial case study of bypass testing on web applications. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 465–474. IEEE, 2008.

[117] Jeff Offutt, Ye Wu, Xaiochen Du, and Hong Huang. Web application bypass testing. In *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.*, volume 2, pages 106–109. IEEE, 2004.

[118] Jeff Offutt, Ye Wu, Xiaochen Du, and Hong Huang. Bypass testing of web applications. In *15th International Symposium on Software Reliability Engineering*, pages 187–197. IEEE, 2004.

[119] Adeyefa Oluwatoba. 2fa bypass vulnerability — part 1 — by adeyefa oluwatoba — medium. https://sainttobs.medium.com/2fa-bypass-vulnerability-part-1-447cf11a1525, 2021.

[120] OpenAI. Terms of use. https://openai.com/policies/terms-of-use, 2023.

[121] OWASP. Customized headers - owasp cheat sheet series. https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html#custom-request-headers, 2023.

[122] OWASP. Double submit cookie - owasp cheat sheet series. https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html#double-submit-cookie, 2023.

[123] OWASP. Token-based mitigation - owasp cheat sheet series. https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html#token-based-mitigation, 2023.

[124] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. Do they really smell bad? a study on developers' perception of bad code smells. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 101–110. IEEE, 2014.

[125] Mike Papadakis and Nicos Malevris. Mutation based test case generation via a path selection strategy. *Information and Software Technology*, 54(9):915–932, 2012.

[126] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. X-force: Force-executing binary programs for security applications. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 829–844, 2014.

[127] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.

[128] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. Practical mutation testing at scale. *arXiv preprint arXiv:2102.11378*, 2021.

[129] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. *arXiv preprint arXiv:1806.01156*, 2018.

[130] Burp Suite Professional PortSwigger. Burp suite professional - portswigger. https://www.kali.org/tools/dirbuster/, 2023.

[131] Pouya. India - otp bypass on phone number verification for account creation. https://www.darabi.me/2015/04/bypass-facebook-csrf.html, 2015.

[132] Mark Probst, Andreas Krall, and Bernhard Scholz. Register liveness analysis for optimizing dynamic binary translation. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, pages 35–44. IEEE, 2002.

[133] yns000 ranulf. Jbrofuzz download — sourceforge.net. https://sourceforge.net/projects/jbrofuzz/, 2013.

[134] Dark Reading. Facebook bug allows 2fa bypass via instagram. https://www.darkreading.com/application-security/facebook-bug-2fa-bypass-instagram, 2023.

[135] Xiaoxue Ren, Jiamou Sun, Zhenchang Xing, Xin Xia, and Jianling Sun. Demystify official api usage directives with crowdsourced api misuse scenarios, erroneous code examples and patches. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*, pages 925–936, 2020.

[136] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, 1995.

[137] Athos Ribeiro, Paulo Meirelles, Nelson Lago, and Fabio Kon. Ranking warnings from multiple source code static analyzers via ensemble learning. In *Proceedings of the 15th International Symposium on Open Collaboration*, pages 1–10, 2019.

[138] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of javascript programs. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 2010.

[139] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. Sibylfs: formal specification and oracle-based testing for posix and real-world file systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 38–53, 2015.

[140] Nayan B Ruparelia. The history of version control. *ACM SIGSOFT Software Engineering Notes*, 35(1):5–9, 2010.

[141] Amir Saboury, Pooya Musavi, Foutse Khomh, and Giulio Antoniol. An empirical study of code smells in javascript projects. In *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*, pages 294–305. IEEE, 2017.

[142] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *NDss*, 2010.

[143] Martin Schäf, Daniel Schwartz-Narbonne, and Thomas Wies. Explaining inconsistent code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 521–531, 2013.

[144] Paul B Schneck. A survey of compiler optimization techniques. In *Proceedings of the ACM annual conference*, pages 106–113, 1973.

[145] David Schuler and Thomas Zimmermann. Mining usage expertise from version archives. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 121–124, 2008.

[146] Hina Shah, Carsten Gorg, and Mary Jean Harrold. Understanding exception handling: Viewpoints of novices and experts. *IEEE Transactions on Software Engineering*, 36(2):150–161, 2010.

[147] Lwin Khin Shar and Hee Beng Kuan Tan. Predicting common web application vulnerabilities from input validation and sanitization code patterns. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 310–313, 2012.

[148] Nishant Sharma. CVE-2020-8772 exploitation under 3 minutes — by nishant sharma — pentester academy blog. https://blog.pentesteracademy.com/cve-2020-8772-exploitation-under-3-minutes-594265b4e26a, 2020.

[149] Raed Shatnawi and Wei Li. An investigation of bad smells in object-oriented design. In *Third International Conference on Information Technology: New Generations (ITNG'06)*, pages 161–165. IEEE, 2006.

[150] Bingyu Shen. *Automatic Methods to Enhance Server Systems in Access Control Diagnosis*. University of California, San Diego, 2022.

[151] Bingyu Shen, Tianyi Shan, and Yuanyuan Zhou. Multiview: Finding blind spots in access-deny issues diagnosis. In *USENIX Security Symposium*, 2023.

[152] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 1–16, USA, 2016. USENIX Association.

[153] Dag IK Sjøberg, Aiko Yamashita, Bente CD Anda, Audris Mockus, and Tore Dybå. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, 39(8):1144–1156, 2012.

[154] Nazari Skrupsky, Prithvi Bisht, Timothy Hinrichs, VN Venkatakrishnan, and Lenore Zuck. Tamperproof: a server-agnostic defense for parameter tampering attacks on web applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 129–140, 2013.

[155] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. Rolecast: finding missing security checks when you do not know what checks are. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 1069–1084, 2011.

[156] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266. ACM, 2016.

[157] Fangqi Sun, Liang Xu, and Zhendong Su. Detecting logic vulnerabilities in e-commerce applications. In *NDSS*, 2014.

[158] Sven. Jsdetox - a javascript malware analysis tool using static analysis / deobfuscation techniques and an execution engine featuring html dom emulation. http://www.relentless-coding.org/projects/jsdetox, 2023.

[159] Lin Tan, Xiaolan Zhang, Xiao Ma, Weiwei Xiong, and Yuanyuan Zhou. Autoises: Automatically inferring security specification and detecting violations. In *USENIX Security Symposium*, pages 379–394, 2008.

[160] the ZAP Dev Team. The zap homepage. https://www.zaproxy.org/, 2024.

[161] Patanamon Thongtanunam, Shane McIntosh, Ahmed E Hassan, and Hajimu Iida. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *Proceedings of the 38th international conference on software engineering*, pages 1039–1050, 2016.

[162] Aaron Tomb and Cormac Flanagan. Detecting inconsistencies via universal reachability analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 287–297, 2012.

[163] Kali Linux Tools. dirbuster — kali linux tools. https://www.kali.org/tools/dirbuster/, 2023.

[164] Twitter. Twitter — bug bounty program policy — hackerone. https://hackerone.com/twitter?type=team, 2023.

[165] Orpheas van Rooij, Marcos Antonios Charalambous, Demetris Kaizer, Michalis Papaevripides, and Elias Athanasopoulos. webfuzz: Grey-box fuzzing for web applications. In *Computer Security–ESORICS 2021: 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4–8, 2021, Proceedings, Part I 26*, pages 152–172. Springer, 2021.

[166] Morteza Verdi, Ashkan Sami, Jafar Akhondali, Foutse Khomh, Gias Uddin, and Alireza Karami Motlagh. An empirical study of c++ vulnerabilities in crowd-sourced code examples. *IEEE Transactions on Software Engineering*, 48(5):1497–1514, 2020.

[167] Mitchell Wand and Igor Siveroni. Constraint systems for useless variable elimination. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 291–302, 1999.

[168] Chuanqi Wang, Yanhui Li, Lin Chen, Wenchin Huang, Yuming Zhou, and Baowen Xu. Examining the effects of developer familiarity on bug fixing. *Journal of Systems and Software*, 169:110667, 2020.

[169] Rui Wang, Shuo Chen, and XiaoFeng Wang. Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services. In *2012 IEEE Symposium on Security and Privacy*, pages 365–379. IEEE, 2012.

[170] Rui Wang, Shuo Chen, XiaoFeng Wang, and Shaz Qadeer. How to shop for free online–security analysis of cashier-as-a-service based web stores. In *2011 IEEE symposium on security and privacy*, pages 465–480. IEEE, 2011.

[171] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. An empirical study of usages, updates and risks of third-party libraries in java projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 35–45. IEEE, 2020.

[172] Ying-Jie Wang, Liang-Ze Yin, and Wei Dong. Amchex: Accurate analysis of missing-check bugs for linux kernel. *Journal of Computer Science and Technology*, 36:1325–1341, 2021.

[173] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 32–41, 2007.

[174] Michael Weissbacher, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. {ZigZag}: Automatically hardening web applications against client-side validation vulnerabilities. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 737–752, 2015.

[175] Chengcheng Xiang, Yudong Wu, Bingyu Shen, Mingyao Shen, Haochen Huang, Tianyin Xu, Yuanyuan Zhou, Cindy Moore, Xinxin Jin, and Tianwei Sheng. Towards continuous access control validation and forensics. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pages 113–129, 2019.

[176] Aiko Yamashita and Leon Moonen. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 682–691. IEEE, 2013.

[177] Evan You. Vue.js - the progressive javascript framework — vue.js. https://vuejs.org/, 2023.

[178] Asimina Zaimi, Apostolos Ampatzoglou, Noni Triantafyllidou, Alexander Chatzigeorgiou, Androklis Mavridis, Theodore Chaikalis, Ignatios Deligiannis, Panagiotis Sfetsos, and Ioannis Stamelos. An empirical study on the reuse of third-party libraries in open-source software development. In *Proceedings of the 7th Balkan Conference on Informatics Conference*, pages 1–8, 2015.

[179] Jie Zhang, Ziyi Wang, Lingming Zhang, Dan Hao, Lei Zang, Shiyang Cheng, and Lu Zhang. Predictive mutation testing. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 342–353, 2016.

[180] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M. Azab, and Ruowen Wang. Pex: A permission check analysis framework for linux kernel. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 1205–1220, USA, 2019. USENIX Association.

[181] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Locating faults through automated predicate switching. In *Proceedings of the 28th international conference on Software engineering*, pages 272–281, 2006.

[182] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. Z3-str: A z3-based string solver for web application analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 114–124, 2013.

[183] Li Zhong. A survey of prevent and detect access control vulnerabilities. *arXiv preprint arXiv:2304.10600*, 2023.

[184] Li Zhong and Zilong Wang. Can chatgpt replace stackoverflow? a study on robustness and reliability of large language model code generation, 2023.

[185] Li Zhong and Zilong Wang. A study on robustness and reliability of large language model code generation. *arXiv preprint arXiv:2308.10335*, 2023.

[186] Li Zhong, Chengcheng Xiang, Haochen Huang, Bingyu Shen, Eric Mugnier, and Yuanyuan Zhou. Effective bug detection with unused definitions. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 720–735, 2024.

[187] Lily Zhong, Zilong Wang, and Jingbo Shang. Ldb: A large language model debugger via verifying runtime execution step-by-step. *arXiv preprint arXiv:2402.16906*, 2024.

[188] Yuchen Zhou and David Evans. Ssoscan: Automated testing of web applications for single sign-on vulnerabilities. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 495–510, 2014.

[189] Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. Authscope: Towards automatic discovery of vulnerable authorizations in online services. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 799–813, 2017.