Author/s:
Pan, Lianglu

Title:
Advancing Practical Automated Security Testing for Web Applications

Date:
2025

Persistent Link:
https://hdl.handle.net/11343/357123

Advancing Practical Automated Security Testing for Web Applications

by

Lianglu PAN
ORCID: 0009-0005-8803-1008

A thesis submitted in total fulfillment for the
degree of Doctor of Philosophy

in the
School of Computing and Information Systems
Faculty of Engineering and Information Technology
THE UNIVERSITY OF MELBOURNE

July 2025

THE UNIVERSITY OF MELBOURNE

Abstract

School of Computing and Information Systems
Faculty of Engineering and Information Technology

Doctor of Philosophy

by Lianglu PAN
ORCID: 0009-0005-8803-1008

Web applications are increasingly central to modern digital infrastructure, yet they remain highly susceptible to security vulnerabilities. Automated testing offers a scalable path toward improving web application robustness, but faces persistent challenges—including the oracle problem, the lack of formal specifications, and the difficulty of detecting subtle vulnerabilities like excessive data exposures.

This thesis advances practical automated security testing by rethinking how human interaction artifacts—such as captured traffic and behavioral relationships—can be leveraged to reduce manual effort while maintaining high testing effectiveness. It introduces EDEFuzz, a novel fuzzing tool for detecting *Excessive Data Exposure (EDE)* vulnerabilities in web APIs. EDEFuzz applies metamorphic testing principles to compare related web responses, enabling effective and scalable detection of sensitive data leaks without requiring exhaustive manual specifications. A controlled user study demonstrates that EDEFuzz improves both the accuracy and efficiency of EDE detection while reducing user cognitive load.

The thesis also presents Trailblazer, an end-to-end fuzzing framework designed to uncover server-side crashes in undocumented or poorly documented APIs. Trailblazer bootstraps its test generation process from real-world API traffic, autonomously identifying endpoints and inferring payload structures without needing an API specification. This black-box approach surfaces crash-inducing edge cases that are often missed by existing tools, highlighting its applicability to complex, real-world web systems.

Together, these contributions demonstrate that reusing naturally generated human interaction artifacts can enable practical, low-overhead, and scalable security testing. The findings point toward new directions for building intelligent, human-informed testing systems that better match the realities of modern web development.

Declaration of Authorship

I, Lianglu PAN, declare that this thesis titled, 'Advancing Practical Automated Security Testing for Web Applications' and the work presented in it are my own. I confirm that:

- The thesis comprises only my original work towards the degree of Doctor of Philosophy except where indicated in the preface;
- due acknowledgement has been made in the text to all other material used; and
- the thesis is fewer than the maximum word limit in length, exclusive of tables, maps, bibliographies and appendices as approved by the Research Higher Degrees Committee.

Lianglu PAN
April 2025

Preface

This thesis is submitted in total fulfilment of the requirements for the degree of Doctor of Philosophy at the University of Melbourne. The research presented here was primarily conducted at the School of Computing and Information Systems, The University of Melbourne, under the supervision of Dr. Van-Thuan Pham, Prof. Toby Murray and Dr. Shaanan Cohney.

Below is the list of publications and manuscripts arising from this thesis. I was the principal author of all work and contributed more than 50% to each work. I was responsible for designing the algorithm architectures, implementation, running experiments, collecting data, and analysing experimental results. My co-authors contributed with suggestions and feedback, as well as the revisions of the manuscripts. Ethics approval to conduct the studies comprising this thesis was provided by The University of Melbourne's human ethics committee (Chapters 4, ID: 24632).

- Contents of Chapter 3 have been published in the following paper: Lianglu Pan, Shaanan Cohney, Toby Murray, and Van-Thuan Pham. Edefuzz: A web api fuzzer for excessive data exposures. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12, 2024.
- Contents of Chapter 5 have been published in the following paper: Lianglu Pan, Shaanan Cohney, Toby Murray, and Van-Thuan Pham. Trailblazer: Practical end-to-end web api fuzzing (registered report). In *Proceedings of the 4th International Fuzzing Workshop*, pages 1–10, 2025.

I gratefully acknowledge Melbourne Research Scholarship for funding.

Acknowledgements

Completing a PhD is an incredible journey, one that's intellectually demanding, emotionally intense, and deeply rewarding. I could not have reached this point without the support, encouragement, and companionship of many people along the way.

First and foremost, I would like to express my deepest gratitude to my supervisors, Dr. Van-Thuan Pham, Prof. Toby Murray and Dr. Shaanan Cohney. Thank you for believing in me and guiding me with patience and wisdom through every phase of this journey. Your dedication, insightful feedback, and unwavering support have been instrumental to my development as a researcher. Your sharp intellect, high standards, and generosity with your time have greatly shaped my thinking and work. I am fortunate to have had the opportunity to learn from all of you.

I would also like to extend my heartfelt thanks to my thesis committee Chairs, Prof. Egemen Tanin, Prof. Alistair Moffat and Prof. Shanika Karunasekera, for their invaluable feedback in each of progress review meeting. The insightful comments and suggestions you shared helped me to refine my research and improve the quality of my work. Your expertise and perspectives have been invaluable in shaping my understanding of the field.

To my friends who made this journey lighter and more joyful—Chenyuan Zhang, Guang Hu, Yujing Jiang, and Ruihan Zhang—thank you for your companionship, support, and shared laughter. Whether it was deep discussions about research or unwinding over meals and games, your presence made all the difference. Being the last among us to start—and now finish—a PhD, I've greatly benefited from the valuable advice and experiences you generously shared. Your guidance smoothed the path ahead of me and made the journey far less daunting.

I'm also grateful to my labmates in our research group: Pengbo Yan, Faxing Wang, Zhenzhi Lai, Zhiyuan Zhang, Wentao Gao, Tian Qiu, and Wenqi Yan. Thank you for your camaraderie and collaborative spirit. From discussing research ideas and progress, to late nights in the lab scrambling to meet conference deadlines, your presence made the tough times more manageable and the victories more meaningful. I fondly remember our board game nights, badminton sessions, weekend getaways, and countless lunches and dinners together—they brought much-needed balance and joy to this journey. Our shared experiences made the lab not just a workplace, but a community I'll always cherish.

To my partner, Xiran Xie—thank you for your love, patience, and unwavering support throughout this journey. The first three years of my PhD were especially difficult as we

were separated by closed borders during the pandemic, with you in China and me in Australia. Despite the distance, you were always there for me—cheering me on, listening through countless video calls, and grounding me when things got tough. Your strength and understanding carried me through many moments of doubt and exhaustion. I'm so grateful for your presence in my life, and I couldn't have done this without you.

Most importantly, I wish to thank my parents for their unconditional love and encouragement. To my father, Daqing Pan, and my mother, Jian Lu—thank you for your constant support, even from afar. Your strength, care, and belief in me have been a steady source of motivation. This thesis would not have been possible without your sacrifices and unwavering faith in my journey.

Thank you all for being part of this chapter in my life.

Lianglu Pan

Contents

List of Figures

List of Tables

Abbreviations

API	**A**pplication **P**rogramming **I**nterface
DOM	**D**ocument **O**bject **M**odel
EDE	**E**xcessive **D**ata **E**xposure
HTTP	**H**yper **T**ext **T**ransfer **P**rotocol
NIST	**N**ational **I**nstitute of **S**tandards and **T**echnology
OAS	**O**pen**A**PI **S**pecification
OWASP	**O**pen **W**eb **A**pplication **S**ecurity **P**roject
REST	**R**epresentational **S**tate **T**ransfer
SUT	**S**ystem **U**nder **T**est
UI	**U**ser **I**nterface
URL	**U**niform **R**esource **L**ocator
XSS	**Cross**-**S**ite **S**cripting

Chapter 1

Introduction

1.1 Overview

Web applications are an integral part of today's software ecosystem, powering a wide range of services from e-commerce and online banking to social media and enterprise systems. Their ubiquity and importance continue to grow as more functionality is moved to the web. According to recent industry reports [1], the usage of web APIs—an essential component of web applications—has surged in recent years, reflecting the increasing reliance on web-based architectures.

Given their critical role, testing web applications is essential but remains inherently challenging [2, 3]. Failures or vulnerabilities can lead to severe consequences, including data breaches, service disruptions, and reputational damage [4]. Effective testing must ensure not only functional correctness but also security and robustness across a wide range of input and interaction patterns. However, generating meaningful test cases and designing effective test oracles is particularly difficult in this domain. Web applications often involve asynchronous communication, dynamic content generation, and stateful behavior [3, 5, 6], while their expected behaviors are frequently implicit and hard to formalize [7]. These challenges complicate automated test generation and evaluation.

Automated testing has become the most scalable and cost-effective way to find bugs in web applications [8–10]. Compared to manual approaches, automated tools can explore large input spaces and diverse interaction sequences with minimal human effort.

Techniques such as fuzzing, symbolic execution, and dynamic analysis have been successfully applied to detect a variety of software flaws. However, their effectiveness depends heavily on access to formal specifications, well-defined test oracles, or exhaustive input models—requirements that are often missing in real-world web systems.

While many existing approaches either rely heavily on specifications or aim for full automation with minimal contextual awareness, there remains a gap in leveraging targeted automation informed by the natural by-products of human interaction. In particular, there is an opportunity to bridge technically grounded fuzzing techniques with lightweight, incidental human input—such as user-generated traffic or UI-triggered API calls—to navigate the middle ground between manual and fully autonomous testing. This thesis explores that space by proposing methods that selectively reuse information generated during routine web application use to inform and enhance automated testing. Rather than requiring users to explicitly configure tests or write specifications, our approach extracts useful artifacts from natural behavior, allowing even non-technical participants to contribute to vulnerability detection indirectly. This enables practical, end-to-end testing techniques that operate with minimal manual setup while still uncovering deep and impactful bugs in realistic applications.

We target two critical classes of web application bugs: (1) *excessive data exposures* (EDEs), where APIs unintentionally leak sensitive data [11], and (2) *crashes in undocumented APIs*, which may result from malformed inputs, weak error handling, or incorrect assumptions about endpoint usage. These two bug classes were chosen not only for their security and reliability impact, but also for the unique challenges they pose to existing automated testing methods. Some bugs, like EDEs, are particularly difficult to detect because they lack clear or easily automated test oracles—what constitutes "excess" data is often subjective and context-dependent, making it easier for humans to recognize but hard to encode programmatically. Similarly, crashes in undocumented APIs are challenging because the most effective testing strategies typically rely on detailed specifications—specifications that are either incomplete or absent for these endpoints. As a result, these bugs tend to be overlooked in both research and practice, requiring either intensive manual effort or domain knowledge to uncover. This makes them ideal candidates for the approach proposed in this thesis, which seeks to smooth out such difficulties by reusing artifacts produced during natural human interaction with web applications. By leveraging what users already generate—such as traffic traces and

behaviorally coupled responses—we demonstrate that it is possible to surface subtle, high-impact bugs with low manual effort and without the need for manually-written formalized input models.

In light of these challenges, this thesis aims to rethink how automated security testing for web applications can be made more practical, scalable, and effective, particularly in environments where traditional specifications, oracles, and expert-driven testing are infeasible. By leveraging artifacts naturally produced during human interaction, we seek to design testing techniques that minimize manual engineering effort while uncovering deep, context-sensitive vulnerabilities.

To realize this vision, we focus on four core research questions that collectively explore new methods for detecting subtle web application flaws, designing low-overhead automation strategies, and ensuring that the resulting tools are both technically effective and practically usable.

1.2 Research Problems

The research questions that guide this study are as follows:

RQ1: How can excessive data exposure vulnerabilities be effectively detected in web applications?

RQ2: Can automated fuzzing uncover server-side crashes in undocumented or poorly documented web APIs without requiring prior specifications?

RQ3: How can information produced during natural human interaction with a web application be leveraged to enhance automated web application testing?

RQ4: How user studies can be designed to evaluate the effectiveness and usability of web application fuzzing tools?

The research questions addressed in this thesis are motivated by pressing security challenges in modern web applications. These challenges stem from the increasing complexity and ubiquity of web APIs, the subtlety of certain vulnerability classes, and the need

to build tools that are not only effective but also usable by developers. Each research question targets a critical gap in current testing and vulnerability detection practices.

The first question addresses a relatively underexplored but increasingly important security concern. Excessive data exposures (EDEs) occur when web applications unintentionally reveal sensitive or contextual information via APIs, even though the disclosure may appear benign in isolation. Unlike traditional vulnerabilities such as SQL injection or cross-site scripting, excessive data exposures (EDEs) involve violations of intended data access boundaries rather than injection of malicious inputs. EDEs often arise due to flawed logic, poor access control, or miscommunication between frontend and backend systems. Detecting them is challenging because, like some other vulnerabilities, they may not cause crashes or visible misbehavior [12, 13]. Instead, identifying EDEs often requires a deep understanding of which data should or should not be transmitted under specific contexts, making conventional automated testing approaches insufficient. Detecting such flaws necessitates novel testing paradigms—such as metamorphic testing—that focus not on individual outputs, but on relational properties between multiple executions. This question is both important and challenging due to the lack of formal specifications defining "excessiveness" and the difficulty of establishing ground truth in automated settings.

The second question addresses a widespread limitation in current web testing tools. Many real-world APIs, particularly internal or rapidly evolving ones, lack comprehensive documentation or formal interface definitions like OpenAPI [14]. As a result, traditional fuzzers that depend on specifications for input generation are rendered ineffective. This question is crucial because undocumented APIs are often among the most vulnerable, as they receive less scrutiny and are more likely to suffer from inconsistencies and latent bugs. Automatically detecting crash-inducing behavior without prior knowledge requires understanding of API endpoints and payload structure. Solving this problem pushes the boundary of black-box and specification-less fuzzing, making it highly relevant to both academia and industry.

The third question builds on the insight that human interactions with web applications—whether during development, testing, or regular usage—naturally generate valuable information that can inform automated testing. This includes HTTP traffic, UI behaviors, sequences of API calls, and context-specific outputs. By repurposing these artifacts, it

may be possible to reduce the need for hand-crafted specifications or manually written test cases, while improving the relevance and effectiveness of automated tests. This question is particularly significant because it opens up a new paradigm for low-effort, high-impact testing: allowing non-technical users to unknowingly contribute meaningful data that enhances test coverage and vulnerability detection. Exploring this question involves both technical and methodological challenges, such as extracting useful signals from noisy interaction traces and designing testing frameworks that can learn and generalize from them.

The fourth question is essential for bridging the gap between research prototypes and tools that developers will actually adopt. Security tools are often evaluated solely in terms of technical performance, such as code coverage or number of bugs found. However, if a tool is too complex, unintuitive, or time-consuming, it is unlikely to see real-world use, regardless of its capabilities. Designing effective user studies to assess developer experience introduces several methodological challenges, including identifying appropriate tasks, measuring subjective perceptions like trust and confidence, and accounting for participants' varied backgrounds. This question matters because usability directly impacts the practical value of a tool: a moderately effective tool that is easy to use may be more valuable in practice than a highly effective one that is difficult to integrate. Moreover, insights from user studies can inform future improvements, enabling iterative development that aligns more closely with real-world needs.

Together, these questions form the backbone of this thesis. They reflect a commitment to advancing both the technical rigor and practical impact of web application testing research, with contributions spanning vulnerability detection techniques, tool development, and empirical evaluation. More importantly, taken together, these questions address a broader problem at the heart of modern security testing: how to reduce the manual effort traditionally required to test complex web applications, without sacrificing precision or depth. By targeting underexplored classes of bugs that resist conventional automation (RQ1 and RQ2), investigating new sources of test-relevant information rooted in human interaction (RQ3), and ensuring that resulting tools are usable and actionable in real-world settings (RQ4), this thesis aims to unlock a new space in the effort-effectiveness tradeoff. It offers a path toward practical, low-effort testing approaches that remain capable of surfacing deep and impactful vulnerabilities in today's web systems.

Each of these research questions is addressed through the technical contributions and evaluations presented in this thesis. **RQ1** is tackled through the design and evaluation of EDEFuzz, described in Chapter 3, which focuses on detecting excessive data exposure vulnerabilities in web APIs. **RQ2** is addressed by the development of Trailblazer, detailed in Chapter 5, which explores automated fuzzing techniques to uncover server-side crashes in undocumented APIs without requiring formal specifications. **RQ3** is interwoven throughout both EDEFuzz and Trailblazer, as both systems leverage artifacts produced during natural human interactions to guide automated testing. Finally, **RQ4** is the focus of the user study presented in Chapter 4, which systematically evaluates the usability and effectiveness of EDEFuzz from the perspective of real-world users. Together, these chapters collectively demonstrate how targeted automation, informed by naturally occurring interaction data, can achieve the broader thesis goal of practical, low-effort yet high-effectiveness web application security testing.

1.3 Research Contributions

This thesis addresses a fundamental challenge in web application testing: how to achieve high testing effectiveness while minimizing manual effort and reliance on formal specifications or expert engineering. By leveraging artifacts naturally produced during human interaction with web applications—such as API traffic and user interface behaviors—we demonstrate that it is possible to build practical, automated testing tools that uncover deep and subtle vulnerabilities with minimal setup. This strategy opens up a new design space between manual testing and traditional automation, allowing even non-technical users to indirectly contribute to security testing workflows.

Building on this overarching idea, this thesis makes several concrete contributions spanning methodological innovations, tool development, empirical evaluation, and practical impact.

First, we developed EDEFuzz, a novel approach and tool designed to detect excessive data exposure (EDE) vulnerabilities in web applications. Unlike traditional web fuzzers that mutate input requests, EDEFuzz introduces the idea of mutating response content to uncover unused data fields. This approach is rooted in the principles of metamorphic testing [15, 16], where we define a metamorphic relation to express whether the web page

should change—or remain stable—when mutated server responses are issued. EDEFuzz supports developers and security analysts in identifying instances where sensitive or unnecessary information is revealed unintentionally, often in subtle or context-dependent ways that bypass standard checks.

Second, to evaluate the effectiveness and usability of EDEFuzz, we conducted a controlled experiment and user study involving participants with varying levels of experience in web security. The study measured the ability of participants to detect EDE vulnerabilities with and without the assistance of EDEFuzz, assessing both objective performance metrics (such as vulnerability identification and time taken) and subjective measures (such as perceived usability and confidence). The results demonstrate that EDEFuzz enhances both the efficiency and accuracy of vulnerability discovery, while also reducing the cognitive load on users. The study design and outcomes contribute to the growing body of empirical work on evaluating security tools in realistic development settings.

Third, we introduce Trailblazer, an end-to-end web API fuzzing framework that targets server crashes in undocumented APIs—a class of vulnerabilities often overlooked by specification-driven tools. Trailblazer is capable of autonomously analyzing the collected web traffic and generate test cases for web APIs without requiring an API specification or prior knowledge of the endpoints. It combines generation-based and mutation-based fuzzing and utilizes the server response code to indicate server crashes. Trailblazer is particularly effective in discovering server crashes that only occur under malformed inputs, making it well suited for testing complex, real-world web services. While Trailblazer is designed to focus on crashes, its detection capability colud be expended with new test oracles for other types of web application bugs.

Both EDEFuzz and Trailblazer instantiate the broader thesis principle: that meaningful security testing can be built upon artifacts captured during natural human interaction. By reusing these naturally occurring behaviors and workflows, our approach minimizes the need for hand-crafted specifications and expert-driven configuration, while maintaining strong practical effectiveness. This allows security testing processes to scale with real-world development practices, enabling broader and more continuous coverage without imposing prohibitive effort requirements.

As part of the evaluation of both EDEFuzz and Trailblazer, we tested a range of widely used web applications. In doing so, we discovered previously unknown bugs and vulnerabilities that had not been identified by existing tools or disclosed by vendors. Upon discovering these issues, we followed responsible disclosure practices and submitted detailed reports to the respective project maintainers. Several of these vulnerabilities have since been acknowledged and patched, demonstrating the practical security impact of this work. These findings underscore the need for more advanced, practical testing strategies and contribute directly to improving the security of the web ecosystem.

This work has also achieved significant practical recognition. A patent application for EDEFuzz has been filed [17], protecting the novel approach of using metamorphic testing to detect excessive data exposures in web APIs. Furthermore, the industrial relevance of the proposed tools is evidenced by two letters of support from leading companies in the web security domain. One of these companies has actively begun piloting the Trailblazer tool within their internal API testing workflows, highlighting the potential of this research to influence real-world security practices and tooling.

Finally, this thesis provides a discussion of the broader implications of our findings for web application security. We reflect on how the design of fuzzing tools and testing frameworks must evolve to address modern challenges such as stateful APIs, implicit security assumptions, and the increasing use of undocumented interfaces. We also outline potential future research directions, including the integration of human feedback, the application of static analysis to guide dynamic testing, and the extension of metamorphic testing principles to other domains. By addressing both theoretical and practical aspects, this work aims to inform future efforts in the secure development and testing of web applications.

1.4 Thesis Outline

The remainder of this thesis is organized to address the research questions introduced in Section 1.2 through a series of technical contributions and evaluations.

Chapter 2 lays the foundation for the rest of the thesis by providing a comprehensive background on web applications and their security landscape. It introduces key concepts in web technology, including the architecture and operation of web APIs, which serve as

the primary interface for modern web applications. This chapter also surveys software testing methodologies, with an emphasis on fuzzing techniques that aim to automate the discovery of vulnerabilities. A thorough review of related work in both vulnerability detection and automated testing is presented, highlighting gaps in existing research that motivate the contributions of this thesis.

Chapter 3 presents the design and implementation of EDEFuzz, a novel approach and tool for detecting excessive data exposure (EDE) vulnerabilities in web applications. The chapter begins by defining EDE and explaining why it represents a distinct and under-addressed class of security issues. The system architecture and design rationale behind EDEFuzz are detailed, including the metamorphic relation it relies on and the mutation strategy. Implementation choices are discussed in the context of offline testing and scalability with minimal manual configuration. The chapter concludes with an evaluation of EDEFuzz against real-world applications, demonstrating its ability to uncover sensitive data leaks that are overlooked by existing tools.

Chapter 4 reports on a user study designed to evaluate the effectiveness and usability of EDEFuzz when used by developers and security analysts. The study measures how well participants can identify and understand EDE vulnerabilities using the tool, compared to manual testing. Methodological details are provided, including the study design, participant demographics, and analysis techniques. Results show that EDEFuzz not only improves vulnerability detection but also supports better decision-making by presenting clearer indicators of risk. User feedback is analyzed to highlight strengths, limitations, and areas for improvement in the tool's interface and workflow.

Chapter 5 introduces Trailblazer, a practical end-to-end testing framework developed to explore crashes in web APIs, without the need of an API specification. This chapter describes the motivation for targeting undocumented APIs, which are often overlooked by specification-based tools but may expose critical backend functionality. The architecture and key components of Trailblazer are presented, including the strategy to identify endpoints, the process to infer the payload structure, and the way it produces new test cases. Trailblazer combines both generation-based and mutation-based fuzzing, which was demonstrated to be effective accross different domains [18–20]. The chapter concludes with case studies that illustrate Trailblazer's ability to detect multiple new bugs in widely used web applications.

Chapter 6 synthesizes the findings of the previous chapters and discusses their broader implications for web application testing and security. It reflects on how the contributions of EDEFuzz and Trailblazer push the boundaries of current vulnerability detection methods by addressing overlooked but impactful categories of bugs. The chapter explores opportunities for extending these approaches through deeper integration with static analysis, machine learning, and human-in-the-loop testing. It also outlines future directions in web security research, such as automated reasoning about data sensitivity, improving oracle generation, and scaling these techniques across large microservice architectures. The thesis is concluded by summarizing the key findings and contributions.

1.5 Thesis Statement

This thesis demonstrates that security testing for modern web applications can be made both practical and effective by leveraging artifacts generated through natural human interactions, such as API traffic and UI behavior, to automate the discovery of deep, context-sensitive vulnerabilities with minimal manual effort or formal specifications.

Chapter 2

Background and Related Work

2.1 Web Applications

Web applications have become an essential part of modern computing, supporting a wide range of services such as online banking, e-commerce, healthcare systems, social media, and enterprise platforms. Their ubiquity and accessibility make them a primary interface between users and digital services. As organizations increasingly migrate services to the web, the scale and complexity of web applications have also grown substantially.

Despite their benefits, web applications remain a common target for security attacks. According to the OWASP Top Ten [4], a regularly updated industry-standard reference for web application security risks, several vulnerabilities—including broken access control, injection attacks, and sensitive data exposure—continue to affect a significant portion of deployed applications. These risks are amplified by the distributed nature of web applications, their reliance on third-party components, and the challenge of securing dynamically generated content.

Ensuring the reliability and security of web applications is therefore critical. However, testing web applications poses unique challenges. Unlike traditional desktop software, web applications operate in a heterogeneous environment involving client-server communication, asynchronous requests, and complex interactions between UI and backend APIs. These characteristics introduce difficulty in replicating user behavior, generating relevant test inputs, and verifying correctness of application behavior.

Consequently, the development of effective and automated testing approaches for web applications has become an active area of research [8, 21–25]. This thesis contributes to this space by exploring automated techniques that reduce manual effort and improve coverage, with a particular focus on uncovering subtle or hard-to-trigger bugs in real-world web systems.

2.2 Web APIs

2.2.1 Overview

Web APIs (Application Programming Interfaces) have become a foundational element of modern software development, powering a vast range of applications across multiple industries. From fetching weather data to checking stock availability, web APIs are widely used to facilitate communication between client applications and remote servers. Their ability to enable seamless integration of services and systems has made them indispensable in areas such as e-commerce, social media, finance, IoT, and more.

At its core, a web API allows one system to interact with another over the Internet using the HTTP protocol [26]. A typical web API request includes essential components such as the request method (e.g., `GET`, `POST`), a URL (Uniform Resource Locator), headers, and an optional body. A response from the server consists of a status code (e.g., `200 OK`, `404 Not Found`), headers, and a response body that contains the requested data or an error message.

For example, consider the HTTP request for fetching weather information, and its response, shown in Listing 2.1. The `GET` represents the HTTP method used in this request. `GET`, `POST`, `PUT` and `DELETE` are the most commonly used HTTP methods in web APIs, they typically indicate that the client is requesting data, creating data, updating data, or deleting data, respectively. The `/weather` is the API endpoint, and at least canonically represents a discrete functionality on the server (in this case, fetching weather information). The `?city=Sydney` is the query parameter, telling the server to fetch the weather information for the given location. Note that for many web APIs, it is common to allow encapsulating a query parameter into the endpoint (e.g. `GET /weather/Sydney`), depending on the server implementation. The next two lines beginning with `Host:` and

`Accept:` are HTTP headers, specifying where the API is hosted, and indicates the client expects the server to return the data in JSON format. Additional headers may also be present for purposes such as authentication and authorization.

LISTING 2.1: An API request (left) for fetching weather information, and its corresponding response (right).

```
GET /weather?city=Sydney HTTP/1.1
Host: api.weatherprovider.com
Accept: application/json
```

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 60

{"city":"Sydney", "temperature":"18°C",
↪  "condition":"Sunny"}
```

Responses from the API follow typical HTTP semantics. See Listing 2.1, which contains the HTTP status code (`200 OK`), followed by remaining metadata headers, and then finally the response body. API requests most typically follow a single request from a client, followed by a single response, which is subsequently handled by the client.

To formalize the syntax and semantics of requests and responses, developers often use a formal specification, such as an *OpenAPI Specification*.

2.2.2 OpenAPI Specification

The OpenAPI Specification is a standard, language-agnostic format for describing RESTful (Representational State Transfer) web APIs [27]. It defines how API requests should be structured, including the endpoints, methods, parameters, and expected responses, making it easier for developers to understand and interact with APIs. An OpenAPI document serves as both a reference and a contract between API developers and users, enabling clear communication of how the API behaves and what responses to expect.

In practice, OpenAPI specifications can be produced through a variety of methods depending on the development workflow, the complexity of the API, and the tools or frameworks in use. Many modern web frameworks offer automated generation of OpenAPI specifications. For instance, frameworks such as Spring Boot (Java), FastAPI (Python), and ASP.NET Core (C#) provide built-in support to automatically produce OAS from the defined routes and data models. These tools leverage the application's existing codebase to generate accurate and up-to-date API documentation, reducing the maintenance overhead.

2.2.2.1 Key Features

The OpenAPI Specification offers several key features that make it an essential tool for API development:

- **Comprehensive API Documentation:** OAS enables the creation of detailed API documentation that is both human-readable and machine-readable.
- **Code Generation:** Tools supporting OAS can generate client libraries, server stubs, and API documentation directly from the specification file.
- **Interoperability:** By using a standard format, OAS promotes interoperability between different tools and frameworks in the API ecosystem.
- **Testing and Validation:** The specification can be used to automate API testing and validate requests and responses for conformance.

2.2.2.2 Structure

An OpenAPI document is typically written in either YAML or JSON format and consists of the following main sections:

- **OpenAPI Version:** Specifies the version of the OpenAPI Specification being used.
- **Info:** Provides metadata about the API, such as its title, version, and description.
- **Servers:** Lists the base URLs where the API is hosted.
- **Paths:** Describes the API endpoints (paths) and the operations available for each endpoint.
- **Components:** Defines reusable objects, such as schemas, parameters, and responses, to avoid redundancy.
- **Security:** Specifies the security mechanisms, such as API keys, OAuth2, or basic authentication, required to access the API.
- **Tags:** Groups operations by category for better organization.

Listing 2.2 shows the OpenAPI Specification for the weather API illustrated in Listing 2.1. This OpenAPI Specification defines the `GET /weather` endpoint, the `city` query parameter, and expected responses.

Listing 2.2: The OpenAPI Specification for the API shown in Listing 2.1.

```yaml
openapi: 3.0.0
info:
  title: Weather API
  description: This API provides weather information for a specified city.
  version: 1.0.0
servers:
  - url: https://api.example.com
paths:
  /weather:
    get:
      description: Returns the current weather conditions for the specified city.
      parameters:
        - name: city
          in: query
          required: true
          description: The name of the city for which weather information is being
          ↪  requested.
          schema:
            type: string
      responses:
        '200':
          description: A JSON object containing the weather details for the city.
        '400':
          description: Bad Request, e.g., missing or invalid city name.
```

2.2.2.3 Advantages

The OpenAPI Specification (OAS) provides a standardized, machine-readable description of RESTful APIs, enabling developers and testers to understand how to interact with a service without inspecting its underlying implementation. A well-defined specification outlines endpoints, request/response formats, and parameters, serving as a single source of truth for both human and automated processes.

The adoption of OAS brings several advantages:

- **Improved Collaboration:** A shared specification aligns development, testing, and operations teams, streamlining communication and reducing ambiguity.
- **Automation Support:** OAS enables automation in documentation generation, mock server creation, test case generation, and CI/CD pipelines.
- **Standardization and Interoperability:** Conforming to a common format makes APIs easier to consume, integrate, and maintain across tools and systems.

Overall, the OpenAPI Specification has become a cornerstone of modern API ecosystems, simplifying the design, documentation, and testing processes, while promoting more robust and reliable software development practices.

2.2.2.4 Challenges in OAS Adoption

Despite these advancements, not all web development frameworks or pipelines support automatic OAS generation. In such cases, developers may need to rely on third-party libraries or continue with manual documentation efforts. Moreover, even among APIs that do have specifications, a significant number suffer from "specification drift", where the documentation becomes outdated or inconsistent with the actual API implementation [28–30]. A study by APIContext revealed that 75% of their tested production APIs had discrepancies between their implementations and their published OpenAPI specifications [14]. Furthermore, the adoption of OpenAPI specifications across the web is not yet universal. According to the same study, only 57% of APIs have public API specifications, indicating that a substantial portion of APIs lack formal documentation. This gap underscores the need for broader adoption of standardized API documentation practices to enhance interoperability and maintainability in web services.

To mitigate the challenges of manual specification and specification drift, several tools and research efforts have emerged to assist in generating or inferring OpenAPI specifications from existing web applications. Research tools such as *APIDiscoverer* [31], *APICARV* [32], *Respector* [33], and *RESTSpecIT* [34] leverage techniques like traffic analysis and static analysis to produce or infer an API specification. In practice, developers can also rely on tools like *SwaggerHub Explore* (formerly *Swagger Inspector*) [35] for testing and documenting APIs, or *AppMap* [36] for visualizing and analyzing application behavior to support documentation efforts. These solutions play a crucial

role in improving specification coverage and accuracy, especially for legacy systems or applications lacking up-to-date documentation.

2.3 Software Testing

2.3.1 Overview

Software testing is a critical component of the software development lifecycle, aimed at ensuring the quality, functionality, reliability, and security of software systems. It involves systematically evaluating a program against its specified requirements to identify defects, ensure correctness, and validate that the system behaves as expected under various conditions [37]. By uncovering errors early in the development process, software testing helps reduce the cost of fixing bugs, enhances user satisfaction, and mitigates potential security risks.

Modern software testing encompasses a wide range of techniques, broadly classified into static and dynamic testing [38–40]. Static testing involves analyzing the code or documentation without executing the program, focusing on identifying potential issues such as coding standard violations, security vulnerabilities, or logical errors. Common static testing techniques include code reviews, walkthroughs [41, 42], and the use of automated tools like linters or static analyzers [43–46].

Dynamic testing, on the other hand, evaluates the system by executing it with different inputs and observing its behavior. This approach can be further categorized into functional and non-functional testing. Functional testing ensures that the software meets its specified requirements, often performed through techniques such as unit testing, integration testing, and system testing. Non-functional testing, in contrast, assesses attributes like performance, scalability, and security, which are crucial for real-world deployment.

Advancements in software testing have introduced automation, enabling testers to efficiently execute large test suites and achieve higher coverage. Techniques like test-driven development (TDD) [47] and continuous integration (CI) [48] pipelines have integrated testing into the development workflow, promoting rapid feedback and iterative improvement. Among dynamic testing methods, fuzzing has gained prominence for its ability

to uncover critical vulnerabilities by systematically generating and testing unexpected inputs [49, 50].

2.3.2 The Test Oracle Problem

One of the fundamental challenges in software testing is the *test oracle problem*—determining whether the output of a program for a given input is correct [51]. While generating test inputs can often be automated using techniques such as random testing, symbolic execution, or model-based generation, verifying the correctness of outputs remains a significant bottleneck in many testing scenarios.

In its most basic form, a test oracle is a mechanism that determines whether a program behaved as expected for a given test input. In practice, however, many systems lack a precise specification or a reliable ground truth against which to validate behavior. This issue is particularly prevalent in complex or interactive systems such as web applications, machine learning pipelines, and systems involving nondeterminism, concurrency, or evolving state.

Researchers have proposed various strategies to address the oracle problem:

- **Assertion-based oracles** rely on pre-defined conditions or properties that must hold during execution, such as assertions in code or postconditions in formal contracts. While effective in some cases, these require prior knowledge of expected behavior and can be incomplete.
- **Regression oracles** compare the behavior of a system against previous versions, assuming the older version's behavior was correct. This can reveal unintended changes, but may miss bugs present in both versions.
- **Differential testing** (or cross-implementation testing) compares outputs across multiple implementations of the same functionality [52]. It has been successfully used in testing compilers and database systems, but is limited to cases where multiple comparable systems exist.
- **Metamorphic testing** leverages known relationships between inputs and outputs (metamorphic relations) to derive new test cases and predict output patterns [53].

This strategy is especially useful when the expected output is unknown or hard to define.

- **Human oracles** rely on manual inspection by developers or testers to judge correctness. While sometimes necessary, this is costly, error-prone, and difficult to scale.
- **Specification mining and inferred models** attempt to learn expected behavior by observing system executions, such as through runtime monitoring or analysis of execution logs [54]. These approaches can be helpful when specifications are missing, but may generalize incorrectly.
- **Machine learning-based oracles** are an emerging area where classifiers or anomaly detectors are trained to distinguish between correct and incorrect behavior. These methods are promising but face challenges in explainability, robustness, and data requirements.

Despite decades of research, the oracle problem remains a fundamental obstacle in automating software testing [55]. Each strategy offers trade-offs in terms of accuracy, scalability, and applicability to different domains. In practice, many modern testing frameworks combine multiple oracle strategies, or rely on partial oracles that can validate certain aspects of behavior while ignoring others.

2.3.3 Fuzzing

Fuzzing is one of the most effective and widely used techniques in software testing, particularly for identifying security vulnerabilities and bugs in complex software systems. The technique involves providing random, invalid, or unexpected inputs to a program in order to discover potential flaws, including crashes, memory leaks, and security vulnerabilities such as buffer overflows or unhandled exceptions. Fuzzing has become a cornerstone of modern security testing, especially in the context of vulnerability discovery for large-scale systems, including web applications, operating systems, and network services [56–60].

Fuzzing techniques can be categorized in multiple ways depending on the criteria used. One widely used classification groups fuzzers based on the level of knowledge they possess

about the system under test: *black-box*, *white-box*, and *grey-box* fuzzing. Black-box fuzzing treats the application as a black box and generates test inputs without any knowledge of the internal structure, often relying solely on output anomalies such as crashes. White-box fuzzing, on the other hand, leverages complete knowledge of the program's source code or binary, employing techniques such as symbolic execution to systematically explore execution paths [61]. Grey-box fuzzing strikes a balance by using lightweight instrumentation or runtime feedback—such as code coverage—to guide the input generation process, with tools like AFL [62] exemplifying this approach. This taxonomy helps clarify the trade-offs between precision, scalability, and required domain knowledge across different fuzzing strategies. Another common distinction is between *generation-based* and *mutation-based* fuzzing, each suited for different scenarios and testing needs, discussed below.

2.3.3.1 Mutation-based Fuzzing

Mutation-based fuzzing generates new test cases by making small, random changes to existing inputs. These modifications may include altering specific fields, flipping bits, or inserting unexpected characters, all of which are designed to trigger unexpected behavior or vulnerabilities in the target system. Mutation-based fuzzing is particularly effective for finding anomalies in applications that accept complex, but relatively predictable inputs, such as network protocols or binary formats.

One of the most notable tools in this category is AFL (American Fuzzy Lop) [62], which combines mutation with feedback-driven exploration. AFL guides the fuzzing process using coverage feedback, enabling the fuzzer to focus on untested or under-executed code paths. This method has been highly successful in finding vulnerabilities in widely used software, including open-source projects and proprietary applications.

AFL has served as the foundation for numerous extensions that aim to adapt its powerful mutation-based fuzzing capabilities to new domains. For instance, *AFLNet* [59] extends AFL to effectively fuzz stateful network protocols by incorporating a model of request-response state transitions, enabling it to maintain semantic correctness across sequences of network interactions. *StateAFL* [63] augments AFL by integrating program state information into the fuzzing feedback loop, guiding the exploration toward semantically rich and previously unvisited states. *AFL++* [64] is a community-driven

successor to AFL that incorporates numerous improvements such as better mutation strategies, persistent mode, and custom mutators, offering a highly extensible and modern fuzzing platform. *FairFuzz* [65] enhances input selection by targeting rare branches, using a probability-based model to generate inputs that maximize coverage of hard-to-reach program paths. These innovations illustrate the flexibility of AFL's core design and its continued relevance in modern fuzzing research.

Mutation-based fuzzing has been effectively applied beyond traditional user-space applications, extending to domains such as operating systems [60, 66–68], embedded systems [66, 69, 70], databases [71–73], and compilers [74–76]. *TriforceAFL* [68] extends AFL to enable syscall-level fuzzing of kernel-space code by leveraging QEMU for full-system emulation, uncovering vulnerabilities in OS kernels. *Avatar* [70] targets embedded firmware by combining symbolic execution with fuzzing across real and emulated peripherals, enabling automated bug discovery in deeply embedded systems. For database systems, *Ratel* [72] is a coverage-guided fuzzer designed for enterprise-level database management systems (DBMS), which tackles key challenges in testing large-scale distributed systems and has successfully uncovered dozens of previously unknown bugs in production-grade databases. In the domain of compilers, *Csmith* [75] uses random program generation with mutation strategies to test C compilers, leading to the discovery of hundreds of bugs in widely used compilers.

Overall, mutation-based fuzzing excels in scenarios where inputs can be derived from a base corpus of known test cases, and havs demonstrated its versatility and impact across diverse and complex software systems.

2.3.3.2 Generation-based Fuzzing

In contrast to mutation-based fuzzing, generation-based fuzzing creates test cases from scratch, typically using predefined rules or specifications of the expected input format. This approach is ideal for applications where input constraints are well understood and explicitly defined, such as in the case of SQL queries, XML parsers, or any system requiring structured input.

One of the primary advantages of generation-based fuzzing is its ability to generate inputs that are syntactically correct according to the input grammar. By adhering to

the expected structure of the input, generation-based fuzzing ensures that test cases are valid and can trigger deeper application logic that mutation-based fuzzing might miss due to incorrect or malformed inputs. For example, in a database system, generation-based fuzzing might create well-formed SQL queries with edge-case values to test the system's handling of complex queries.

Tools such as *DynSQL* [77] and *Skyfire* [78] focus on generation-based fuzzing for specific application domains, such as database query engines and web applications, respectively. DynSQL generates SQL queries based on a predefined grammar and uses a feedback loop to iteratively refine inputs to find vulnerabilities in SQL parsers, including SQL injection vulnerabilities. Skyfire, on the other hand, generates and mutates XML documents to test XML parsers, allowing the identification of flaws in XML handling that could lead to security vulnerabilities or data corruption. Other popular work include *Peach* [79], which is a general-purpose fuzzing framework that supports both generation-based and mutation-based fuzzing, *Domato* [80], a document object model fuzzer, *Atheris* [81], a Python fuzzing library that allows for the generation of structured inputs based on user-defined schemas, and *Fuzzilli* [20], a JavaScript engine fuzzer. These tools demonstrate the effectiveness of generation-based fuzzing for discovering vulnerabilities in structured data handling and parsing.

Generation-based fuzzing is highly effective when working with well-documented input formats, providing a more targeted and efficient approach to discovering vulnerabilities in structured data handling.

2.3.3.3 Fuzzing in Software Testing

Fuzzing has gained prominence in the software testing community due to its effectiveness in finding security-critical vulnerabilities that might otherwise go unnoticed through traditional testing methods. Its success is partly attributed to its ability to generate a high volume of diverse inputs, which allows for the exploration of edge cases that may not be covered by human testers or other automated testing techniques.

Moreover, fuzzing's ability to automatically detect crashes, memory leaks, and unexpected behavior makes it a vital tool in security testing. It is particularly crucial in the context of modern software systems, where complex input validation is required to

protect against threats such as buffer overflow attacks, denial of service (DoS), and remote code execution vulnerabilities [82]. As a result, fuzzing is increasingly integrated into continuous integration pipelines and is used in production environments to ensure software robustness.

Despite its success, fuzzing faces certain challenges, including the test oracle problem (i.e., determining the correctness of a test case's result) [51, 55], the difficulty of generating inputs for highly complex or stateful systems [83, 84], and the need for domain-specific knowledge to guide test case generation [85–87]. Despite these challenges, ongoing research continues to refine fuzzing techniques, with approaches like hybrid fuzzing [18–20] combining generation-based and mutation-based methods, demonstrated the effectiveness across different domains.

Overall, fuzzing remains one of the most powerful tools for discovering security vulnerabilities and ensuring the reliability of software systems. By leveraging both mutation-based and generation-based approaches, fuzzing allows for broad coverage and deep exploration of software behavior. As the field evolves, further innovations in feedback mechanisms, state modeling, and hybrid approaches will continue to improve fuzzing's applicability across different domains, making it an essential component of modern software testing.

2.3.4 Metamorphic Testing and Fuzzing

Metamorphic testing is an advanced software testing technique particularly effective in scenarios where the test oracle problem arises—that is, when it is difficult or infeasible to determine the expected output for a given test input [15, 16]. Instead of verifying individual outputs, metamorphic testing focuses on the relationships between multiple inputs and their outputs, known as metamorphic relations. These relations express necessary properties that should hold across transformed inputs and their corresponding outputs, allowing for indirect validation of software correctness.

Consider the following toy example of metamorphic bug finding: we can test a function that reverses a list by testing, for an arbitrary input list x, whether reversing the reverse of x yields x itself; or for a function that calculates distance between a pair of points,

whether the distance from point a to c is always smaller or equal to the sum of the distances from a to b and from b to c, etc.

Metamorphic testing serves both as a method for test case generation and test result verification [15]. By systematically applying transformations to inputs (or outputs) and checking the consistency of the results, it enables the discovery of faults that traditional testing approaches may overlook. Over the years, metamorphic testing has been successfully applied in diverse domains, including machine learning [88, 89], database systems [90–92], compilers [93], machine translation [94–96] and web services [97, 98].

A key challenge in applying metamorphic testing lies in identifying suitable metamorphic relations, which often requires deep domain knowledge and creativity. To address this, Segura et al. proposed six abstract metamorphic relation patterns that serve as reusable templates, from which concrete, domain-specific metamorphic relations can be derived. In their study, they identified 60 web API-specific metamorphic relations, most of which describe how related web requests should lead to related responses, for instance, adding a query parameter might result in a superset of results.

2.4 Web Application Testing

2.4.1 Overview

Web application testing plays a critical role in ensuring the functionality, reliability, and security of modern web systems. As applications grow more complex—often structured as interconnected microservices and exposing public-facing APIs—they become increasingly susceptible to a wide range of vulnerabilities. Effective testing must therefore address not only functional correctness but also robustness against malicious input, protocol compliance, and stateful behavior across interactions.

Research in this area spans both traditional vulnerability detection and modern API-specific testing techniques. Vulnerability detection efforts have extensively explored attacks such as SQL injection, cross-site scripting, and remote code execution, while more recent work has addressed emerging threats like server-side request forgery and excessive data exposure (EDE). In parallel, automated techniques have evolved to test RESTful APIs [84, 100–103], leveraging formal specifications like OpenAPI, dynamic feedback,

and access to source code. These approaches include grammar-based fuzzing, coverage-guided mutation, and metamorphic testing, each aiming to improve test effectiveness and vulnerability discovery.

This section surveys key techniques and tools across web application testing. It also highlights current gaps—such as oracle limitations and challenges with stateful systems—that continue to drive innovation in the field.

2.4.2 Web Application Failures

Failures in web applications are manifestations of underlying software bugs, which can vary widely in visibility and impact. Some bugs are easily observable, such as crashes or exceptions triggered by malformed input; others are silent, resulting in incorrect or insecure behavior without immediate symptoms. Among these, a subset of bugs have security implications and are referred to as *vulnerabilities*.

2.4.2.1 Security Vulnerabilities

A vulnerability is a bug that introduces a security weakness in the design, implementation, or configuration of a web API. Such issues may allow attackers to compromise confidentiality, integrity, or availability of the application. They often stem from missing input validation, flawed authentication and authorization logic, insecure data handling, or misunderstandings between frontend and backend components. Well-known examples include SQL injection, broken access control, server-side request forgery (SSRF), and excessive data exposure (EDE). These vulnerabilities can lead to data breaches, privilege escalation, or unauthorized operations.

Mitigating vulnerabilities involves secure development practices, continuous testing, and vigilant monitoring. Techniques like static analysis [104–106], fuzzing [107–109], and penetration testing [110, 111] are commonly used to uncover vulnerabilities, while guidelines from organizations like OWASP [112, 113] and NIST [114] help standardize best practices across the industry.

2.4.2.2 Other Bugs and Crashes

Not all bugs are exploitable in a security context, but they can still undermine the reliability and robustness of web applications. Crashes—such as segmentation faults, unhandled exceptions, or out-of-memory errors—are often caused by malformed inputs, logic errors, or improper resource management. While these failures may not pose immediate security risks, they can degrade availability, expose implementation details, or become entry points for further investigation.

Detecting such reliability-related bugs relies on systematic input exploration, fault injection, and stress testing. Fuzzing is particularly effective in this domain [84, 100, 101, 115, 116], as it can generate unexpected input combinations that uncover subtle and rarely exercised failure conditions.

Overall, both vulnerabilities and non-security bugs highlight the importance of comprehensive testing strategies. Identifying and addressing both types is essential to building secure, stable, and user-trustworthy web applications.

2.4.3 Vulnerability Detection in Web Applications

Web vulnerability detection focuses on identifying weaknesses in web applications that attackers may exploit to compromise confidentiality, integrity, or availability. The OWASP Top 10 [112] is a widely adopted resource that highlights the most critical security risks to web applications. While the OWASP list does not prescribe a strict technical ranking of individual vulnerabilities, it reflects a consensus view of the most prevalent and impactful classes of risks based on data from industry sources. Factors such as exploitability, detectability, and real-world impact all inform their inclusion.

In this section, we summarize common web vulnerabilities, many of which align with the OWASP Top 10. Rather than offering a formal ranking, we provide context for each issue's practical relevance, potential impact, and the corresponding research efforts aimed at detecting and mitigating them.

SQL Injection (SQLi) vulnerabilities enable attackers to manipulate backend database queries via crafted input, potentially leading to unauthorized data access or full system

compromise. Halfond et al. provide a taxonomy of SQLi attack types and countermeasures, emphasizing the limitations of many existing defenses that only address narrow subsets of attacks. Alwan and Younis and Sarmah et al. further survey classical and modern techniques for detecting and preventing SQLi across web, mobile, and desktop applications.

Several tools and approaches have emerged to address SQLi at scale. `SQLMap` [120] automates the discovery and exploitation of SQLi vulnerabilities using a variety of injection strategies. Boyd and Keromytis propose *SQLrand*, a defense based on instruction-set randomization to make injected queries syntactically invalid. Appelt et al. introduce *µ4SQLi*, a mutation-based input generator capable of producing executable attack payloads that bypass common filtering mechanisms. More recently, Trickel et al. presented *Witcher*, a grey-box, coverage-guided fuzzing framework that outperforms black-box scanners by using fault escalation and coverage feedback to detect SQL and command injection vulnerabilities. Witcher's ability to generate inputs based on observed output improves input-space exploration and has uncovered dozens of previously unknown vulnerabilities across multiple web frameworks.

Cross-Site Scripting (XSS) remains one of the most prevalent and dangerous vulnerabilities in web applications, allowing attackers to inject and execute malicious scripts in users' browsers. According to Rodríguez et al., XSS was responsible for 40% of web attacks as of 2018, affecting up to two-thirds of web applications. Traditional mitigation approaches focus on server-side filtering and sanitization, but as Vogt et al. and Wassermann and Su highlight, these are often error-prone or incomplete, leading to persistent exposure.

Martin and Lam introduced *QED*, which uses goal-directed model checking to generate multi-request XSS attacks without false positives, showing the potential of formal techniques in practical vulnerability detection. Duchene et al. proposed *KameleonFuzz*, a black-box fuzzer leveraging genetic algorithms and taint inference to evolve XSS payloads and reliably detect vulnerabilities. Client-side taint tracking approaches, such as dynamic defense mechanism [123], and Wang et al.'s *TT-XSS* framework, further extend protection by monitoring JavaScript and DOM APIs during runtime. *TT-XSS* rewrites JavaScript features to track taints through browser rendering, enabling detection and automated verification of DOM-based XSS vulnerabilities with higher precision. These

hybrid and dynamic methods collectively improve resilience against the evolving XSS threat landscape.

Remote Code Execution (RCE) is a high-impact vulnerability that enables attackers to execute arbitrary code on a server, often by injecting malicious inputs that are later interpreted as executable commands. RCE is considered a severe form of XSS when attacker-supplied input is stored and executed as server-side script [104]. Zheng and Zhang introduce a path- and context-sensitive interprocedural static analysis framework that models both string and non-string input behaviors to detect RCE vulnerabilities in PHP applications. Their approach handles the coordination of multiple requests and control-flow complexities inherent to RCE scenarios, leading to the discovery of previously unknown vulnerabilities. Biswas et al. present a case study, examining how insecure input handling practices during development contribute to RCE exposure, and highlighting common patterns across vulnerable applications. Hassan et al. propose an exploitation algorithm for identifying RCE vulnerabilities, and evaluate it through manual penetration testing of over 1000 web applications. Their empirical study correlates specific input handling flaws with successful exploitation, providing insights for improving preventive practices.

Logic Bugs are often the result of faulty application behavior rather than simple input validation issues. These vulnerabilities are difficult to identify because they are highly specific to the application's intended functionality. Felmetsger et al. propose a dynamic analysis-based approach to automate the detection of application logic vulnerabilities. Their method infers a set of behavioral specifications from observing the normal operation of the application and then uses model checking to identify deviations that may indicate logic flaws. Similarly, Deepa et al. developed *DetLogic*, a black-box tool that models the intended application behavior using an annotated finite state machine. This model helps derive constraints related to input parameters, access-control mechanisms, and workflows. Violations of these constraints are then simulated to detect vulnerabilities like parameter manipulation, access-control breaches, and workflow bypasses. Deepa and Thilagam also highlight that while injection vulnerabilities such as SQL Injection (SQLi) and Cross-Site Scripting (XSS) have well-established detection mechanisms, logic vulnerabilities have been gaining more attention due to their specific impact on application functionality. They emphasize the need for continued research in fixing these flaws, especially as attackers increasingly exploit them to compromise application behavior.

Authentication and Session Management Flaws arise when web applications fail to correctly implement mechanisms for user identity verification and session control. To address these issues, several works propose techniques for both understanding their root causes and automating their detection or mitigation. Huluka and Popov apply Root Cause Analysis (RCA) to identify systemic causes behind session management and authentication vulnerabilities. Their work isolates 11 causes for session management flaws and 9 for broken authentication, offering a structured approach to designing more robust authentication mechanisms based on these insights. Dalton et al. introduce *Nemesis*, a system that leverages Dynamic Information Flow Tracking (DIFT) to prevent authentication bypass and access control flaws. It tracks the flow of user credentials through an application's runtime and cross-references them with access control policies to enforce proper authorization. Takamatsu et al. propose an automated technique to detect session management vulnerabilities using simulated attacks. Their approach reduces the need for manual setup and expert knowledge by requiring only minimal input from testers. Hassan et al. conduct a large-scale case study on 267 web applications and apply manual double-blind penetration testing to assess the prevalence of broken authentication and session management vulnerabilities, revealing a high incidence of flaws and highlighting the importance of rigorous and systematic evaluation methods during deployment and maintenance.

Server-Side Request Forgery (SSRF) vulnerabilities occur when a server-side application fetches resources from user-supplied URLs without proper validation, enabling attackers to make requests to internal services. Several approaches have been proposed to detect and prevent such flaws. Jabiyev et al. analyzed over 60 SSRF vulnerability reports and proposed a novel defense mechanism by extending a reverse proxy to enforce SSRF-aware URL filtering. Al-talak and Abbass took a machine learning approach, developing an LSTM-based deep learning model capable of detecting SSRF attacks from request patterns, achieving high accuracy in experimental settings. Static analysis techniques have also evolved to address limitations in detecting SSRF in PHP-based applications. Ji et al. identified the shortcomings of existing taint analysis tools, particularly over- and under-tainting due to lack of SSRF-specific modeling. They proposed a refined taint analysis technique that extracts application-specific sources and sinks and builds accurate call graphs to improve detection. Building on this, Ji et al. introduces an LLM-assisted inter-procedural, path-sensitive taint analysis engine that

incorporates SSRF semantics and PHP-specific features to prune false positives and increase precision.

In summary, prior work on web application security has explored both targeted and general-purpose techniques for vulnerability detection and prevention. While some research has focused on specific classes of vulnerabilities, others aim for broader coverage. Tools like *webFuzz* [115], *Enemy of the State* [139], and *Burp Suite* [116] apply fuzzing and state-aware crawling techniques to uncover a wide range of vulnerabilities. Empirical studies, such as the one by Fonseca and Vieira, have analyzed the underlying software bugs that commonly lead to security issues, contributing to more realistic threat models and testing methodologies. Widely-used platforms like *OWASP ZAP* [141] further support comprehensive web vulnerability scanning across different attack surfaces. Together, these approaches and tools provide a foundation for securing web applications against both known and emerging threats. While many of these well-known vulnerabilities have been extensively studied, certain categories—such as excessive data exposures (EDE), where sensitive but seemingly innocuous data is unintentionally revealed—have received comparatively little attention in the academic literature, despite their potential for serious security and privacy implications.

2.4.4 Testing Web APIs using OpenAPI Specification

Testing web APIs presents unique challenges compared to traditional software testing. APIs often expose complex input-output behaviors through HTTP interfaces, and their correctness depends not only on the functional responses but also on adherence to protocol standards, appropriate status codes, and correct state transitions [101, 139].

One major focus in API testing has been the generation of test cases from formal specifications, particularly OpenAPI Specifications (OAS). Tools like *RESTler* [84] leverage OAS to automatically generate test sequences that account for dependencies among API endpoints. *RESTler* uses server responses to detect crashes and infers API dependencies to guide test generation. Later work [142] extended it to check for common REST rule violations in addition to server crashes. *Schemathesis* [100] supports property-based testing from OAS and GraphQL, employing fuzzing and hypothesis-based input generation to systematically explore edge cases and malformed inputs. *EvoMaster* [101] takes a white-box approach, using evolutionary algorithms to generate HTTP request

sequences that maximize code coverage and fault detection. Designed with microservice architectures in mind, it benefits from source code access and instrumentation to guide test generation, while also applying smart sampling strategies based on common REST design patterns.

Several techniques use feedback-guided fuzzing or dynamic analysis to enhance test effectiveness. For instance, RESTler was evolved to support grammar-based fuzzing [143], allowing the generation of semantically meaningful inputs that traverse deeper application logic. *Pythia* [144] incorporates coverage-guided feedback and statistical learning to model API usage patterns and generate more targeted tests. Doupé et al. presents a state-aware black-box web vulnerability scanner that infers a web application's internal state machine by observing output differences during navigation. By using the inferred state model to guide input fuzzing, the scanner is able to explore more of the application and discover vulnerabilities missed by traditional black-box tools.

In addition to feedback-driven and dynamic techniques, another line of research has focused on improving the breadth and usability of specification-based testing. Several recent tools build on OpenAPI specifications to support automated black-box testing with varying strategies and levels of effort. *Morest* [145] adopts a model-based approach, constructing a dynamically updated RESTful-service property graph (RPG) to capture interdependencies among endpoints and guide API call sequence generation. Its adaptive strategy enables broader coverage and deeper testing compared to static dependency-based methods. *QuickREST* [102] offers a lightweight property-based testing technique that generates both test inputs and validation oracles directly from OpenAPI documents, helping uncover misalignments between API implementations and their specifications with minimal manual effort. In a similar vein, *RESTest* [146] and *RestTestGen* [103] are general-purpose black-box testing frameworks that leverage API definitions (OpenAPI or Swagger) to generate input values and oracles. RESTest is designed for extensibility and integration into continuous testing pipelines, while RestTestGen emphasizes testing both nominal and error-handling behaviors using dual oracles to detect faults across a wide range of real-world APIs.

Together, these tools demonstrate the growing sophistication of specification-driven API testing. However, their effectiveness often hinges on the availability and accuracy of OpenAPI documents—an assumption that does not always hold in practice. To address

scenarios where specifications are missing, outdated, or misaligned with the actual implementation, recent research has explored complementary directions that aim to infer specifications or derive test artifacts dynamically. Yandrapally et al. proposed *API-CARV*, a technique that "carves" API tests and OpenAPI specifications from UI tests. Their dynamic analysis infers endpoints and parameters by navigating the UI, allowing them to increase test coverage when integrated with existing tools. Decrop et al. introduced *RESTSpecIT*, an LLM-driven black-box testing framework that generates and mutates HTTP requests to infer REST API specifications with minimal user input. It was shown to discover undocumented endpoints, generate usable specifications, and detect server faults.

Kim et al. provided a comprehensive evaluation of ten state-of-the-art API testing tools on 20 open-source RESTful services, highlighting the trade-offs between techniques and the need for unified benchmarks and evaluation metrics. Meanwhile, Martin-Lopez et al. studied the synergy between black-box and white-box approaches, showing that hybrid strategies can outperform either technique alone in both coverage and fault detection. Extending this line of work, Martin-Lopez et al. demonstrated the viability of *online testing*—continuous testing in production—using *RESTest* [150] on industrial APIs. Their system generated over a million test cases and uncovered hundreds of real-world faults, showing the practical impact and scalability challenges of automated API testing.

While these methods have made substantial progress, challenges remain, particularly around the oracle problem, stateful API testing, and scalability across large distributed systems. Ongoing research continues to refine API testing by integrating static analysis, symbolic execution, and leveraging human-generated behaviors or feedback.

Chapter 3

EDEFuzz: Detecting Excessive Data Exposures in Web Server Responses with Metamorphic Fuzzing

Contribution

This chapter presents EDEFuzz, an approach and tool for detecting excessive data exposure (EDE) vulnerabilities in web APIs. Aligned with the broader goal of this thesis—to develop practical automated testing techniques that reduce reliance on formal specifications and extensive manual engineering—EDEFuzz demonstrates how relational properties between web responses and UI behaviors can serve as implicit oracles for uncovering subtle, context-dependent leaks. Although it requires users to document basic interaction steps to target specific API calls, EDEFuzz leverages these naturally occurring workflows to drive test generation and validation. In doing so, it offers a low-overhead, developer-friendly approach to identifying data exposures that are often missed by conventional scanners.

This work was published at ICSE 2024 and received a Distinguished Paper Award.

3.1 Overview

Every week, another leak! Server-side APIs of web applications frequently transmit more data than is needed for their corresponding clients. This may not have been an issue, were it not for the fact that these APIs are often publicly accessible. API vulnerabilities of this type are known as *Excessive Data Exposures* (EDEs). Despite ranking as OWASP's #3 most significant API vulnerability for 2019 [113], technology to detect these vulnerabilities remains underdeveloped.

We thus develop the first automated and systematic fuzzing tool, EDEFuzz, to detect EDEs. As the "gold standard for finding and removing costly, exploitable security flaws", fuzzing is a key tool for cost-effectively detecting and remediating such issues [151].

> *"Automatic tools usually can't detect this type of vulnerability because it's hard to differentiate between legitimate data returned from the API, and sensitive data that should not be returned without a deep understanding of the application."*
>
> — The Open Web Application Security Project (OWASP)

> *"This vulnerability is so prevalent (place 3 in the top 10) because it's easy to miss. Automation is near useless here because robots can not tell what data should not be served to the user without telling them exactly how the application should work. This is bad because API's are often implemented in a generic way, returning all data and expecting the front-end to filter it out."*
>
> — Wallarm End-to-End API Security Solution

FIGURE 3.1: **Industry views on the EDEs.** These indicate the prevalence of EDEs and limitations of existing detection tools

We posit that the lack of automated tools to detect EDEs is due to their *semantic* nature. Specifically, EDEs do not manifest through explicit, abnormal behaviours (e.g., program crashes). Detecting them thus requires a model of what constitutes an EDE.

We start with a definition: an API is vulnerable to EDE if it exposes meaningfully more data than what the client legitimately needs [113].

Consider a simple example of an online storefront. When a user views the page for a specific product, an API call may be made to fetch stock levels, informing the user whether the item is in stock. However, the API may also return extraneous data (such as the profit margin on the item) that is not displayed to the user but is nonetheless transmitted. The transmission of the extra data constitutes an "excessive data exposure". This leads to our motivating question:

How to detect if a web API exposes unnecessary data?

The question is related to the famous test oracle problem. How can a tester or an automated testing technique distinguish desired, correct behaviour from undesired or incorrect behaviour [55]. The common wisdom in industry (see Figure 3.1) is that the test oracle problem renders EDE detection beyond current testing approaches.

We address this challenge with the following key insight:

Data returned from an API endpoint is more likely excessive if it has no impact on the content displayed to a user.

Specifically, we develop the following novel metamorphic relation[1] to side-step this problem. Through the relation, automated testing approaches can check if a data field in an API response is excessive by checking for difference between what a client displays when the field is present in an API response, versus when the field is deleted.

Formally, assume we have an API response under analysis $\mathcal{R}_{\mathsf{origin}}$ comprising a set of data fields. A web client (e.g., a web browser) uses $\mathcal{R}_{\mathsf{origin}}$ to render a page that can be represented by a Document Object Model (DOM) tree $\mathcal{D}_{\mathsf{origin}}$. A data field $d \in \mathcal{R}_{\mathsf{origin}}$ is considered non-excessive if the following inequality holds:

$$\mathsf{diff}_{\mathtt{DOM}}(\mathcal{D}_{\mathsf{origin}}, \mathcal{D}_{\mathsf{mutated}}) \neq 0, \tag{3.1}$$

where $\mathsf{diff}_{\mathtt{DOM}}$ calculates the difference between two DOM trees $\mathcal{D}_{\mathsf{origin}}$ and $\mathcal{D}_{\mathsf{mutated}}$. $\mathcal{D}_{\mathsf{mutated}}$ is constructed from $\mathcal{R}_{\mathsf{mutated}}$ which we obtain by removing the in-question data field d from $\mathcal{R}_{\mathsf{origin}}$. If a data field violates Equation (3.1), it is deemed excessive.

From this relation we build a system that significantly reduces the potential for false negatives that hinder competing approaches—manual inspection and keyword-matching. Notably, keyword matching techniques often use a list of terms ("key", "token", "password" etc) in order to flag exposures [152] and therefore any excessive data field that does not match any known keywords is erroneously ignored.

In contrast to these approaches, our tool EDEFuzz leverages the metamorphic relation to detect EDEs. It mutates and replays API responses into the client side of a web

[1] A metamorphic relation is one that holds between two different program inputs and their corresponding outputs [53]

application and compares the generated DOM tree with the original tree in each fuzzing iteration.

Building the tool required surmounting two major challenges.

First, we needed to build an API fuzzer with *response determinism*. Existing mutation algorithms used in Web API testing/fuzzing [84] focus on mutating API *requests* which introduces random and untargeted changes in server *responses*. However, our metamorphic relation requires that the responses differ only in a single field.

Second, like other fuzzing tools, the usefulness of our tool depends on its ability to achieve reasonable throughput (represented in tests per second). This challenge is particularly acute in the context of web fuzzing as tools are rate limited by both bandwidth, and server load. For public sites, the challenge is further compounded by both server-side rate-limiting and the need to minimize disruption. These hinder the timely progress of a fuzzing tool.

To address these two challenges, we adopt a "record-replay" model [153]. We combine a web proxy and a custom-built simulated server to minimize interactions with sites under-test. Prior to beginning the fuzzing process, our tool initiates a "record" phase: a web proxy captures all client requests and server responses, including the request sent to the targeted API and the corresponding response. Note that in each fuzzing campaign EDEFuzz targets only one API. Following the record phase, fuzzing begins (i.e., the "replay" phase).

In the replay phase, *no communication with the actual remote server is necessary*. Our lightweight simulated server handles all requests. If a request is sent to the targeted API, the simulated server transmits a mutated version of the original server response. Otherwise, the simulated server merely replays the recorded transmissions.

This architecture yields several benefits. First, test executions (i.e., sending requests and getting responses) are performed locally—leading to much lower latency. Second, changes to the remote server do not impact test results, making them highly deterministic. Maintaining deterministic results is a critical requirement for fuzzing in general because it helps reduce false positives. However, when detecting EDEs, this also helps reduce false negatives. Absent this determinism, an application change that yields a different web page may cause EDEFuzz to incorrectly flag a field as non-excessive–believe

the DOM change to be caused by changes in the server-response and not in the application itself. Third, the architecture permits running tests in parallel, which minimizes the burden of scaling the tool.

We evaluate the tool in two different settings. First motivated by a recent massive Web API leak in Australia [154], we applied EDEFuzz to several comparable Australian web properties. We perform a detailed comparison of the tool's results against a corresponding manual effort to assess the severity and accuracy of the findings. Second, we run our tool against a broader set of sixty-nine web applications—the complete set of applicable targets from the Alexa Top-200. We use this evaluation to assess the scalability of our tool as well as its applicability to a representative set of global web applications.

Our overall contributions are as follows:

- We identify a novel metamorphic relation to address the test oracle problem in the context of detecting excessive data exposure.
- We develop the first systematic and automated fuzzing tool for detecting excessive data exposure vulnerabilities, EDEFuzz.
- We empirically evaluate the accuracy of our approach, its applicability to popular websites, and its efficiency (both in terms of computational time and human effort). Our results demonstrate EDEFuzz's effectiveness for discovering unknown sensitive data leakage via EDE also, whose prevalence we also investigate. We found that our approach is
  - highly accurate: 98.65% of the fields flagged by the tool in a controlled study were true excessive data exposures.
  - widely applicable to popular websites, requiring modest computational costs and human effort to employ.
  - able to discover zero-day EDE vulnerabilities. Specifically, it found five zero-day EDE vulnerabilities serious enough to merit immediate disclosure.

To support future research in this interesting topic, EDEFuzz is made open-sourced. The source code, along with some tutorials will be available at `https://github.com/Broken-Assumptions/EDEFuzz`.

The reminder of this chapter is structured as follow. In **Section 3.2**, we provide the necessary preliminaries on Web APIs, Excessive Data Exposures and Metamorphic Fuzzing. In **Section 3.3**, we motivate our work with several real-world vulnerabilities *discovered by our tool*. In **Section 3.4**, we present our automated approach to detect EDEs and our implementation. In **Section 3.5**, we report our experimental results. We consider related work in **Section 3.6** followed by a brief discussion in **Section 3.7**.

3.1.1 Research Ethics

We considered both the propriety of our scanning and fuzzing techniques and engaged in vulnerability disclosure.

We discussed our research in a series of conversations with our research ethics office who ultimately deemed it exempt from a full review process. Our research involves scraping and scanning commensurate with ordinary activity by both search engines and the research community. Our methodology minimizes interaction with remote servers by performing all fuzzing offline using a simulated replica of the target server. Given the low impact of capturing the outcome of a limited number of HTTP requests and the potential benefits of our research it was determined that our work adheres to the principle of beneficence that is the hallmark of research ethics.

As our work notes, EDEFuzz flags fields for further human analysis (rather than indicating vulnerabilities with certainty). As a result, our assessment of whether a flagged field rises to a reportable level requires human judgement about whether an EDE leaks sensitive information and the potential harms from that leak. In the five instances where we discovered sensitive data leakage we contacted the affected entities. By time of submission, all five entities had acknowledged our disclosures; two had remediated the issues.

3.2 Background

3.2.1 Excessive Data Exposure

Web applications often expose API endpoints to the public internet. Exposing the endpoint allows the application to separate front- and back-end logic. While the front-end

components focus on rendering visual elements and their associated interactive components, back-end logic is more closely tied to long-term data storage. The API allows the font-end to query the back-end and in many cases serves a response in either JSON or XML. While an API ought to narrowly tailor the data in its response to match the request, this practice is often ignored. OWASP terms this excessive data exposure (EDE) [11].

One cause for EDEs is that API developers over-rely on API clients to perform data filtering. This eases the cognitive burden on back-end developers to determine the specific needs of the client *a priori*.

When present, EDEs are often trivial to exploit. To obtain the excess (or even sensitive) data, it is often sufficient to simply examine response traffic from the target API.

Since technology to scan for and detect EDEs remains underdeveloped, OWASP only provides general advice [11] on how to prevent them, such as "Never rely on the client to filter data!", "Review all API responses and adapt them to match what the API consumers really need", and "Enforce response checks to prevent accidental leaks of data or exceptions". However, the prevalence of EDEs shows that advice alone is insufficient. We need effective automation!

3.2.2 Fuzzing

Fuzzing is a process of repeatedly generating (random) inputs and feeding them to a system under test (SUT) to discover bugs. In its traditional use, a fuzzer detects issues through aberrant program behaviour, such as program crashes. This indicates a potential security bug in the SUT. In response, the fuzzer will preserve the bug-triggering input for further analyses (e.g., manual debugging). A more detailed discussion about fuzzing is given in Section 2.3.3.

While we preserve the input generation phase above, our work notably deviates in that we detect potential errors through *the lack of change* in program output, rather than a spec violation or crash.

However, our work is not the first to address web application testing. Web application fuzzing recently garnered increased interest from both industry and academia, as discussed in Section 2.4.

A common classification of fuzzers is based on the fuzzer's awareness of the internal structure of the SUT [155], as discussed in Section 2.3.3. Our tool EDEFuzz is a black-box fuzzer.

3.2.3 Metamorphic Testing/Fuzzing

We adopt fuzzing as our approach to detecting EDEs because of its demonstrated success in discovering security flaws.

Highlighting the effectiveness of fuzzing, as of May 2022, Google's fuzzing infrastructure had detected over 25,000 bugs [156]. However, these bugs were detected using explicit test oracles. Bugs detected by test oracles either cause program crashes (e.g., segmentation faults) or are caught by instrumentation-based checkers (e.g., Address Sanitizer, etc). In contrast, semantic bugs like EDEs do not manifest through explicit abnormal behaviours, and cannot be reliably detected by observing *single* program executions.

How do we build tools to detect semantic bugs? Metamorphic testing and fuzzing, which leverage metamorphic relations, are a promising approach. At its core, metamorphic fuzzing involves *comparing* multiple executions of the SUT under different inputs and observing whether some relation (called the *metamorphic relation*) holds between their corresponding outputs, as discussed in Section 2.3.4.

Metamorphic relations are properties that must necessarily hold with respect to the correct functioning of the SUT. In metamorphic testing, the violation of a metamorphic relation indicates a potential bug [53]. Past work successfully identified and applied metamorphic relations to discover bugs across various systems, as discussed in Section 2.3.4.

3.3 Motivating Examples

In this section, we present a selection of real-world EDEs detected by our tool to demonstrate their prevalence and implications, the challenges in detecting them using prior approaches, and the motivation for our tool (which we comprehensively evaluate in Section 3.5).

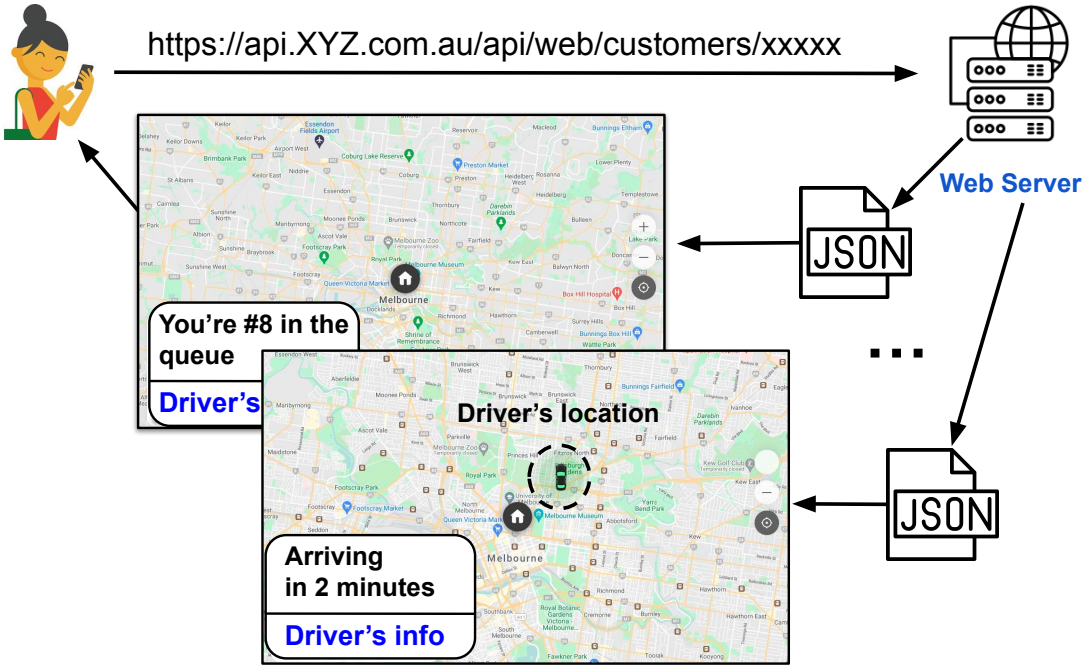

FIGURE 3.2: API flow for a package delivery service. A web application requests tracking information, returned in a JSON object.

Vulnerability 1 - Locations and Contact Details EDEFUZZ discovered a vulnerability while testing the live delivery tracking service offered by an Australian last-mile delivery service. The vulnerability has been reported and fixed. As shown in **Figure 3.2**, a customer receives a unique link on the day that an item is on board for delivery. The link opens a web page which displays the name and a photo of the delivery driver, an Estimated Time of Arrival of the delivery, and the position of the item in the driver's queue. The page sends an API request to the server regularly, and updates the contents on the page based on the API response. Part of this response is shown in **Listing 3.1**, with sensitive information removed.

The client-side logic allows the webapp to display the accurate geographic location of the delivery driver *only* when the item to be delivered is at the front of the queue. It suggests that while the driver is delivering an item to a customer, another customer should not be able to ascertain the location of the driver—which would leak the location of other deliveries. However, EDEFUZZ detected that the API response always contains rich information about the delivery driver, including accurate latitude and longitude (the `location` field), direction of facing (the `bearing` field), and speed of traveling (the `speed` field). EDEFUZZ also identified the driver's manager's information in the API response.

Knowing the timestamped location of the delivery driver, a customer may recover the route that the delivery driver is traveling, or even be able to identify the address of other customers who receive parcel from the same delivery driver. This information is ideal for enabling further attacks, such as social engineering [157].

LISTING 3.1: **An API response to a query for delivery status.** The authors have redacted or adjusted sensitive information.


```
{
   "driver":{
      "id":353,
      "url":"/api/web/drivers/353",
      "full_name":"[NAME]",
      "car":{
         "car_type":1,
         "car_type_name":"Car",
         "capacity":33
      },
      "member_id":37270063,
      "avatar":"[URL_TO_AVATAR]",
      "is_online":true,
      "work_status":"working",
      "phone":"[PHONE_NUMBER]",
      "status":"in_progress",
      "location":{
         "id":1081725,
         "timestamp":1632107809.0,
         "speed":7.60149761928202,
         "accuracy":4.88680554160758,
         "location":"-37.79998905537697,144.9940922248341",
         "created_at":"2021-09-20T13:16:49.365492+10:00",
         "bearing":277.508087158203,
         ...
      },
      ...
   },
   "manager":{
      "id":305,
      "full_name":"[NAME]",
      "email":"[EMAIL]",
      "avatar":null,
      "is_online":false,
```

```
35          "work_status":"not_working",
36          "phone":"[PHONE_NUMBER]",
37          "can_make_payment":false,
38          "merchant_position":"Manager",
39          "role":"Manager",
40          ...
41      },
42      ...
43  }
```

Vulnerability 2 - Stock Levels This second vulnerability exposed detailed stock availability for a retailer, which could be considered sensitive information. Many retailers allow potential customers to check their stock availability online before visiting their shops. Some retailers reveal precise stock values on their websites, while other retailers decided to only display categorised values such as "in stock", "low stock" and "out of stock". Interestingly, we noticed a few such online services in which the server transmits the precise stock values but the web application only displays categorised values based on thresholds.

We observed this design in retailer companies such as Company-C, Company-D and Company-E. Company-D claimed that their accurate stock level is non-sensitive, whereas Company-E removed their stock level from their API response before we tried to contact them.

Regarding attack scenarios, if a retailer knows about the stock details of their competitors in different locations, they could adjust their logistic plan accordingly to increase their sales and gain more profit. Individual suppliers might also take advantage of this vulnerability to increase their prices for specific retailers when they know that they are low on stock. Having access to that kind of information, a 3rd-party company (e.g., a shopping suggestion service) could also develop an app and give more precise suggestions to their customer, leading to some monetary benefit.

Vulnerability 3 - Network bandwidth Transmitting excessive data fields reduces performance, thereby negatively impacting the user experience. Further, it imposes increased bandwidth requirements creating accessibility issues. EDEs imply a lack of consensus between an application's back- and front-ends. Like other "code smells" EDEs

are indicative of poor development practices that may cause other vulnerabilities. We found a case in Company-C's API where the front-end consumed only 4.4% of the response data fields.

The Bugs Escaped Detection! There are a variety of ways in which these flaws may have escaped notice. Developers, while aware of security flaws, are heavily reliant on tooling to detect bugs. As discussed in **Section 3.2** keyword-matching tools [152] to discover EDEs are of low effectiveness and prone to false negatives. To confirm this hypothesis we applied the popular vulnerability scanner Burp Suite [152] to attempt to identify EDE and sensitive data leakage to sites from our test set. As expected it failed to identity Vulnerability-2 in Company-D (a false negative); it also failed to identify almost all of the EDE and sensitive data leakage in Company-I (identifying only the disclosed email addresses in Listing 3.1). It also reported false positives (mistaking public postcode data for credit card numbers).

A tool-free approach requires back-end developers to carefully ascertain if a given field is needed, which is not always feasible.

As discussed in the case of Vulnerability-2, while Company-D claimed that their accurate stock level is non-sensitive, Company-E decided to remove that information from their API responses. This further indicates the need for a principles based approach that reduces the burden on developers to determine the balance between business need and sensitivity.

3.4 EDEFuzz Workflow

We now provide an overview of EDEFuzz and detail the design of its components. We depict the main workflow of EDEFuzz when testing a website (given its URL and a targeted API endpoint) in **Figure 3.3**.

3.4.1 Overview

Identifying API Endpoints. Identifying target API endpoints is an orthogonal problem to testing them for the presence of EDEs, and is thus out of scope for this work. One may leverage tools like crawlers to automatically explore the website and detect APIs

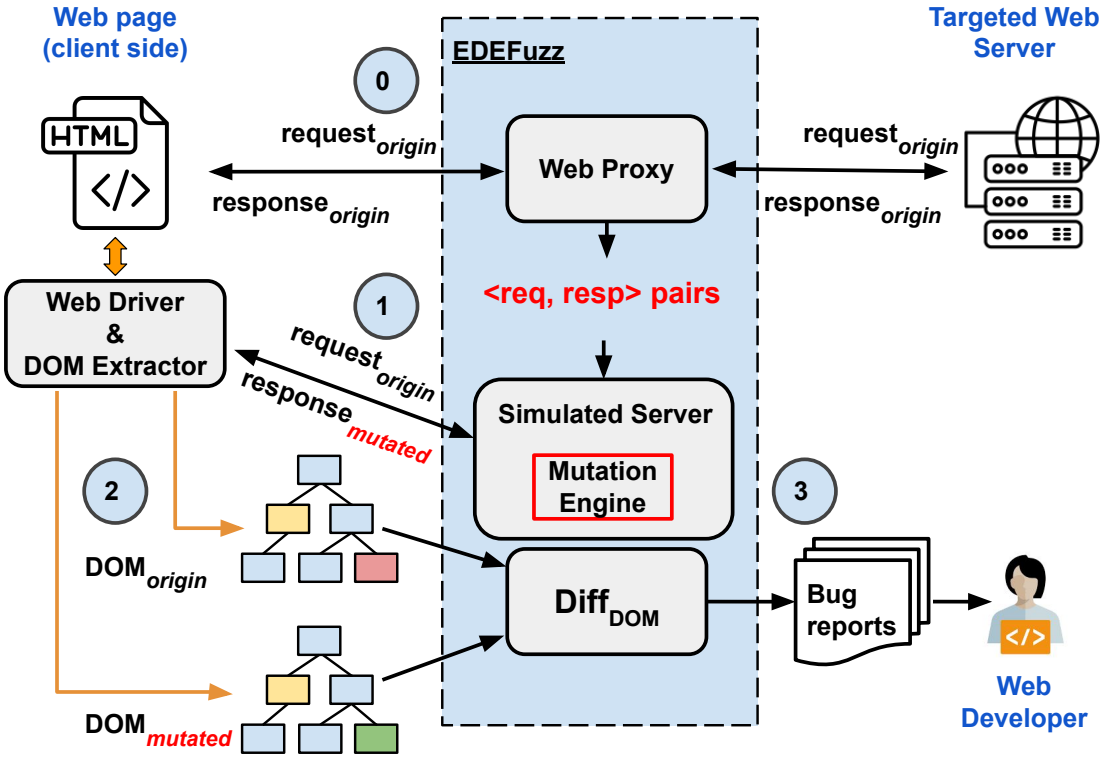

FIGURE 3.3: The workflow of our approach to identify excessive data exposure vulnerabilities in web APIs.

for targeting. However, this automated approach could be destructive and potentially illegal without permission from the website owners. In our experiments (as detailed in **Section 3.5**) we identify target APIs by examining browser behavior during manual website interaction.

Having identified a target API, EDEFUZZ's workflow consists of four main steps which are in turn divided into two phases: recording/preparation phase (Step 0) and relaying/-fuzzing phase (Steps 1-3), in line with its record/replay design discussed in **Section 3.1**.

Recording/Preparation Phase. In this semi-automated phase, the goal is to generate a configuration file denoted as C, which sets the client under test to a baseline state. This file will be used in the subsequent replaying/fuzzing phase. To that end, we use a Web Proxy to capture the traffic between a client app, which is a web browser in our experiments, and the targeted web server. Specifically, we start the client app and capture its initial state S_0. After that, we let the client open the given URL, wait for the web page to be fully loaded. The tester then interacts with the page (e.g., fill in text boxes, click buttons) to trigger a request to the target API. We denote S_1 as the client state at which the request has just been completely sent. We develop a lightweight

browser plugin to capture the interaction steps required to traverse from state S_0 to S_1 and save them into the configuration file so that they can be played back in subsequent steps.

The standard most common for responses to web API requests is JSON. Under AJAX or similar paradigms, when the API response is received a client uses the JSON response to update the web page (e.g., showing more information) and the update typically alters only some parts of the web page, rather than changing the entire page. We denote S_2 as the client state immediately following the update. This state can be typically identified by the existence of certain page elements. The steps required to identify the transition from S_1 to S_2 (typically of the form "wait until page element X appears") are also recorded in the configuration file, meaning it now stores all steps required to traverse from state S_0 to S_2. At S_2, a baseline DOM tree (denoted as $\mathcal{D}_{\mathsf{origin}}$) of the web page is extracted using the DOM Extractor component. This DOM tree will be compared with other trees to be generated in the fuzzing phase to check for potential excessive data exposures based on the metamorphic relation defined in **Equation (3.1)**.

Before the targeted request-response pair has been exchanged, the browser and the server might have completed other exchanges for fetching HTML documents and other resources such as images, stylesheets and Javascript files. Thus all request-response pairs and resources, denoted P are recorded and stored for the replaying/fuzzing phase.

Replaying/Fuzzing Phase. The input for this phase includes: 1) the configuration file C, 2) the original DOM tree $\mathcal{D}_{\mathsf{origin}}$, and 3) all request-response pairs P recorded in the recording phase.

LISTING 3.2: **A configuration file generated in recording phase.**

```
1 TARGET /api/v2/stock/get
2
3 LOAD https://www.example.com/path/page
4 INPUT //input[@id="text-postcode"] 3000
5 CLICK //span[text()="Check availability"]
6 WAIT_LOCATE //div[@id="stock-info"]/div[2]
7 FUZZ
```

Listing 3.2 shows a sample configuration file. The first line specifies the target API under test. The rest define a sequence of user interactions to be performed to reach state

S_2. The example loads a web page, enters the number `3000` into a text box, clicks a button and then waits for a specific element to appear on the web page before capturing the state. Apart from these actions, EDEFuzz also supports HOVER, SLEEP, and SCROLL for (i) hovering the mouse on a specific element, (ii) waiting for a specified period of time, and (iii) scrolling up/down, respectively.

In each fuzzing iteration, EDEFuzz goes through three steps (Steps 1-3). In the first step (Step-1), the Web Driver component, which is built on top of the Selenium Web Driver [158], uses the configuration file C to replay all the steps until the client reaches the state S_1. Before that state has been reached, the Simulated Server responds to client requests with the corresponding recorded responses stored in P with no modification. Once state S_1 is reached and the Simulated Server receives the request sent to the targeted API, it mutates the originally recorded response by deleting a specific data field and transmits to the client. We describe the mutation algorithm in detail in **Section 3.4.3**.

After the baseline state is reached, in Step-2 the client uses the mutated response to update the page accordingly. If this leads to any error, EDEFuzz moves to the next fuzzing iteration. Otherwise, EDEFuzz waits until the page is fully updated (i.e. until state S_2 is reached) and uses the DOM Extractor to extract the current DOM tree denoted as $\mathcal{D}_{\mathsf{mutated}}$. In Step-3, EDEFuzz compares $\mathcal{D}_{\mathsf{mutated}}$ and $\mathcal{D}_{\mathsf{origin}}$ using a comparison algorithm described in **Section 3.4.4**. If the two DOMs are the same (based on our definition of similarity) then we flag the deleted data field. According to the metamorphic relation, the field is excessive. Once all fuzzing iterations are completed, EDEFuzz reports all the potential excessive data fields to the tester for further analysis and confirmation (see **Section 3.4.5**).

Randomness: The explanation so far assumes that web replaying is fully deterministic: given a configuration file, EDEFuzz—without applying any mutations on the server response—produces the same DOM tree across all runs. Though uncommon, we identified a few cases in which this assumption does not hold. For instance, a social media platform may randomly insert advertisement between user posts during runtime. This causes the web page to be visually different in each run, even if all contents transmitted from the server were identical. We discuss the issues in details and how we address them in **Section 3.4.2** and **Section 3.4.4**.

Implementation Details. We implemented EDEFuzz in Python3 using Selenium as a Web Driver to control web browser. We successfully tested our tool on widely used platforms including Ubuntu 18.04, Ubuntu 20.04 and Windows 10. Currently, EDEFuzz supports two web browsers: Google Chrome and Mozilla Firefox. Our design is modular to support future extensions.

3.4.2 Simulated Server

Our design, which employs a Simulated Server, brings several benefits to EDE testing. The server supplies locally recorded contents with minimal delay—reducing test latency. Secondly, EDEFuzz does not need to communicate with the targeted remote server during the fuzzing phase, allowing testing in parallel without affecting the targeted server. This allows developers to easily test their servers without impacting production services. This design also ensures consistency within a given run, ensuring test results are not affected by potential changes to the state of the remote server during the testing process.

Since the simulated server provides a snapshot of states on the remote server, there is no need to handle cookies or sessions within EDEFuzz. The simulated server matches the requested URL from the collection of recorded request-response pairs and simply sends back the recorded response. This also supports sites the user needs to first log-in: as long as the remote server sent responses representing the session of a user who was logged-in during the recording phase, our simulated server can reproduce it in the fuzzing phase.

Websites that fetch resources by randomised URLs instead require a fuzzy matching algorithm in the simulated server. A typical case is when requesting a resource from the server, the web page generates a random token to be included in the request URL. This case causes a request to be generated for which the simulated server has no recorded response. Our experimental evaluation in Section 3.5.2 shows that this applies to only a fraction of popular websites (8.7% of our evaluated target set). We believe it can be handled by applying fuzzy matching for request URLs, but leave this extension for future work.

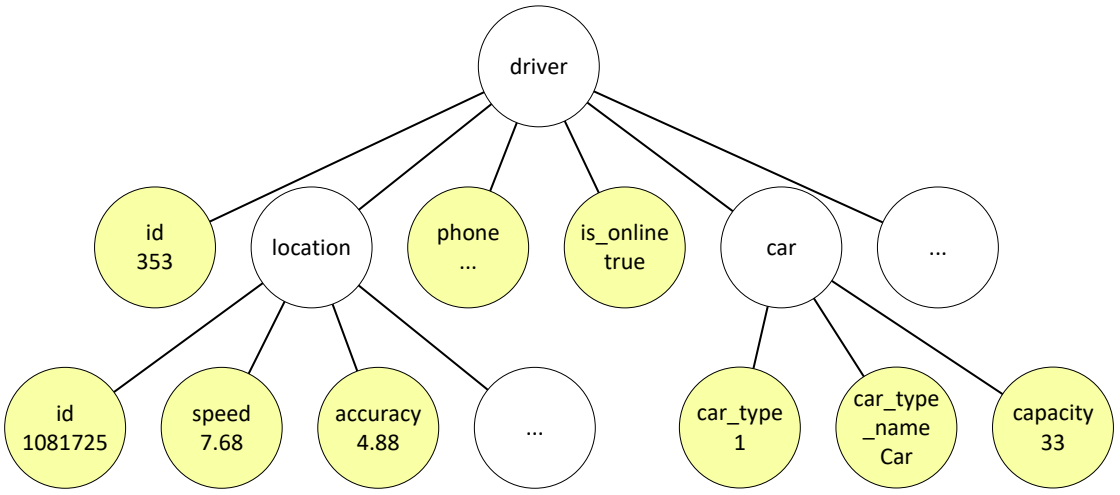

FIGURE 3.4: **Tree representation of the JSON object in Listing 3.1.** The mutation engine uses this represenation to determine what to mutate. We highlight leaf nodes in yellow.

3.4.3 Mutation Engine

The original API response (JSON) produced by the server is assumed to be valid both structurally and semantically. EDEFUZZ generates test cases by mutating the original API response.

We represent the server-supplied JSON object using a tree. Each leaf node is potentially an excessive data field. For example, **Figure 3.4** shows the (partial) tree representation of the JSON object from COMPANY-I shown in **Listing 3.1**. In this tree, all leaf nodes are shaded.

We generate each mutation (test case) by removing a leaf node from the tree. For example, a valid mutation of the JSON object from COMPANY-I shown in **Listing 3.1** could remove the key-value pair `id: 353` from the `driver` dictionary, or `capacity: 33` from the `car` dictionary.

Unlike other fuzzing approaches which may generate an infinite number of test cases (e.g by using genetic mutation operators such as bit flips and splicing [62]), our approach produces a fixed number of test cases corresponding to the leaf nodes on the tree. Standard techniques to further reduce that number using approaches like binary tree search and delta debugging [159] are not applicable because there is no way to determine whether a subtree (or set of fields) contains at least *one* field that constitutes an EDE (if we delete all of them and notice a change in the DOM, then it might mean that

none are excessive, or all-but-one are excessive, or something in-between). Instead our metamorphic relation can tell us only if *all* fields in some set are subject to EDE (when we remove them all and no change is detected in the DOM). It is worth noting that this design decision also ensure more predictable test run times.

3.4.4 Similarity Check for DOM Trees

A web page (an HTML document) has a hierarchical structure that can be represented using a DOM tree. In **Section 3.4.4** and **Figure 3.5** we show a sample HTML snippet and its DOM tree, respectively. This is a simplified version of one of our testing targets COMPANY-I, which we shared in **Section 3.3**.

```
<div class="container">
  <div class="logo">
    <img src="..." />
  </div>
  <div class="location_icon">
    <div class="icon">
      <i class="icon_car"></i>
    </div>
  </div>
  <div class="driver_info">
    <p>Name: ...</p>
    <p>Phone: ...</p>
  </div>
</div>
```

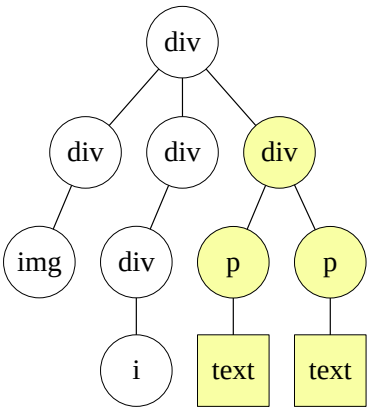

FIGURE 3.5: **A simple HTML snippet and its corresponding tree representation.** Each rounded node represents a tag element while each square node represents a string. The hierarchical structure allows a human to easily specify a subtree (in yellow) for the tool to evaluate.

Recall, a DOM tree has nodes and each tree node represents a tag element in the HTML document. Each tree node has zero or more attributes—which are stored as key-value pairs—and each node may have children, which are either further tree nodes or plain strings.

When comparing two web pages, one considers both the DOM tree structure and the content within each tree node. According to our metamorphic relation, if an API response with a particular data field removed would still result in an identical DOM tree, compared to the DOM tree produced using the original API response, that data field is reported as excessive. Two DOM trees are considered identical if and only if *all* of the following conditions hold:

- **(C1)** Their root nodes have the same tag name
- **(C2)** Their root nodes have the same number of attributes
- **(C3)** Each corresponding pair of attributes have the same value
- **(C4)** Their root nodes have the same number of children
- **(C5)** Each corresponding pair of children representing a tag element is identical, with respect to conditions **C1-4**
- **(C6)** Each corresponding pair of children representing a string is identical

To check for all of these conditions, the most simple yet effective approach is to compare the string representations of the corresponding HTML documents. Our experiments showed that this works for 47 out of 54 targets. However, we observed that several web pages contain elements which are not affected by the API response. For example, the value of the `class` attribute within a `<div>` tag could be randomly generated at run-time. Another common case is when the web page displays the current date and time on it. Apparently, these cases could yield false negatives using the straightforward string-based comparison approach.

To address this issue, we relax the conditions **C3** and **C6**. That is, we accept the differences in string leaf nodes and in attributes' values caused by randomness. To that end, EDEFuzz runs a pre-processing step before the fuzzing phase. In this step, EDEFuzz uses the configuration file C to replay and generate a few DOM trees, all generated from replaying the *same* server response. After that, it recursively traverses and compares $\mathcal{D}_{\mathsf{origin}}$ with each of the newly generated DOM trees to look for those parts that differ due to randomness, and marks those random elements and attributes of $\mathcal{D}_{\mathsf{origin}}$ that should be "ignored" in the comparison step of the fuzzing phase. It is worth noting that, in this pre-processing step, if EDEFuzz finds that the generated DOM trees are *structurally* different from $\mathcal{D}_{\mathsf{origin}}$, i.e. violating conditions **C1**, **C2**, **C4** and **C5**, it will decide to terminate the testing process. In our experiment, 9 of 69 targets were flagged at this step, meaning that they each produced web pages with different structures, even using the same response.

In the fuzzing phase, EDEFuzz compares the DOM tree $\mathcal{D}_{\mathsf{mutated}}$ produced from each test case, with the pre-processed $\mathcal{D}_{\mathsf{origin}}$. Basically, EDEFuzz (i) recursively traverses

through every node in $\mathcal{D}_{\mathsf{origin}}$ and its corresponding node in $\mathcal{D}_{\mathsf{mutated}}$, and (ii) compares each pair of nodes. EDEFuzz will skip all string nodes and attributes on $\mathcal{D}_{\mathsf{origin}}$ that have been marked as "ignored" in the pre-processing step. Two nodes are deemed structurally different if any of the conditions **C1**, **C2**, and **C4** is violated. Moreover, they are considered different in terms of content if there is a discrepancy in the values of the nodes (in the case of non-ignored string nodes) or the non-ignored attributes (for other types of nodes).

While comparing the entire DOM tree can identify if a web page is different from another, in many cases a response will affect only specific areas of a web page. Our approach can optionally utilise human knowledge to allow the user to specify an *area-of-interest* on the web page. The area-of-interest is a subtree in the DOM tree that contains contents (that the user believes are) affected by the API response. The area-of-interest in the Figure 3.5 example could be the subtree rooted at the node `<div class="driver_info">` (highlighted in yellow in the tree representation). This helps increase efficiency, and it also avoids other components on the web page affecting the comparison (e.g. if they are randomly generated).

3.4.5 Result Inspection

The final step of our approach is manually inspecting the results. This involves inspecting each of the flagged data fields to determine what kind of data it exposes (e.g. sensitivity) and, therefore, whether the web application should be modified to avoid this exposure.

We suggest the following approach to determine whether a given field is indeed unnecessary (i.e., a true positive), and how sensitive the data is (the severity of a leak).

False positives. Like other testing/fuzzing approaches, EDEFuzz could yield false positives. For instance, a value included in an API response may only be populated on the web page with further user interactions. This is not uncommon when using an overlay window to display certain information. A tester typically verifies a flag by further interacting with the web-page, discovering any hidden dependencies on the field.

Sensitivity of the excessive data. To determine the sensitivity of an excessive data field, first we can assess whether the data field is human interpretable. This may include a minimal decoding process to parse information represented under a given scheme (eg;

base64, GPS coordinates, Unix timestamp, gzip). Next, we determine whether the user could legitimately acquire the data some other way. If not, we consider it 'sensitive'. We present real examples from our dataset. We note that some non-interpretable may nonetheless be sensitive when parsed appropriately. We merely present a usable heuristic applicable to our evaluation.

3.5 Evaluation and Results

We designed our experimental evaluation to answer the following research questions.

(RQ1) Accuracy. Of the data fields flagged by EDEFuzz, what proportion are true excessive data exposures (i.e. unused by the web page). This evaluates the usefulness of our metamorphic relation.

(RQ2) Applicability. To what proportion of widely used websites can EDEFuzz be applied successfully? This helps to understand limitations of our approach, both inherent and those that arise from EDEFuzz's current implementation.

(RQ3) Efficiency. How much human effort and computational time is required to apply EDEFuzz? This sheds light on the scalability of our approach.

(RQ4) Prevalence of Sensitive Data Leakage. Of those fields flagged by EDEFuzz as excessive, what proportion contain sensitive data? This helps us understand how prevalent sensitive data leakage is amongst excessive data exposure issues.

Note the distinction between **RQ1** and **RQ4**. Specifically, given some set of data fields reported by EDEFuzz as excessive, the former measures the proportion of *true positives* whereas the latter measures the proportion that are *sensitive*. These notions are entirely orthogonal: a flagged data field is a true positive precisely when it is really excessive (i.e. unused by the web page), regardless of whether the data it contains is sensitive or not. Likewise, a field contains sensitive data precisely when that data should not be revealed by the web application, regardless of whether the field is excessive (i.e. is unused by the web page) or not. Flagged fields that are not true positives are *false positives*. False positives can exist, for instance, when a field contained in a response is used to affect the DOM only after subsequent user interaction with the web page. Importantly, whether a data field is a true or false positive is a property of the behaviour of the web page. That

is, distinguishing between true and false positives requires carefully understanding *all* behaviours of the web application, including by reading its code and interacting with it, to determine whether the data field contained in the response is ever used in future by the page. Whether a field is sensitive or not is simply a property of the data that field contains, and can be relatively quickly ascertained by inspecting just the field itself.

This means that accurately evaluating **RQ1** requires a set of websites whose behaviour is (or can be) well-understood by the humans inspecting EDEFuzz's results. Carefully understanding the behaviour of an individual site can be very time consuming. Therefore, for **RQ1** we assembled a set (see Table 3.1) of eight (8) popular websites within a single country (Australia) that were familiar and whose individual behaviours of the web page were therefore able to be well-understood. This purposeful restriction was necessary to ensure that the proportion of true positives could be accurately evaluated.

In contrast, adequately evaluating the remaining research questions requires a dataset comprising widely used websites. Therefore the remaining RQs were evaluated on a data set drawn from the Alexa Top-200 list of websites. While not applicable to **RQ1** (since enumerating all of their possible behaviours to accurately determine true positives is infeasible), this set forms a representative best-of sample, suitable for assessing EDEFuzz's performance in general (**RQ2** and **RQ3**), as well as to understand the prevalence of sensitive data leakage via EDE among common websites (**RQ4**).

3.5.1 Procedure

We follow the 4-step procedure below to fuzz test a given website using EDEFuzz. The procedure is well-aligned with the workflow of the tool, as discussed in Section 3.4.1.

(Step-1) Identifying the target API endpoint. Given a primary domain, we identify API endpoint(s) for inclusion in the testing regime. Each domain accessed a variety of APIs, some internal to the domain (eg., a shopping website accessing the site's stock inventory and pricing endpoint) and some external (eg., an analytics endpoint to retrieve recent visitor counts). Of these, we manually identify the internal API most relevant to the function of the site in question. For instance, in Table 3.1 we list the selected Australian sites and endpoints used in our evaluation.

Target	Rank	Used when
Company-A	37	Load tracking history of parcel
Company-B	–	List members of a school subject
Company-C	121	Check stock availability of item
Company-D	–	Check stock availability of item
Company-E	59	Check stock availability of item
Company-F	166	Get flight prices for next 30 days
Company-G	123	Check stock availability of item
Company-H	2770	List vehicles available for sale

Table 3.1: **Australian websites tested.** We include short descriptions of the purpose of the targeted APIs on which we performed deeper evaluation. Where available we provide Alexa rankings within Australia (extracted 29th April 2022).

(Step-2) Writing a configuration file. We compose a configuration file that specifies how to correctly trigger the selected endpoint—this includes the sequences of user interactions that preclude the execution of a request to the API.

This process is semi-automated via our custom-built web browser plugin. However the plugin still requires the user to manually perform interactions. After the file is generated, the operator reviews the file to make any necessary changes.

We specified an area-of-interest (see Section 3.4.4) on all our evaluation targets, at the cost of a few seconds per target.

(Step-3) Running EDEFuzz. After devising an appropriate configuration, we then execute EDEFuzz, time its execution, and collate the results for analysis. We chose not to repeat the fuzzing process for each target because, unlike in traditional fuzzing, EDEFuzz's mutation process is deterministic by design.

(Step-4) Analysing results. This involves the manual classification step, discussed in Section 3.4.5.

We adhere to a systematic classification model to decide if a reported EDE is sensitive or not. We first assess whether the data field is readily human interpretable (including simple encoding schemes such as base64, timestamps). We mark non-human interpretable values non-sensitive. Next, we determine whether the user could legitimately acquire the data otherwise. If yes, mark the field non-sensitive; if not, mark it sensitive.

Target	Data fields	Reported	Confirmed	True Positive	Preparation (min)	Execution (min)	Classification (min)	Sensitive	Non-sensitive
Company-A	189	124	124	100.00%	10	11	5	0	124
Company-B	18	16	14	87.50%	20	2	2	2	12
Company-C	2600	2580	2504	97.05%	5	306	3	104	2400
Company-D	545	506	479	94.66%	15	43	10	9	470
Company-E	4249	4147	4127	99.52%	10	755	15	0	4127
Company-F	778	749	749	100.00%	15	103	5	0	749
Company-G	120	100	100	100.00%	5	12	3	0	100
Company-H	1465	1066	1066	100.00%	15	79	20	19	1047

Table 3.2: **Summary statistics from the Australian sites. Data fields** reports the total number of fields contained in the API response of each target, **Reported** is the number of fields flagged by EDEFuzz as excessive; **Confirmed** is the number of fields manually confirmed to be excessive, i.e. true positives, **TP**. The time taken to configure EDEFuzz for each target is reported in **Preparation**, as is the duration of test execution (**Duration**) and the human effort required to manually classify the flagged fields as sensitive or not (**Classification**), all measured in minutes. We also report (**Sensitive**) the number of fields we classified as containing sensitive data, after manual inspection.

3.5.2 Results

RQ1. Accuracy.

We evaluated the accuracy of our metamorphic relation, as implemented in EDEFuzz, for identifying excessive data exposures against the eight sites listed in Table 3.1. The TP column of Table 3.2 summarises the results by recording the true positive rate, namely the proportion of reported excessive fields (Reported) that were actually excessive (Confirmed). This was determined by manually inspecting the web pages and carefully understanding their behaviours, including via manual interaction, to check whether, for each reported data field, they had any behaviours that made use of the data field. If no such behaviours were identified, the field was classified as a true positive; otherwise it constitutes a false positive.

The overall true positive rate was 98.65%, confirming the exceptionally high degree of accuracy of our approach. (Later, **RQ4** investigates which of these EDEs actually leaked sensitive data which, as argued in Section 3.5, is a separate concern.)

EDEFuzz, like other testing/fuzzing approaches, may yield false positives. For instance, a value included in an API response may only be populated on the web page after further user interaction. This pattern arises for instance when using an overlay window to display certain information. We manually investigated each false positive (1.35% overall) and found that each was an instance of this pattern.

RQ2. Applicability.

We evaluated the applicability of EDEFuzz on a subset of the top 200 sites (as recorded by the Alexa ranking). Of the 200 sites, we excluded (see Figure 3.6) from analysis those that had no web APIs (and, hence, no possibility for EDE); those requiring payment; those in a language that none of our authors understand; those comprising adult or illegal content; those that were geoblocked; and those that required solving a CAPTCHA. Doing so excluded around 60% of the 200 sites, after deduplication. None of these exclusions represent limitations of our approach or implementation. Of the remaining sites, we additionally excluded 12 sites that used `HTTP_POST` requests to query their APIs with query parameters included in the request body, since EDEFuzz's simulated server currently relies solely on the request URL to supply the response, though this implementation-level limitation could be resolved with modest future work. This left 69 sites for our evaluation of EDEFuzz.

Of the 69 evaluated sites, EDEFuzz was successfully applied to 53 targets (76.8%). Of the 16 unsuccessful targets, all but one failed due to non-deterministic behavior in the web application. Specifically, nine sites (13.0%) failed the pre-processing step (Section 3.4.4) because key elements of the page were populated non-deterministically, resulting in inconsistent DOM states across repeated loads. An additional six sites (8.7%) issued requests containing session-specific or random tokens required to load resources, which EDEFuzz could not reliably reproduce. These sources of non-determinism introduce significant false negatives, as mismatches between test and reference states may not reflect genuine differences in data exposure. Consequently, our approach is not applicable to web applications that rely heavily on dynamic content or randomised behaviors unless additional measures are taken to stabilise execution. While experiments were conducted in a controlled, isolated environment to minimize variability, complete elimination of such non-determinism remains a challenge. The final failure case involved

a site that used shadow DOM for critical content, which EDEFuzz does not currently support, preventing full DOM extraction and analysis.

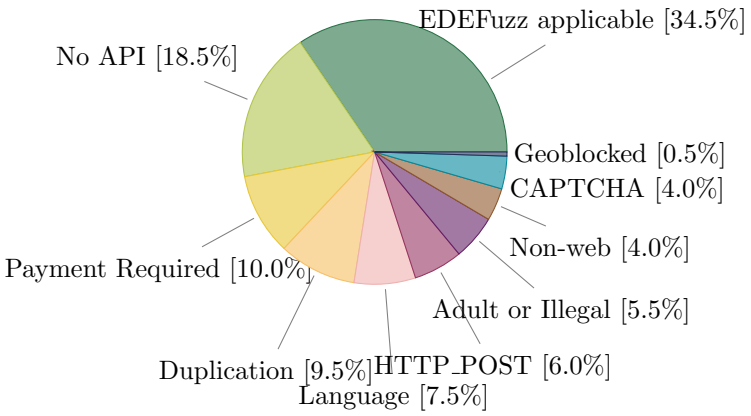

FIGURE 3.6: **Applicable websites from Alexa Top-200.** Of the 200 we found 69 (34.5%) appropriate for our testing-set. We excluded domains for the following reasons: duplication of a single service across domains, adult and illegal content, geoblocking, payment required for access, lack of an API, foreign language and encoding of parameters in POST requests (discussed in **Section 3.5.2**).

We further assessed applicability by performing an additional validation step on those 53 successful targets. This was done to test the implicit assumption of our mutation strategy that, given any two fields of a response, whether one is excessive is independent of the presence or absence of the other, i.e. that each field of a response can be assessed independently of the others. We tested this assumption for each of the 53 successful targets by taking the entire set of fields flagged by EDEFuzz as excessive and removing *all* of them simultaneously from the recorded response, and replaying that mutated response to see whether the resulting DOM passed the similarity check (Section 3.4.4) to the original. Surprisingly we found that for three (3) of the sites, removing multiple fields at once caused a difference in the DOM even when removing each of the fields individually did not (behaviour we confirmed via manual testing for each of them). For these sites, one might reasonably debate whether those flagged fields really are excessive or not, a question we leave for future work.

EDEFuzz is highly applicable; however sites that employ randomness, either in their DOM *structure* or the requests they generate, are beyond EDEFuzz's current design. The latter we posit could be addressed by fuzzy matching in the simulated server (see Section 3.4.2).

RQ3. Efficiency.

Efficiency measures not only the amount of computational time to employ EDEFuzz, but also the amount of human effort both to configure the tool and to inspect its results to determine which reported excessive data fields contain sensitive data. We evaluated efficiency across both the Australian sites (Table 3.2) and the Top 200 data set. Experiments for the Australian sites were carried out on a commodity PC with an Intel Core i7-9600K, 32GB of RAM, running Windows 10 21H1. Those for the Top 200 data set were carried out on an AWS VPS with a 16-core Intel CPU and 32 GB of memory, running Ubuntu 20.04.

The time spent on test execution was roughly linear in the number of data fields included in the response from the target API, as expected since our mutation strategy must necessarily mutate one field at a time. On average, about 8 test cases were executed per minute and this figure is consistent between the two data sets. However, computation time does not present a significant bottleneck, especially since our approach is trivially parallelised.

Regarding human effort, it took a maximum of twenty minutes per website for us to identify an appropriate API endpoint that it made use of and to then compose a configuration file, representing minimal overhead.

While naturally some human effort is required to inspect the flagged fields, to determine which are sensitive, this again took no longer than 20 minutes per website, even when EDEFuzz reported many thousands of fields as excessive. This was because many API responses contained large numbers of repeated structures, which allowed us to quickly classify thousands of flagged fields. However, we found certain flagged data fields required lengthier and more comprehensive analysis. We hypothesise that this is a fundamental limitation on automation (for the near future) as deciding whether a field is indeed sensitive is an exercise of human judgement, involving considering the application's function, what data is already publicly available, and privacy expectations, etc.

Overall we conclude therefore that our approach requires only a modest amount of human effort.

RQ4. Prevalence of Sensitive Data in EDEs.

Finally, our results allow us to draw conclusions about the prevalence of sensitive data leakage via excessive data exposure. Such conclusions necessarily *underestimate* the true extent of sensitive data leakage, even on the sites used in our evaluation, since we tested on each site only one—highly-visible and, hence, likely to be widely-tested—API.

Among the Australian websites, we find that sensitive data leakage is much more prevalent (present in 4 out of the 8 cases evaluated) than among the Alexa Top 200 sites (where it is present in only one of the 53 cases successfully evaluated). We conjecture that this should be expected, since popular sites are more widely used (and thus tested) by definition. Yet even among very popular sites, EDEFuzz still found sensitive data leakage.

We already discussed sensitive data leakage discovered by EDEFuzz in the Australian websites Company-C, Company-D, and Company-E in Section 3.3. (We also discussed vulnerabilities it found in Company-I and Company-J during testing, which are not part of our evaluation set). EDEFuzz also identified sensitive data leakage in Company-B, the learning management system with the largest market share ($\approx 34\%$). Company-B has a feature to create student groups within a subject. Instructors of a subject can view and assign group members of each group. Our tool flagged the API that lists group members. While the web page only displayed a list of names in a group, the API response contained the full list of subjects each student is enrolled into. We further found that this API is accessible from a student account as well, allowing any student to gain knowledge about into which subjects their classmates have ever enrolled.

The one instance of sensitive data leakage found by EDEFuzz in the Alexa Top 200 (Rank 91) affected an API called by a web page for downloading device drivers, which inadvertently exposed employee names as excessive data, as well as hardware IDs.

Summary.

EDEs appear prevalent, though many are relatively harmless. However, much like memory corruption vulnerabilities (whose severity can vary wildly), they can also be severe

and leak sensitive information. EDEFuzz is effective at diagnosing such vulnerabilities via its highly accurate metamorphic relation, requiring modest human effort and computational cost, while being widely applicable.

3.6 Related Work

Detecting Vulnerabilities in Web Applications Researchers have made satisfying progress in detecting and preventing *certain classes* of webapp vulnerabilities. Much work [115, 119, 126, 127, 139] exists for detecting cross-site scripting [125]. The techniques vary, including black-box [126, 139], grey-box [115] and white-box [127] approaches. Very recently, a novel grey-box fuzzing approach was proposed to detecting SQL and command injections [109].

However, excessive data exposure has received comparatively little attention despite being one of the most common vulnerabilities [112]. To the best our knowledge, there is no published research focusing on an automated approach to detect this class of vulnerability in web applications: our tool EDEFuzz is the first of its kind.

[160] studied a white-box and semi-automated mechanism to identify EDE vulnerabilities in Android applications—not in web applications. Unlike EDEFuzz which runs the program on two different outputs from the web server (one with a data field deleted) and looks for the absence of difference in the DOM to detect leakage, their approach requires (decompiled) source code of the applications to do instrumentation and static data-flow analysis. Its static analysis identifies potential EDEs by flagging data received by the app over the network (source) that is then serialised to a Java object but that then never propagates to the user interface (sink). A subsequent dynamic analysis that relies on program instrumentation and manual app interaction is used to confirm potential vulnerabilities, wherein the human analyst must manually generate tests that attempt to trigger the EDE. EDEFuzz also requires manual effort to interact with a web application to trigger the web API under test, and like Koch et al. also to confirm the sensitivity of leaked information. So [160] and our approach are complementary. As demonstrated in fuzzing research [155], combinations of complementary approaches could yield better results and we leave that for future work.

Metamorphic Testing/Fuzzing As discussed in Section 3.2, one of the most important steps of metamorphic testing is to identify metamorphic relation(s). This requires creativity and a good understanding of the system under test. To ease this crucial step, Segura et al. proposed six abstract relations from which concrete relations can be defined. Specifically, the authors identified 60 API-specific metamorphic relations in their work. Their relations specify how related web requests should produce related responses, and so are inapplicable to detecting EDE. The fundamental insight of EDEFuzz that shows how metamorphic fuzzing is applicable to detecting EDE is not to mutate the *request* (as in all prior web fuzzing work), but to mutate the *response* instead.

RESTful Web API Testing RESTler [84]—the state-of-the-art RESTful API fuzzing approach—used server states, relying on response codes to identify server crashes on APIs used by cloud services. Their tool infers dependencies among requested APIs to guide the generation of new test cases. [142] suggested an extension to RESTler to report the violation of four rules commonly applied to REST services, in addition to server crashes. [143] improved RESTler's test generation algorithm by representing the payloads in a tree structure on which structure-aware mutation operators can be applied. Pythia [144] augmented RESTler with coverage-guided feedback and it implemented a learning-based mutation strategy. Specifically, it utilised a statistical model to gain knowledge about frequent ordering of calling APIs from seed inputs. It used a regular grammar to encode an API request and perform mutation by adding noise to the request. Neither RESTler nor its follow-up works can detect EDE vulnerabilities. Moreover, this line of work focuses on mutating the API requests while EDEFuzz modifies the API responses.

Record-replay Mechanism in Web Application Testing Record-replay models are popular in testing web applications and services. We use two types of record-replay in our work. The first bears similarity to WaRR [161], which records the interaction between a user and a web application. It allows the recorded traces to be later replayed to simulate the user interacting with the web application. Another similar work is Timelapse [162], in which researchers log unexpected behaviours in web applications to help developers visualise, demonstrate and better understand bugs. We follow their mechanism, automating interactions with a web application through a headless web-driver. The second form of record-replay tools capture communications between a server and client, to be replayed at a later time [153]. While existing work focuses on producing

an exact replication during a replay stage, EDEFuzz uses a simulated server to instead supply mutated server responses.

Web Change Detection The components of a web application may change over time, hindering research and testing that relies on consistency. Researchers have looked into different strategies to compare two pages and identify their differences. One strategy was proposed over twenty years ago and relies on the HTML DOM tree to monitor structural changes on the web page [163]. Other relevant work includes X-Diff [164] which aimed at detecting changes in an XML document, and [165] which improved the efficiency of Hungarian algorithm in detecting web page changes.

Modern web applications have increased in complexity and often break from prior design paradigms. As a result, past approaches for page comparison are less effective than they once were. To address this challenge, Waterfall [166] uses two versions of the same web application, in detecting locator changes and applying fixes. WebEvo [167] attempts to identify evolution of a web application through detecting semantic structure changes. Both approaches aim at matching contents between two structural different web pages, while our work focus on identifying both structural difference and content difference.

3.7 Discussion and Future Directions

Simple ideas are often the best. From our evaluation we conclude that EDEs appear prevalent, and that EDEFuzz is effective at finding them, including sensitive data leakage, with acceptable efficiency and requiring a modest amount of human work. Its metamorphic relation yields precise results in practice (a TP rate of 98.65%). It is also generally applicable, and it can be parallelized easily. At the same time, our results suggest avenues for improvement.

Handling HTTP POST Requests by parsing request bodies, which affected 6% of our evaluation targets from the Alexa Top-200 websites.

Improving Efficiency. Even though our Simulated Server helps EDEFuzz achieve a reasonable fuzzing throughput, we can improve it further by leveraging the recent advancement in the topic of snapshot-based fuzzing [168]. As the design of EDEFuzz is modular, the change could be minimal. Specifically, we can take a snapshot of the client

at the state S_1 when a request to the target API has just departed (See **Section 3.4**). We can then restore the snapshot for each fuzzing iteration instead of replaying requests using the Web Driver.

Fuzzy Matching. We posited in **Section 3.4.2** that EDEFuzz's current Simulated Server has implementation limitations that prevent it from being able to respond to randomised requests. It could be overcome by implementing fuzzy matching. While affecting only 8.7% of our sites, this is a straightforward avenue for future investigation.

Mutation Algorithm. In this work, we only apply data field deletion to mutate the server responses. It would be interesting to study other mutation strategies and update/improve our metamorphic relation accordingly. For instance, consider a website that queries the available stock of an item and converts the numeric response into one of two answers it displays to the user: "out-of-stock", and "in stock". Suppose deleting this field causes neither answer to be displayed. While this case didn't arise in our evaluation, EDEFuzz would consider this a false negative, even though the response leaks more information than is displayed to the user. A mutation strategy that modifies this response value without deleting it would detect this form of EDE.

Expanding the Evaluation Set. A promising direction for future work is to expand the evaluation set of EDEFuzz by including a broader range of open-source web applications. While the current study demonstrates strong results across a diverse set of real-world systems, applying EDEFuzz to openly available targets could provide additional insights into its generalisability, reveal corner cases not encountered in proprietary deployments, and support more reproducible benchmarking by the research community. This broader evaluation could also help identify new patterns of excessive data exposure and guide further refinement of the tool's heuristics and configuration model.

Chapter 4

A User-Centered Evaluation of EDEFuzz

Contribution

This chapter presents a user-centered evaluation of EDEFuzz, aimed at understanding its effectiveness and usability from a developer's perspective. While much of this thesis focuses on technical innovations to minimize human effort in automated web application testing, this chapter emphasizes that practical adoption also depends on usability and developer experience. Through a controlled user study, we investigate how real-world users interact with EDEFuzz and assess its impact on vulnerability discovery in realistic settings. These insights reinforce the broader thesis objective of creating security tools that are not only technically effective but also accessible and usable in everyday development workflows.

4.1 Introduction

The rapid proliferation of web APIs has revolutionized how modern software systems interact, fostering innovation and efficiency across industries. From powering e-commerce platforms to enabling seamless integration between services, web APIs play an essential role in the digital ecosystem. However, as APIs grow in complexity and adoption, so do the associated security risks. Among these, excessive data exposure (EDE) has emerged as a particularly insidious vulnerability, where APIs unintentionally reveal sensitive or unnecessary data in their responses. Ranked as the third most critical API security issue

by OWASP in 2019 [113], EDE poses significant threats to user privacy, organizational compliance, and system integrity.

To address this vulnerability, various tools and methodologies have been developed, including EDEFuzz (presented in Chapter 3)—an innovative fuzzing tool for systematically detecting EDEs. While EDEFuzz has demonstrated strong technical capabilities, its practical utility and usability remain largely unexplored. Understanding how such tools can aid developers in identifying vulnerabilities and their associated costs is critical for advancing secure software practices.

This chapter presents a user study aimed at evaluating the usefulness and usability of EDEFuzz. Participants were tasked with identifying EDE vulnerabilities in web applications with and without EDEFuzz, allowing for a comparison of task efficiency and effectiveness. Additionally, participants composed configuration files for EDEFuzz, providing insights into the labor cost and complexity of using the tool. To further understand user perspectives, participants were asked to rank the OWASP API Top 10 vulnerabilities from 2019 based on perceived prevalence and impact, shedding light on their awareness of EDE and other API security concerns.

The study's findings highlight the potential of EDEFuzz to assist developers in identifying EDEs more efficiently while providing actionable insights into its adoption and associated challenges. This introduction sets the stage for discussing the study's design, results, and implications for secure API development.

4.2 Background

4.2.1 EDEFuzz

EDEFuzz (presented in Chapter 3) is a systematic and automated tool designed to detect EDE vulnerabilities in web APIs. It addresses the limitations of manual vulnerability detection methods by leveraging a fuzzing-based approach combined with a novel metamorphic relation. This relation allows the tool to detect excessive data exposure by comparing the changes in a web page's Document Object Model (DOM) when specific data fields are removed from API responses. If the removal of a data field does not impact the rendered content, the field is flagged as excessive.

At its core, EDEFuzz generates and sends diverse payloads to API endpoints and analyzes their responses systematically. By dynamically mutating input data and monitoring the output, EDEFuzz can detect instances where APIs disclose more data than intended. The tool integrates heuristics and detection strategies to maximize accuracy and reduce false positives. To further enhance reliability, EDEFuzz employs a record-replay mechanism, using a simulated server to replay recorded interactions. This approach ensures deterministic results and minimizes reliance on external servers during fuzzing, enabling efficient and scalable testing.

EDEFuzz has demonstrated high applicability and accuracy. In experiments with popular web applications, it achieved a true positive rate of 98.65% and identified previously unknown sensitive data leaks. Additionally, it provides user-friendly features, such as detailed reports of detected vulnerabilities, including the specific endpoints and data fields affected. These reports help developers quickly understand the scope and severity of the issue, enabling timely remediation. By automating the detection process and providing actionable insights, EDEFuzz streamlines the security assessment of web APIs, reducing the burden on developers and security professionals.

4.3 Methodology

4.3.1 Study Design

The primary purpose of this study was to evaluate the usability and effectiveness of EDEFuzz as a tool for identifying excessive data exposure (EDE) vulnerabilities in web applications. In particular, we aimed to assess whether EDEFuzz could improve the accuracy and efficiency of vulnerability discovery while reducing human effort, compared to traditional manual inspection methods. Additionally, we sought to explore how developers perceived the tool's usefulness and ease of use, given that usability is a key factor influencing real-world tool adoption.

To achieve these goals, we employed a mixed-methods study design, combining both

quantitative and qualitative elements. Quantitative data were collected through time-on-task measurements, Likert scale ratings [169], and controlled comparisons of manual versus tool-assisted performance. Qualitative insights were gathered through semi-structured interviews and open-ended feedback questions, allowing participants to express subjective experiences, challenges, and suggestions for improvement. This combination of methods was chosen to provide both objective evidence of EDEFuzz's impact and a deeper understanding of user perceptions.

We chose a task-based evaluation approach, in which participants engaged in realistic vulnerability discovery activities on custom-built web applications containing known excessive data exposure issues. Task performance was complemented by semi-structured interviews that explored participants' impressions of the tool, their mental models of its operation, and their broader thoughts on the problem of EDE vulnerabilities. This design allowed us to simulate realistic workflows while maintaining sufficient control to support hypothesis-driven analysis.

Overall, the study was designed to align with the broader goals of the thesis: advancing practical, low-effort automated security testing techniques that integrate smoothly into natural development and usage patterns. Evaluating EDEFuzz through both technical and user-centered lenses provides critical evidence for its effectiveness, usability, and potential for real-world adoption.

4.3.2 Research Questions and Hypotheses

The study was guided by the following hypotheses:

- **H1:** Developers will rank both the prevalence and impact of EDE in the lower half (ranks 6–10) of the OWASP API Security Top 10.
- **H2:** Using EDEFuzz can reduce the human effort (measured as time spent) required to identify excessive data exposure vulnerabilities, compared to manual testing.
- **H3:** Developers will perceive EDEFuzz as both helpful and easy to use.

These hypotheses were designed to evaluate both the technical effectiveness and the perceived usability of EDEFuzz, aligning with the broader goals of practical, low-effort security testing.

4.3.3 Participants

We recruited 20 participants for the user study through internal advertising at the University of Melbourne. Our recruitment targeted individuals with relevant background knowledge in security and software testing. Specifically, we focused on three groups: (1) Master's students who had completed the "Security and Software Testing" course, (2) Master's students who had taken the "Web Security" course, and (3) PhD students conducting research in the areas of security and/or software testing. This sampling strategy was intended to ensure participants had sufficient foundational knowledge to engage meaningfully with the tasks and the EDEFuzz tool.

4.3.4 Materials and Tools

We developed two custom web applications that simulate realistic API-driven interactions, each embedding excessive data exposure vulnerabilities. Specifically, certain sensitive data fields were returned via APIs but were unused in the frontend UI.

Participants used the Google Chrome browser and its built-in Developer Tools (DevTools) to inspect network traffic and API responses. During the tool-assisted task, participants were provided with EDEFuzz's output, which highlights potentially excessive fields. Participants were introduced to the configuration grammar and shown examples of configuration files used by EDEFuzz.

All sessions were conducted in person, using participants' own laptops preloaded with necessary applications. Task durations were recorded for subsequent analysis.

4.3.5 Procedure

Each interview session lasted approximately 90 minutes and followed a semi-structured protocol consisting of the following stages:

1. **Introduction and Background**

   Participants were verbally introduced to the concept of excessive data exposure vulnerabilities and their relevance in modern web applications. Participants were asked to rank the OWASP API Top 10 vulnerabilities by perceived prevalence and impact. During this task, we clarified any other type of vulnerabilities participants were unfamiliar with.

2. **Demonstration of Excessive Data Exposure**

   We walked participants through a sample web application to demonstrate how excessive data fields might appear in real-world scenarios. Using Chrome DevTools, participants observed API calls and corresponding response data. Together, we examined several fields and discussed which ones were necessary for the web application's functionality and which were excessive.

3. **Tool-Assisted Detection Task**

   Participants were then asked to analyze a different web application (developed for the study) to identify unused or excessive data fields. EDEFuzz output was provided, highlighting data fields that were not used by the frontend web page. Participants were instructed to manually inspect the application and verify which fields were excessive. The time taken to complete this task was recorded.

4. **Manual Detection Task**

   Using a second web application, participants repeated the same detection task without the aid of EDEFuzz. They relied solely on Chrome DevTools and manual inspection to identify unused data fields. Task duration was again recorded for comparison.

5. **Configuration Task**

   Participants were introduced to EDEFuzz's configuration file format and shown the configuration file used in the earlier tool-assisted task. They were then asked to write a configuration file for the second web application, capturing the necessary steps to trigger relevant API calls. A configuration was considered complete if it could be executed successfully with EDEFuzz. Time to completion was recorded.

6. **Perceived Usefulness and Usability**

   Participants rated EDEFuzz's usefulness (for identifying excessive data fields) and usability (in terms of creating the configuration file and interpreting output).

Ratings were provided using a 5-point Likert scale: *strongly disagree, disagree, neutral, agree, strongly agree.*

7. **Open-Ended Feedback**

   Finally, participants were invited to share any additional comments or suggestions about EDEFuzz. This included aspects they felt the tool did well and areas where it could be improved.

4.3.6 Data Analysis

Data analysis followed the plan pre-registered on AsPredicted.org. The study was designed to evaluate participants' awareness of excessive data exposure (EDE), the efficiency of EDEFuzz compared to manual detection, and the perceived usability and usefulness of the tool.

- **Perceived Prevalence and Impact of EDE**

  Participants ranked EDE among the OWASP API Top 10 security risks. We calculated the average rank for both perceived prevalence and impact. A one-sample t-test was used to determine whether the mean rank for EDE fell in the lower half of the list (i.e., between ranks 6 and 10), in support of Hypothesis 1.

- **Efficiency of EDEFuzz Compared to Manual Detection**

  Each participant completed one task using EDEFuzz and one manually, across two web applications, in a crossover design. We recorded the time taken (in minutes) for both tasks. A paired samples t-test was performed to compare the task durations and test Hypothesis 2 regarding the tool's ability to reduce human effort.

- **Usability and Usefulness of EDEFuzz**

  Participants rated their agreement with two statements: (1) "EDEFuzz is helpful in identifying excessive data fields", and (2) "EDEFuzz is easy to use". Responses were encoded numerically on a 5-point scale: *strongly disagree* (-2) to *strongly agree* ($+2$). We computed the mean and standard deviation for each item, and conducted one-sample t-tests to assess whether the average response was significantly greater than zero (neutral), corresponding to Hypothesis 3.

- **Configuration Task Performance**

  We recorded the time (in minutes) required by participants to compose a valid configuration file for EDEFuzz. Although no specific hypothesis was associated with this task, the data provide supporting context for assessing usability.

- **Outlier Handling and Exclusion Criteria**

  As pre-registered, participants who were unable to identify any excessive data fields during the manual testing task were excluded from the analysis. Additionally, task durations were screened for outliers using the 1.5×IQR rule, and sensitivity analyses were conducted with and without those values to confirm robustness of the results.

All statistical analyses were conducted using *R*, with significance threshold of $\alpha = 0.05$.

4.3.7 Ethical Considerations

This study was approved by the Human Research Ethics Committee at the University of Melbourne. Ethical approval was obtained prior to any data collection, and the study was pre-registered on AsPredicted.org to promote transparency and reproducibility.

Participants were provided with a plain language statement and required to sign a consent form before participating. The consent form stated that participation was voluntary, and that participants could withdraw from the study at any time without explanation or penalty, and withdraw any unprocessed data they had provided. It also clarified that no audio or video recording would occur and that only basic background information would be collected.

Participants were informed of the study's purpose—evaluating the usability and usefulness of the EDEFuzz tool for identifying excessive data exposure vulnerabilities. They were assured that their responses and data would remain confidential, securely stored at the University of Melbourne, and destroyed five years after publication.

4.4 Results

4.4.1 User Awareness of EDE

To understand the user awareness of Excessive Data Exposure, we asked participants to order the 10 vulnerability types in OWAPS API top 10 [113], regarding their prevalence and impact. The average ranking is presented in Table 4.1.

TABLE 4.1: Average ranking of 10 vulnerabilities regarding their prevalence and impact. In our survey, 1 means most prevalent or serious, while 10 means least prevalent or serious.

Vulnerability	Avg. Rank (Prevalence)	Avg. Rank (Impact)
Broken Function Level Authorization	7.10	4.40
Broken Object Level Authorization	7.15	4.90
Broken User Authentication	6.75	2.40
Excessive Data Exposure	3.80	6.35
Improper Assets Management	5.90	7.65
Injection	5.00	2.65
Insufficient Logging & Monitoring	3.45	8.45
Lack of Resources & Rate Limiting	5.00	6.75
Mass Assignment	6.35	5.75
Security Misconfiguration	4.50	5.70

The survey results suggest that people perceive Excessive Data Exposure as a relatively common vulnerability, with a prevalence score of 3.80, indicating they rank it among the more frequently encountered issues, second only to **Insufficient Logging & Monitoring** (3.45). However, participants seem to view its potential impact as relatively moderate, with an average rank of 6.35, which is lower than that of six other vulnerabilities (broken function level authorization, broken object level authorization, broken user authentication, injection, mass assignment and security misconfiguration). This perception indicates that while developers may recognize Excessive Data Exposure as a frequent issue, they might underestimate the severity of its consequences, reflecting a possible gap in understanding its critical implications for API security.

A Wilcoxon Signed-Rank Test was performed to compare the rank of Excessive Data Exposure against every other vulnerability to test if it is consistently ranked higher or lower. The result is presented in Table 4.2.

Table 4.2: Raw Results of Wilcoxon Signed-Rank Test for Prevalence and Impact of OWASP API Top 10 Vulnerabilities

Vulnerability	**Prevalence**		**Impact**	
	W	**P-Value**	**W**	**P-Value**
Broken Function Level Authorization	29.5	0.00499	161.0	0.03740
Broken Object Level Authorization	33.5	0.00792	146.0	0.12711
Broken User Authentication	42.0	0.01924	191.5	0.00123
Improper Assets Management	33.5	0.00775	54.0	0.05408
Injection	72.0	0.22160	200.5	0.00038
Insufficient Logging & Monitoring	119.5	0.59166	33.5	0.00767
Lack of Resources & Rate Limiting	69.5	0.18924	93.5	0.67842
Mass Assignment	36.5	0.01083	129.5	0.36618
Security Misconfiguration	82.0	0.39782	126.5	0.43026

The results of the Wilcoxon Signed-Rank Test provide significant insights into participants' perceptions of the prevalence and impact of Excessive Data Exposure against other vulnerabilities from the OWASP API Top 10.

For **prevalence**, vulnerabilities like **Broken Function Level Authorization** ($p = 0.005$), **Broken Object Level Authorization** ($p = 0.008$), and **Improper Assets Management** ($p = 0.008$) were identified as significantly distinct (less prevalent) from EDE. However, **Injection** ($p = 0.222$), **Insufficient Logging & Monitoring** ($p = 0.592$), **Lack of Resources & Rate Limiting** ($p = 0.189$) and **Security Misconfiguration** ($p = 0.398$) showed no significant difference, suggesting these vulnerabilities are considered to have a similar prevalence by participants.

For **impact**, vulnerabilities such as **Broken User Authentication** ($p = 0.001$) and **Injection** ($p = 0.0004$) were rated as highly impactful, indicating participants' recognition of their potential to cause severe damage if exploited. On the other hand, vulnerabilities like **Lack of Resources & Rate Limiting** ($p = 0.678$), **Mass Assignment** ($p = 0.366$) and **Security Misconfiguration** ($p = 0.430$) were perceived as similar to Excessive Data Exposure (less impactful).

4.4.2 Usability of EDEFuzz

4.4.2.1 Human Effort in Determining Excessive Fields

To understand the usability of EDEFuzz, we asked each participant to highlight excessive data fields in two given APIs, each within a simple web application. Each participant

was given the output of EDEFuzz (indicate which data fields were flagged by EDEFuzz) in one task to aid their judgement, while no additional material was given in the other task. By using this set up, we aim at simulating the scenario where a developer or tester uses EDEFuzz to identify EDEs, and using manual testing to identify EDEs. We recorded the time (in minutes) each participant spent on each task, and present the raw result in Table 4.3.

Table 4.3: Time spent on Experiment 1 (manual testing without EDEFuzz) and Experiment 2 (testing with EDEFuzz) for each participant.

Participant	**Without EDEFuzz (min)**	**With EDEFuzz (min)**
1	23	9
2	17	5
3	6	14
4	8	11
5	13	6
6	18	8
7	8	4
8	8	5
9	3	3
10	5	10
11	3	2
12	3	2
13	8	8
14	3	3
15	4	2
16	3	3
17	8	7
18	7	2
19	4	5
20	6	4

We applied *Repeated Measures ANOVA* to test for differences across the twe experiments, while considering variability between participants. This is to explore if participant-level differences impact the results. ANOVA reported a $Pr(> F)$ value of 0.0413. With a chosen significance level of 0.05, ANOVA indicates the difference between Experiment 1 and Experiment 2 is statistically significant.

4.4.2.2 Human Effort in Preparing

Each participant was introduced how a configuration file is prepared, and then asked to compose a configuration file on their own. We recorded the time they spent to write a configuration file that can be used by EDEFuzz to complete the fuzzing process. All

20 participants successfully composed a configuration file for a given web application within 10 minutes (despite it was their first time to do so). The average time spent was 5.05 minutes and 14 participants (70%) completed this task with no more than 5 minutes. This result shows manageable and scalable human effort of using EDEFuzz.

4.4.3 Usefulness of EDEFuzz

After the practical experiments, we asked participants if they thought the output of EDEFuzz was helpful in determining excessive data fields. Out of 20 participants, all agreed that the output of EDEFuzz was helpful in identifying excessive data exposures (EDEs). Specifically, 10 participants (50%) selected **Agree**, while the remaining 10 participants (50%) chose **Strongly Agree**. This unanimous positive feedback suggests a high level of perceived utility and effectiveness of EDEFuzz in identifying EDEs.

The majority of participants found EDEFuzz easy to use, including the processes of preparing a configuration file, executing test cases, and analyzing outputs. Specifically, 60% (12 participants) strongly agreed, and 30% (6 participants) agreed. However, 5% (1 participant) remained neutral, and 5% (1 participant) disagreed. These results indicate overall positive feedback on usability, with a small minority suggesting areas for potential improvement.

4.4.4 Limitations and Future Research

While this study provides valuable insights into the usability and effectiveness of EDEFuzz, several limitations should be noted.

First, participants' assessments of vulnerability prevalence and impact were based on subjective rankings. While these rankings help capture developer perceptions, they may not align with real-world exploit frequency or severity. Future research could compare these perceptions against empirical vulnerability data to gain a more accurate understanding of how developers evaluate security risks.

Second, many participants had limited prior exposure to excessive data exposure (EDE) vulnerabilities. For most, this study was their first hands-on experience with such issues. As a result, participants likely focused on different aspects of the task and adopted

varying manual strategies. This variability may have introduced inconsistencies in how tasks were approached and completed.

In some cases, participants performed manual testing even during the tool-assisted task—comparing their manual results with the output of EDEFuzz. In these instances, the intended comparison between tool-assisted and manual effort was compromised, as both tasks effectively involved manual analysis. A more controlled experimental design or clearer task guidelines could help reduce such confounding behavior in future studies.

Third, the study involved a relatively small and specialized sample—primarily students and early-career researchers with a background in security or software testing. While this ensured a baseline level of technical familiarity, it limits the generalizability of findings to broader developer populations. Future work could include participants from industry, with diverse roles and varying levels of experience.

Finally, the web applications used in the study were developed in-house and may not fully represent the complexity or variability of real-world systems. Evaluating EDEFuzz on larger-scale, production-grade, or open-source applications would help assess its generalizability and scalability in practical settings.

4.5 Discussion

This study investigated the awareness developers have of Excessive Data Exposure (EDE) vulnerabilities, as well as the usability and usefulness of a new detection tool, EDEFuzz. The results reveal several key insights.

First, participants ranked EDE as one of the more prevalent vulnerabilities in web APIs, suggesting general awareness of its frequency in practice. However, its perceived impact was rated significantly lower compared to threats like injection and broken authentication. This mismatch highlights a potential gap in developer understanding: although EDE is commonly encountered, its consequences may be underestimated. Bridging this gap could be critical in encouraging developers and organizations to take EDE more seriously.

Second, the evaluation of EDEFuzz provided encouraging evidence for its practical utility. Participants spent significantly less time identifying EDEs when aided by EDEFuzz than when using manual testing alone. This supports our hypothesis that the tool can reduce the human effort involved in EDE detection. However, as noted in the limitations, a few participants applied manual methods even during the tool-assisted task, complicating the time-based comparison. Nonetheless, the general trend indicates time-saving potential with tool support.

In terms of usability, participants were able to compose effective configuration files in a short amount of time—even without prior experience. This demonstrates that EDEFuzz requires minimal onboarding effort and can be adopted quickly, particularly by developers with some familiarity in security or testing workflows. Feedback on the tool's interface and outputs was overwhelmingly positive, reinforcing its perceived usefulness in aiding EDE detection.

These findings suggest that EDEFuzz can play an important role in improving the developer workflow for detecting excessive data exposure. The tool is accessible, reduces effort, and provides relevant and helpful insights. Moreover, the study contributes to a growing understanding of how developers perceive security vulnerabilities—showing that while some issues like EDE are recognized, their full implications may still be underappreciated.

Future work could further refine EDEFuzz's interface and documentation based on the small portion of neutral or negative usability feedback. Additionally, broader deployment and testing in industrial contexts could validate the tool's effectiveness in more complex and realistic environments. Cross-referencing developer perceptions with actual vulnerability data could also yield a deeper understanding of where educational or tooling interventions are most needed.

Overall, this study demonstrates that EDEFuzz is a promising tool for enhancing developer capabilities in addressing Excessive Data Exposure vulnerabilities, and it opens the door for more systematic and scalable approaches to secure web application design and testing.

Chapter 5

TrailBlazer: Practical End-to-end Web API Fuzzing

Contribution

This chapter introduces TrailBlazer, a specification-less web API fuzzer designed to detect server-side crashes in undocumented or poorly documented API endpoints. In line with the overarching goal of this thesis—to explore how naturally generated human interaction data can be repurposed to enable low-effort, high-impact web application testing—TrailBlazer bootstraps its testing process from web traffic collected during ordinary application use. By combining generation-based and mutation-based fuzzing without requiring formal API specifications, TrailBlazer expands the reach of automated testing into a critical but often overlooked part of the web application attack surface.

5.1 Introduction

In the preceding chapters, we presented EDEFuzz, a testing technique specifically designed to detect excessive data exposure (EDE) vulnerabilities in web applications. EDEFuzz introduced a novel metamorphic relation tailored to API response consistency and demonstrated effectiveness in uncovering EDEs through both tool-based analysis and a controlled user study. Like many other web API testing tools [84, 100, 101], it assumes that the relevant API endpoints are already known and that valid test inputs can

be constructed—assumptions that limit their applicability in broader web API testing contexts.

This chapter introduces TRAILBLAZER, a practical end-to-end web API fuzzing framework that addresses a more general class of challenges in API testing. Unlike EDEFUZZ, which focuses on a specific vulnerability class, TRAILBLAZER tackles the fundamental problems of discovering where to send API requests and how to construct valid, semantically meaningful payloads—crucial steps in testing any web API effectively. Importantly, TRAILBLAZER is designed to operate even in the absence of a formal API specification, enabling black-box testing of real-world web applications at scale.

Web APIs are the backbone of virtually every online service. However, despite their importance, the scale and dynamic nature of web APIs present unique challenges for effective (security) testing. With APIs now widely used across diverse applications, a substantial number remain inadequately tested, it is necessary to develop automated approaches (such as improved fuzz-testing) that obviate the need to manually assess each API. This chapter takes steps to address this challenge, aiming to provide a tool that can be set loose on such an API and attain strong test performance, regardless of if the API is documented with a machine-readable specification or not.

Recent tools such as RESTler [84] and Schemathesis [100] leverage OpenAPI specifications [27] to perform generation-based fuzzing, aiming to uncover bugs and vulnerabilities automatically. Yet, as effective as these tools are, they cannot handle natively most web APIs, as the tools are dependent on the availability of high-quality specifications.

API specifications are generally human-written and, due to the labor- and time-cost of keeping the specs current, are often incomplete and outdated, where they exist at all (see Figure 5.1). Outdated specs in turn, limit the efficacy of the aforementioned tools, which rely on the specs. Consequently, fuzzing based solely on these specs misses critical API functionality and hampers test effectiveness. Additionally, existing generation-based approaches rely heavily on syntactic rules but lack support for enforcing semantic constraints, which are crucial for generating meaningful test cases.

To address the need for better specs, prior research has attempted to infer API specs from network traffic [31, 32, 34]. However, these methods typically assume prior knowledge

> *"API testing continues to be a top priority for almost all developers, with over 90% of developers reporting testing or planning to test their APIs."*
>
> — State of APIs 2022 (RapidAPI) [170]

> *"56% of APIs are private APIs. While 58% of developers rely on internal documentation to learn APIs, 39% say inconsistent docs are the biggest roadblock ."*
>
> — 2024 State of API Report (Postman) [1]

FIGURE 5.1: Industry views on the importance of web API testing and challenges raised by inconsistent API docs.

of endpoints and/or produce specifications that modern tools cannot fully utilize, thus only partially solving the problem.

In this work, we propose TRAILBLAZER, the first practical end-to-end web API fuzzing framework that overcomes these limitations. TRAILBLAZER captures real-time traffic between a web application and the remote server through a custom plugin, automatically identifying APIs and inferring an OpenAPI specification—a standard for API specs—that can be used directly with existing approaches to testing. The captured traffic and inferred OpenAPI specification enable us to combine generation-based fuzzing and structure-aware, mutation-based fuzzing. Prior studies have demonstrated the effectiveness of combining generation and mutation-based fuzzing across different domains [18–20]; We believe TRAILBLAZER is the first to apply this powerful combination to web API fuzzing.

We implemented TRAILBLAZER and evaluated it on popular content management systems (CMS), a category of software that underpins a wide variety of services from e-commerce to company websites. CMS platforms, due to their open-source nature and the ethical considerations they satisfy for testing, offer an ideal testing ground for our approach.

We find that TRAILBLAZER works effectively on a broad-cross section of web applications that utilize APIs in their web interface. Further, when pointed at a web application, our implementation is able to automatically discover API endpoints and generate test payloads via mutation (with a spec). By combining generation-based fuzzing with mutation-based fuzzing, TRAILBLAZER achieved high code coverage and identify previously unknown bugs, with minimal human effort.

In summary, this paper makes the following contributions:

- We analyze the limitations of existing web API fuzzing approaches, particularly RESTler and Schemathesis, underscoring their dependence on OpenAPI specifications and their lack of structure-aware mutation support.
- We present TRAILBLAZER, the first practical end-to-end web API fuzzing framework that automates endpoint discovery and specification inference, enabling both generation-based and structure-aware mutation-based fuzzing.
- We empirically evaluate TRAILBLAZER on widely-used CMS platforms. Our preliminary results show that TRAILBLAZER is effective in discovering zero-day vulnerabilities. Specifically, TRAILBLAZER identified 12 previously unknown crash-triggering bugs, with six already confirmed and fixed by maintainers, and another four confirmed.

5.2 Background

5.2.1 REST API and GraphQL API

REST (Representational State Transfer) [171] is a popular architectural style for designing web APIs that emphasize stateless interactions and the use of standard HTTP methods. In a REST API, each URL represents a specific resource, and interactions are performed via standard HTTP methods. REST APIs are designed to be stateless, meaning that each request from a client must contain all the necessary information for the server to process it, without relying on any stored state.

While many modern Web APIs follow REST principles, not all do. Conventional Web APIs may utilize HTTP as the transport protocol and JSON as the data format but may not strictly adhere to REST's constraints, such as statelessness or resource orientation. Additionally, some APIs may use alternative architectures like GraphQL, which allows clients to query exactly the data they need, or follow custom protocols that don't fit neatly into either REST or GraphQL paradigms.

APIs examined in this work can operate in various architectures, allowing for more flexibility in how they are implemented and tested. While this flexibility offers developers more freedom, it also introduces challenges, particularly the lack of standardization. This absence of clear architectural conventions complicates the process of automated testing,

as there are fewer consistent patterns to rely on when designing and generating test cases.

5.2.2 Web API Testing

Web API fuzzing recently garnered increased interest from both industry and academia. As discussed in Section 2.4, a substantial body of work in web API testing has focused on leveraging formal specifications—particularly OpenAPI Specifications (OAS)—to automate test generation. Numerous tools and frameworks have been developed to exploit these specifications in diverse ways, including *RESTler* [84], *Schemathesis* [100], *EvoMaster* [101], *Morest* [145], *QuickREST* [102], *RESTest* [146] and *RestTestGen* [103]. Collectively, these tools reflect a sustained interest in using API specifications to automate the discovery of functional bugs, rule violations, and security vulnerabilities in web services.

5.2.2.1 Schemathesis

Schemathesis is an automated tool designed for testing web APIs, leveraging OpenAPI specifications to generate and execute a variety of test cases. Built on top of property-based testing principles, Schemathesis systematically generates a range of inputs based on the constraints defined in the API specification. It then checks how the API responds to these inputs, detecting potential issues like server errors, incorrect data handling, or violations of the OpenAPI-defined schema. Schemathesis supports both positive and negative testing through built-in checks, including checks for server error responses (`not_a_server_error`) and validation that APIs reject data outside the expected schema (`negative_data_rejection`). This allows developers to ensure robust API behavior across diverse cases, while also identifying undocumented edge cases. Schemathesis is particularly powerful when paired with a well-documented OpenAPI specification, as it can more thoroughly test the API's responses, content types, and expected data formats.

5.2.2.2 Limitations of Existing Tools

Here we emphasize limitations of existing tools, which we address in this chapter. Firstly, a large proportion of web applications do not have a machine-parseable specification for

their APIs. In a scoping analysis we performed of 68 open-source content management systems (CMS) we found that while 59 (86.8%) had public APIs, only 23 (33.8%) provided an OpenAPI specification. Web applications without such a specification impose two key challenges: (1) a test must first identify the appropriate API endpoints, and (2) the syntax and semantics of a valid payload must be determined. Furthermore, existing specifications may be inaccurate or outdated. As we discovered in our scoping evaluation, this may include both API endpoints that are not part of specification or API functionalities that are undocumented.

5.3 Workflow

TRAILBLAZER is designed to address the challenge of testing web APIs when no OpenAPI specification is provided. Current state-of-the-art tools like RESTler and Schemathesis rely heavily on OpenAPI specifications to generate effective test cases. However, many web applications do not publish these specifications, making it difficult for developers and testers to perform automated API testing. We propose an end-to-end workflow to automate web API testing, by producing test cases from captured API traffic. Below, we depict the detail of TRAILBLAZER in Figure 5.2, and outline its key steps.

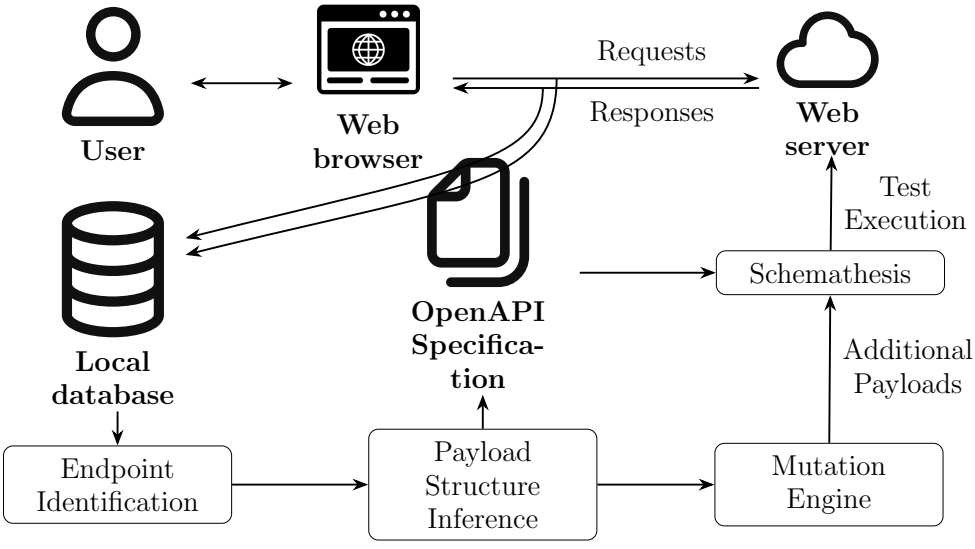

FIGURE 5.2: **The workflow of Trailblazer**. A user interacts with the web application via a web browser. A web browser plugin captures all API traffic and stores into a local database. The captured traffic is analysed to identify API endpoints and infer the structure of request payloads. An OpenAPI specification is then generated, which serves as input to automated testing tools such as Schemathesis. Additionally, our mutation engine generates supplemental test cases to further enhance test coverage.

Step-1 Capturing API Traffic We developed a web browser plugin capable of capturing all API traffic generated by a web application. This plugin hooks into both the `XHR` and `Fetch` methods in JavaScript, ensuring that every API request and its corresponding response are captured and stored. Users (developers, testers, or regular users) interact with the web application, exploring various features to trigger as many API calls as possible. API traffic can be captured in two scenarios: (a) by a single user on a single device, or (b) by multiple users across different devices. In the latter scenario, different users may have distinct roles, leading to API calls that are role-specific. This design allows capturing diverse API usages that may not be triggered by a single user.

Step-2 Endpoint identification and requests grouping Once API traffic is captured, the next step is to identify API endpoints and group requests accordingly. API endpoints are identified by analyzing the captured traffic. For example, `GET /api/user/1` and `GET /api/user/456` would be grouped under the same endpoint, as they share the same HTTP method (`GET`) and have a similar pattern (`/api/user/{id}`). However, `PUT /api/user/123`, `GET /api/user/456` and `GET /api/article/789` are considered distinct, since they either differ in their HTTP methods (`PUT` vs. `GET`) or have different endpoint patterns (`/api/user/{id}` vs. `/api/article/{id}`). We define a set of rules to group requests, ensuring that the identified endpoints are accurate and comprehensive.

Step-3 OpenAPI Specification Inference The captured traffic is analyzed to infer an OpenAPI specification. While generating specifications for `GET` and `DELETE` methods is relatively straightforward, as they do not require request payloads, `POST` and `PUT` methods present a greater challenge. These methods typically involve payloads, often in JSON format, that must adhere to a specific structure. If the inferred specification does not accurately define these payloads, the testing phase may produce invalid test cases that are rejected by the server. This hinders the exploration of execution paths and limits test coverage. To overcome this, our approach employs a combination of rules and heuristics to infer the structure of valid request payloads from the captured traffic.

Step-4 Test Execution The inferred OpenAPI specification serves as input for automated testing tools like Schemathesis and RESTler, which generate test cases based on the specification. To further enhance test coverage, our approach leverages the request

payloads captured during the traffic collection phase. By applying mutation-based techniques, we generate additional test payloads, increasing the diversity of test cases and improving the chances of uncovering edge cases and bugs.

Through this approach, we enable automated API testing for web applications even without publicly available OpenAPI specifications. Our method not only generates the necessary specifications from real-world traffic but also augments standard testing tools with richer test cases derived from real usage scenarios. Below we discuss the design and implementation detail of each component.

5.3.1 Capturing Web Traffic

The first step involves collecting API traffic generated during normal use of a web application. We developed a Chrome browser plugin that automatically records API requests sent from the browser to a remote server, along with their corresponding responses. Although web APIs can exchange information in various formats, such as JSON, HTTP form-data, or XML, our approach focuses specifically on JSON-formatted APIs. The focus on JSON offers several advantages: JSON is widely adopted, lightweight, and easier to parse, making it a natural choice for automated testing. While we concentrate on JSON-based APIs for the purposes of this research, the core principles of our proposed approach are generalizable and can be extended to support other data formats, with appropriate adjustments. By modifying the payload analysis and parsing methods, the same automated testing and fuzzing techniques could be applied to APIs using different data formats, maintaining the versatility and applicability of our workflow across a broader range of web APIs.

An example of API request and its corresponding response is given in Listing 2.1. In this step, we record the following information for each API call: (1) the endpoint URL; (2) the HTTP method used; (3) query parameters (if any); (4) the request payload (if any); (5) the HTTP response code; and (6) the response payload (if any).

5.3.2 Endpoint Identification

An API endpoint consists of two components: an HTTP method (commonly `GET`, `POST`, `PUT`, or `DELETE`) and an endpoint URL (e.g., `/api/user/123`). While the HTTP method

is straightforward and can be extracted from each captured API call, the main challenge is identifying a common pattern that groups multiple requests under the same API endpoint. We address this by following a four-step procedure to analyze each captured API call and infer the corresponding endpoints, given in Listing 5.1.

Listing 5.1: The algorithm to extract endpoint pattern.

```
1: function GetPattern(url)                          ▷ example input: PUT /api/v2/user/25?update=true
2:     method ← GetMethod(url)                       ▷ method: PUT
3:     url ← RemoveParameter(url)                    ▷ url: /api/v2/user/25
4:     segments ← Segment(url)                       ▷ segments: ["api", "v2", "user", "25"]
5:     for each segment in segments do
6:         result.add(CheckDynamic(segment))         ▷ checks if a segment is dynamic
7:     return result                                 ▷ result: [PUT, "api", "v2", "user", ID_NUMERIC]
```

First, we strip any query parameters from the API calls. While query parameters can influence the server's behavior (e.g., applying filters), we treat them as secondary to the endpoint itself. For instance, `GET /api/user` and `GET /api/user?role=admin` are considered to target the same API endpoint, as the query parameters do not change the fundamental resource being accessed.

Second, we divide the path into its constituent components. For example, `/api/user/12` is broken into three segments: `api`, `user`, and `12`. This segmentation provides a basic structure for determining endpoint patterns.

Then, we apply a heuristic to determine whether each segment is static or dynamic. Consider the API calls `/api/user/12` and `/api/user/45`; here, `api` and `user` are static segments, while `12` and `45` are dynamic. In an OpenAPI specification, this could be represented as `/api/user/{id}` where `{id}` is a parameter. From our analysis of captured traffic, dynamic segments are typically resource IDs on the server, which are commonly integers, hexadecimal hash values, or UUIDs. To differentiate between static and dynamic segments, we use a heuristic to check if a segment is numeric, follows a UUID format, or consists solely of hexadecimal characters with a minimum length of 16 characters. If any of these conditions are met, the segment is classified as dynamic; otherwise, it is static. This simple heuristic was proven effective in our experiments on popular content management systems.

Finally, we group captured API requests based on their inferred endpoints. Requests are placed in the same group if they share the same HTTP method and endpoint pattern. This grouping allows us to systematically organize API requests for further analysis.

5.3.3 Request Payload Structure Inference

As the collected traffic is already grouped by endpoints, we proceed by analysing each group of API requests to generate an internal representation for each endpoint. This internal representation can then be transformed into an OpenAPI specification, which allows us to systematise the knowledge captured in the traffic. The resulting specification can be used by automated testing tools such as Schemathesis or RESTler to facilitate systematic testing and exploration of the API.

First, we classify endpoints based on whether or not they include a request payload. While there are exceptions, typically, `GET` and `DELETE` requests do not contain a request payload, whereas `POST` and `PUT` requests often do. For requests without a payload, we only need to record the endpoint URL and identify any dynamic segments, as described in Section 5.3.2. For requests with a payload, further analysis is required to determine the structure of the payload.

The captured API request payloads (in JSON format) are presumed to be structurally and semantically valid, since they are generated by the web application's own interface, and any invalid payload would disrupt the application's functionality. We represent a JSON object using a tree structure. In this structure, each non-leaf node represents one of the two composite JSON types: object and array, while leaf nodes correspond to primitive types, such as string, number, boolean, and null. Let $\mathcal{R}$ be the set of collected request payloads sent to the same API endpoint. Our goal is to generate a grammar, $\mathcal{G}$, capable of representing all elements in $\mathcal{R}$.

We process each JSON object in a top-down manner, beginning with its outermost structure (typically of type *object*). We apply the following rule-set:

Below, we detail the process of generating a grammar $\mathcal{G}$ by analyzing all requests in $\mathcal{R}$.

Object	An *object* consists of zero or more *key-value* pairs, where each *key* is a string, with its *value* being of any type. In the tree representation, an *object* is represented as a parent node, each of its *key-value* pair is a child.
Array	An *array* consists of zero or more elements of the same type. Although JSON syntax allows mixed types within an array, all APIs from our evaluation use arrays to store single-typed data. In the tree representation, an *array* itself is depicted as a parent node, with each element represented as a child node.
String	We further check whether it follows a specific format. Common formats we can identify include email, formatted datetime, numerical strings, UUID, and hexadecimal.
Number	We check whether the value conforms to specific formats such as Unix timestamps or integers.
Boolean/Null	No additional rules.

Step 1: Initial Grammar Tree Construction

A request payload (a JSON object) $\mathcal{R}_0$ is randomly selected from the set of request payloads $\mathcal{R}$. The initial grammar tree $\mathcal{G}_0$ is created following the structure of $\mathcal{R}_0$, where each node in the grammar tree records the following properties of the corresponding node in $\mathcal{R}_0$:

- **Data Type**: The type of data for the current node (e.g., *object*, *array*, *string*, *number* and *boolean*).
- **Subtype**: For *string* and *number* types, a subtype is inferred (e.g., UUID, timestamp).
- **Object Keys**: If the node is an *object*, the *keys* present as child nodes are recorded.
- **Optionality**: All nodes are initially set to non-optional.

Step 2: Refining the Grammar Tree with Additional Payloads

Each of the remaining request payloads in $\mathcal{R} \setminus \{\mathcal{R}_0\}$ is processed to refine the generated grammar tree $\mathcal{G}$. The following scenarios are handled:

- **New Key in Object Node**: If an object node contains a *key* that does not exist in the corresponding node in $\mathcal{G}$, a new optional child node is created.
- **Missing Key in Object Node**: If an expected *key* is absent in an object node (but marked non-optional in $\mathcal{G}$), the corresponding key is updated to optional.
- **Subtype Mismatch for Strings or Numbers**: If a *string* or *number* node has a value not conforming to the subtype in $\mathcal{G}$, the stored subtype is updated.
- **Non-Null to Null**: If a node appears as non-null in $\mathcal{R}_0$ but null in $\mathcal{G}$, the grammar updates to non-null.
- **Null Conflicts**: If a node presents null in $\mathcal{R}_0$ but as non-null in $\mathcal{G}$, no update occurs.

- **Conflicting Non-Null Values**: If a node is non-null in $\mathcal{R}_0$ but conflicts with a different non-null value in $\mathcal{G}$, the grammar does not update. (No such conflicts arose in our experiments.)

Following the above procedure, we generate a grammar $\mathcal{G}$ that represents all elements in $\mathcal{R}$. This grammar is then transformed into an OpenAPI specification, which serves as input for automated testing tools to generate test cases. Additionally, we embed the captured request payloads as explicit examples within the generated OpenAPI specification. These examples act as the seed corpus for producing further test cases using mutation-based techniques, as we will discuss in Section 5.3.4.

While the OpenAPI specification supports documenting inter-request dependencies, such as when a resource created by one API request is consumed by another, inferring these dependencies remains an open problem. Tools like Schemathesis and RESTler utilize such dependencies documented in the specification for stateful fuzzing, but TrailBlazer focuses primarily on inferring the structure of request payloads. Identifying inter-request dependencies poses significant challenges. First, determining whether a dependency exists between two API endpoints is inherently complex. Second, even when dependencies are suspected, pinpointing which parameter in an API response might be consumed by another API is particularly difficult, especially for APIs that deviate from RESTful conventions. Addressing these challenges is beyond the scope of this work and remains an area for future research.

5.3.4 Mutations

While the generated grammar $\mathcal{G}$ can be used to produce new test payloads, we observed that these payloads are not always valid. This is due to two key factors. First, multiple data fields within a request payload may not be independent. For example, the length of an array may need to match the value of another field (e.g., an integer), or certain fields may be mutually exclusive. Second, a string field might need to adhere to a specific format, such as Markdown or HTML. In both cases, although the generated payloads are syntactically valid, they are semantically incorrect, which prevents them from passing server validation. Unfortunately, the widely adopted OpenAPI Specification (up to

version 3.0.x) lacks native support for handling such dependencies. Current state-of-the-art API testing tools, such as Schemathesis and RESTler, also face these limitations.

To address this, we incorporate mutation-based test generation. The collected traffic contains valid payloads, which we modify to increase the likelihood of producing test cases that pass validation. We apply three fundamental mutation strategies, shown in Figure 5.3: (1) *Recombining valid payloads* merges elements from different requests within the same endpoint, maintaining structural integrity while exploring new data combinations. (2) *Removing fields* selectively deletes fields to test how the API handles missing data, potentially uncovering hidden execution paths. (3) *Invalidating data types* changes field types to assess the robustness of API validation and error handling. These mutations allow us to generate more meaningful test cases while preserving key aspects of valid payloads.

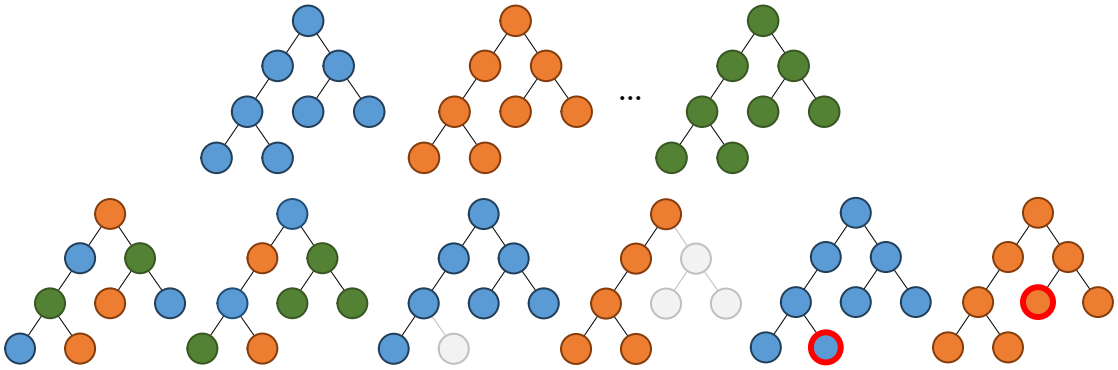

FIGURE 5.3: **Mutation strategies used to generate request payload**. The trees in the first row represent the original JSON payloads captured from API traffic. Each pair of trees in the second row illustrates examples of new JSON payloads generated using the three mutation strategies. The colour of each node indicates which original payload provided its value. Grayed-out nodes (in the second pair) indicate deleted nodes, while nodes with a red border (in the third pair) highlight an invalid data type.

5.3.5 Test Execution

Thus far, we have a list of API endpoints, as well as the corresponding inferred payload structure and seed payloads. This not only allows us to generate new test cases via mutation, but also enables us to produce an OpenAPI specification for generation-based fuzzing.

TRAILBLAZER leverages Schemathesis for test execution. Two checks in Schemathesis are particularly relevant: `not_a_server_error` and `negative_data_rejection`. The

former check identifies any 5xx HTTP response code indicating unhandled server errors, while the latter generates invalid payloads and checks if the server can properly handle them. Other builtin Schemathesis checks are disabled because our inferred OpenAPI specification is based on the observed traffic, which cannot be utilized to verify if a new request or response is valid. In addition, TRAILBLAZER incorporates mutation-based fuzzing by producing additional test cases, as discussed in Section 5.3.4.

5.4 Evaluation

We designed our experimental evaluation to answer the following research questions.

(RQ1) Quality of inferred specifications. Does the inferred OpenAPI specification adequately capture the full range of APIs offered by the web application?

(RQ2) Effectiveness and efficiency of Trailblazer. We aim to evaluate the effectiveness of TRAILBLAZER by measuring the code coverage it achieves and assessing its ability to uncover system bugs. Additionally, we investigate the human effort and computational time required to apply our workflow to a web application, providing insights into the scalability of our approach.

5.4.1 Target Selection

We selected four CMS platforms as our evaluation targets: *Cockpit* [172], *Directus* [173], *Ghost* [174] and *Strapi* [175]. Additionally, we included *RealWorld* [176] as a representative benchmark from recent related work [32]. We focused our experiments on CMS platforms as they are widely used in both personal and enterprise environments, making them representative of real-world modern web applications. Many of these systems expose a variety of API endpoints, handling complex interactions such as user authentication, content creation, and data management, which are ideal for evaluating the robustness of TRAILBLAZER. Additionally, many CMS platforms have transparent source code, allowing us to quantitatively analyze and compare test results across different implementations. We designed our selection to validate the effectiveness of our approach in diverse, real-world scenarios.

5.4.2 Evaluation Setup

We deployed the web-servers on Ubuntu 22.04 (through VMware ESXi hypervisor), allocated two *efficient*-cores of an Intel Core i7-1240P processor and 4GB of RAM. We ran TRAILBLAZER on another node within the same local network, equipped with an Intel Core i5-13490F, 128GB of RAM, running Windows 10 22H2.

For each evaluation target, we followed the following steps. (1) one of our author manually interacts with the web application for 20 minutes to collect API traffic, (2) we process the collected traffic following the steps discussed in Section 5.3.2 and Section 5.3.3, and (3) performed test execution as discussed in Section 5.3.4 and Section 5.3.5, and collected code coverage information. As an end-to-end fuzzing approach, TRAILBLAZER performs steps (2) and (3) in a fully automatic manner. In step (3), we configured Schemathesis to generate 100 test cases per API endpoint, and TRAILBLAZER generates 100 additional test cases through mutation. We measured code coverage using `c8` [177] for NodeJS web applications (*Directus*, *Ghost*, *RealWorld* and *Strapi*), while `Xdebug` [178] and `php-code-coverage` [179] were used for the PHP application (*Cockpit*). We also ran Schemathesis using web application's official specification and collected code coverage information.

5.4.3 Results

5.4.3.1 (RQ1) Quality of inferred specifications.

To evaluate the quality of our inferred OpenAPI specifications, we compare the endpoints in our inferred specification with the official specification to assess how many endpoints and operations our approach successfully identifies.

TABLE 5.1: **Number of endpoints in a) official API documentation b) inferred OpenAPI specification and c) overlaps.** API endpoints may be generated by interacting with the system. Numbers in this table reflect endpoints in our locally deployed systems, after 20 minutes of manual exploration.

System under Test	# Official	# Inferred	Overlap
Cockpit	11	20	0
Directus	129	75	28
Ghost	-	46	-
RealWorld	19	13	13
Strapi	37	65	4

Table 5.1 presents a comparison between the number of API endpoints in our inferred OpenAPI specifications and those in the official specifications. The results highlight both the strengths and limitations of our approach.

TRAILBLAZER successfully identifies a substantial number of API endpoints but does not capture all endpoints present in the official specifications. In the case of *Directus*, our approach inferred 75 endpoints, covering only 28 of the 129 endpoints officially documented. Similarly, in *Strapi*, we inferred 65 endpoints but found an overlap of only 4 with the 37 officially specified ones. These gaps arise for two main reasons. First, some functionalities in the web interface do not rely on API calls (e.g., generating hash values locally). Second, this result relies on manual exploration of the web application within a limited 20-minute window, meaning certain API-triggering actions may not have been fully exercised. With a longer exploration period or a better understanding of the application's features, more endpoints could likely be uncovered.

Interestingly, TRAILBLAZER discovered a considerable number of undocumented API endpoints—those not present in the official specification. For all tested systems, TRAILBLAZER identified API endpoints that do not present in their official documentaiton. Notably, in *Strapi* and *Cockpit*, our inferred specification contains nearly twice the number of endpoints in the official documentation, suggesting the presence of hidden or unofficially supported API functionalities.

In practice, API traffic could be collected in a distributed manner from multiple users interacting with the application over time, creating a more comprehensive set of observed endpoints. The numbers reported in our study represent a lower bound, reflecting what can be achieved with minimal human effort in a short timeframe. Despite its limitations, our approach demonstrates strong potential for uncovering both documented and undocumented API functionalities.

5.4.3.2 (RQ2) Effectiveness and efficiency.

Table 5.2 summarizes the code coverage achieved by TRAILBLAZER, the baseline, and *Schemathesis*. The baselines were measured by replaying all captured API requests, while the Schemathesis results are based on tests generated using the official OpenAPI

specification. All reported code coverage values are averaged over five runs for consistency.

TABLE 5.2: **Code coverage and number of unique bugs found in each system.** -: For both *Cockpit* and *Ghost*, there is no offical OpenAPI speficiation available, thus *Schemathesis* were unable to be applied to test these two systems. *: We were unable to measure branch coverage for *Cockpit* due to the limitation of the tool we used for code coverage calculation.

Target	Test Setup	Statement	Branch	Bugs	Execution time (s)
Cockpit	Baseline	441 (8.51%)	*	0	52
	Schemathesis	-	-	-	-
	TRAILBLAZER	577 (11.02%)	*	3	119
Directus	Baseline	16679 (23.52%)	1435 (59.34%)	0	9
	Schemathesis	19504 (27.51%)	1998 (61.93%)	14	589
	TRAILBLAZER	17733 (25.01%)	1645 (60.70%)	3	174
Ghost	Baseline	30232 (39.87%)	1617 (63.67%)	0	6
	Schemathesis	-	-	-	-
	TRAILBLAZER	33069 (43.61%)	2108 (64.03%)	0	86
RealWorld	Baseline	3805 (30.73%)	249 (64.50%)	0	13
	Schemathesis	1227 (13.80%)	101 (43.91%)	3	57
	TRAILBLAZER	4493 (34.75%)	335 (68.50%)	2	59
Strapi	Baseline	194147 (64.54%)	1212 (39.45%)	0	7
	Schemathesis	192777 (64.09%)	976 (35.47%)	2	393
	TRAILBLAZER	194554 (64.68%)	1310 (41.05%)	4	90

Across all evaluated targets, TRAILBLAZER consistently achieves higher code coverage than the baseline. This improvement is primarily because the baseline consists solely of replaying previously captured API requests, which contain only valid payloads observed during manual interaction. In contrast, TRAILBLAZER leverages the inferred OpenAPI specification to systematically generate both valid and invalid payloads. This allows it to exercise additional execution paths, including error-handling routines and edge cases that are not typically covered by normal API usage. As a result, TRAILBLAZER explores a broader range of server-side behaviors, leading to increased coverage.

When comparing TRAILBLAZER to Schemathesis, which relies on the official OpenAPI specification, the relative effectiveness varies by target. In *Strapi*, TRAILBLAZER achieves higher coverage because it uncovers undocumented API endpoints that are not included in the official specification. This highlights one of the key advantages of our approach—its ability to infer and test hidden API functionalities that would otherwise remain untested. However, in *Directus*, Schemathesis attains slightly higher coverage. This is

likely due to the fact that many API endpoints in *Directus* cannot be triggered through its web interface, limiting the effectiveness of our traffic-based inference approach. Consequently, our inferred specification lacks coverage for those endpoints, leading to a lower overall test effectiveness compared to Schemathesis.

We analyzed the code coverage results for *Directus* and *Strapi* to understand the contributions of each test setup. Figure 5.4 shows the breakdown of statement coverage across different approaches. TRAILBLAZER improves coverage beyond baseline by generating both valid and invalid payloads. Invalid payloads trigger error-handling logic rarely exercised in normal use due to front-end validation, while valid payloads explore API functionalities that the web UI does not fully expose or users did not interact with. As a result, TRAILBLAZER were able to cover statements that neither captured traffic nor Schemathesis reached. This result highlights the complementary strengths of both approaches: Schemathesis systematically exercises documented APIs, while TRAILBLAZER infers and explores APIs based on real traffic.

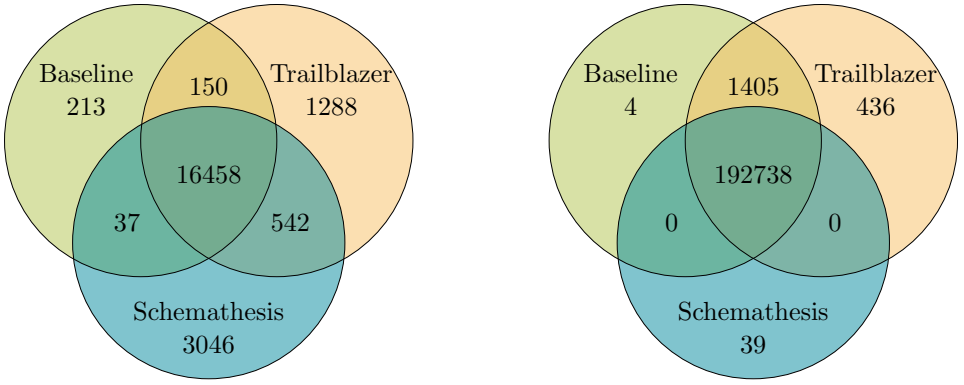

FIGURE 5.4: **Venn diagram of statement coverage across three test setups for *Directus* (left) and *Strapi* (right).** Each circle represents the code coverage achieved by a specific test setup. The overlap between circles indicates the shared coverage between setups.

In terms of bug discovery, TRAILBLAZER uncovered 12 new bugs across the tested systems [180–188]. For *Strapi*, it identified four unique bugs, surpassing the two found using the official specification. In Directus, TRAILBLAZER found three bugs, whereas Schemathesis uncovered more (14). Notably, all 11 bugs found using the official specification but missed by TRAILBLAZER were associated with endpoints that were not covered during our manual exploration of the web application. This result highlights a limitation of our approach——its reliance on observed traffic for API inference. However,

it also demonstrates that TRAILBLAZER can enhance bug discovery, even when an official specification exists, by testing undocumented or underutilized API functionalities that may not be exercised through conventional specification-based testing. We present selected bugs in Section 5.4.4.

Regarding efficiency, we consider both computational time and human effort. Computationally, our workflow includes inferring an OpenAPI specification and executing tests using Schemathesis. Test execution time varies based on the number of tested endpoints and configuration settings. As shown in Table 5.2, across both tested targets, TRAILBLAZER completes testing within three minutes, making it practical for real-world use. The inference of OpenAPI specifications is also highly efficient, taking only a few seconds for both targets, as it primarily involves retrieving API traffic from a local Postgres database and generating the specification.

In terms of human effort, we spent 20 minutes per web application to collect API traffic, though this process can be distributed among users naturally interacting with the system. Our code coverage results in Table 5.2 represent a lower bound on TRAILBLAZER's effectiveness, as extended exploration would likely improve coverage.

5.4.4 Selected New Bugs

When running TRAILBLAZER against locally deployed CMSes, we found a total of 12 new unique bugs [180–188]. We present the detail of three bugs in this section.

5.4.4.1 A Carefully Crafted API Payload can Crash Strapi CMS Server

In most cases, when a web server encounters an unhandled exception, it responds with a 500 error, signaling a server-side issue. Tools like RESTler and Schemathesis often rely on such signals to detect faults. Typically, the server application can handle these errors gracefully, ensuring that even if an unhandled exception occurs, it only affects the response for that specific request. However, through our approach, we uncovered an implementation flaw in Strapi CMS, that can crash the entire server application. Specifically, the `/content-type-builder/content-types` endpoint with `POST` method expects object types within the `attributes` object. Due to incomplete input validation, an invalid input that has `null` in the `attributes` object (minimal PoC:

`{"contentType": {"attributes": {"": null}}}`) would trigger an unhandled exception. Upon receiving such a request payload, the server not only triggered an unhandled exception but also caused the entire server application to crash. This bug affects both development and production environment of Strapi CMS.

5.4.4.2 Missing Validation

TRAILBLAZER uncovered two flaws in the `POST /collections` endpoint in Directus. This endpoint creates custom collection types. The payload should follow the format: "`{"fields": [...], "collection": "a", "schema": {}}`", where `collection` defines the name of the new collection type, and `"fields"` specifies the fields within the collection.

TRAILBLAZER found that supplying a non-string `"collection"` (e.g., `"collection":1`), triggers an unhandled server exception. We confirmed that the server attempted to call `.startsWith()` on the submitted `"collection"` parameter, without validating whether it was a string.

The second flaw was related to the `"fields"` parameter. A valid API request to `POST /collections` creates a database table on the server. Fields in the newly created table are specified by the `"fields"` parameter. However, by submitting an empty array (i.e., `"fields": []`), the server would produce an invalid SQL statement such as "`CREATE TABLE` `` `a` `` `();`". This SQL statement is invalid because there must be at least one field defined when creating a table. The error reported by the database was not properly handled by Directus, causing a 500 server error to be reported to the user.

5.5 Related Work

5.5.1 Vulnerability Detection in Web Applications

Web vulnerability detection aims to identify weaknesses in web applications that could be exploited by attackers, potentially compromising data integrity, confidentiality, or availability. The OWASP Top 10 [112] is a widely recognized list published by the Open Web Application Security Project, highlighting the most critical security risks to

web applications. Updated periodically, the OWASP Top 10 serves as a guideline for developers, security professionals, and organizations to better understand and mitigate common web vulnerabilities. Common vulnerabilities include SQL injection, cross-site scripting (XSS), and remote code execution, which can stem from insufficient input validation, poor session management, or insecure configuration. Various tools and research efforts have been directed at identifying and addressing these specific vulnerabilities, such as [104, 117, 118, 120, 122, 139, 189].

5.5.2 Web API Testing

5.5.2.1 Testing Web APIs using OpenAPI Specification

RESTler [84] represents a current state-of-the-art RESTful API fuzzing approaches. It uses server states, relying on response codes to identify server crashes on APIs used by cloud services. It infers dependencies among requested APIs to guide the generation of new test cases. [142] suggested an extension to RESTler to report the violation of four rules commonly applied to REST services, in addition to server crashes. [143] improved RESTler's test generation algorithm by representing the payloads in a tree structure on which structure-aware mutation operators can be applied. Pythia [144] augmented RESTler with coverage-guided feedback and it implemented a learning-based mutation strategy. Specifically, it utilized a statistical model to gain knowledge about frequent ordering of calling APIs from seed inputs. It used a regular grammar to encode an API request and perform mutation by adding noise to the request.

TRAILBLAZER builds on Schemathesis [100], discussed earlier, and RESTler, but with the advantage that contrary to those tools, we do not require a pre-existing API specification.

5.5.2.2 OpenAPI Specification Inference

While some modern web development frameworks can automatically generate OpenAPI Specifications (OAS) from source code, many web APIs either lack formal specifications altogether or rely on outdated and inaccurate ones. This absence of reliable specifications complicates efforts to understand, test, and interact with such APIs. To address this issue, researchers have proposed various approaches to infer OpenAPI specifications

through code analysis or network traffic inspection. This section reviews state-of-the-art techniques in OpenAPI specification inference, focusing on their capabilities and limitations.

APIDiscoverer [31] introduces a tool that generates OpenAPI specifications by analyzing pairs of HTTP requests and responses. However, this approach has several limitations: (1) its web-based interface requires manually submitting or triggering individual API calls, which makes it impractical for large-scale or automated analysis; (2) it only supports OpenAPI version 2.0, which is outdated given the widespread adoption of version 3.x since its release in 2017; and (3) in our experiments using real-world API traffic, the tool failed to generate valid OpenAPI specifications.

APICARV [32] is more closely related to TRAILBLAZER, leveraging automated UI tests to collect API traffic and subsequently infer an OpenAPI specification. While APICARV primarily emphasizes a graph-based technique for identifying path parameters, TRAILBLAZER goes further by also inferring the structure of request payloads. Moreover, TRAILBLAZER integrates both generation-based and mutation-based fuzzing strategies, which significantly enhance its capability to test and uncover issues in web APIs.

Respector [33] is a recent tool designed to generate OpenAPI specifications directly from source code by performing static and symbolic program analysis. While Respector demonstrates promising results, it has several limitations compared to TRAILBLAZER. First, Respector requires access to the source code of the web application, whereas TRAILBLAZER operates without such access, making it more versatile for scenarios where the source code is unavailable. Second, Respector has been evaluated exclusively on Java-based web APIs, leaving its effectiveness on APIs implemented in other programming languages uncertain. Finally, adapting Respector to work with web applications written in other programming languages would likely require significant additional engineering effort, whereas TRAILBLAZER is designed to be language-agnostic and does not impose such constraints.

As large language models (LLMs) have demonstrated remarkable effectiveness across multiple domains in recent years, there has also been growing interest in leveraging LLMs to assist in software testing. **RESTSpecIT** [34] is one such approach that uses LLMs to infer OpenAPI specifications and generate corresponding test cases. However, its evaluation is limited to APIs that support only the `GET` method, leaving the effectiveness

of the approach on state-changing requests (e.g., `POST`, `PUT`) uncertain. Moreover, the method relies heavily on the computational resources required to run large language models, making local deployment on commodity hardware infeasible. In contrast, our approach addresses both limitations: it supports APIs that require complex payloads and can be executed efficiently on standard desktop machines.

5.5.3 Black-box Grammar Inference

Black-box grammar inference techniques such as L* [190] and Arvada [191] are highly relevant to the problem space addressed by TRAILBLAZER, especially in terms of learning protocol structure or state machines without prior specifications. While TRAILBLAZER does not currently adopt these techniques directly, they represent promising directions for future work.

5.6 Discussion and Future Direction

There is often a mismatch between the endpoints observed in traffic and those documented in official API specifications. This suggests that specifications may not fully capture real-world API usage. Leveraging traffic to infer or enhance API specifications could help verify consistency, serving as a diagnostic tool for API maintenance and improving the reliability of API ecosystems.

TRAILBLAZER combines generation- and mutation-based testing to enhance API fuzzing, though each method has trade-offs. Generation-based testing increases input diversity, potentially uncovering edge cases, yet often produces invalid payloads that are rejected early by API validation. Mutation-based testing, by modifying valid payloads from captured traffic, is more likely to exercise meaningful application logic, but its effectiveness depends on the completeness of the observed requests. Our planned future evaluation seeks to investigate the respective contributions of the two strategies in web API testing. A promising direction for future work is dynamically adjusting the balance between the two strategies. For instance, using heuristics or leveraging server feedback to guide when to favor mutation-based tests and when to introduce generation-based tests.

Another promising avenue for future work is the use of automated bug seeding techniques, such as mutation analysis, to strengthen the evaluation of security testing tools. While the current evaluation relies on code coverage and the discovery of previously unknown bugs, introducing synthetic vulnerabilities could offer controlled benchmarks for measuring tool sensitivity and robustness. Implementing this, however, would require substantial extensions to the experimental framework, including the development of meaningful mutation strategies tailored to web APIs and the design of suitable oracles capable of detecting the seeded bugs. Nonetheless, such enhancements could provide deeper insight into the strengths and limitations of the proposed tools.

Chapter 6

Discussion and Future Directions

This chapter reflects on the contributions made by this thesis to the field of automated web application security testing, particularly through the development and evaluation of EDEFuzz and Trailblazer. It highlights key insights drawn from the research, acknowledges current limitations, and explores potential future directions to extend and deepen this line of work.

6.1 Summary of Contributions

At the heart of this thesis is the idea that effective security testing does not need to depend on high levels of manual effort, formal specifications, or intensive expert intervention. Instead, we propose a new strategy: leveraging information naturally produced during regular human interaction with web applications. This includes traffic captured during ordinary use or testing, behavioral relationships implicitly defined by user actions or page behavior. By reusing these artifacts, automated tools can operate with greater contextual relevance and lower manual overhead. In doing so, this approach allows even non-technical personnel to contribute indirectly to vulnerability discovery, simply through their natural engagement with the application.

This principle was instantiated through two concrete systems: EDEFuzz, which detects excessive data exposures by comparing related web responses, and Trailblazer, which bootstraps fuzzing from captured traffic to uncover server crashes in undocumented or poorly specified APIs. Both tools demonstrate that meaningful security testing can be

achieved with only lightweight configuration, surfacing subtle and security-critical issues that conventional approaches often overlook. While this thesis specifically focused on EDEs and server crashes, the broader idea—repurposing human-generated artifacts to inform and guide automated testing—is a general strategy that can be extended to a wide range of vulnerability classes, testing domains, and application types.

Figure 6.1 provides a qualitative comparison of common web application testing mechanisms along two conceptual axes: the amount of human effort required and their relative effectiveness in uncovering bugs and vulnerabilities. Traditional approaches like manual penetration testing and model-based testing are highly effective but demand significant expertise and labor. On the other end of the spectrum, methods such as random fuzzing typically require less human effort but struggle to uncover deeper, context-sensitive bugs; similarly, static analysis tools, while automated, often generate high volumes of false positives that still require substantial human triage.

Static analysis occupies a nuanced position in this space. While static analysis techniques require low human effort in principle, their high false positive rates have historically limited wide adoption in practice. Tools such as Infer [192, 193] and CodeQL [194] illustrate how industrial-grade static analysis must deliberately trade off some detection power (accepting higher false negatives) in order to dramatically reduce false positives and gain user acceptance. As high-false-positive static analysis is rarely used at scale [195, 196], we place static analysis in the "low human effort, medium effectiveness" region of the map to represent the best-in-practice, lower-false-positive variants—rather than earlier, more academic prototypes.

In contrast, the approach proposed in this thesis—leveraging artifacts generated through natural human interaction—aims to strike a favorable and relatively underexplored balance. It sits in a promising part of the design space, requiring only low to medium manual effort while maintaining relatively high effectiveness. More importantly, it suggests a new direction for the field: instead of attempting to eliminate human involvement altogether, or requiring costly manual testing, future systems can actively incorporate incidental human behavior as a low-effort, high-value artifact for automated testing.

This rethinking of how human interaction artifacts are used has significant implications. It opens the possibility for broader democratization of security testing, enabling contributions not only from security experts, but also from developers, testers, and even end

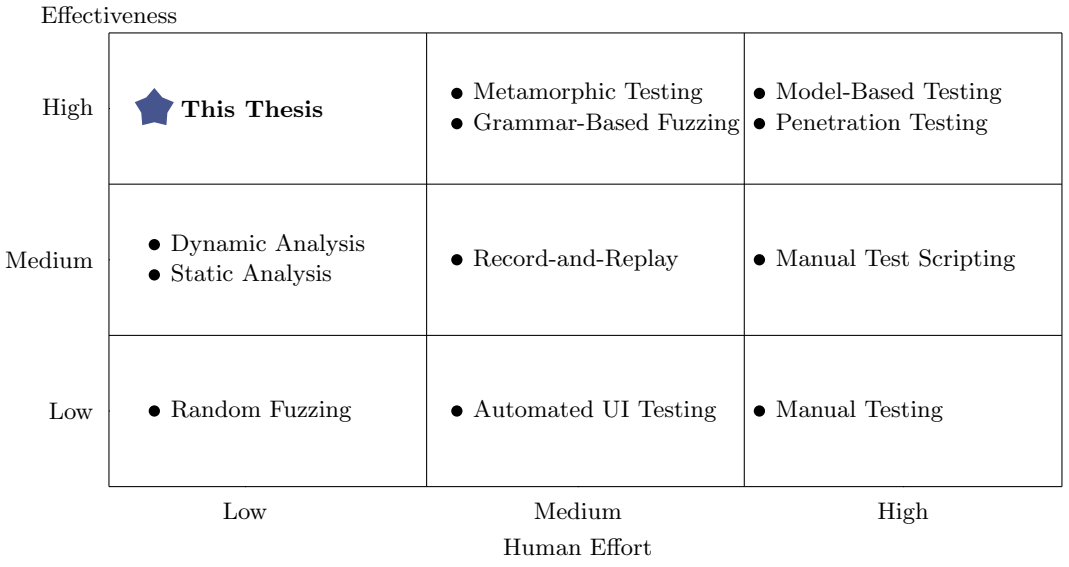

FIGURE 6.1: Qualitative comparison of web application testing mechanisms by human effort and effectiveness.

users through their natural usage patterns. It points toward a future where security validation is less a standalone activity and more an integrated, continuous by-product of application use and development. As software systems become increasingly complex, distributed, and dynamic, approaches that can blend human intuition with scalable automation will be critical. This thesis lays the foundation for such a direction, showing that with the right techniques, we can rethink the effort-effectiveness tradeoff and unlock new opportunities for practical, scalable, and inclusive security testing.

6.2 Research Impact

The research presented in this thesis has already begun to make a tangible impact on both academia and industry, as well as on the broader web security ecosystem. The following summarizes the key impacts of this work:

Firstly, the novel approach of using metamorphic testing to detect excessive data exposures in web APIs has been recognized through the filing of a patent application for EDEFuzz [17]. This demonstrates the uniqueness and practicality of the methodology developed in this research.

Secondly, the industrial relevance of the tools developed—EDEFuzz and Trailblazer—has been evidenced by letters of support from leading companies in the web security domain. These endorsements highlight the potential for these tools to address real-world security challenges and improve testing practices within industry.

Moreover, one of these companies has actively begun piloting the Trailblazer tool within their internal API testing workflows. This indicates a clear pathway for the research to influence practical security testing procedures and underscores its relevance in addressing contemporary web application bugs.

The academic impact of this work is further demonstrated by the recognition it has received in the form of a distinguished paper award at ICSE 2024 for the EDEFuzz work. This accolade reflects the high quality and significance of the research, as well as its contribution to advancing the field of automated web application security testing. In addition, it has garnered interest from other researchers who are eager to explore its capabilities and contribute to its development.

Additionally, the practical impact of this research is evident in the discovery of around 20 vulnerabilities using the proposed approach and tools. These findings were disclosed to corresponding vendors, with some already fixed, thereby contributing directly to improving the security of web applications and services. This work has thus made a concrete contribution to enhancing the overall web security environment.

In summary, this research has achieved significant impacts across multiple dimensions—from theoretical contributions recognized through academic awards, to practical tools adopted by industry, and vulnerabilities identified and remediated in real-world systems. These outcomes demonstrate the transformative potential of leveraging human-generated artifacts for automated security testing and underscore the importance of continuing to explore such approaches in future work.

6.3 Key Insights

Several important insights emerged from the development and evaluation of EDEFuzz and Trailblazer. First, the use of metamorphic testing in EDEFuzz illustrates how

relational properties between inputs and outputs can serve as implicit test oracles, enabling automated detection of vulnerabilities without requiring exact expected outputs. This technique helped address the oracle problem in scenarios involving complex, data-rich API responses. Second, our user study with EDEFuzz showed that usability and developer experience are critical factors in the adoption of security tools. Participants were more confident and efficient when using EDEFuzz, highlighting the importance of intuitive interfaces and clear testing workflows. Third, Trailblazer demonstrated the power of reusing real-world interaction traces to uncover bugs in endpoints that lack formal documentation or specification. By learning from these traces, the tool was able to generate meaningful test inputs for complex APIs. This further revealed that undocumented or internal-facing APIs form a growing and under-tested part of the modern web application attack surface. Trailblazer's success in uncovering bugs in such endpoints supports the argument for testing techniques that do not depend on complete specifications or explicit schemas.

6.4 Limitations

While this thesis demonstrates promising results through the development of EDEFuzz and Trailblazer, several limitations remain—both in the individual tools and in the broader landscape of automated web application testing.

First, although both tools aim to minimize manual effort, they are not fully autonomous. EDEFuzz relies on a human tester to compose a configuration file (even though it can be partially aided by a web browser extension), while Trailblazer requires access to user-generated or captured web traffic to bootstrap test generation. These dependencies reflect a broader tension in the field: many modern testing techniques either require labor-intensive configurations or rely on incidental artifacts (e.g., logs, UI flows, or telemetry) that may not always be available or complete. Achieving meaningful automation without sacrificing test relevance remains a difficult balance, particularly in systems with rich state or complex, domain-specific logic.

Second, there is an inherent challenge in testing highly stateful or dynamically evolving

web applications. In such systems, the behavior of an API endpoint may depend on complex sequences of prior interactions, authenticated sessions, or contextual data. Trailblazer currently has limited capability in reasoning about or generating such sequences. While state-aware fuzzing and model inference are active areas of research [139, 143, 197–199], the combinatorial explosion of possible states and transitions continues to pose a scalability bottleneck. Some progress may be made through hybrid approaches that combine static analysis or symbolic reasoning with dynamic fuzzing, but fully solving this challenge may require domain-specific models or human-in-the-loop assistance.

Third, the absence of formal specifications or ground-truth oracles limits the reliability of many testing techniques, including those proposed here. Metamorphic testing, as used in EDEFuzz, offers a promising way to overcome the oracle problem in specific cases, but it may not generalize easily to all classes of vulnerabilities. Similarly, crash detection in Trailblazer is based on observable errors like HTTP 500 responses, which may not capture deeper logic flaws or silent security issues. The field continues to struggle with the problem of test oracle design: how to automatically determine whether a system's behavior is correct, especially in open-ended or user-driven contexts [55].

Fourth, there is a broader issue of evaluation realism and generalizability. Although our tools were tested on widely-used or open-source applications, the diversity of web application architectures, frameworks, and deployment environments makes it difficult to claim universal applicability. Many academic tools, including ours, are evaluated in controlled or curated environments, raising concerns about how they perform in the wild. Industrial-scale adoption requires tools to be robust against edge cases, interoperable with heterogeneous stacks, and usable by non-experts—goals that remain aspirational in many current approaches.

Lastly, most existing tools, including those in this thesis, focus on vulnerability **detection** rather than **prevention** or **remediation**. While finding bugs is a critical first step, helping developers understand, prioritize, and fix these issues is equally important. Bridging this gap will require integration with developer workflows, clearer reporting mechanisms, and possibly advances in root cause analysis or automated repair [200–202].

Some of these challenges—such as improving state modeling or reducing manual setup—are likely to see incremental progress through better tooling, data collection, or integration with development environments. Others, like oracle inference and evaluation scalability, represent more fundamental barriers that will require deeper shifts in methodology or the incorporation of new technologies such as machine learning or formal methods. Acknowledging these limitations is essential not only for framing the contributions of this thesis but also for identifying future research directions that can move the field forward.

6.5 Future Directions

This work opens several promising avenues for future research, both in extending the capabilities of the proposed tools and in addressing broader challenges in automated web application testing.

A natural next step is to explore deeper integration between human input and automated fuzzing. While traditional *human-in-the-loop* fuzzing incorporates developer feedback during test generation [203–206]—such as confirming the relevance of findings or directing exploration—this thesis also introduces and motivates a complementary paradigm: *human-before-the-loop* fuzzing. In this model, information generated during routine human interaction with web applications (e.g., user-driven traffic, page behaviors, or UI flows) is captured and repurposed before any automated testing begins. This approach has the advantage of requiring little to no active involvement from technical users, yet can provide rich contextual artifacts to guide fuzzing. Future work could explore how these two models might be combined: allowing systems to first bootstrap from passively collected artifacts and then adapt based on interactive feedback during testing.

Another important direction is enhancing support for stateful and complex APIs. Many modern web services exhibit behaviors that depend on prior request sequences, user sessions, or data-dependent transitions. Addressing these challenges may require the use of state inference techniques, such as learning finite-state models from traffic, or applying symbolic reasoning to extract dependencies across calls [139, 143, 197–199]. Incorporating such capabilities into Trailblazer would expand its applicability to

richer API ecosystems, including GraphQL or event-driven architectures. Similarly, extending metamorphic testing beyond excessive data exposure–such as to verify correct access control enforcement or idempotent behavior—would increase the generality of EDEFuzz's core methodology.

The issue of test oracle design also remains an open challenge [55]. While this thesis uses metamorphic relations and observable server crashes as proxies for correctness, many security-relevant properties remain implicit and context-dependent. Future work could investigate ways to infer likely oracles using machine learning, specification mining, or program analysis. For instance, expected field-level behavior might be learned from training examples or inferred from UI logic, providing additional validation power without requiring hard-coded assertions.

At a systems level, improving the realism and scalability of evaluations is another key direction. Integrating the proposed tools into CI/CD pipelines, DevSecOps workflows, or security testbeds would enable longitudinal studies on their practical effectiveness and usability. This would also help assess how the tools perform under diverse workloads and evolving application states. Additionally, research should continue to explore techniques for automated triage, root cause analysis, and remediation assistance—capabilities that are currently lacking but essential for real-world adoption.

Finally, the broader field of web application testing must continue to grapple with foundational tensions: between automation and context-awareness, scalability and specificity, ease of use and analytical depth. The approaches presented in this thesis suggest that it is possible to make progress along multiple axes simultaneously—by reusing natural human interaction data, by defining weaker yet useful correctness conditions, and by designing tools with both technical and usability goals in mind. These directions hold potential for building the next generation of security testing tools that are not only technically sophisticated, but also aligned with how modern web applications are built, tested, and used.

6.6 Concluding Remarks

In summary, this thesis demonstrates that practical, automated testing tools can be both effective and low-effort when they are designed to reuse artifacts produced through

natural human interaction with web applications. By applying this philosophy to the detection of excessive data exposures and undocumented endpoint crashes, we show that critical classes of vulnerabilities can be detected with minimal configuration and no formal specifications. The insights and techniques developed here contribute to a growing body of work on practical, intelligent, and scalable web application security testing.

Bibliography

[1] Postman, Inc. 2024 state of the api report: Api use and discovery. https://www.postman.com/state-of-api/2024/api-use-and-discovery/, 2024. Accessed: 2024-10-20.

[2] Giuseppe A Di Lucca and Anna Rita Fasolino. Testing web-based applications: The state of the art and future trends. *Information and Software Technology*, 48 (12):1172–1186, 2006.

[3] Ali Mesbah and Arie Van Deursen. Invariant-based automatic testing of ajax user interfaces. In *2009 IEEE 31st International Conference on Software Engineering*, pages 210–220. IEEE, 2009.

[4] OWASP. Owasp top 10: 2021. https://owasp.org/Top10/, 2021. Accessed: 2025-02-12.

[5] Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd international conference on software engineering*, pages 1–10, 2011.

[6] Dirk Beyer and Thomas Lemberger. Software verification: Testing vs. model checking: A comparative evaluation of the state of the art. In *Hardware and Software: Verification and Testing: 13th International Haifa Verification Conference, HVC 2017, Haifa, Israel, November 13-15, 2017, Proceedings 13*, pages 99–114. Springer, 2017.

[7] I Luk Kim, Weihang Wang, Yonghwi Kwon, and Xiangyu Zhang. Bftdetector: Automatic detection of business flow tampering for digital content service. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 448–459. IEEE, 2023.

[8] Serdar Doğan, Aysu Betin-Can, and Vahid Garousi. Web application testing: A systematic literature review. *Journal of Systems and Software*, 91:174–201, 2014.

[9] Saru Dhir and Deepak Kumar. Automation software testing on web-based application. In *Software Engineering: Proceedings of CSI 2015*, pages 691–698. Springer, 2019.

[10] Filippo Ricca and Andrea Stocco. Web test automation: Insights from the grey literature. In *SOFSEM 2021: Theory and Practice of Computer Science: 47th International Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM 2021, Bolzano-Bozen, Italy, January 25–29, 2021, Proceedings 47*, pages 472–485. Springer, 2021.

[11] Api3:2019 excessive data exposure. https://github.com/OWASP/API-Security/blob/master/2019/en/src/0xa3-excessive-data-exposure.md, 2019. Accessed: 2021-12-31.

[12] Ganesan Deepa, P Santhi Thilagam, Amit Praseed, and Alwyn R Pais. Detlogic: A black-box approach for detecting logic vulnerabilities in web applications. *Journal of Network and Computer Applications*, 109:89–109, 2018.

[13] G Deepa and P Santhi Thilagam. Securing web applications from injection and logic vulnerabilities: Approaches and challenges. *Information and Software Technology*, 74:160–180, 2016.

[14] J. Simpson. Most apis suffer from specification drift. https://nordicapis.com/most-apis-suffer-from-specification-drift/, September 2024. Accessed: 2025-01-22.

[15] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, TH Tse, and Zhi Quan Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys (CSUR)*, 51(1):1–27, 2018.

[16] Sergio Segura, Gordon Fraser, Ana B Sanchez, and Antonio Ruiz-Cortés. A survey on metamorphic testing. *IEEE Transactions on software engineering*, 42(9):805–824, 2016.

[17] Lianglu Pan, Shaanan Cohney, Toby Murray, and Van-Thuan Pham. System, method and device for detecting excessive data exposures. https://patentscope.wipo.int/search/en/detail.jsf?docId=WO2024086877, 2024.

[18] Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 47(9):1980–1997, 2019.

[19] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for deep bugs with grammars. In *NDSS*, 2019.

[20] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. Fuzzilli: Fuzzing for javascript jit compiler vulnerabilities. In *NDSS*, 2023.

[21] Sebastian Balsam and Deepti Mishra. Web application testing—challenges and opportunities. *Journal of Systems and Software*, page 112186, 2024.

[22] 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. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4765–4782, 2024.

[23] Antoine Chevrot, Alexandre Vernotte, Jean-Rémy Falleri, Xavier Blanc, and Bruno Legeard. Are autonomous web agents good testers? *arXiv preprint arXiv:2504.01495*, 2025.

[24] Yan Zheng, Yi Liu, Xiaofei Xie, Yepang Liu, Lei Ma, Jianye Hao, and Yang Liu. Automatic web testing using curiosity-driven reinforcement learning. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 423–435. IEEE, 2021.

[25] Rahul Krishna Yandrapally and Ali Mesbah. Fragment-based test generation for web apps. *IEEE Transactions on Software Engineering*, 49(3):1086–1101, 2022.

[26] Roy T. Fielding and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. https://datatracker.ietf.org/doc/html/rfc7230, 2014. RFC 7230.

[27] OpenAPI Initiative. Openapi specification. https://swagger.io/resources/open-api/, 2020. Accessed: 2023-07-06.

[28] Martin P Robillard and Robert DeLine. A field study of api learning obstacles. *Empirical Software Engineering*, 16:703–732, 2011.

[29] Mohamed Aymen Saied, Houari Sahraoui, and Bruno Dufour. An observational study on api usage constraints and their documentation. In *2015 IEEE 22nd International conference on software analysis, evolution, and reengineering (SANER)*, pages 33–42. IEEE, 2015.

[30] Christoph Treude and Martin P Robillard. Augmenting api documentation with insights from stack overflow. In *Proceedings of the 38th International Conference on Software Engineering*, pages 392–403, 2016.

[31] Hamza Ed-Douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. Example-driven web api specification discovery. In *European Conference on Modelling Foundations and Applications*, pages 267–284. Springer, 2017.

[32] Rahulkrishna Yandrapally, Saurabh Sinha, Rachel Tzoref-Brill, and Ali Mesbah. Carving ui tests to generate api tests and api specification. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1971–1982. IEEE, 2023.

[33] Ruikai Huang, Manish Motwani, Idel Martinez, and Alessandro Orso. Generating rest api specifications through static analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[34] Alix Decrop, Gilles Perrouin, Mike Papadakis, Xavier Devroey, and Pierre-Yves Schobbens. You can rest now: Automated specification inference and black-box testing of restful apis with large language models. *arXiv preprint arXiv:2402.05102*, 2024.

[35] SmartBear Software. Swaggerhub explore. https://swagger.io/api-hub/explore/, 2024. Accessed: 2025-02-05.

[36] Appmap. https://appmap.io/. Accessed: 2025-04-13.

[37] Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990. doi: 10.1109/IEEESTD.1990.101064.

[38] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[39] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[40] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering (FOSE'07)*, pages 85–103. IEEE, 2007.

[41] Michael E Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 38(2.3):258–287, 1999.

[42] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 712–721. IEEE, 2013.

[43] Darko Stefanović, Danilo Nikolić, Sara Havzi, Teodora Lolić, and Dušanka Dakić. Identification of strategies over tools for static code analysis. In *IOP Conference Series: Materials Science and Engineering*, volume 1163, page 012012. IOP Publishing, 2021.

[44] Wachiraphan Charoenwet, Patanamon Thongtanunam, Van-Thuan Pham, and Christoph Treude. An empirical study of static analysis tools for secure code review. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 691–703, 2024.

[45] Darko Stefanović, Danilo Nikolić, Dušanka Dakić, Ivana Spasojević, and Sonja Ristić. Static code analysis tools: A systematic literature review. In *Ann. DAAAM Proc. Int. DAAAM Symp*, volume 31, pages 565–573, 2020.

[46] Nathaniel Ayewah, William Pugh, J David Morgenthaler, John Penix, and YuQian Zhou. Using findbugs on production software. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 805–806, 2007.

[47] Kent Beck. *Test driven development: By example*. Addison-Wesley Professional, 2022.

[48] Paul M Duvall, Steve Matyas, and Andrew Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[49] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. Fuzzing: Challenges and reflections. *IEEE Software*, 38(3):79–86, 2020.

[50] Marcel Böhme, Maria Christakis, Rohan Padhye, Kostya Serebryany, Andreas Zeller, and Hasan Ferit Eniser. Software bug detection: Challenges and synergies (dagstuhl seminar 23131). *Dagstuhl Reports*, 13(3):92–105, 2023.

[51] Elaine J Weyuker. On testing non-testable programs. *The Computer Journal*, 25 (4):465–470, 1982.

[52] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[53] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. Metamorphic testing: a new approach for generating next test cases. *arXiv preprint arXiv:2002.12543*, 2020.

[54] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st international conference on Software engineering*, pages 213–224, 1999.

[55] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525, 2014.

[56] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1:1–13, 2018.

[57] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. Nyx-net: network fuzzing with incremental snapshots. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 166–180, 2022.

[58] Penghui Li and Mingxue Zhang. Fuzzcache: Optimizing web application fuzzing through software-based data cache. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 511–524, 2024.

[59] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Aflnet: a greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 460–465. IEEE, 2020.

[60] Libo Chen, Quanpu Cai, Zhenbang Ma, Yanhao Wang, Hong Hu, Minghang Shen, Yue Liu, Shanqing Guo, Haixin Duan, Kaida Jiang, et al. Sfuzz: Slice-based fuzzing for real-time operating systems. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 485–498, 2022.

[61] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[62] American fuzzy lop. https://lcamtuf.coredump.cx/afl/. Accessed: 2023-10-03.

[63] Roberto Natella. Stateafl: Greybox fuzzing for stateful network servers. *Empirical Software Engineering*, 27(7):191, 2022.

[64] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. {AFL++}: Combining incremental steps of fuzzing research. In *14th USENIX workshop on offensive technologies (WOOT 20)*, 2020.

[65] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 475–485, 2018.

[66] Yuheng Shen, Yiru Xu, Hao Sun, Jianzhong Liu, Zichen Xu, Aiguo Cui, Heyuan Shi, and Yu Jiang. Tardis: Coverage-guided embedded operating system fuzzing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(11):4563–4574, 2022.

[67] Su Yong Kim, Sangho Lee, Insu Yun, Wen Xu, Byoungyoung Lee, Youngtae Yun, and Taesoo Kim. {CAB-Fuzz}: Practical concolic testing techniques for {COTS} operating systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 689–701, 2017.

[68] NCC Group. Triforceafl: Afl/qemu fuzzing with full-system emulation, 2017. URL https://github.com/nccgroup/TriforceAFL. Accessed: 2025-04-20.

[69] Joobeom Yun, Fayozbek Rustamov, Juhwan Kim, and Youngjoo Shin. Fuzzing of embedded systems: A survey. *ACM Computing Surveys*, 55(7):1–33, 2022.

[70] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *NDSS*, volume 14, pages 1–16, 2014.

[71] Xiyue Gao, Zhuang Liu, Jiangtao Cui, Hui Li, Hui Zhang, Kewei Wei, and Kankan Zhao. A comprehensive survey on database management system fuzzing: Techniques, taxonomy and experimental comparison. *arXiv preprint arXiv:2311.06728*, 2023.

[72] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. Industry practice of coverage-guided enterprise-level dbms fuzzing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 328–337. IEEE, 2021.

[73] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. Griffin: Grammar-free dbms fuzzing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.

[74] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1044–1051, 2019.

[75] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 283–294, 2011.

[76] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F Donaldson. Many-core compiler fuzzing. *ACM SIGPLAN Notices*, 50(6):65–76, 2015.

[77] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. {DynSQL}: Stateful fuzzing for database management systems with complex and valid {SQL} query generation. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4949–4965, 2023.

[78] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 579–594. IEEE, 2017.

[79] Michael Eddington. Peach fuzzer: Smart fuzzing framework. https://www.peach.tech/products/peach-fuzzer, 2004. Accessed: 2025-01-07.

[80] Ivan Fratric. Domato: Dom fuzzer. https://github.com/googleprojectzero/domato, 2017. Accessed: 2025-04-20.

[81] Google. Atheris: A coverage-guided, native python fuzzer. https://github.com/google/atheris, 2020. Accessed: 2025-04-20.

[82] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019.

[83] Cristian Daniele, Seyed Behnam Andarzian, and Erik Poll. Fuzzers for stateful systems: Survey and research directions. *ACM Computing Surveys*, 56(9):1–23, 2024.

[84] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Restler: Stateful rest api fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 748–758. IEEE, 2019.

[85] Jiho Shin, Sepehr Hashtroudi, Hadi Hemmati, and Song Wang. Domain adaptation for deep unit test case generation. *arXiv e-prints*, pages arXiv–2308, 2023.

[86] Congying Xu, Valerio Terragni, Hengcheng Zhu, Jiarong Wu, and Shing-Chi Cheung. Mr-scout: Automated synthesis of metamorphic relations from existing test cases. *ACM Transactions on Software Engineering and Methodology*, 33(6):1–28, 2024.

[87] Teemu Kanstrén and Olli-Pekka Puolitaival. Using built-in domain-specific modeling support to guide model-based test generation. *Model-Driven Engineering of Information Systems: Principles, Techniques, and Practice*, pages 295–319, 2012.

[88] Xiaoyuan Xie, Joshua WK Ho, Christian Murphy, Gail Kaiser, Baowen Xu, and Tsong Yueh Chen. Testing and validating machine learning classifiers by metamorphic testing. *Journal of Systems and Software*, 84(4):544–558, 2011.

[89] Shin Nakajima and Hai Ngoc Bui. Dataset coverage for testing machine learning computer programs. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 297–304. IEEE, 2016.

[90] Zeyang Zhuang, Penghui Li, Pingchuan Ma, Wei Meng, and Shuai Wang. Testing graph database systems via graph-aware metamorphic relations. *Proceedings of the VLDB Endowment*, 17(4):836–848, 2023.

[91] Mikael Lindvall, Dharmalingam Ganesan, Ragnar Árdal, and Robert E Wiegand. Metamorphic model-based testing applied on nasa dat–an experience report. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 129–138. IEEE, 2015.

[92] Manuel Rigger and Zhendong Su. Testing database engines via pivoted query synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 667–682, 2020.

[93] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices*, 49(6):216–226, 2014.

[94] Liqun Sun and Zhi Quan Zhou. Metamorphic testing for machine translations: Mt4mt. In *2018 25th Australasian Software Engineering Conference (ASWEC)*, pages 96–100. IEEE, 2018.

[95] Daniel Pesu, Zhi Quan Zhou, Jingfeng Zhen, and Dave Towey. A monte carlo method for metamorphic testing of machine translation services. In *Proceedings of the 3rd International Workshop on Metamorphic Testing*, pages 38–45, 2018.

[96] Pinjia He, Clara Meister, and Zhendong Su. Testing machine translation via referential transparency. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 410–422. IEEE, 2021.

[97] Phu X Mai, Fabrizio Pastore, Arda Goknil, and Lionel Briand. Metamorphic security testing for web systems. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 186–197. IEEE, 2020.

[98] Wing Kwong Chan, Shing Chi Cheung, and Karl RPH Leung. A metamorphic testing approach for online testing of service-oriented software applications. *International Journal of Web Services Research (IJWSR)*, 4(2):61–81, 2007.

[99] Sergio Segura, José A Parejo, Javier Troya, and Antonio Ruiz-Cortés. Metamorphic testing of restful web apis. *IEEE Transactions on Software Engineering*, 44 (11):1083–1099, 2017.

[100] Zac Hatfield-Dodds and Dmitry Dygalo. Deriving semantics-aware fuzzers from web api schemas. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 345–346, 2022.

[101] Andrea Arcuri. Restful api automated test case generation with evomaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(1):1–37, 2019.

[102] Stefan Karlsson, Adnan Čaušević, and Daniel Sundmark. Quickrest: Property-based test generation of openapi-described restful apis. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 131–141. IEEE, 2020.

[103] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. Resttestgen: automated black-box testing of restful apis. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 142–152. IEEE, 2020.

[104] Yunhui Zheng and Xiangyu Zhang. Path sensitive static analysis of web applications for remote code execution vulnerability detection. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 652–661. IEEE, 2013.

[105] 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.

[106] 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.

[107] 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.

[108] Dennis Appelt, Cu Duy Nguyen, Lionel C Briand, and Nadia Alshahwan. Automated testing for sql injection vulnerabilities: an input mutation approach. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 259–269, 2014.

[109] Erik Trickel, Fabio Pagani, Chang Zhu, Lukas Dresel, Giovanni Vigna, Christopher Kruegel, Ruoyu Wang, Tiffany Bao, Yan Shoshitaishvili, and Adam Doupé. Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect sql and command injection vulnerabilities. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 116–133. IEEE Computer Society, 2022.

[110] Sangeeta Nagpure and Sonal Kurkure. Vulnerability assessment and penetration testing of web application. In *2017 International Conference on Computing, Communication, Control and Automation (ICCUBEA)*, pages 1–6. IEEE, 2017.

[111] Esra Abdullatif Altulaihan, Abrar Alismail, and Mounir Frikha. A survey on web application penetration testing. *Electronics*, 12(5):1229, 2023.

[112] OWASP. Owasp top 10: 2021. https://owasp.org/www-project-top-ten/, 2021. Accessed: 2024-09-25.

[113] OWASP API Top 10: 2019. https://owasp.org/www-project-api-security/, 2019. Accessed: 2022-03-18.

[114] Murugiah Souppaya, Karen Scarfone, and Donna Dodson. Secure software development framework (ssdf) version 1.1: Recommendations for mitigating the risk of software vulnerabilities. Technical Report NIST Special Publication 800-218, National Institute of Standards and Technology, Gaithersburg, MD, February 2022. Accessed: 2024-07-06.

[115] Orpheas van Rooij, Marcos Antonios Charalambous, Demetris Kaizer, Michalis Papaevripides, and Elias Athanasopoulos. webfuzz: Grey-box fuzzing for web applications. In *European Symposium on Research in Computer Security*, pages 152–172. Springer, 2021.

[116] Burp suite. https://portswigger.net/burp. Accessed: 2021-12-31.

[117] William GJ Halfond, Jeremy Viegas, Alessandro Orso, et al. A classification of sql injection attacks and countermeasures. In *ISSSE*, 2006.

[118] Zainab S Alwan and Manal F Younis. Detection and prevention of sql injection attack: a survey. *International Journal of Computer Science and Mobile Computing*, 6(8):5–17, 2017.

[119] Upasana Sarmah, DK Bhattacharyya, and Jugal K Kalita. A survey of detection methods for xss attacks. *Journal of Network and Computer Applications*, 118: 113–143, 2018.

[120] SQLMap team. Sqlmap: Automatic sql injection and database takeover tool. https://sqlmap.org/, 2024. Accessed: 2024-09-25.

[121] Stephen W Boyd and Angelos D Keromytis. Sqlrand: Preventing sql injection attacks. In *International conference on applied cryptography and network security*, pages 292–302. Springer, 2004.

[122] Germán E Rodríguez, Jenny G Torres, Pamela Flores, and Diego E Benavides. Cross-site scripting (xss) attacks and mitigation: A survey. *Computer Networks*, 166:106960, 2020.

[123] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *NDSS*, volume 2007, page 12, 2007.

[124] Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *Proceedings of the 30th international conference on Software engineering*, pages 171–180, 2008.

[125] Michael C Martin and Monica S Lam. Automatic generation of xss and sql injection attacks with goal-directed model checking. In *USENIX Security symposium*, pages 31–44, 2008.

[126] Fabien Duchene, Sanjay Rawat, Jean-Luc Richier, and Roland Groz. Kameleonfuzz: evolutionary fuzzing for black-box xss detection. In *Proceedings of the 4th ACM conference on Data and application security and privacy*, pages 37–48, 2014.

[127] Ran Wang, Guangquan Xu, Xianjiao Zeng, Xiaohong Li, and Zhiyong Feng. Tt-xss: A novel taint tracking based dynamic detection framework for dom cross-site scripting. *Journal of Parallel and Distributed Computing*, 118:100–106, 2018.

[128] S Biswas, M Sohel, MM Sajal, T Afrin, T Bhuiyan, and MM Hassan. A study on remote code execution vulnerability in web applications. In *International conference on cyber security and computer science (ICONCS 2018)*, pages 50–57, 2018.

[129] Md Maruf Hassan, Umam Mustain, Sabira Khatun, Mohamad Shaiful Abdul Karim, Nazia Nishat, and Mostafijur Rahman. Quantitative assessment of remote code execution vulnerability in web apps. In *InECCE2019: Proceedings of the 5th International Conference on Electrical, Control & Computer Engineering, Kuantan, Pahang, Malaysia, 29th July 2019*, pages 633–642. Springer, 2020.

[130] Viktoria Felmetsger, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Toward automated detection of logic vulnerabilities in web applications. In *19th USENIX Security Symposium (USENIX Security 10)*, 2010.

[131] Daniel Huluka and Oliver Popov. Root cause analysis of session management and broken authentication vulnerabilities. In *World Congress on Internet Security (WorldCIS-2012)*, pages 82–86. IEEE, 2012.

[132] Michael Dalton, Christos Kozyrakis, and Nickolai Zeldovich. Nemesis: Preventing authentication & [and] access control vulnerabilities in web applications. 2009.

[133] Yusuke Takamatsu, Yuji Kosuga, and Kenji Kono. Automated detection of session management vulnerabilities in web applications. In *2012 Tenth Annual International Conference on Privacy, Security and Trust*, pages 112–119. IEEE, 2012.

[134] Md Maruf Hassan, Shamima Sultana Nipa, Marjan Akter, Rafita Haque, Fabiha Nawar Deepa, Mostafijur Rahman, Md Asif Siddiqui, Md Hasan Sharif, et al. Broken authentication and session management vulnerability: a case study of web application. *Int. J. Simul. Syst. Sci. Technol*, 19(2):1–11, 2018.

[135] Bahruz Jabiyev, Omid Mirzaei, Amin Kharraz, and Engin Kirda. Preventing server-side request forgery attacks. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, pages 1626–1635, 2021.

[136] Khadejah Al-talak and Onytra Abbass. Detecting server-side request forgery (ssrf) attack by using deep learning techniques. *International Journal of Advanced Computer Science and Applications*, 12(12):12, 2021.

[137] Yuchen Ji, Ting Dai, Yutian Tang, and Jingzhu He. Poster: Whether we are good enough to detect server-side request forgeries in php-native applications? In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 4928–4930, 2024.

[138] Yuchen Ji, Ting Dai, Zhichao Zhou, Yutian Tang, and Jingzhu He. Artemis: Toward accurate detection of server-side request forgeries through llm-assisted interprocedural path-sensitive taint analysis. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA1):1349–1377, 2025.

[139] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Enemy of the state: A {State-Aware}{Black-Box} web vulnerability scanner. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 523–538, 2012.

[140] Jose Fonseca and Marco Vieira. Mapping software faults with web security vulnerabilities. In *2008 IEEE international conference on dependable systems and networks With FTCS and DCC (DSN)*, pages 257–266. IEEE, 2008.

[141] OWASP Foundation. Owasp zed attack proxy (zap). https://www.zaproxy.org/, 2010. Accessed: 2024-09-25.

[142] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Checking security properties of cloud service rest apis. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 387–397. IEEE, 2020.

[143] Patrice Godefroid, Bo-Yuan Huang, and Marina Polishchuk. Intelligent rest api data fuzzing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 725–736, 2020.

[144] Vaggelis Atlidakis, Roxana Geambasu, Patrice Godefroid, Marina Polishchuk, and Baishakhi Ray. Pythia: grammar-based fuzzing of rest apis with coverage-guided feedback and learning-based mutations. *arXiv preprint arXiv:2005.11498*, 2020.

[145] Yi Liu, Yuekang Li, Gelei Deng, Yang Liu, Ruiyuan Wan, Runchao Wu, Dandan Ji, Shiheng Xu, and Minli Bao. Morest: Model-based restful api testing with execution feedback. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1406–1417, 2022.

[146] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. Restest: automated black-box testing of restful web apis. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 682–685, 2021.

[147] Myeongsoo Kim, Qi Xin, Saurabh Sinha, and Alessandro Orso. Automated test generation for rest apis: No time to rest yet. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 289–301, 2022.

[148] Alberto Martin-Lopez, Andrea Arcuri, Sergio Segura, and Antonio Ruiz-Cortés. Black-box and white-box test case generation for restful apis: Enemies or allies? In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, pages 231–241. IEEE, 2021.

[149] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. Online testing of restful apis: Promises and challenges. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 408–420, 2022.

[150] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. Restest: Black-box constraint-based testing of restful web apis. In *Service-Oriented Computing: 18th International Conference, ICSOC 2020, Dubai, United Arab Emirates, December 14–17, 2020, Proceedings 18*, pages 459–475. Springer, 2020.

[151] Microsoft announces new Project OneFuzz framework. https://www.microsoft.com/security/blog/2020/09/15/microsoft-onefuzz-framework-open-source-developer-tool-fix-bugs/, 2020. Accessed: 2022-05-06.

[152] Using burp to test for sensitive data exposure issues. https://portswigger.net/support/using-burp-to-test-for-sensitive-data-exposure-issues, 2022. Accessed: 2023-03-21.

[153] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: Accurate {Record-and-Replay} for {HTTP}. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 417–429, 2015.

[154] Scott Ikeda. Massive optus data leak prompts new privacy rules in australia. *CPO Magazine*, 2022.

[155] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *TSE*, 2019.

[156] Google clusterfuzz. https://google.github.io/clusterfuzz/, 2022. Accessed: 2022-12-22.

[157] What is social engineering? https://www.kaspersky.com.au/resource-center/definitions/what-is-social-engineering, 2022. Accessed: 2022-12-22.

[158] Webdriver. https://www.selenium.dev/documentation/webdriver/, 2022. Accessed: 2022-12-22.

[159] Andreas Zeller. Yesterday, my program worked. today, it does not. why? *ACM SIGSOFT Software engineering notes*, 24(6):253–267, 1999.

[160] William Koch, Abdelberi Chaabane, Manuel Egele, William Robertson, and Engin Kirda. Semi-automated discovery of server-based information oversharing vulnerabilities in android applications. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 147–157, 2017.

[161] Silviu Andrica and George Candea. Warr: A tool for high-fidelity web application record and replay. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 403–410. IEEE, 2011.

[162] Brian Burg, Richard Bailey, Andrew J Ko, and Michael D Ernst. Interactive record/replay for web application debugging. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*, pages 473–484, 2013.

[163] Sergio Flesca, Filippo Furfaro, and Elio Masciari. Monitoring web information changes. In *Proceedings International Conference on Information Technology: Coding and Computing*, pages 421–425. IEEE, 2001.

[164] Yuan Wang, David J DeWitt, and J-Y Cai. X-diff: An effective change detection algorithm for xml documents. In *Proceedings 19th international conference on data engineering (Cat. No. 03CH37405)*, pages 519–530. IEEE, 2003.

[165] Imad Khoury, Rami M El-Mawas, Oussama El-Rawas, Elias F Mounayar, and Hassan Artail. An efficient web page change detection system based on an optimized hungarian algorithm. *IEEE Transactions on Knowledge and Data Engineering*, 19 (5):599–613, 2007.

[166] Mouna Hammoudi, Gregg Rothermel, and Andrea Stocco. Waterfall: An incremental approach for repairing record-replay tests of web applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 751–762, 2016.

[167] Fei Shao, Rui Xu, Wasif Haque, Jingwei Xu, Ying Zhang, Wei Yang, Yanfang Ye, and Xusheng Xiao. Webevo: taming web application evolution via detecting semantic structure changes. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 16–28, 2021.

[168] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. Nyx-net: Network fuzzing with incremental snapshots. *arXiv preprint arXiv:2111.03013*, 2021.

[169] Rensis Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.

[170] State of apis 2024. https://stateofapis.com/, 2024. Accessed: 2024-10-20.

[171] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.

[172] Cockpit CMS. Cockpit cms. https://github.com/Cockpit-HQ/Cockpit, 2024. Accessed: 2024-10-23.

[173] Directus. Directus cms. https://github.com/directus/directus, 2024. Accessed: 2024-10-21.

[174] Ghost CMS. Ghost cms. https://github.com/TryGhost/Ghost, 2024. Accessed: 2024-10-23.

[175] Strapi. Strapi cms. https://github.com/strapi/strapi, 2024. Accessed: 2024-10-20.

[176] RealWorld. Realworld. https://github.com/cirosantilli/node-express-sequelize-nextjs-realworld-example-app, 2025. Accessed: 2024-02-12.

[177] Node.js Contributors. c8, 2019–2024. URL https://github.com/bcoe/c8. Code coverage tool using V8's built-in coverage, compatible with Node.js.

[178] Derick Rethans. Xdebug, 2002–2024. URL https://xdebug.org. PHP debugging and profiling tool.

[179] Sebastian Bergmann. php-code-coverage, 2009–2024. URL https://github.com/sebastianbergmann/php-code-coverage. Library for generating code coverage reports for PHP.

[180] Strapi. Bug report 1. https://github.com/strapi/strapi/issues/21935, 2024. Accessed: 2024-10-22.

[181] Strapi. Bug report 2. https://github.com/strapi/strapi/issues/21936, 2024. Accessed: 2024-10-22.

[182] Strapi. Bug report 3. https://github.com/strapi/strapi/issues/23166, 2025. Accessed: 2025-03-17.

[183] Directus. Bug report 1. https://github.com/directus/directus/issues/23939, 2024. Accessed: 2024-10-30.

[184] Directus. Bug report 2. https://github.com/directus/directus/issues/23940, 2024. Accessed: 2024-10-30.

[185] Directus. Bug report 3. https://github.com/directus/directus/issues/23941, 2024. Accessed: 2024-10-30.

[186] Cockpit CMS. Bug report 1. https://github.com/Cockpit-HQ/Cockpit/issues/227, 2024. Accessed: 2024-10-30.

[187] Cockpit CMS. Bug report 2. https://github.com/Cockpit-HQ/Cockpit/issues/228, 2024. Accessed: 2024-10-30.

[188] Cockpit CMS. Bug report 3. https://github.com/Cockpit-HQ/Cockpit/issues/229, 2024. Accessed: 2024-10-30.

[189] Yinhao Xiao, Yizhen Jia, Chunchi Liu, Xiuzhen Cheng, Jiguo Yu, and Weifeng Lv. Edge computing security: State of the art and challenges. *Proceedings of the IEEE*, 107(8):1608–1631, 2019.

[190] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[191] Neil Kulkarni, Caroline Lemieux, and Koushik Sen. Learning highly recursive input grammars. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 456–467. IEEE, 2021.

[192] Facebook. Infer: A static analyzer for java, c, c++, and objective-c. https://fbinfer.com/, 2015. Open-sourced by Facebook in June 2015.

[193] Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W O'Hearn. Finding real bugs in big programs with incorrectness logic. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):1–27, 2022.

[194] GitHub. Codeql: Semantic code analysis engine. https://codeql.github.com/, 2020. Originally developed by Semmle; acquired by GitHub in 2019.

[195] Zhaoqiang Guo, Tingting Tan, Shiran Liu, Xutong Liu, Wei Lai, Yibiao Yang, Yanhui Li, Lin Chen, Wei Dong, and Yuming Zhou. Mitigating false positive static analysis warnings: Progress, challenges, and opportunities. *IEEE Transactions on Software Engineering*, 49(12):5154–5188, 2023.

[196] Aniruddhan Murali, Noble Mathews, Mahmoud Alfadel, Meiyappan Nagappan, and Meng Xu. Fuzzslice: Pruning false positives in static analysis warnings through function-level fuzzing. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13, 2024.

[197] Shisong Qin, Fan Hu, Zheyu Ma, Bodong Zhao, Tingting Yin, and Chao Zhang. Nsfuzz: Towards efficient and state-aware network service fuzzing. *ACM Transactions on Software Engineering and Methodology*, 32(6):1–26, 2023.

[198] Joachim Viide, Aki Helin, Marko Laakso, Pekka Pietikäinen, Mika Seppänen, Kimmo Halunen, Rauli Puuperä, and Juha Röning. Experiences with model inference assisted fuzzing. *WOOT*, 2:1–2, 2008.

[199] HyungSeok Han and Sang Kil Cha. Imf: Inferred model-based fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2345–2358, 2017.

[200] Xiang Gao, Sergey Mechtaev, and Abhik Roychoudhury. Crash-avoiding program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 8–18, 2019.

[201] YoungJae Kim, Seungheon Han, Askar Yeltayuly Khamit, and Jooyong Yi. Automated program repair from fuzzing perspective. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 854–866, 2023.

[202] Marcel Böhme, Charaka Geethal, and Van-Thuan Pham. Human-in-the-loop automatic program repair. In *2020 IEEE 13th international conference on software testing, validation and verification (ICST)*, pages 274–285. IEEE, 2020.

[203] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1597–1612. IEEE, 2020.

[204] Chijin Zhou, Mingzhe Wang, Jie Liang, Zhe Liu, Chengnian Sun, and Yu Jiang. Visfuzz: Understanding and intervening fuzzing with interactive visualization. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1078–1081. IEEE, 2019.

[205] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, pages 224–234, 2013.

[206] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. Fuzzfactory: domain-specific fuzzing with waypoints. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.