Issues and Their Causes in WebAssembly Applications: An Empirical Study

Muhammad Waseem
Faculty of Information Technology, University of Jyväskylä
Jyväskylä, Finland
muhammad.m.waseem@jyu.fi

Teerath Das
Faculty of Information Technology, University of Jyväskylä
Jyväskylä, Finland
teerath.t.das@jyu.fi

Aakash Ahmad
School of Computing and Communications, Lancaster University Leipzig
Leipzig, Germany
a.ahmad13@lancaster.ac.uk

Peng Liang
School of Computer Science, Wuhan University
Wuhan, China
liangp@whu.edu.cn

Tommi Mikkonen
Faculty of Information Technology, University of Jyväskylä
Jyväskylä, Finland
tommi.j.mikkonen@jyu.fi

ABSTRACT

WebAssembly (Wasm) is a binary instruction format designed for secure and efficient execution within sandboxed environments - predominantly web apps and browsers - to facilitate performance, security, and flexibility of web programming languages. In recent years, Wasm has gained significant attention from the academic research community and industrial development projects to engineer high-performance web applications. Despite the offered benefits, developers encounter a multitude of issues rooted in Wasm (e.g., faults, errors, failures) and are often unaware of their root causes that impact the development of web applications. To this end, we conducted an empirical study that mines and documents practitioners' knowledge expressed as 385 issues from 12 open-source Wasm projects deployed on GitHub and 354 question-answer posts via Stack Overflow. Overall, we identified 120 types of issues, which were categorized into 19 subcategories and 9 categories to create a taxonomical classification of issues encountered in Wasm-based applications. Furthermore, root cause analysis of the issues helped us identify 278 types of causes, which have been categorized into 29 subcategories and 10 categories as a taxonomy of causes. Our study led to first-of-its-kind taxonomies of the issues faced by developers and their underlying causes in Wasm-based applications. The issue-cause taxonomies - identified from GitHub and SO, offering empirically derived guidelines - can guide researchers and practitioners to design, develop, and refactor Wasm-based applications.

CCS CONCEPTS

• **Software and its engineering → Designing software**; • **General and reference → Empirical studies**.

EASE 2024, 18–21 June, 2024, Salerno, Italy

ACM ISBN 978-1-4503-9053-8/24/06…$15.00
https://doi.org/10.1145/xxxxxxx.xxxxxxx

KEYWORDS

WebAssembly, Wasm, Issues, Causes, Mining Software Repositories

ACM Reference Format:
Muhammad Waseem, Teerath Das, Aakash Ahmad, Peng Liang, and Tommi Mikkonen. 2024. Issues and Their Causes in WebAssembly Applications: An Empirical Study. In *Proceedings of The 28th International Conference on Evaluation and Assessment in Software Engineering (EASE 2024).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/xxxxxxx.xxxxxxx

1 INTRODUCTION

WebAssembly (Wasm) as a binary instruction format enhances the performance and security of applications in web-based execution environments [1]. It serves as a potential compilation target for a variety of programming languages including C, C++, and Rust, marking a significant milestone in web development [2]. Wasm enables developers to employ their chosen programming languages and execute them swiftly in browsers, elevating the overall web experience that can range from gaming to multimedia and scientific simulations [3]. One of the key features of Wasm is its sandboxed execution environment - a compelling alternative to JavaScript, generally regarded as default language for web applications - offering an efficient and secure web interactions [4].

Recent research (e.g., [5][6]) has shown a significant rise in the utilization of Wasm beyond web browsers. This entails adapting code from various programming languages to operate on a range of devices via the Wasm Interpreter, aiming to establish a unified software architecture for web systems, softwares, and services. Despite these advantages, a thorough understanding of the challenges encountered by developers working with Wasm applications is yet to be fully explored. Wasm is a promising technology, however; its ecosystem and associated tools are in a phase of continuous evolution and often regarded as unstable that can impede the development practices for web applications [7]. Wasm applications may have an *additional* or *specific* set of **issues**. Borrowing the idea from [8] [9], we define **issues** in this study as errors, faults, failures, and bugs that occur in Wasm applications and consequently impact their quality and functionality. To address this *knowledge gap*, we conducted an empirical study of developer interactions on

GitHub and Stack Overflow (SO) concerning Wasm applications. By scrutinizing the information within these exchanges, we aim to identify the common problems faced by developers and the underlying causes of these issues. This initiative will help identify the common issues and their causes, which can also highlight areas requiring further research.

Motivating Scenario: To contextualise the issues and causes in Wasm, we have provided a representative example in Figure 1. The particular example is taken from the Assemblyscript project, an open-source Wasm based project hosted on GitHub (see Table 1). Figure 1 provides essential information about Assemblyscript, including its project description, the number of stars it has received, and its contributor count. As demonstrated in the example, a contributor not only writes code but may also provide additional explanations through comments. Once the code has been compiled, if issues arise, any contributor can report an *issue*, such as "*How to support function callback and Polymorphism*". It is also possible that the same or another contributor may identify the *cause* of the issue, for instance, stating that "*Class inheritance features are not complete. Operation not supported error when using super() keyword*".

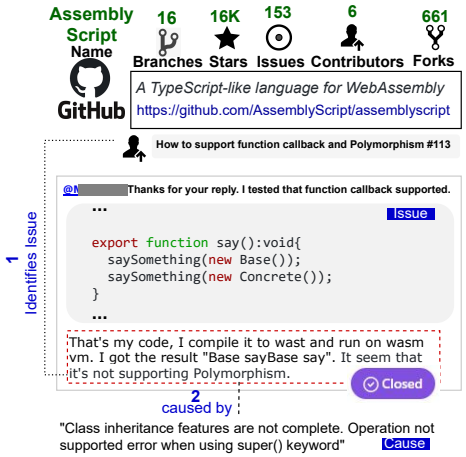

Figure 1: Example: Github based Issue and its Cause in Wasm

Objectives and key Findings: This work aims to *analyse developers' knowledge (available GitHub projects and SO posts) to systematically and comprehensively classify the issues and root causes associated with Wasm-based applications.* To this end, we conducted an empirical investigation on 385 issues from projects hosted on GitHub and 354 issue releated questions and answers posts from SOF. The key findings of the study are structured as taxonomies of issues and their causes indicating (1) the issues in Wasm application are classified into 9 categories, of which Infrastructure and Compatibility Issues (28.16%), Language Features and Documentation Issues (18.00%), and Code Implementation and Build Issues (13.83%) are the most frequently reported; and (2) the leading causes behind these issues are Syntactic and Semantic Errors (25.77%), Configuration and Compatibility Constraints (20.1%), and Operational Limitations (12.98%). Primary contributions of this research are:

- Mining GitHub (social coding platform) and SO (Q&A forum) to collect and analyze practitioners' perspectives, such as code snippets, comments, scripts, queries, and responses, on predominant issues and their most frequent causes.
- Taxonomic classification of issue-cause types, synthesizing available evidence (Figure 3, Figure 4), to categorize, visualize, and understand the nature of issues and causes.
- Providing publicly available data [10] and outlining research implications as recommended guidelines for researching, designing, developing, and refactoring WebAssembly-based applications. The issue-cause taxonomies lay the foundation for discovering and documenting recurring solutions as patterns to address these issues (ongoing future work).

The taxonomies of issues and causes derived from our study offer a structured framework that can guide developers in diagnosing and addressing several types of problems in Wasm applications. Furthermore, these findings provide an empirical foundation for researchers to target specific areas for tool and language improvement, enhancing the overall robustness and usability of the WebAssembly ecosystem.

2 RESEARCH METHOD

The methodology employed for this study is divided into three phases, elaborated below and illustrated in Figure 2.

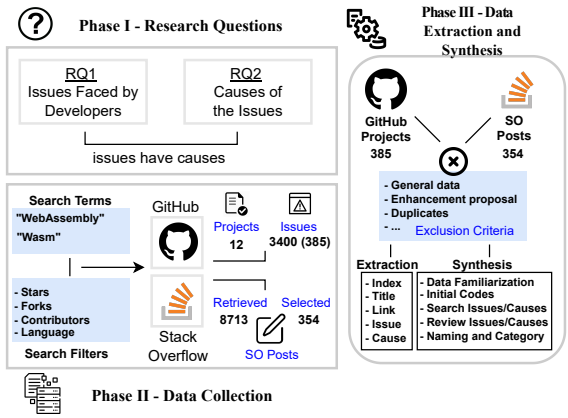

Figure 2: Overview of the research method

2.1 Phase I - Research Questions (RQs)

- **RQ1**: *What issues do developers face when working with WebAssembly applications?* The **objective** of this RQ is to systematically identify and categorize the issues faced by developers in working with WebAssembly applications.
- **RQ2**:*What are the causes of issues that occur in WebAssembly applications?* The **objective** of this RQ is to systematically investigate the root causes of the identified issues in WebAssembly applications.

2.2 Phase II - Data Collection

Data for this study was gathered from two primary platforms.

GitHub: We collected data from a diverse range of open-source Wasm applications hosted on GitHub (see Table 1). To explore

Wasm projects as in 1, we executed the search with the terms "*WebAssembly*" and "*Wasm*" in the GitHub search bar, yielding 11,366 repository results as of March 26, 2023. We then filtered these results based on "languages" using the GitHub sidebar, resulting in 392 repositories. We discovered that several projects were aimed at developing Wasm applications but utilized other programming languages (e.g., Go, Java). To exclude such projects, we manually examined the 392 repositories, selecting only those that (i) utilized the Wasm language for over 50% of the project, and (ii) had more than 100 closed issues. Ultimately, we identified 12 projects (see Table 1). Our selected projects range from *Runtime Environments* like Wasmer, which serves as a critical platform for executing Wasm code, to *Specifications and Proposals* such as `WebAssembly/spec` and `WebAssembly/simd` that guide the platform's evolution. We also explored *Toolchains and Compilers*, exemplified by projects like `WebAssembly/binaryen` and `AssemblyScript/assemblyscript`, which facilitate the development and optimization of Wasm modules. Our study also investigates *Applications* related to Wasm like `torch2424/wasmboy`, a Game Boy emulator that showcases Wasm's performance capabilities, as well as *Blockchain and Smart Contract systems* like `near/nearcore`, which demonstrate Wasm's versatility. Similarly, we also found *UI and Frontend frameworks* for Wasm like `unoplatform/Uno.Wasm.Bootstrap`, which leverage Wasm to extend traditional web development boundaries.We extracted the developer discussions with the help of issue tracking systems of the identified 12 projects, encompassing on closed issues, in order to enhance the likelihood of discovering Wasm-related discussions. In total, we obtained 6,448 closed issues.

Stack Overflow: We also collected Question and Answer (Q&A) pairs related to Wasm from SO. We initiated our data collection process by conducting an automated search on SO using two terms (i.e., "*WebAssembly*", "*Wasm*") aligning with our GitHub search criteria. We then implemented a custom script to extract the relevant Q&A posts retrieved from SO and store the information in data extraction sheets [10]. This encompassed posts where the terms appeared in the post title, body of questions, and contents of answers. Initially, this search yielded 4,518 posts as of April 12, 2023.

Random Sampling: To conduct a comprehensive analysis of the 6,448 issue discussions from GitHub and 4,518 Q&As from SO, we employed a random sampling formula:

$$n = \frac{N \cdot X}{X + N - 1}, \quad \text{with } X = \frac{Z^2 \cdot P \cdot (1 - P)}{E^2}$$

In this formula, N is the population size, which is 6,448 and 4,518 , Z is the Z-score, which is 1.96, P is the assumed population proportion, which is 0.5, and E is the margin of error, which is 0.05. This approach facilitated a balanced subset selection from our dataset, thereby mitigating bias and ensuring the generalizability of our findings. Additionally, it allowed for equitable comparisons among various groups within the dataset, all while optimizing resource utilization. Our selection procedure involved the random sampling of 385 issues from a pool of 6,448 issue discussions and 354 Q&As posts from SO. These samples were drawn while maintaining a 95% confidence level and a 5% margin of error [11].

Table 1: List of Identified WebAssembly Applications

#	Project Name	Closed Issues	Fork	Star
1	Wasmerio/wasmer	917	631	14.7K
2	WebAssembly/spec	517	438	2.9K
3	WebAssembly/binaryen	693	640	6.3K
4	AssemblyScript/assemblyscript	1169	636	15.3K
5	Torch2424/wasmboy	120	55	1.3K
6	WebAssembly/simd	120	46	503
7	WebAssembly/gc	258	51	700
8	WebAssembly/exception-handling	89	33	115
9	Near/nearcore	2201	436	2K
10	PollRobots/scheme	283	5	141
11	Unoplatform/Uno.Wasm.Bootstrap	91	49	312
12	Brson/wasm-opt-rs	13	6	22

Table 2: Data items extracted

#	Data item	Description
D1	Index	ID of the GitHub discussion and SO post
D2	Title	Title of the discussion and SO post
D3	Link	Weblink of the the discussion and SO post
D4	Issue	Key point(s) of the issue from discussion and posts
D5	Cause	Key point(s) for the cause from discussion and posts

2.3 Phase III - Extract and Synthesize Data

Issues and Causes Extraction: After selecting 12 projects, 385 developer discussions from GitHub, and 354 Q&A posts from SO, the first and second authors manually retrieved the background information (e.g., issue label, URL) about the developer discussions and Q&A posts from SO. In the case of GitHub, we only selected closed issues that could potentially provide answers to our research questions. During this step, the first and second authors thoroughly analyzed each of the 385 issues and 354 Q&A, and excluded all those that consisted of (i) general questions, opinions, feedback, and ideas; (ii) enhancement proposals; (iii) general announcements; (iv) duplicated issues or repeated questions; and (v) issues and Q&A posts without detailed descriptions. During the data extraction, there were several discussions from GitHub and Q&A posts from SO where the first and second authors were not able to decide whether to include them for further analysis. In such situations, the first and second authors discussed those issues with all authors to gather their opinions about inclusion or exclusion. Any disagreements about the results of the screening process were discussed among all the authors to reach a consensus.

Data Extraction: We defined a set of data items (see Table 2) to answer the RQs formulated in Section 2.1. The first and second authors of the study conducted a pilot data extraction involving 30 GitHub discussions and 30 SO Q&A posts, and the remaining authors evaluated the extracted data. Subsequently, the first and second authors employed a revised set of data items for formal data extraction from the selected issues. Data items (D1-D3) provide basic information, while data items (D4, D5) used to extract data to answer RQ1 and RQ2.

Data Synthesis: We employed the thematic analysis approach [12] to analyze and classify issues and causes which is consists of five key steps: (i)*Familiarizing with data*: The first and second authors thoroughly reviewed the GitHub discussion and SO pots and documented the key points related to issues and causes. (ii) *Preparing initial codes*: After familiarizing, the same authors compiled an initial list of codes for the identified issues and causes (refer to

the Initial Codes sheet in [10]). (iii) *Searching for the types of issues and causes*: Following the preparation of the initial codes, both the first and second authors analyzed them and categorized them into specific types of issues and causes, (iv) *Reviewing types of issues and causes*: All authors collaboratively reviewed and refined the coding results, organizing them under the relevant types of issues and causes. During this process, we engaged in discussions, separating, merging, or discarding several issues and causes. (v) *Defining and naming categories*: We precisely defined and further refined all types of issues and causes by creating clear subcategories and categories. By following these steps, we established three levels of categories for effectively managing the identified issues and causes for Wasm applications.

3 RESULTS – ISSUES AND CAUSES IN WASM

This section presents the study results, addressing the two RQs outlined in Section 2.1. The results are organized into categories, subcategories, and types. Categories are presented in **boldface**, subcategories in *italic*, and types in SMALL CAPITALS. At the end of each section, based on the study results, a 'Takeaways' box provides the key messages for Wasm researchers and practitioners.

3.1 Types of Issues (RQ1)

The taxonomy of Wasm application issues is shown in Figure 3. Developed from analyzing GitHub developer discussions and SO Q&As, it categorizes 739 issues into 9 main categories with 19 subcategories, totaling 120 types. Descriptions of each category are provided below, with detailed data in our replication package [10].

1. Infrastructure and Compatibility Issues: This category broadly covers issues related to system architecture, integration of various components, and compatibility issues in Wasm applications. It is composed of three subcategories: *Infrastructure Management Issues*, which collect concerns in setting up and maintaining the infrastructure necessary for Wasm; *Application Integration Issues*, which gather problems in the integration of Wasm with various programming languages and platforms; and *Compatibility and Configuration Issues*, which amass issues related to the compatibility of Wasm across different systems. In total, there are 169 issues, constituting 28.16% of all identified issues.

Examples of issues within these subcategories include TESTING ISSUES, TOOLING ISSUES, and INTEGRATION ISSUES, which are related to compromising the reliability and efficiency of the infrastructure. Additionally, COMPATIBILITY ISSUES and SYMBOL RENAMING ISSUES, are crucial to ensure that Wasm modules operate correctly across different environments. Tackling these issues is critical for the robust deployment and functioning of Wasm applications.

2. Language Features and Documentation Issues: This category encompasses concerns related to the features of the programming languages that are available in Wasm, along with issues pertaining to the associated documentation. It consists of two subcategories: *Language Feature Issues*, which collects challenges related to language use, specifications, and the introduction of new features, and *Documentation Issues*, which aggregates problems involving existing documentation, licensing, intellectual property rights, and queries regarding pricing. Altogether, there are 108 issues noted, which constitute 18.00% of all issues identified.

Within these subcategories, examples of issues such as LANGUAGE USAGE ISSUES, LANGUAGE SPECIFICATION ISSUES, and LANGUAGE FEATURE REQUESTS are significant, as they directly influence the efficacy with which developers can leverage Wasm. In parallel, DOCUMENTATION ISSUES and LICENSE/INTELLECTUAL PROPERTY ISSUES underscore the importance of having clear, accessible, and legally robust support materials to facilitate the adoption and effective use of Wasm. Addressing these issues is essential for fostering a comprehensive understanding and application of Wasm within the developer community.

3. Code Implementation and Build Issue: This category is concerned with challenges encountered during the coding and build phases in software development, which are especially pertinent for Wasm given its need for compilation. It includes two main subcategories: *Code Implementation Issues*, representing the range of problems that can arise during the actual coding process, and *Build Issues*, that pertain to complications that occur during the software build process, such as dependency management. This category has a total of 83 issues, representing 13.83% of all the issues identified.

Issues within these subcategories, such as CODE QUALITY ISSUES, CODE ANALYSIS ISSUES, and CODE REVIEW AND FEEDBACK ISSUES, are crucial as they directly impact the efficacy and maintainability of Wasm modules. The compilation process and associated challenges, including BUILD ISSUES and DEPENDENCY MANAGEMENT, are important to address because they affect the performance, reliability, and the smooth deployment of Wasm applications, thus influencing their stability and the cycle of updates.

4. User Interface and Performance Issue: This category encapsulates concerns with the user-facing aspects and the efficiency of Wasm applications. It is divided into *User Interface Issues*, which pertains to the design and interactivity components, including elements like button functionality and UI customization, and *Performance Issue*, which deals with the speed and responsiveness of the application, including performance optimization and graphics rendering. In total, this category includes 68 issues, constituting 11.33% of all issues identified.

Within these subcategories, specific challenges like UI RENDERING ISSUES and UI DESIGN ISSUES are crucial as they directly influence user engagement and satisfaction. Performance concerns, such as PERFORMANCE OPTIMIZATION ISSUES and TIMING AND SYNCHRONIZATION ISSUES, are fundamental to the functionality of Wasm applications, affecting their operational capability and the user experience. Addressing these issues is critical to enhancing both the interface and the performance of Wasm applications for end users.

5. Error Management and Debugging Issues: This category addresses the crucial aspects of identifying, handling, and resolving errors in Wasm applications. It is categorized into *Debugging Issues*, which includes problems like bug regressions, debugging intricacies, and bug fuzzing, and *Error Management Issue*, which covers the systematic approach to error and exception handling, and issues arising from unexpected behavior or integrity errors. There are 65 issues in total, accounting for 10.83% of all issues documented.

Specifically, within these subcategories, challenges such as EXECUTION ERRORS, ERROR/EXCEPTION HANDLING, and UNEXPECTED BEHAVIOR are pivotal, as they impact the stability and reliability of Wasm applications. Issues like FUNCTION SIGNATURE MISMATCH

ERROR, TYPE MISMATCH ERROR, and VALUE ASSIGNMENT ERROR underscore the complexities of ensuring accurate execution and data integrity. Effective management and resolution of these issues are essential for the development of robust, error-resistant Wasm applications.

6. Network and Operational Issues: This category pertains to challenges associated with networking and the day-to-day operational aspects of Wasm applications. It is subdivided into *Functional & Operational Issues*, which comprise problems affecting application functionality and operations such as event handling, module imports, and system integrations, and *Network and Communication Issues*, which deal with data transmission, protocol adherence, and network requests. There are 40 documented issues in total, which account for 6.66% of all issues reported.

In these subcategories, specific concerns like FUNCTIONALITY ISSUES, PRERENDERING ISSUES, and SOCKET INTEGRATION ISSUES are significant as they directly influence the operational effectiveness of Wasm applications. Network-related issues such as NETWORK/PROTOCOL ISSUES and NETWORK COMMUNICATION ISSUES are critical for maintaining robust communication channels within and across Wasm applications.

7. Security Issues: This category encompasses the various aspects of security within Wasm applications. It includes *Authentication Issues*, which cover problems related to user verification, assertion checks, certificate integration, cryptographic operations, and role-based authorization. Another critical area is *Compliance Issues*, comprising 13 issues related to adherence to platform standards, browser compatibility, and environmental regulations, as well as the challenges in porting applications while maintaining compliance. There are 29 documented issues in total, which account for 4.83% of all issues reported.

Specific challenges within these subcategories, such as AUTHENTICATION ISSUES and CRYPTOGRAPHIC OPERATIONS, are fundamental to the secure operation of Wasm applications. Compliance concerns, including PLATFORM COMPATIBILITY ISSUES and ADVERTISING COMPLIANCE, are crucial for the applications to operate within the legal and technical frameworks of various environments. Effectively managing these security and compliance issues is paramount for the integrity and reliability of Wasm applications.

8. Concurrency and Memory Management Errors: This category addresses critical issues related to the simultaneous operation of multiple processes and the efficient management of memory in Wasm applications. It accounts for 20 issues in total, comprising 3.33% of all recorded problems. Within this category, there are *Concurrency Issues*, which include challenges like managing asynchronous execution and synchronization, and *Memory Management Errors*, which involve a variety of concerns ranging from memory access and allocation issues to questions about memory usage and the limitations inherent in dynamic loading. There are 20 issues in total, comprising 3.33% of all recorded problems.

9. State and Data Management: This category is concerned with issues related to maintaining the state of applications and the management of data within Wasm applications, accounting for 18 issues and representing 3% of all issues. It is divided into *State Management Issues*, which includes problems like state serialization and general state management concerns, and *Data Management Issues*, which covers a broader range of data-related challenges such as database integration, caching strategies, asset management, cookie handling, and file management issues.

Within these subcategories, issues such as STATE SERIALIZATION ISSUES and SERIALIZATION ISSUES are critical because they affect how application state is maintained and restored, which is vital for the user experience. On the data management side, issues like DATABASE ISSUES, CACHING ISSUES, and FILE MANAGEMENT ISSUES are essential for the efficient operation and scalability of Wasm applications.

Takeaways

1. **Infrastructure and Compatibility**: Leading issues include system architecture and API complexities, affecting Wasm's seamless integration with existing systems.
2. **Operational Issues**: Networking and communication issues notably impact the stability and reliability of Wasm applications.
3. **Code and Build Issues**: Implementation, optimization, and dependency management are key areas needing attention for quality Wasm application development.

3.2 Causes of Wasm Issues (RQ2)

The taxonomy of causes of Wasm issues is detailed in Figure 4. The cause taxonomy is based on data mined from developer discussions on GitHub and SO. It is important to note that not all discussions on these platforms provide cause information. Therefore, we identified only 516 cause instances from both sources. This analysis identified 278 cause types, categorized into 10 main categories and 29 subcategories. Detailed information is available in the dataset [10].

1. Syntactic and Semantic Errors: This cause category encompasses causes that originate from syntactic and semantic inconsistencies within Wasm code, often leading to compilation and runtime issues , or unexpected behavior. It includes a total of 133 reported causes. Subcategories within this category are *Syntax Errors and Inconsistencies*, *Initialization and File Handling Anomalies*, *Type Mismatches and Inconsistencies*, and *Logic Errors and Bugs*. Some of the leading key types of causes such as BUG IN THE CODE, INTERNAL ERROR, and SYNTAX UNFAMILIARITY within *Syntax Errors and Inconsistencies* are critical as they directly affect the correct interpretation and execution of Wasm code. In *Initialization and File Handling Anomalies*, causes like MISMANAGEMENT OF DATA BUFFERS and SEGMENTATION FAULT ISSUES are significant, as they can lead to crashes and unpredictable behavior. *Type Mismatches and Inconsistencies* include crucial causes such as MISSING IDENTIFIERS AND FUNCTIONS, which can prevent code from compiling or running correctly, while *Logic Errors and Bugs*, with causes like BRANCHING LOGIC IN TRANSACTIONS and COMPONENT INITIALIZATION ISSUES, can result in flawed application logic and runtime errors.

2. Configuration and Compatibility: This cause category is central to issues that arise from the setup and interoperability of Wasm systems, featuring 104 reported causes. Within this category, we have three subcategories: *Compatibility and Specification Issues*, *Build and Configuration Conflicts*, and *Environment & Setup Issues*.

Notable causes within *Compatibility and Specification Issues* include INTEROPERABILITY CHALLENGES and LIBRARY COMPATIBILITY ISSUES, which are critical for ensuring that Wasm modules work across different platforms and with various libraries. In *Build and*

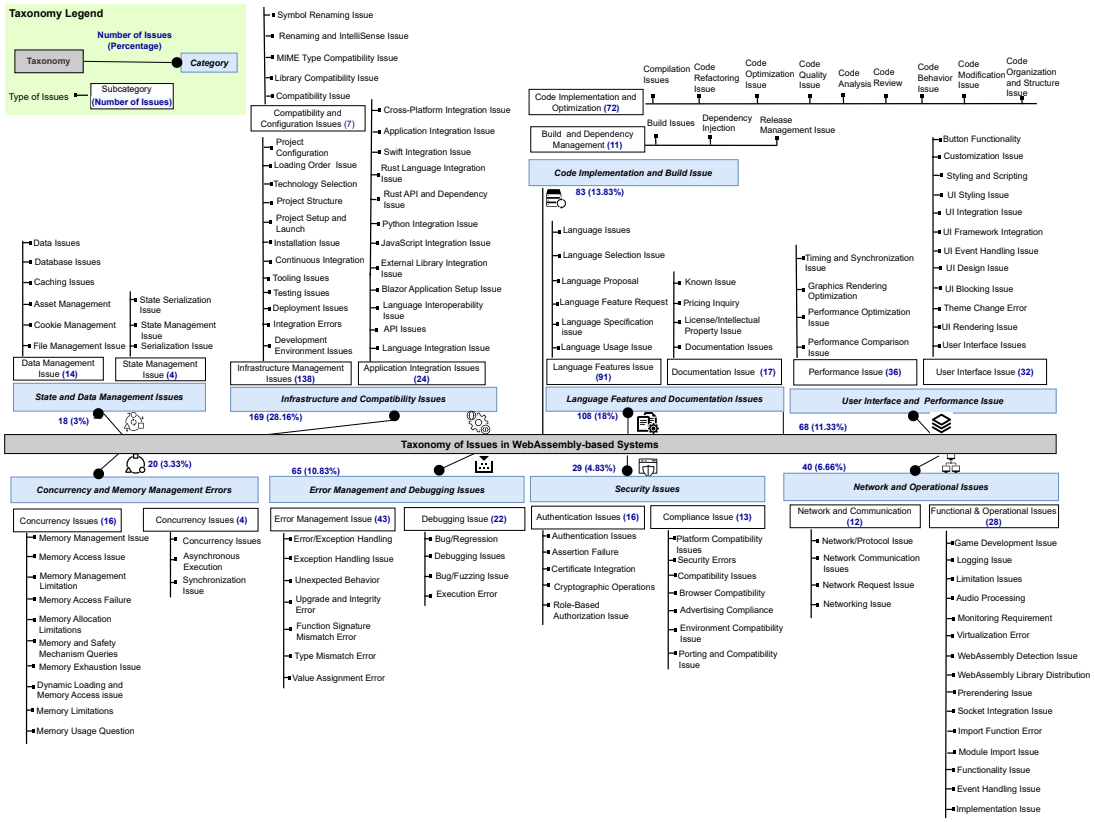

Figure 3: A Taxonomy of Issues in Wasm Applications

Configuration Conflicts, causes such as ENVIRONMENT VARIABLE ISSUES and SSL CONFIGURATION ISSUES can lead to significant deployment problems. Moreover, *Environment & Setup Issues* like LOGGING CONFIGURATION CONFLICTS affect the operational aspect of Wasm applications, highlighting the need for meticulous configuration management. Addressing these configuration and compatibility causes is essential to prevent disruptions in Wasm application development and deployment.

3. Operational Limitations: This cause category deals with constraints that impact the functionality and security of Wasm applications during their operation, totaling 62 reported causes. Within this category, we have two subcategories: *Performance deficiencies*, *Technical Limitations*, and *Security Constraints*. Key types of causes within *Performance deficiencies*, such as PERFORMANCE LIMITATION and PERFORMANCE REGRESSION, are critical as they can significantly degrade user experience and application responsiveness. Within *Technical Limitations*, causes like LACK OF SUPPORT FOR GLOBAL VARIABLES and ABSENCE OF SPECIFICATION TESTS pose challenges for developers by restricting the functionality and verifiability of Wasm modules. *Security Constraints* involve issues like AUTHENTICATION/TOKEN ISSUES and SECURITY/BROWSER POLICY ISSUES, which are essential for maintaining the integrity and trustworthiness of applications. Effectively addressing these operational limitations is crucial for the advancement and secure deployment of Wasm applications.

4. Infrastructure Limitations: This cause category encompasses foundational concerns that impact the operation of Wasm applications. It includes two subcategories: *Network and Platform limitation*, *I/O Handling limitation* and *Synchronization Limitation*.

Significant causes within *Network and Platform limitation* involve challenges like NETWORK CONNECTION ISSUES and PLATFORM-SPECIFIC COMPATIBILITY ISSUES, which can severely restrict an application's functional scope and connectivity. *I/O Handling limitation* is vital for the application's interface with the user and the system, where issues such as HTTP RESPONSE HANDLING and CORS ISSUES are key operational concerns. In the realm of *Synchronization Limitation*, causes such as SINGLE-THREADING CONSTRAINTS and DEADLOCKS highlight the complexities of managing concurrent operations in Wasm. Effectively tackling these infrastructure limitations is essential for the seamless operation and scalability of Wasm applications.

5. Low Code Quality: Gathers the causes related to inadequately maintained and poor-quality codebases in JavaScript and Wasm interactions, along with a deficiency in essential supportive elements such as libraries and documentation. It includes *Poor Code Quality and Maintenance* and *Poor Dependency and Integration Limitations* subcategories.

Key causes such as SCRIPT PATH ISSUES and INEFFICIENCIES IN ORIGINAL CODING DESIGN from the *Poor Code Quality and Maintenance* subcategory are pivotal, as they can directly impact the

functionality and extendibility of the code. Additionally, *Poor Dependency and Integration Limitations* present significant causes like AUTHENTICATION INTEGRATION SHORTCOMINGS and SERVICE CONTAINER DEPENDENCY MISCONFIGURATIONS, which can complicate the integration process and affect the stability of the application. Addressing these causes is essential for the development of high-quality, maintainable Wasm applications that are well-integrated within their respective ecosystems.

6. Language and Library Constraints:Consolidates the causes related to the limitations within programming languages and their associated libraries in the context of Wasm. This category combines causes into three subcategories *API and Functionality Constraints*, *Language & Library Limitations*, and *Web Platform Limitations* subcategories.

For example, *API and Functionality Constraints* involve critical causes such as API LIMITATIONS that can significantly hamper the integration and operational capabilities of Wasm modules. *Language & Library Limitations* highlight causes like LANGUAGE INTEROPERABILITY ISSUES, which affect the seamless integration of Wasm with other programming environments. Furthermore, *Web Platform Limitations* bring attention to causes such as TOOLING LIMITATIONS, emphasizing the need for up-to-date and compatible tools to support the evolving landscape of Wasm. Navigating these causes is key to enhancing Wasm's adaptability and ensuring its effective deployment across various platforms.

7. Documentation and Technologies Causes: This category identifies causes related to informational discrepancies and technical limitations that affect the use and development of Wasm. This category combines causes into two subcategories: *Documentation Inaccuracy* and *Technological and Licensing Constraints*.

For instance, *Documentation inaccuracy* covers causes such as LACK OF CLEAR DEFINITION AND ERRORS IN DOCUMENT, which can create barriers to correctly implementing and leveraging Wasm's functionalities. *Technological and Licensing Constraints* highlight issues like RUST SEGMENTATION FAULT and RUST/WASM INTEROPERABILITY MISMATCH, pinpointing the technical hurdles that can arise due to language-specific features or the integration of different technologies. These constraints underscore the importance of accurate documentation and adaptable technology solutions to support the evolving needs of Wasm applications and their users. Addressing these causes is essential to foster a clear understanding and effective utilization of Wasm across various domains.

8. Data Handling and Design Anomalies: This category captures causes concerning the integrity and structure of data within Wasm applications. It category combines causes into three subcategories: *Data inconsistency*, *Database Anomalies*, *File Access and Handling Anomalies*, and *Serialization Anomalies*.

For example, causes such as *anomalies in atomic type implementation* and *state management issues* within the *Data inconsistency* subcategory can directly affect the accuracy and reliability of data processes. Causes in *Database Anomalies*, like *data conversion errors* and *missing database files*, are pivotal as they influence the robustness of database operations. Within *File Access and Handling Anomalies*, causes such as *I/O constraints* and *file system access restrictions* can severely limit application functionality. Causes in *Serialization Anomalies*, including *discrepancies in serialization of memory operations offset*, can lead to data integrity concerns.

9. Memory and Storage Anomalies: This category collects causes associated with the mismanagement and technical challenges of memory and storage within Wasm applications. It combines causes into two subcategories *Memory and Storage Anomalies* and *Block and Caching Anomalies*.

Causes like *mismanagement of memory resources* and *issues with garbage collection* within the *Memory and Storage Anomalies* subcategory are critical as they directly influence the application's stability and resource optimization. In the *Block and Caching Anomalies* subcategory, causes such as *block download failures* and *cache problems* highlight the importance of reliable data storage and efficient retrieval mechanisms. Addressing these memory and storage causes is fundamental to ensuring that Wasm applications maintain their integrity and provide a responsive user experience.

10. User Interaction Anomalies: This category encompasses causes that negatively impact the user's ability to interact with Wasm applications effectively. It includes *UI Control Inconsistencies*, *UI Rendering Anomalies*, and *Browser-Specific Limitations* subcategories. For instance, within *UI Control Inconsistencies*, causes such as *compatibility issues with UI controls* and *confusion due to formatting* can disrupt the user's navigation and interaction with the application. *UI Rendering Anomalies* highlight causes like *animation and timing flaws*, which are crucial for a seamless and intuitive user interface. *Browser-Specific Limitations* bring to light causes such as *security constraints in browser*, which can limit functionality and affect the overall accessibility of Wasm applications.

Takeaways

4 **Diversity in Language Compilation**: The diversity of source languages compilable to Wasm causes inconsistencies and errors in resultant applications.
5 **Security Vulnerabilities due to Wasm's Structure and Execution Model**: Wasm's structure and execution model are inherent causes of new security vulnerabilities in applications.
6 **Complexities in Optimizing Compiled Code**: The inherent complexities in optimizing Wasm code cause performance bottlenecks affecting user experience.

4 DISCUSSION AND IMPLICATIONS

This section presents the discussion on the key takeaways along with implications for researchers and practitioners based on the study results. Section 4.1 outlines the potential implications associated with WebAssembly issues, while section 4.2 discusses into the various WebAssembly causes.

4.1 Wasm Issues

1 **Infrastructure and Compatibility Issues**: Among the key insights gained from mining GitHub and SO discussions is the recurring theme of infrastructure and compatibility issues with Wasm. The consistent mention of these challenges among developers suggests that integrating Wasm into existing systems remains a significant obstacle. This aligns with existing studies (e.g., [13]) highlight the difficulty in adopting new technologies due to system architecture complexities [13] and API-related issues [14]. The online developer discussions illuminate a gap between academic understanding and real-world practice. Although the academic literature may describe the theoretical benefits of Wasm [15], the actual integration into existing architectures proves to be a complex issue not fully addressed [16]. *Implications* : The persistent nature of

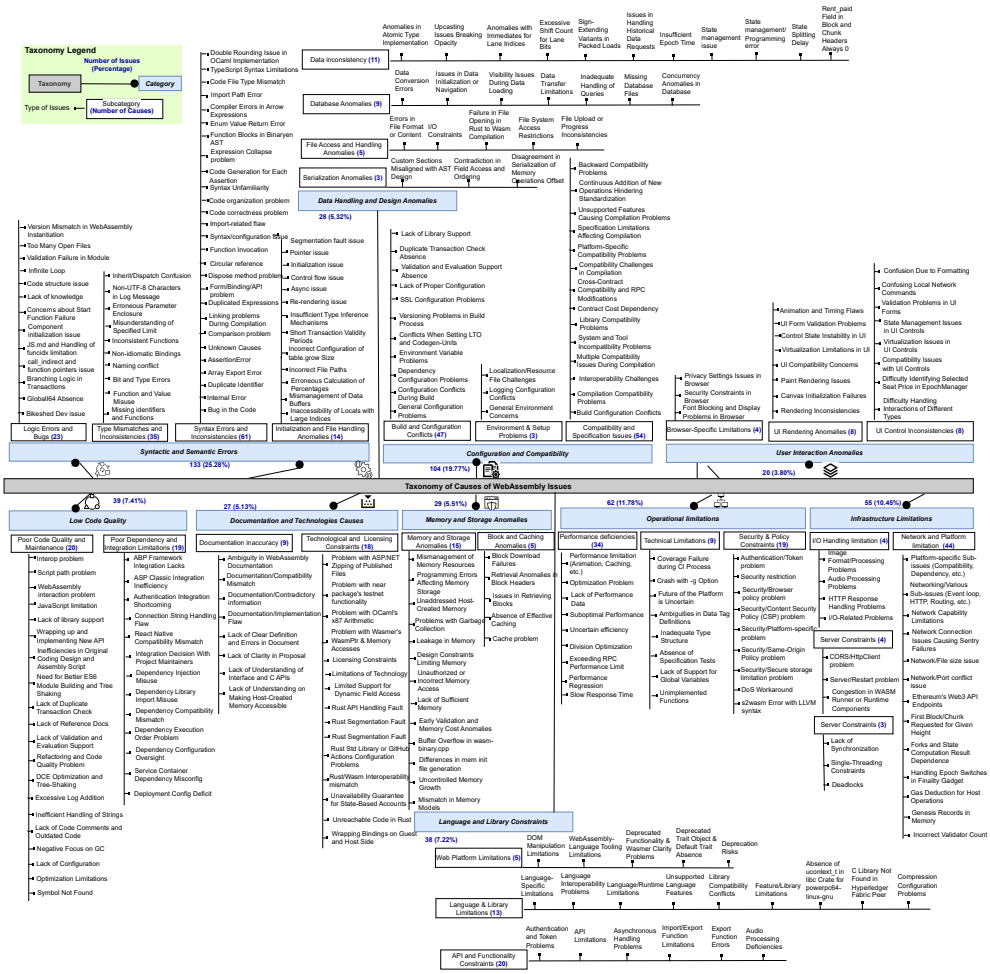

Figure 4: A Taxonomy of Causes of Issues in Wasm Applications

these issues highlights the need for academic research that bridges theory and practice, focusing on creating more robust integration methods or frameworks for Wasm. Practitioners can benefit from more actionable guidance, possibly in the form of best practice documents.

❷ **Operational Issues**: A critical aspect uncovered pertains to operational challenges in networking and communication, as highlighted in discussions across platforms like GitHub and SO. Developers have raised concerns about the reliability of these aspects in Wasm [2], echoing literature that points to new technologies often grappling with underdeveloped networking protocols (e.g.,[17]). There is a discernible lack of literature focusing specifically on Wasm's operational capabilities in these areas. Insights from real-world scenarios, as seen in the aforementioned platforms, are crucial in bridging this knowledge gap [18]. *Implications* : This presents a fertile ground for researchers to delve into Wasm's networking functionalities and suggest improvements. Similarly, practitioners are advised to examine the operational elements of Wasm meticulously, potentially utilizing external libraries or modules as interim solutions.

❸ **Code Implementation and Optimization Issues**: Data mined from developer conversations also highlight challenges in code implementation and optimization, including build and dependency management. The online discussions complement the literature (e.g., [2, 19]) which often talks about the lack of mature toolsets for new technologies, emphasizing that Wasm development is still in its infancy stage [18, 20]. Although academic discourse may emphasize the computational efficiency of Wasm, it seems to overlook the practical aspects of code implementation and optimization, an area clearly fraught with challenges according to GitHub and SO data. *Implications* : Researchers could aim to develop better tools for Wasm development, possibly in collaboration with industry stakeholders, to address the implementation challenges. Practitioners could consider incorporating emerging best practices and tools as they become available, staying up-to-date through both academic and community channels.

4.2 Causes of Wasm Issues

❹ **Diversity in Language Compilation**: We find the diversity in language compilation to Wasm is causing significant discrepancies

and anomalies in the resultant applications, creating inconsistencies in application behavior and functionality. This issue is consistent with existing studies (e.g., [21]) and practitioner perspective (e.g., [21–23]), highlighting the challenges and irregularities arising due to compiling a variety of languages like C, C++, and Rust to Wasm. It reaffirms the prevailing knowledge base, emphasizing the problems in maintaining consistency during compilation processes. The diverse origin of source languages necessitates a more universal and standardized compilation strategy to prevent the resultant inconsistencies and anomalies in Wasm-based applications. *Implications*: There is a need to develop more comprehensive and robust compilation methods to accommodate the diversity in source languages. Developers should be aware of the complications arising from language diversity and consider the compatibility of origin languages with Wasm during the development phase.

❺ **Security Vulnerabilities due to Wasm's Structure and Execution Model**: Our research identifies that the unique structure and execution model of Wasm are introducing new security vulnerabilities and expanding the application's attack surface. This finding align with some of the earlier studies (e.g., [1, 24]) that depicted Wasm as a more secure alternative to JavaScript. The findings also reveals potential gaps in our understanding of Wasm's security model and necessitates further exploration into its unique vulnerabilities. The alignment between our results and previous studies highlights the evolving and dynamic nature of Wasm, suggesting continuous emergence and evolution of potential security threats and vulnerabilities. *Implications*: This contradiction prompts a deeper examination of Wasm's security framework, urging further exploration and research into its vulnerabilities and mitigation strategies. Developers need to implement rigorous security protocols and continuously monitor and update the security features of applications to mitigate the risks associated with the unique vulnerabilities of Wasm.

❻ **Complexities in Optimizing Compiled Code**: The research indicates that the complexities involved in optimizing the compiled code are creating substantial performance bottlenecks, affecting the user experience and application response times adversely. The findings align well with existing literature (e.g., [19, 25, 26]), emphasizing the critical need to address these performance bottlenecks by developing advanced optimization techniques to improve the efficiency and response time of Wasm applications. *Implications*: The recurring issues related to performance bottlenecks in our findings indicate the need of optimization techniques and methodologies to enhance the user experience and application efficiency. Developers and IT professionals should prioritize resolving these performance bottlenecks by exploring and implementing new optimization solutions and techniques to enhance application performance and user experience.

5 RELATED WORK

This section overviews the most relevant existing research, classifying and analyzing empirically-based studies focused on (i) bugs and security issues along with (ii) performance challenges in Wasm applications. A conclusive summary highlights the scope and contributions of the proposed research in the context of related work.

5.1 Bugs and Security Issues in Wasm

Bugs in Wasm applications are among the prevailing challenges including issues that relate to not the bugs, errors, and security risks during application compilation [27–29]. Specifically, Romano et al. [30] conducted an empirical study to analyze 1,054 bugs in Wasm compilers The study investigated 'lifecycle', 'impact', and 'sizes' of bug-inducing inputs and bug fixes and highlighted the need for further research on principles and practices to debug Wasm applications. Security-critical issues in Wasm have gained significant attention of researchers with web application development for blockchain solutions [31, 32]. Lehmann et al. [1] examined security vulnerabilities to analyze the extent vulnerabilities are exploitable in WebAssembly binaries, and how this compares to native code in Wasm and proposed solutions. Similar studies such as [27–29] address the security of Wasm-based smart contracts for blockchain systems. Compared to conventional Ethereum smart contracts, Wasm smart contracts have shown growing popularity for web-based blockchains, however, they suffer from various attacks exploiting their vulnerabilities [28].

5.2 Performance Issues

Performance issues in Wasm applications can jeopardise time-critical transactions and user experience in web systems. The research by Jangda et al. [26] empirically compares native and Wasm code to identify the bottlenecks that slow down application execution. A similar study by Yan et al. [33] compared Wasm and JavaScript performance to guide developers in identifying optimization opportunities in web development. Furthermore, Andre et al. [34] investigate Wasm-related discussions on Stack Overflow, revealing security concerns and frequent requests for bug-fixing corresponding to the performance of Wasm-based web applications.

Conclusive Summary: Based on the review above, we conclude that the proposed research is closely aligned and complements the existing body of knowledge on empirical studies on identifying bugs [31] and experimental analysis of security-critical issues in Wasm application development [34]. The proposed research has investigated data from social coding and discussion platforms (GitHub, SO) in an attempt to identify, classify, and conceptualize the issues faced by developers and their causes in Wasm application development cycle.

6 THREATS TO VALIDITY

External validity refers to how generalizable the study's findings are to other contexts or settings related to Wasm issues and causes. One of the possible threat could be missing out some Wasm issues or getting different results from various other platforms/data sources such as GitLab and Bitbucket. In order to minimize this potential biases, we gathered data from two widely-used and popular platforms, namely GitHub and Stack Overflow. These two platforms contains the millions of developers user base. Another potential threat may be not considering all data points for our analysis. To ensure a well-rounded representation of the data, we followed a standard random sampling technique with 95% confidence level and 5% margin of error [11].

Internal validity relates to how well a study minimizes bias collection. One of the possible risks includes the qualitative analysis

and taxonomy synthesis from the discussions of GitHub and Q&A posts on Stack Overflow. More specially, the annotation phase could inject subjective bias among the annotators. To mitigate this risk, we conducted a pilot study to establish a shared comprehension of the attributes of Wasm issues. This initial phase also aided in the creation of a robust coding schema for the subsequent annotation process. Furthermore, two authors construct the taxonomies, with a third author conducting a comprehensive validation of the results and resolving any discrepancies through ongoing consensus discussions. Additionally, we calculated Cohen Kappa values to assess the agreement among all authors. Another potential threat to internal validity concerns the selection of open-source GitHub projects.

7 CONCLUSIONS

In this research, we developed the first-of-its-kind taxonomies for Wasm issues and their causes. *Implications:* This study provides researchers and practitioners with valuable insights into the challenges and complexities involved in the development and deployment of Wasm application. The taxonomy and empirical findings contribute as an evidence-based understanding that is essential for advancing the research in Wasm, which has seen rising attention but still lacks comprehensive issue-related research.

Needs for future research: We have three main objectives: (i) To propose a taxonomy of solutions, mapping the relationships among issues, causes, and potential solutions. (ii) To validate the proposed taxonomy of issues, causes, and solutions through an industrial survey, seeking insights from the practitioners' perspective. (iii) To investigate the difficulty and priority levels associated with the identified issues in practical settings.

ACKNOWLEDGMENTS

This research is funded by Business Finland through the LiquidAI (8542/31/2022) and 6G Soft (8541/31/2022) projects, and by the NSFC China under Grant No. 62172311.

REFERENCES

[1] D. Lehmann, J. Kinder, and M. Pradel, "Everything old is new again: Binary security of WebAssembly," in *Proceedings of the 29th USENIX Security Symposium (USS).* USENIX, 2020, pp. 217–234.

[2] A. Haas, A. Rossberg, D. L. Schuff, B. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with WebAssembly," in *Proceedings of the 38th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI).* ACM, 2017, pp. 185–200.

[3] T. Ketonen, "Examining performance benefits of real-world WebAssembly applications: a quantitative multiple-case study," Bachelor's Thesis, 2022.

[4] D. Goltzsche, M. Nieke, T. Knauth, and R. Kapitza, "Acctee: A WebAssembly-based two-way sandbox for trusted resource accounting," in *Proceedings of the 20th Int. Middleware Conf. (Middleware).* ACM, 2019, pp. 123–135.

[5] P. Kotilainen, V. Järvinen, J. Tarkkanen, T. Autto, T. Das, M. Waseem, and T. Mikkonen, "WebAssembly in iot: Beyond toy examples," in *Proceedings of the 23rd Int. Conf. on Web Engineering (ICWE).* Springer, 2023, pp. 93–100.

[6] P. Kotilainen, T. Autto, V. Järvinen, T. Das, and J. Tarkkanen, "Proposing isomorphic microservices based architecture for heterogeneous iot environments," in *Proceedings of the 23rd Int. Conf. on Product-Focused Software Process Improvement (PROFES).* Springer, 2022, pp. 621–627.

[7] D. Herrera, H. Chen, E. Lavoie, and L. Hendren, "WebAssembly and javascript challenge: Numerical program performance using modern browser technologies and devices," *University of McGill, Montreal: QC, Technical report SABLE-TR-2018-2*, 2018.

[8] M. Waseem, P. Liang, M. Shahin, A. Ahmad, and A. R. Nassab, "On the nature of issues in five open source microservices systems: An empirical study," in *Proceedings of the 25th Int. Conf. on Evaluation and Assessment in Software Engineering (EASE).* ACM, 2021, pp. 201–210.

[9] M. Waseem, P. Liang, A. Ahmad, A. A. Khan, M. Shahin, P. Abrahamsson, A. R. Nasab, and T. Mikkonen, "Understanding the issues, their causes and solutions in microservices systems: An empirical study," *arXiv preprint arXiv:2302.01894*, 2023.

[10] M. Waseem, T. Das, A. Ahmad, P. Liang, and T. Mikkonen, "Dataset for the Paper: Issues and Their Causes in WebAssembly Applications: An Empirical Study," https://zenodo.org/record/10528608, Jan. 2024.

[11] G. D. Israel, "Determining sample size," Florida Cooperative Extension Service, Institute of Food and Agricultural Sciences, University of Florida, Florida, U.S.A, Fact Sheet PEOD-6, November 1992.

[12] D. S. Cruzes and T. Dyba, "Recommended steps for thematic synthesis in software engineering," in *Proceedings of the 5th ACM/IEEE Int. Symposium on Empirical Software Engineering and Measurement (ESEM).* IEEE, 2011, pp. 275–284.

[13] R. Li, P. Liang, and P. Avgeriou, "Warnings: Violation symptoms indicating architecture erosion," *Information and Software Technology*, vol. 164, p. 107319, 2023.

[14] L. Li, T. F. Bissyandé, H. Wang, and J. Klein, "Cid: Automating the detection of api-related compatibility issues in android apps," in *Proceedings of the 27th ACM SIGSOFT Int. Symposium on Software Testing and Analysis (ISSTA).* ACM, 2018, pp. 153–163.

[15] D. Eleskovic, "A closer look at WebAssembly," Bachelor's Thesis, 2020.

[16] B. Bosshard, "On the use of web assembly in a serverless context," in *Proceedings of the 21st Int. Conf. on Agile Software Development (XP) Workshops.* Springer, 2020, pp. 141–145.

[17] P. P. Ray, "An overview of WebAssembly for iot: Background, tools, state-of-the-art, challenges, and future directions," *Future Internet*, vol. 15, no. 8, p. 275, 2023.

[18] M. Šipek, D. Muharemagić, B. Mihaljević, and A. Radovan, "Next-generation web applications with WebAssembly and trufflewasm," in *Proceedings of the 44th Int. Convention on Information, Communication and Electronic Technology (MIPRO).* IEEE, 2021, pp. 1695–1700.

[19] W. Wang, "Empowering web applications with WebAssembly: are we there yet?" in *Proceedings of the 36th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE).* IEEE, 2021, pp. 1301–1305.

[20] T. Nießen, M. Dawson, P. Patros, and K. B. Kent, "Insights into WebAssembly: compilation performance and shared code caching in node.js," in *Proceedings of the 30th Annual Int. Conf. on Computer Science and Software Engineering (CASCON).* ACM, 2020, pp. 163–172.

[21] P. Krill. (2023) Direct WebAssembly compilation comes to rust language. [Online]. Available: https://www.infoworld.com

[22] H. Patel. (2023) WebAssembly: Unlocking performance and portability for web applications. [Online]. Available: https://javascript.plainenglish.io

[23] C. Popoviciu. (2023) Use the language of your choice with pages functions via WebAssembly. [Online]. Available: https://blog.cloudflare.com

[24] Q. Stiévenart, C. De Roover, and M. Ghafari, "The security risk of lacking compiler protection in WebAssembly," in *Proceedings of the 21st IEEE Int. Conf. on Software Quality, Reliability and Security (QRS).* IEEE, 2021, pp. 132–139.

[25] J. Cabrera Arteaga, S. Donde, J. Gu, O. Floros, L. Satabin, B. Baudry, and M. Monperrus, "Superoptimization of WebAssembly bytecode," in *Proceedings of the 4th Int. Conf. on Art, Science, and Engineering of Programming (PROGRAMMING): Companion.* ACM, 2020, pp. 36–40.

[26] A. Jangda, B. Powers, E. D. Berger, and A. Guha, "Not so fast: Analyzing the performance of WebAssembly vs. native code," in *Proceedings of the USENIX Annual Technical Conf. (ATC).* USENIX, 2019, pp. 107–120.

[27] W. Chen, Z. Sun, H. Wang, X. Luo, H. Cai, and L. Wu, "Wasai: uncovering vulnerabilities in wasm smart contracts," in *Proceedings of the 31st ACM SIGSOFT Int. Symposium on Software Testing and Analysis (ISSTA).* ACM, 2022, pp. 703–715.

[28] L. Quan, L. Wu, and H. Wang, "Evulhunter: Detecting fake transfer vulnerabilities for eosio's smart contracts at WebAssembly-level," *arXiv preprint arXiv:1906.10362*, 2019.

[29] J. Zhou and T. Chen, "Wasmod: Detecting vulnerabilities in wasm smart contracts," *IET Blockchain*, 2023.

[30] A. Romano, X. Liu, Y. Kwon, and W. Wang, "An empirical study of bugs in WebAssembly compilers," in *Proceedings of the 36th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE).* IEEE, 2021, pp. 42–54.

[31] Y. Wang, Z. Zhou, Z. Ren, D. Liu, and H. Jiang, "A comprehensive study of WebAssembly runtime bugs," in *Proceedings of the 30th IEEE Int. Conf. on Software Analysis, Evolution and Reengineering (SANER).* IEEE, 2023, pp. 355–366.

[32] Y. Zhang, S. Cao, H. Wang, Z. Chen, X. Luo, D. Mu, Y. Ma, G. Huang, and X. Liu, "Characterizing and detecting WebAssembly runtime bugs," *ACM Transactions on Software Engineering and Methodology*, 2023.

[33] Y. Yan, T. Tu, L. Zhao, Y. Zhou, and W. Wang, "Understanding the performance of WebAssembly applications," in *Proceedings of the 21st ACM Internet Measurement Conf. (IMC).* ACM, 2021, pp. 533–549.

[34] P. M. André, Q. Stiévenart, and M. Ghafari, "Developers struggle with authentication in blazor WebAssembly," in *Proceedings of the 38th IEEE Int. Conf. on Software Maintenance and Evolution (ICSME).* IEEE, 2022, pp. 389–393.