

Research on WebAssembly Runtimes: A Survey

YIXUAN ZHANG, Peking University, China

MUGENG LIU, Peking University, China

HAOYU WANG, Huazhong University of Science and Technology, China

YUN MA, Peking University, China

GANG HUANG, Peking University, China

XUANZHE LIU, Peking University, China

WebAssembly (abbreviated as Wasm) was initially introduced for the Web but quickly extended its reach into various domains beyond the Web. To create Wasm applications, developers can compile high-level programming languages into Wasm binaries or manually convert equivalent textual formats into Wasm binaries. Regardless of whether it is utilized within or outside the Web, the execution of Wasm binaries is supported by the Wasm runtime. Such a runtime provides a secure, memory-efficient, and sandboxed execution environment designed explicitly for Wasm applications. This paper provides a comprehensive survey of research on WebAssembly runtimes. It covers 98 articles on WebAssembly runtimes and characterizes existing studies from two different angles, including the "internal" research of Wasm runtimes (Wasm runtime design, testing, and analysis) and the "external" research (applying Wasm runtimes to various domains). This paper also proposes future research directions about WebAssembly runtimes.

CCS Concepts: • General and reference → Surveys and overviews.

Additional Key Words and Phrases: WebAssembly, WebAssembly runtime, WebAssembly System Interface

1 INTRODUCTION

WebAssembly (abbreviated as Wasm) was initially proposed for the Web as a binary instruction format for a stack-based virtual machine. The paper introducing WebAssembly was published in 2017 [57]. It outlines the motivation, design, and formal semantics of WebAssembly while offering initial insights into its implementation. Wasm is crafted to serve as a universally compatible compilation target for programming languages, facilitating web deployment for client and server applications [26]. Currently, major web browsers, including Chrome, Firefox, Safari, and Edge, support the execution of WebAssembly (Wasm) [26]. Although Wasm was initially proposed for the Web, it rapidly expanded into multiple domains beyond the Web [24], including the Internet of Things (IoT) [141], blockchain [3, 4, 39], serverless computing [51], edge computing [50, 95], etc.

To develop Wasm applications, developers can compile high-level programming languages into Wasm binaries or manually convert the equivalent textual format [25] into Wasm binaries. Whether inside or outside the web, the execution of Wasm binaries relies on the Wasm runtime. A Wasm runtime offers a secure, memory-efficient, and sandboxed execution environment tailored for Wasm applications [26]. Specifically, Wasm binaries are executed in the JavaScript engine with JavaScript glue code within the Web [134]. Outside the Web, Wasm binaries can independently run in standalone Wasm runtimes [150]. This paper will refer to all tools used to execute Wasm binaries as the Wasm runtime. Wasm runtime has attracted widespread attention in the academic community since the birth of Wasm. Figure 1 shows the number of publications on the topic of Wasm runtimes between January 1st, 2017, and January 8th, 2024 (we introduce how we collected these articles in Section 4.1). The number of papers related to Wasm runtime is rapidly increasing, attracting growing attention to this field.

Authors' addresses: Yixuan Zhang, Peking University, Beijing, China, zhangyixuan.6290@pku.edu.cn; Mugeng Liu, Peking University, Beijing, China, lmg@pku.edu.cn; Haoyu Wang, Huazhong University of Science and Technology, Wuhan, China, haoyuwang@hust.edu.cn; Yun Ma, Peking University, Beijing, China, mayun@pku.edu.cn; Gang Huang, Peking University, Beijing, China, hg@pku.edu.cn; Xuanzhe Liu, Peking University, Beijing, China, liuxuanzhe@pku.edu.cn.

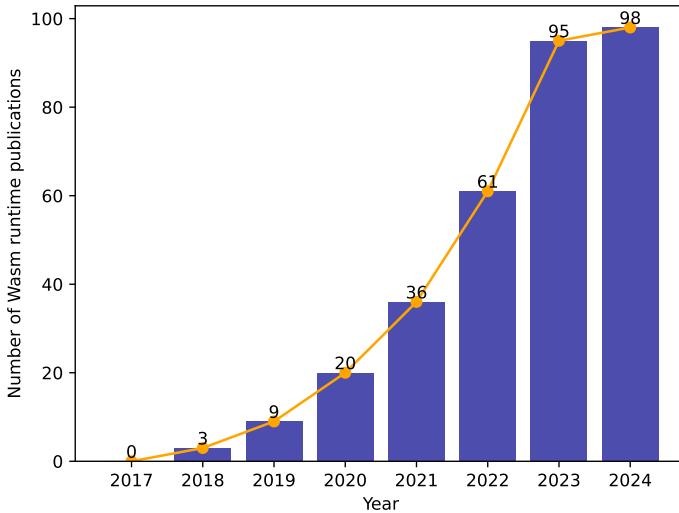


Fig. 1. Wasm runtime publications (accumulative) during 2017-2024. (The publication count for the year 2024 is recorded up to January 8th.)

To the best of our knowledge, this is the first survey of research on Wasm runtimes. We follow the traditional systematic literature review process to collect 98 research papers related to Wasm runtimes. Then we characterize existing studies from two different angles. The first angle is the “internal” research of Wasm runtimes, including analysis, testing, and design of Wasm runtimes, aiming to improve the security, performance and scalability of Wasm runtimes. The other angle is the “external” research of Wasm runtimes, i.e., applying Wasm runtimes to various kinds of domains (e.g., IoT and edge computing) and addressing the challenges of different scenarios. Moreover, we discuss the current issues and the future research directions. As far as we know, this is the first survey focusing on Wasm runtimes. In summary, the paper makes the following contributions:

- *Definition* This paper defines WebAssembly runtime (Wasm runtime), the workflow of WebAssembly, the architecture and features of Wasm runtimes.
- *Survey* The paper provides the first comprehensive survey of 98 Wasm runtime articles across various publishing areas such as software engineering, security, and programming languages.
- *Analyses* This paper illustrates and discusses these related Wasm runtimes in two aspects: application scenarios and research directions.
- *Future directions* To facilitate the development of Wasm runtimes and the Wasm community, this paper identifies the current issues and promising research directions for Wasm runtimes.

Figure 2 depicts the paper structure. The details of the review schema can be found in Section 4.

2 PRELIMINARIES OF WEBASSEMBLY

This section illustrates the background and fundamental terminologies in WebAssembly. WebAssembly is a binary format that was first proposed for the Web [57] but is now widely used inside and outside the Web, including Internet of things(IoT) [111], edge computing [50], and blockchain [39].

Workflow This section first depicts the workflow of Wasm inside and outside the Web. As shown in Figure 3, the Wasm frontend compilers could compile high-level languages, such as C/C++

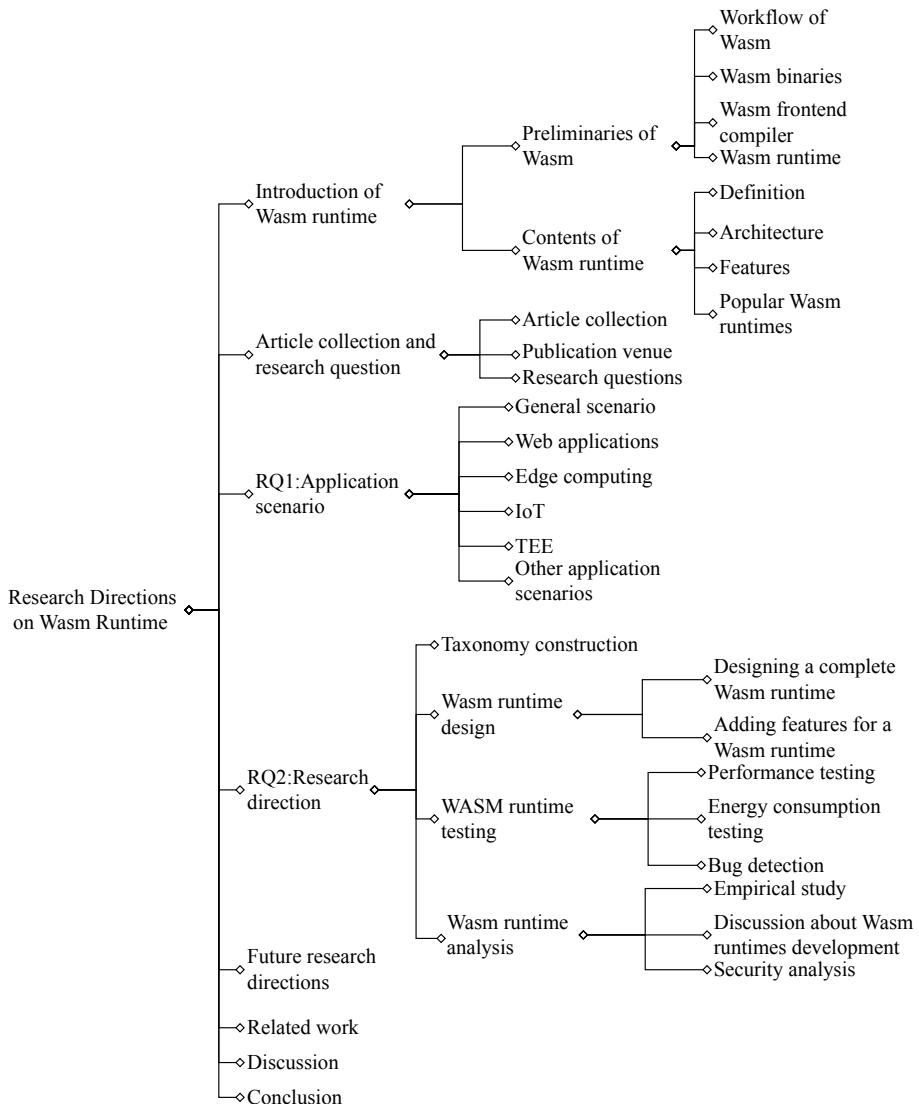


Fig. 2. Tree structure of the contents in this paper.

and Rust, into Wasm binaries with the corresponding JavaScript(JS) glue code. The JS glue code and Wasm binaries are executed in the JS engines in the Web browsers. As shown in Figure 4, the Wasm frontend compilers could compile high-level language into Wasm binaries, and the Wasm binaries are executed in standalone Wasm runtimes which are directly deployed in operating systems(OSes). In this paper, both JS engines that execute Wasm binaries inside the Web and standalone Wasm runtimes that execute Wasm binaries outside the Web are regarded as Wasm runtimes. That is to say, developers could develop applications inside or outside the Web with high-level languages and generate Wasm binaries to be executed in Wasm runtimes. The following subsections depict each part of the Wasm workflow.

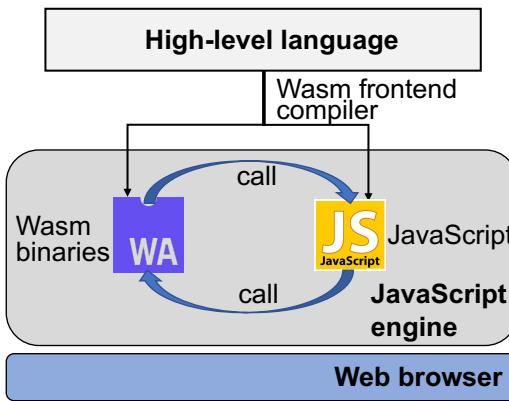


Fig. 3. Wasm workflow inside the Web.

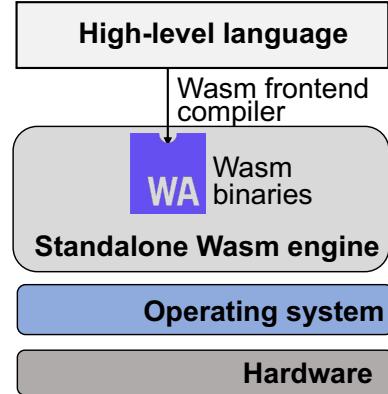


Fig. 4. Wasm workflow outside the Web.

Wasm binaries The WebAssembly (Wasm) binary file is compact, similar to Java class files, and is stored in the file with .wasm suffix [149]. The WebAssembly (Wasm) specification [17] outlines a conceptual stack virtual machine, where the majority of Wasm instructions operate by popping and pushing numbers, leaving the results on the stack. As shown in Figure 5, Wasm binaries could be converted into a pretty-printed textual format (i.e., .wat) with wabt [8] or the reverse process. The wat format [25] is easy to read and can be used to learn the syntax, debug the Wasm code, understand the Wasm modules, manually write Wasm programs, etc. The Wasm binary format begins with a magic number and a version number, followed by the main body of the Wasm module, as shown in Figure 5. A Wasm module is the fundamental unit of Wasm binaries. The specification [17] defines the Wasm module in the format of tree nodes. These nodes are written in the style of S-expressions, which goes inside a pair of parentheses (...). The main body of the Wasm module is organized into different sections, known as segments, where each segment can contain multiple items. These segments include code segments, export segments, function segments, etc.

```

1  (module
2    (func (;0;) (result i64)
3      i64.const 1
4      i64.const 2
5      i64.rot)
6      (export "main" (func 0)))
                                wabt
    1  0061 736d 0100 0000 0105 0160 0001 7e03
    2  0201 0007 0801 046d 6169 6e00 000a 0901
    3  0700 4201 4202 8a0b

```

Fig. 5. The conversion between Wasm binaries and textual format.

Wasm frontend compiler Wasm frontend compiler refers to the compilers which could compile high-level language programs into Wasm binaries. The Wasm frontend compiler first translates high-level language source code into an intermediate representation (IR) and then generates Wasm binaries from the IR [149]. During this process, the Wasm frontend compilers include bindings for existing libraries, enabling the use of standard libraries available in the source language within the Wasm runtime [18, 149]. As shown in Table 1, almost all the most widely used high-level languages could be compiled into Wasm binaries through different Wasm frontend compilers.

Wasm runtime As shown in Figure 3 and Figure 4, Wasm binaries could be executed inside and outside the Web with different types of Wasm runtimes. Inside the Web, the Wasm runtimes are deployed in the web browsers. Moreover, the Wasm binaries have to be executed with the JS glue

Table 1. The Wasm frontend compilers.

Wasm frontend compiler	Source language	Commits	Stars
Emscripten	C/C++	26,521	24.8k
Rustc	Rust	243,427	88.9k
Ppc1-Mirror	Python	1,118	0
TinyGo	Go	3,614	13.9k
Bytecoder	Java	1,422	836
AssemblyScript	TypeScript	1,564	16.2k

code. However, the Wasm runtimes outside the Web are directly deployed on the OSes. And the Wasm binaries could be executed standalone or interact with high-level languages [150].

3 WEBASSEMBLY RUNTIME

This section gives a definition and analysis of Wasm runtimes. It describes the architecture (components), the most popular Wasm runtimes, and the scenarios of Wasm runtimes.

3.1 Definition

The execution of Wasm binaries is defined in an abstract virtual machine (VM). This VM should contain a stack to record the operand values and control constructs and contain an abstract area to store the global state. This VM is expected to support all the instructions in the Wasm binary format and the interaction between Wasm binaries and the environment in which the VM exists [9]. In this paper, we refer to the term *Wasm runtime* for the programs that could execute Wasm binaries inside or outside the Web.

3.2 Architecture

As shown in Figure 6, a Wasm runtime could be divided into four components [150], including the Wasm compiler, Wasm interpreter, runtime environment, and the interaction part with the underlying environment (WASI implementation or the JS glue code). The execution of Wasm binary instructions includes two ways: interpretation or compilation into native code. Some Wasm runtimes support to execute Wasm binaries with **Wasm interpreters**, such as wasm-micro-runtime (WAMR) [16]. Some Wasm runtimes support first compiling the Wasm binaries into an intermediate representation (IR) and then compiling the IR into native code for different hardware, such as wasmtime [14] and wasmer [12]. This paper uses the term **Wasm compiler** to refer to the compilers that could compile Wasm binaries into native code for different CPUs, including *x86_64*, *amd64*, etc. The Wasm runtimes also provide the **runtime environment** to support allocating memory, performing stack operations, reporting execution error messages, and other features.

In order to **interact with the environment** the runtime exists, Wasm runtimes in the Web could call the Web APIs through the **JS glue code**, and standalone Wasm runtimes provide the **implementation of WebAssembly System Interface (WASI)** to call the system calls. WASI is defined by the bytecode alliance [23], containing several OS-like features, including the operations of file systems, clock, thread, etc. WASI is the bridge between the sandbox environment and the environment in which it exists. In order to be available in several OSes, WASI unifies the system calls offered by various OSes, such as POSIX in Linux [23]. The standalone Wasm runtimes, such as WAMR [16], wasmtime [14], and WasmEdge [11], could be called in the programs written in high-level languages as a library to allow developers execute Wasm binaries in any possible cases with various languages. And some standalone Wasm runtimes also offer additional tools to users,

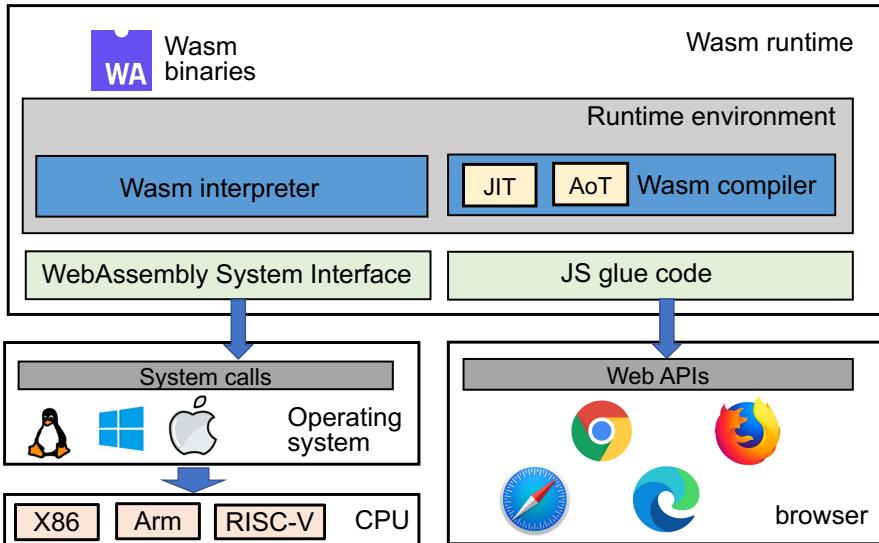


Fig. 6. The architecture of Wasm runtimes.

including Wasm module caching, validation of the Wasm textual file format, and more, such as wasmer.

3.2.1 Wasm interpreter. Wasm is designed as a low-level, cross-platform execution format that can be directly executed by an interpreter. A Wasm interpreter is a tool that interprets WebAssembly binary code instructions individually. One advantage of this approach is that there is no need to wait for a compilation process during loading, which speeds up startup time. However, interpreted execution is generally slower compared to native machine code. For example, WAMR [16] and wasm3 [10] provide the interpretation mode for executing Wasm binaries.

3.2.2 Wasm compiler. The other way to execute Wasm binaries is to first compile the Wasm bytecode into native code, and then execute the native code. There are two types of Wasm compilers: just in time (JIT) compiler and the ahead of time (AoT) compiler. In general, Wasm compilers convert Wasm binaries into their intermediate representation (IR), allocate registers and optimize the IR code to generate the native code. For example, wasmer [12] use three Wasm compilers, singlepass [7], cranelift [27] and LLVM [5]. This process should consider different OSes and CPU architectures. That is to say, the same Wasm binary file could be compiled into different native codes for different OSes and CPUs.

3.2.3 WebAssembly System Interface. Although Wasm was first proposed for the Web, it has already extended beyond it. However, when extending Wasm beyond the confines of the browser, a challenge arises. That is how to ensure that Wasm binaries can consistently interact with different OSes, as the Wasm binaries are compiled from different high-level languages. There is where WASI exists [18]. Specifically, WASI is a middle layer of Wasm runtimes and the underlying OSes. WASI unifies the system calls from various OSes. The specification of WASI [23] follows the modular principle, i.e., it allows runtimes to design and implement other modules [19] based on the foundational `wasi-core` module. The most necessary functions, including files, clocks, and random number generation, are

Table 2. The most popular Wasm runtimes.

Wasm runtime	standalone	interpreter	JIT	AoT	Stars
V8(Chrome/Edge)	○	○	●	○	22.2k
SpiderMonkey(Firefox)	○	○	●	●	241
WebKit-JavaScriptCore(Safari)	○	●	○	○	684
wasmtime	●	○	●	●	13.8k
wasmer	●	○	●	●	16.9k
WAMR	●	●	●	●	4.3k
WasmEdge	●	○	○	●	7.5k
wasm3	●	●	○	○	6.8k

* ● and ○ refers to the positive and negative value individually.

defined in the `wasmi-core` module. Now, almost all the standalone Wasm runtimes implement the WASI functions, including wasmtime [14], wasmer [12], WAMR [16], WasmEdge [11], etc.

3.3 Features

In order to meet the efficient and secure characteristics of Wasm, Wasm runtimes could include the following features [26, 84]: 1) **Safe**. A Wasm runtime is expected to provide a memory-safe, sandboxed environment for executing Wasm binaries, by memory bound checking, etc. 2) **Efficient**. As Wasm aims to execute at native speed, a Wasm runtime needs to be designed to be efficient and fast. 3) **Compatible**. Since Wasm is proposed as a general execution binary format, Wasm runtimes needs to be available on different platforms. 4) **Lightweight**. In order to start up at a fast speed and be compatible with resource-constrained platforms, a Wasm runtime needs to be lightweight.

3.4 Popular Wasm runtimes

This section illustrates the most popular Wasm runtimes. As shown in Table 2, according to whether a Wasm runtime could run outside the Web, the Wasm runtimes could be divided into two types: standalone Wasm runtimes and integrated Wasm runtimes.

3.4.1 Inside the Web. When executing Wasm binaries in the Web, Wasm binary files are expected to leverage Web APIs through JS code while supporting the security model in the Web. As shown in Table 2, the most widely used four browsers, including Chrome, firefox, safari, and Edge, all support executing Wasm binaries in their JavaScript engines. V8 is a JavaScript and Wasm runtime written in C++, used in Chrome, Edge browser and Node.js [31]. SpiderMonkey serves as Mozilla’s engine for JavaScript and WebAssembly, utilized in Firefox, Servo, and several other projects [30]. The implementation is done using C++, Rust, and JavaScript. WebKit-JavaScriptCore is the Apple’s JavaScript engine, and mainly served for iOS and macOS [33]. The Wasm runtimes mentioned above are the runtimes that support executing Wasm binaries on the client side of the Web. Moreover, on the Web’s server side, Wasm binaries are also supported. For example, Wasm binaries could be executed in Node.js on the server side [29].

3.4.2 Outside the Web. As shown in Table 2, we illustrate the five most popular Wasm runtimes in GitHub with the top stars. Wasmtime is a standalone Wasm runtime proposed by the Bytecode Alliance, designed for both WebAssembly and the WebAssembly System Interface (WASI) [14]. It executes WebAssembly code outside the Web, serving as a command-line utility and a library embedded in large-scale applications. It is a highly configurable and embeddable runtime that runs on applications of any scale. Wasmer is a fast and secure WebAssembly runtime that allows

lightweight containers to run anywhere: from desktop applications to the cloud, edge, and IoT devices [12]. Wasmer supports different compiler frameworks(LLVM [5], cranelift [27], singlepass [7]) and can generate native code through these Wasm compilers. Similar to wasmtime, wasmer also provides high-level language APIs for developers. The Wasm bytecode can be embedded into programs developed in other languages through these high-level language APIs. WAMR is a lightweight standalone Wasm runtime with high performance and highly configurable features [16], suitable for applications in various cases, including embedded systems, IoT, trusted execution environments(TEE), smart contracts, cloud-native, etc. WAMR supports all the execution modes: interpreter, JIT, and AoT. WasmEdge is a lightweight, high-performance, and extensible Wasm runtime [11]. It is currently used in edge clouds, serverless software as a service(SaaS) APIs, embedded functions, smart contracts, and intelligent devices. Wasm3 is a high-performance Wasm interpreter [10]. It is designed to be lightweight, focusing on the size of the runtime executable file and memory usage. This makes it suitable for embedded systems, IoT devices, and other resource-constrained environments.

4 ARTICLE COLLECTION AND RESEARCH QUESTION

A systematic literature review assesses and interprets all existing research pertinent to a specific subject area. Adhering to Kitchenham's established guidelines [72], we obey the subsequent review protocol in this section. This section outlines the survey scope, research questions, the methodology used for article collection, an initial analysis of the gathered articles, and the structure of our survey.

Survey Scope The scope of our article is the software used to execute Wasm binaries, whether on the Web or standalone. When gathering articles, we adhere to the subsequent inclusion criteria. It will be incorporated if an article meets one or more of the listed criteria.

- The articles design a Wasm runtime or add features to an existing Wasm runtime.
- The articles discuss or test the bugs, performance or security in a Wasm runtime.
- The article illustrates the application or future direction of Wasm runtime or analysis of the Wasm runtime.

4.1 Article Collection

We collect relevant published articles about Wasm runtimes to answer the above research questions. This section describes the article collection criteria. We first explore relevant articles by specifying keywords on widely used search engines and databases. Subsequently, we establish corresponding selection criteria to gather the conclusive articles for the literature review analysis. Figure 7 illustrates the detailed process of our methodology for article collection.

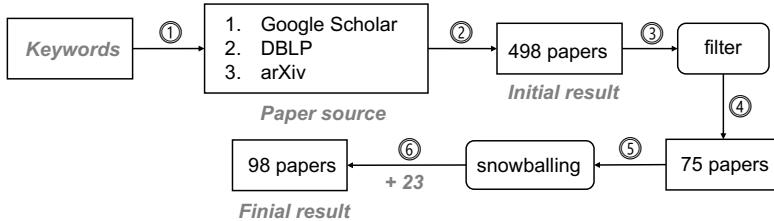


Fig. 7. The article collection process of our methodology.

4.1.1 Article Sources. Following the Zhang.etc’s work [148], we have chosen three popular scientific databases as article sources for the survey. These sources encompass (1) Google Scholar [22], (2) DBLP [21], (3) arXiv [20].

4.1.2 Keywords Searching. To collect the articles related to Wasm runtimes, we follow the previous work [59, 90, 143] to define search keywords and apply them to article sources. First, we include all equivalent names for the Wasm runtime, including runtime, virtual machine (VM), and engine. According to the Wasm runtime architecture [150], we include the terms to describe components of a Wasm runtime, including compiler, interpreter, and WebAssembly System Interface (WASI). Second, we consider the terms of Wasm runtime inside Web and outside the Web. We include the names of JS engines to execute Wasm binaries and names of Web browsers, including V8, SpiderMonkey, JavaScriptCore, chrome, firefox, safari, Edge, and browser. We include the names of the most popular Wasm runtimes: wasmtime [14], wasmer [12], WasmEdge [11], wasm3 [10], and WAMR [16]. Third, we also consider features and popular scenarios of Wasm runtime as the keywords, including security, bug, sandbox, trust, performance, IoT, serverless, blockchain, application, Web, embedded system, cloud, and mobile. Finally, we conducted $(2^*28 + 1 + 5)^*3 = 186$ searches across the three repositories before January 8th, 2024.

- Wasm|WebAssembly runtime|virtual machine|engine|VM
- Wasm|WebAssembly compiler|interpreter|WASI
- WebAssembly System Interface
- Wasm|WebAssembly V8|SpiderMonkey|JavaScriptCore
- Wasm|WebAssembly chrome|firefox|safari|Edge|browser
- wasmtime|wasmer|WasmEdge|WAMR|wasm3
- Wasm|WebAssembly performance|security|bug|sandbox|trust
- Wasm|WebAssembly IoT|serverless|blockchain|application|Web|embedded system|cloud|mobile

Specifically, we collect articles published between 2017 (the publication time of the paper introducing Wasm) and January 8, 2024. Moreover, we obtained 498 articles initially. To find articles that study Wasm runtimes, we only searched for articles whose titles contain these keywords.

4.1.3 Filtering. Although some articles may contain relevant keywords, they may be unavailable, repetitive, or not necessarily about Wasm runtime research. For example, they might be about the compilers that compile high-level languages, such as C in Wasm binaries, rather than the component in a Wasm runtime. Therefore, we have established filtering rules to exclude articles from the initial results. We illustrate the filter criteria process to select the relevant research articles about Wasm runtimes. An article is removed if it satisfies any of the following filter criteria: (1)The complete articles are not available; (2)It is not a research article, such as bachelor, master, or doctoral dissertations; (3)Books, blogs about Wasm runtimes; (4)Moreover, we removed duplicates of articles that appeared in different article sources or searched by different keywords; (5)It is not related to Wasm runtimes. Specifically, we check the remaining articles individually to select the articles related to Wasm runtime. To minimize the risk of erroneous exclusions, the first two authors independently reviewed the articles and discussed whether they were related to Wasm runtime. After the filter process, we obtained 75 relevant research articles.

4.1.4 Snowballing. Following the commonly adapted snowballing process used in other surveys [59, 87, 143], we apply this approach to each obtained article to increase the number of related articles. The snowballing approach includes a forward step and a backward step. These two steps refer to searching the references in each obtained article and other relevant articles from those that cite the obtained articles. We complement 23 articles to the list. Finally, 98 papers are included in the survey.

4.2 Publication Venues

To ascertain the publication venues of research articles on Wasm runtimes, we conduct searches for each article on Google Scholar to identify where they are published, such as conferences or journals.

As shown in Figure 8, we examine the publication venues of research articles utilizing the widely recognized Computer Science Rankings [28]. We also explore other significant publication outlets, such as those related to software engineering, database research, and more. Figure 8 also presents the publication venues' abbreviation names and corresponding full names. Research articles related to Wasm runtime have been published in 74 different venues. Among them, arXiv has the highest proportion, with 8.2% (8/98) of articles published in arXiv. What's more, 12.2% (12/98) of articles have been published in PACM PL, ASE, or USENIX Security. Additionally, 16 articles have been published in 8 venues, including IMC, Middleware, etc. Furthermore, the remaining 62 articles are published in different venues, such as PPoPP (ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming), TOSEM (ACM Transactions on Software Engineering and Methodology), USENIX ATC (USENIX Annual Technical Conference), demonstrating the richness of Wasm runtime-related research across publication venues.

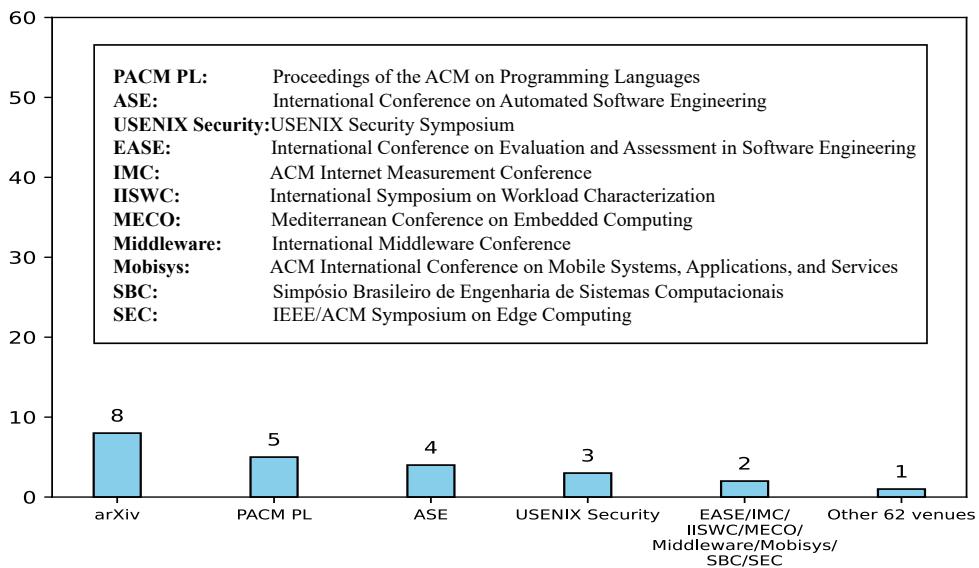


Fig. 8. The main distribution of publication venues for research articles about Wasm runtimes.

Figure 9 shows the distribution of papers published in different academic conferences or journals. Among all the papers, 41.8 percent papers are published in conferences and journals about systems and networks, including MobiSys (ACM International Conference on Mobile Systems, Applications, and Services) and INFOCOM (IEEE International Conference on Computer Communications); 19.4 percent of papers are published in software engineering venues, including ASE (International Conference on Automated Software Engineering), SANER (IEEE International Conference on Software Analysis, Evolution, and Reengineering), ICSE(International Conference on Software Engineering), and TOSEM (ACM Transactions on Software Engineering and Methodology). Besides, 9.2% of the articles are published in conferences and journals about security, such as USENIX Security

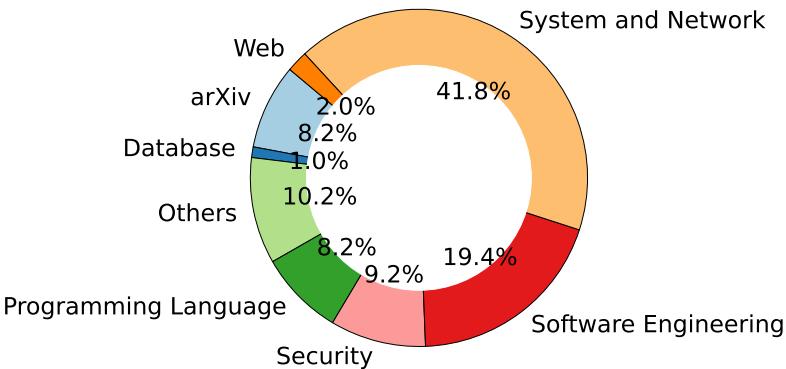


Fig. 9. The publication venue distribution.

(USENIX Security Symposium) and CCS (ACM Conference on Computer and Communications Security). There are small-part papers published in programming languages, databases, the Web, or other venues. Additionally, 8.2 percent of the papers have not been published in any conference or journal (the arXiv part).

4.3 Research Questions

To analyze various research directions related to the Wasm runtime, we propose two research questions from two angles.

RQ1 Application Scenario. (the "external" research)

What scenarios do the Wasm runtimes apply in? This research question explores the wide range of scenarios that Wasm could be used for and the role Wasm runtimes play.

RQ2 Research Direction. (the "internal" research)

What research directions do these articles focus on? This research question delves into the diverse avenues pursued by articles concerning Wasm runtimes. It comprises three sub-questions, delineated in Section 6.1.

5 RQ1: APPLICATION SCENARIO

Although Wasm was initially proposed for the Web, it has been extended to several scenarios. To answer the research question of which scenarios the existing studies apply Wasm runtimes in, the first authors read all the collected articles and classify the articles according to the application scenarios. This section elaborates on the application scenarios in which the Wasm runtimes are applied. Some articles apply Wasm runtimes in more than one scenario, so we calculated the data for each scenario when analyzing such papers. As shown in Figure 10, among the articles about Wasm runtimes, the most attention is focused on the general scenario, such as studying the performance of Wasm runtimes that can be universally applied to various scenarios. A significant number of articles also focus on the applications of Wasm runtimes in Web applications, edge computing, and IoT. For example, discussions on the potential and challenges of Wasm runtime in edge computing are prevalent. Some articles also focus on the scenarios of TEE, serverless computing, etc. Five articles focus on other scenarios, including databases, aviation systems, etc. The articles spanning multiple application scenarios are counted in all the scenarios to which they belong. These show the broad adoption of Wasm and Wasm runtimes. This section dived deep into these scenarios.

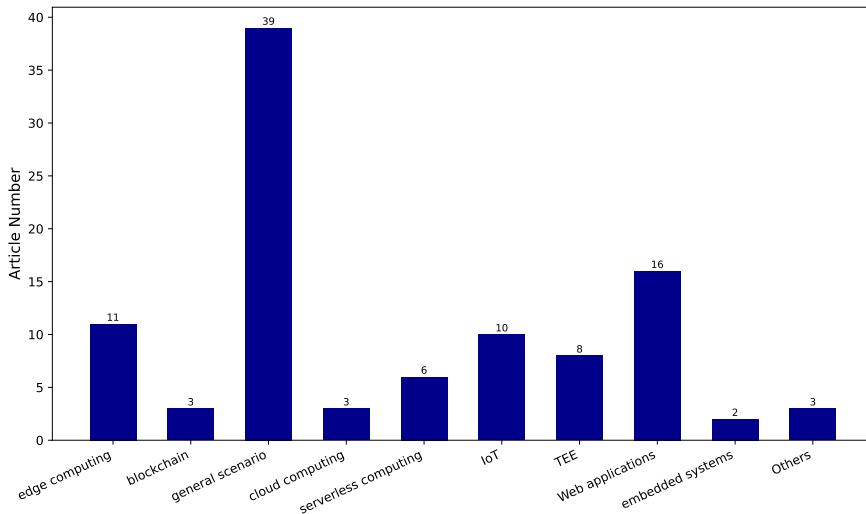


Fig. 10. The application scenarios of the articles.

5.1 General scenario

Most articles do not confine Wasm runtime to a specific scenario but instead explore Wasm runtime in a general, universal context. As shown in Figure 11, more than half of the articles design a new Wasm runtime or add features for existing Wasm runtimes in general scenarios. The other part focuses on bug testing, performance testing, etc.

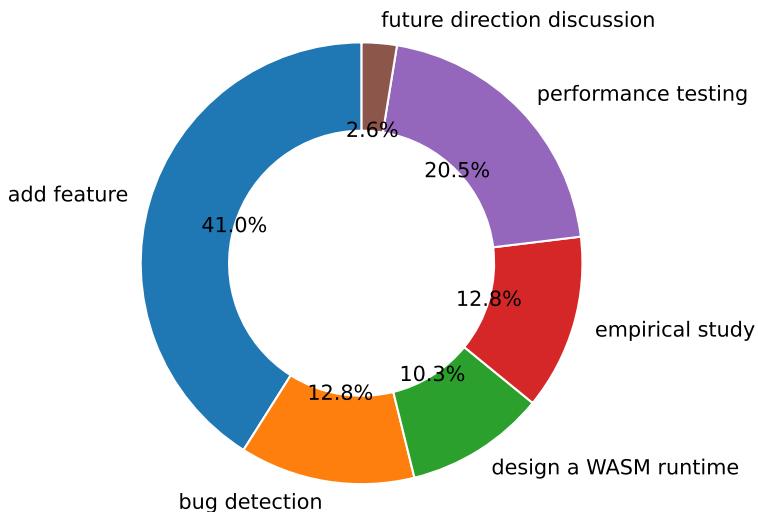


Fig. 11. The research purpose of the articles is for general scenario Wasm runtimes.

10.3% of the articles of general scenarios design a Wasm runtime. For example, Bosamiya et al. design a Wasm runtime with the highest priority goal of security guarantee. This Wasm runtime adheres to two principles: employing conventional formal methods to generate mathematically

verified safety proofs and integrating Wasm semantics into secure Rust code to generate executable code safely via the Rust compiler [40].

41.0% of the articles in general scenarios add features for the Wasm runtime. For example, Titzer et al. design and implement a fast in-place interpreter for Wasm, which could achieve faster startup and lower memory consumption compared to the previous Wasm runtimes [127]. Zhang et al. focus on linear memory protection in the Wasm runtime [151]. They first apply a canary approach in code and define two unique Wasm instructions with instrumentation in the Wasm binaries. Gurdeep Singh et al. broaden the Wasm VM to explore the viability of implementing Wasm on Arduino-compatible microcontrollers [56]. Specifically, they support 1) safe live code updates, 2) remote debugging at the VM level, and 3) programmer configurable modules to keep the Wasm runtime's footprint, enabling the Wasm runtime to have better performance than the interpreter and increasing the ease of development.

What's more, 20.5% of the articles of general scenarios test the performance of different Wasm runtimes. For example, Hockley et al. use micro-benchmarks and a macro-benchmark to compare the execution times in Wasm runtimes of Wasm binaries and the native code directly on the machine [62]. Specifically, they use wasmer [12] as the testing target, a standalone Wasm runtime that could be used in general scenarios. Ultimately, they discover a performance loss of 5-10 times in the Wasm runtime when compared to native execution. Szewczyk et al. focus on the performance consumption due to the Wasm's bounds checked memory access safety mechanism [126]. They extend four popular general scenarios of Wasm runtimes with modern bounds-checking mechanisms, comparing the performance of each with the execution of native code on three different CPU architectures.

There are also 12.8% articles that provide empirical studies for the general-purpose Wasm runtimes, including characterizing and detecting Wasm runtime bugs [136, 150], the issues met by the practitioners of Wasm runtime and application developers [138], characterizing of standalone Wasm runtimes through WABench [135], etc.

Moreover, there are other articles focusing on bug detection, such as WADIFF [154], WRTester [41], Wasmfuzzer [66]; some focusing on the discussion of the future directions, such as Joshi's work [69].

5.2 Web application

As shown in Figure 10, 16 of 98 articles about Wasm runtimes are in the Web application scenario, which occupies the second position. Wasm was first proposed for the Web, and this field continues to attract widespread research interest.

There are 62.5%(10/16) papers focusing on performance testing of Wasm execution on the Web. For example, Jangda et al. expand BROWSIX into BROWSIX-Wasm, facilitating the direct execution of unmodified Unix applications compiled to Wasm within the browser environment [65]. Moreover, they conduct the performance evaluation of Wasm vs. native through BROWSIX-Wasm. Because of inadequate optimizations and issues with code generation, the performance of Wasm code is reduced by approximately 50% in Firefox and 89% in Chrome, on average. Liu et al. find that while Wasm has the potential to enhance Web XR performance, a performance gap exists between Web-based XR and standalone XR environments. [88]. Through comparison between Wasm and JS in different browsers with the three types of benchmarks, Stotoglou et al. figure out which browser is suitable for which class of problems [122].

Besides, three articles dive deep into the energy consumption testing of Wasm on the Web, showing its consumption conservation potential. Pockstaller et al. find that the execution of Wasm could bring 20% to 30% consumption save than JS [113]. Moreover, the difference in consumption cost is also found between Firefox and Chrome. This study could help reduce carbon dioxide emissions. Hasselt et al. [129] and Macedo et al. [47] also find that Wasm could reduce the consumption cost on the Web and improve the battery life of Android devices.

Furthermore, there are 25.0%(4/16) of the articles about Web applications complement features for the Wasm runtimes [78, 107, 125, 139].

5.3 Edge computing

Edge computing occupies a small portion of the application scenarios(11.2%, 11/98) for Wasm runtimes. In this scenario, eight runtimes are fully designed or designed based on the foundation of another runtime. Sebrechts's runtime [119] is based on wasmtime [14]. They extend the WASI component in wasmtime, enabling Kubernetes controllers being executed as lightweight Wasm modules. The idle controllers will be swap into the disk and activated once they are needed. Finally, the evaluation shows that this runtime achieved a 64% reduction in memory usage. To satisfy the essential low latency requirement and the need for lightweight migration on edge services, Nieke et al. design a platform based on Wasm to support portable, secure, and provider-independent services on the edge [106]. Besides, different communication stacks also attract attention. The various communication stacks block the reuse of the services across edge computing systems. To tackle this issue, Kreutzer et al. propose a Wasm runtime to run such services as Wasm modules, which oversees the communication between the Wasm module and other components of the system [79].

Two groups of researchers focus on and discuss the future development direction of Wasm in edge computing scenarios. Hoque et al. explore the potential of utilizing Wasm for edge computing with a comprehensive view. They offer four possible approaches for future development for using Wasm to achieve migratability in edge computing, including code migration, interpreter-based, Wasm-instrumentation, and binary instrumentation [63]. Kakati et al. offer a systematic overview of the current state of Wasm used in edge computing, including performance optimization, interoperability, and security [71]. They hold the view that Wasm is an ideal resolution with a compact format and high-performance runtime environment for the tasks in edge computing.

5.4 Internet of things

In the application scenario of IoT, seven Wasm runtimes are designed fully or based on other runtimes for IoT devices, including Wasmachine [141], ThingSpire OS [83], Aerogel [89], Wiprog [82], WAIT [84], WOOD [42] and VM by Jacobsson et al [64].

Wasmachine could securely execute Wasm binaries in IoT and Fog devices, with 11% faster than Linux execution by ahead-of-time compiling and the zero-cost system calls in kernel mode. Wen et al. implement the OS kernel in Rust for the Wasm's memory safety and utilize the sandboxing features provided by the Wasm runtime [141]. Li et al. design an IoT operating system that uses AoT compilation to achieve seamless inter-module communication when executing Wasm binaries with levels of optimizations [83]. Aerogel strongly emphasizes access control, addressing the disparity between bare-metal IoT devices and Wasm runtimes [89]. The Wasm runtime is crafted as a multi-tenant platform, treating each Wasm application as a distinct tenant. The sandbox feature within the Wasm runtime plays a pivotal role in effectively managing access control for sensors or actuators [89]. Li et al. provide an integrated programming approach for programmers to easily develop IoT applications in Wiprog [82]. Programmers could utilize the API offered by Wiprog to access the IoT device operations and assign the placement of the operation by annotations. WAIT is designed as a secure and energy-efficient WebAssembly (Wasm) runtime. Similarly, to enhance execution efficiency, WAIT also employs Ahead-of-Time (AoT) compilation [84]. Differently, Jacobsson et al. devise a WebAssembly (Wasm) runtime specifically crafted for wearable devices, employing an interpreter-based approach to enable over-the-air programming through Bluetooth low energy [64].

Table 3. The Wasm runtimes designed for TEEs.

Wasm runtime	TEE architecture	Description
TWINE [97, 100]	Intel SGX	A secure and trusted version of SQLite within a TEE.
The runtime by Ménétrey et al. [96]	Different TEE architectures	A secure and attested pub/sub system.
PRIVATON [123]	Different TEE architectures	A runtime based on finite state automations.
The runtime by Pop et.al [114]	Arm TrustZone	An enclave software design based on Wasm runtime.
AccTEE [54]	Intel SGX	A two-way sandbox offering accounting of resource usage.
WATZ [98]	Arm TrustZone	A runtime for the efficient and secure execution of Wasm binaries.

In general, due to the specific nature of IoT devices, the primary considerations in the design of Wasm runtimes are performance and energy consumption. In addition to this, researchers have customized distinct Wasm binary execution strategies and interfaces for various devices.

5.5 Trusted execution environment

As shown in Table 3, most of the articles (7/8, 87.5%) about TEE application for Wasm runtimes focus on designing new WebAssembly (Wasm) runtimes entirely or based on existing Wasm runtimes for Trusted Execution Environment (TEE) environments, and only one article analyzes the security issues for extending Wasm runtimes in TEE.

Ménétrey et al. first designed a Wasm runtime based on the existing standalone Wasm runtime: WAMR [16] in TEE in 2021 [97] and evaluate the performance of the designed runtime in 2023 [100]. To implement a secure version of the SQLite, they extend the WASI component of WAMR to interact with the underlying system calls and secure libraries supported by Intel SGX. When TWINE was implemented in production with the company Credora, the results demonstrated a performance range from a 0.7x slowdown to a 1.17x speedup compared to the native runtime. The study shows that library optimization could improve the performance to a notable 4.1× speedup. Furthermore, they propose the first Wasm runtime for Arm TrustZone in 2022 [98]. WATZ is a trusted Wasm runtime with remote attestation capabilities, suitable for TrustZone and applicable to IoT. Moreover, they extended Wasm runtimes to various TEE architectures in 2023 [96]. They designed a sub/pub system in 2023, modifying the TLS protocol to introduce mutual authentication in network communications.

As for privacy preservation, TEE is a good choice, and some researchers combine TEE and Wasm runtimes for privacy preservation. Subramanyan et al. design a Wasm runtime for the general TEE architecture, utilizing the potential of TEE and Wasm to privacy-preserving computations [123]. PRIVATION lets users define a set of privileges under control, enabling explicit control over computation capabilities. Pop et al. extend the existing Wasm runtime Wasmi [13] to run services in TEE to protect the privacy-sensitive data [114].

As for the remote computation field, due to the trust gap between code providers and code executors, the computation is always executed in a sandbox. Goltzsche et al. provide a two-way sandbox Wasm runtime to account for the resource usage of remote computation [54].

5.6 Other application scenarios

Wasm runtimes are also applied in other scenarios, such as serverless computing [49, 74–76, 93, 152], blockchain [36, 146, 153], embedded systems [118, 133], aviation systems [147], etc.

Gackstatter et al. design a Wasm runtime, WOW, as a serverless container runtime, with the reduction of cold-start latency by 99.5%, suitable for various serverless computing workloads [49]. Kjorveziroski et al. evaluate the performance of Wasm execution in serverless computing [74]. They apply three widely used Wasm runtimes: wasmtime [14], wasmer [12] and WasmEdge [11] to compare the cold start delay and the total execution times, finding that Wasm runtimes perform faster on 10 of 13 tests compared to container runtimes. Moreover, wasmtime is the best.

The blockchain community also embraces Wasm, such as Ethereum, which favors the smart contracts to be executed as Wasm binaries on Ethereum 2.0 [2]. Zheng et al. compare the performance of smart contracts executed in Wasm or in the previous language used in Ethereum, Solidity, however, finding that executing smart contracts in Wasm is still far from ideal [153]. Yang et al. propose a tool called Seraph to automatically analyze the security issues for the platforms executing smart contracts in Wasm [146].

Moreover, it is promising that Zaeske et al. bring Wasm into aviation systems in 2023 [147]. They integrate the Wasm interpreter into the current avionics software stack, utilizing Wasm to implement the common application binary interface.

In general, Wasm shows significant application value in various fields, and the execution of these Wasm applications relies heavily on the design, updates, and analysis of the Wasm runtimes.

Summary of answers to RQ1:

- (1) Due to compactness, safety, and other features, Wasm has been widely used in different scenarios inside or outside the Web, including Web applications, edge computing, IoT, blockchain, embedded systems, serverless computing, etc. 39.8% (39/98) of the articles do not confine the runtime to a specific scenario, which could be used in a general scenario.
- (2) Wasm runtimes are fundamental to utilizing Wasm in various scenarios. They support the safe, efficient execution of Wasm binaries. High-quality Wasm runtime promotes broader applications of Wasm, and the widespread use of Wasm also imposes higher requirements on Wasm runtime. When introducing Wasm into a new scenario, the Wasm runtime needs to be modified to adapt to the new environment, considering the characteristics of the scenario.

6 RQ2:RESEARCH DIRECTION

6.1 Taxonomy and sub-questions

6.1.1 Taxonomy construction. As shown in Figure 12, we construct a taxonomy of the research directions of the Wasm runtime literature, following the methodology used in other taxonomy construction works [45, 143, 150]. No more than ten articles could focus on more than one direction; as for these articles, we assign them according to the primary purpose. For example, WAFL detects bugs [60]. However, the primary purpose is bug detection, with adding features to achieve the goal. Thus, this article is divided into the *bug detection* category. Our taxonomy includes three root categories of research directions in the grey boxes, such as *Wasm runtime design* and *testing*. Moreover, the number in the parentheses represents the specific data of articles in the corresponding categories. For example, 33 articles test the Wasm runtimes. Furthermore, the boxes in the white color represent the sub-directions under a specific research direction scope. For instance, the category *B. Wasm runtime testing* contains three sub-categories, including *performance testing*,

Table 4. The Publication Venues and their Covered Research Directions.

Publication venues	Covered research directions
arXiv	Adding features for a Wasm runtime, Bug detection, Empirical study, Security analysis, Performance testing
PACM PL	Adding features for a Wasm runtime
ASE	Performance testing, Bug detection
USENIX Security	Adding features for a Wasm runtime, Designing a complete Wasm runtime
EASE/IMC/IISWC/MECO/ Middleware/Mobisys/	Performance testing, Energy consumption testing, Empirical study, Adding features for a Wasm runtime, Discussion about future directions, Designing a complete Wasm runtime
SBC/SEC	Designing a complete Wasm runtime
Other 62 venues	All the leaf research directions.

*The publication venues are in abbreviation version.

energy consumption testing and *bug detection*. In total, there are 33 articles focusing on the testing of Wasm runtimes.

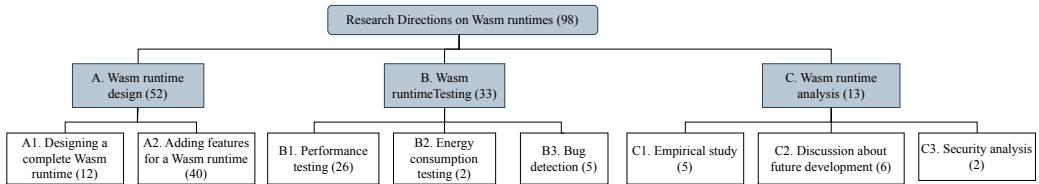


Fig. 12. A taxonomy of research directions in the Wasm runtime literature.

This sub-section will explain our taxonomy in detail. First, articles focusing on the design of Wasm runtimes account for 53.1% (52/98) of all the articles about Wasm runtimes, which occupies the most significant portion. This direction contains 23.1% (12/52) design of a complete Wasm runtime for different usage (*A1. Designing a complete Wasm runtime*), 76.9% (40/52) complement features based on fundamental Wasm runtimes (*A2. Adding features for a Wasm runtime*).

Second, in the taxonomy of Wasm runtime literature, *testing* takes the second portion of the articles. In this category, researchers mainly focus on *performance testing* of Wasm execution, which accounts for 78.8% (26/33). The high performance of Wasm is a significant highlight, and as a result, performance issues have captured enough interest among researchers. Furthermore, there are 6.1% (2/33) and 15.2% (5/33) articles that focus on the *energy consumption* and *bug detection* individually.

Third, our taxonomy shows that the analysis of Wasm runtimes accounts for the third largest percentage, i.e., 13.3% (13/98). About half of these studies (5/13) are empirical studies about Wasm runtimes. Furthermore, there are also 46.2% (6/13) articles discuss the future development directions, and 15.4% (2/13) analyze the security issues in Wasm runtimes.

As shown in Table 4, we summarize the main publication venues and the corresponding research directions. The articles in arXiv cover 62.5% (5/8) leaf categories, focusing on the diverse research directions of Wasm runtimes. The articles published in ASE mainly focus on the Wasm runtime testing, aligned with the scope of software engineering conferences. The articles published in other 62 venues cover all the leaf research directions.

6.1.2 Sub-questions. According to the constructed taxonomy, we propose four sub-questions for RQ2:

RQ2-1 **Wasm runtime design.** What components are modified or considered in designing a Wasm runtime? This sub-research question investigates the components and features (what to consider and how to design) in the articles designing Wasm runtimes.

RQ2-2 **Wasm runtime testing.** What benchmarks and methods are used in testing Wasm runtimes? This sub-research question investigates the benchmarks (how to build and the benchmark content) and methods (how to test and what to test) used in the articles testing Wasm runtimes.

RQ2-3 **Wasm runtime analysis.** What aspects do the empirical studies focus on? What directions do they propose for Wasm runtimes? What are the security issues they are concerned about?

6.2 RQ2-1:Wasm runtime design

Most researchers in the Wasm runtime field seek to contribute to the Wasm runtime itself. 23.1% (12/52) of researchers contribute to the overall runtime design, and 76.9% (40/52) introduce new features to the runtime. This section compares the Wasm runtime architectures and modified Wasm runtime components of the 52 related articles.

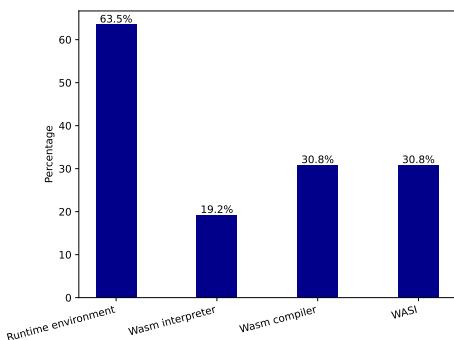


Fig. 13. The components modified or mainly considered when designing Wasm runtimes.

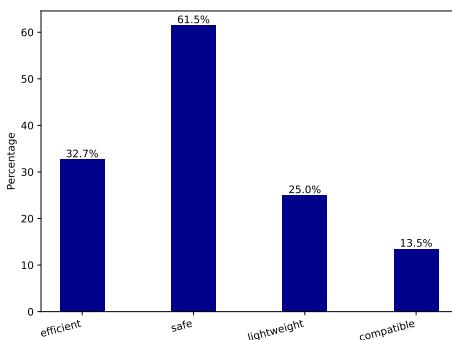


Fig. 14. The features utilized when designing Wasm runtimes.

Figure 13 illustrates the proportions of various components mainly considered or utilized in designing the Wasm runtime. Due to the consideration of multiple components in the design of some runtimes, these runtimes are included in the counts for different component proportions. More than half of the papers in the total 52 Wasm runtime designing articles (*Category A*) focus on the runtime environment, including linear memory protection [81, 120, 151], safety checking [53], workload scheduling [155], etc.

Figure 14 shows the percentages of Wasm runtime features utilized when designing Wasm runtimes. An article could be counted more than once since some leverage multiple features. The safety feature, encompassing memory safety, sandbox mechanisms, and more, is the most compelling aspect capturing researchers' attention. Over half of the 52 articles (*Category A.1*) primarily utilize or contribute to the safety feature of Wasm runtimes.

As shown in Table 5, the Wasm runtimes designed by the researchers are listed, including the runtimes designed completely and the runtimes designed by adding features. The content *original* in the fourth line means the runtime is designed thoroughly without being based on other runtimes.

Table 5. The runtimes designed by the researchers.

Wasm runtime	Components	Features	Based Runtime
Sledge [50]	Wasm compiler(AoT), runtime environment	safe, lightweight	original
TruffleWasm [116]	Wasm interpreter, Wasm compiler(JIT), runtime environment, WASI	safe, efficient	original
Wasmachine [141]	Wasm compiler(AoT), runtime environment, WASI	safe, efficient	original
Bosamiya et al.'s runtime [40]	Wasm compiler(AoT), runtime environment	safe, efficient	original
WAIT [84]	Wasm compiler(AoT), runtime environment, WASI	safe, lightweight, efficient	original
WATZ [98]	Wasm compiler(AoT), runtime environment, WASI	safe, efficient	original
WaVe [68]	runtime environment, WASI	safe	original
Jacobsson et al.'s runtime [64]	Wasm interpreter, WASI	lightweight	original
eWasm [110]	Wasm compiler(AoT), runtime environment	safety, lightweight	original
ThingSpire OS [83]	WASM compiler(AoT)	compatible, safe	original
WasmAndriod [142]	Wasm compiler(AoT), runtime environment	compatible	original
PRIVATION [123]	runtime environment, WASI	safe	original
WiProg [82]	Wasm interpreter, Wasm compiler(JIT), runtime environment, WASI	compatible	wasmer, wasm3
AccTee [54]	runtime environment	safe	V8 engine
TWINE [97, 100]	WASI	lightweight, safe	WAMR
Aerogel [89]	runtime environment	safe	WAMR
Titzer et al.'s runtime [127]	Wasm interpreter	efficient	Wizard
Pop et al.'s runtime [114]	Wasm interpreter	compatible	WASMI
ContractBox[77]	WASI	safe	WAMR
metaSafer [120]	runtime environment	safe	JS VM
WasmRef-Isabelle [140]	Wasm interpreter	efficient	wasmtime
PKUWA [81]	runtime environment	safe	wasmtime
Moron et al.'s runtime [102]	Wasm compiler(JIT)	efficient	wasm3
MPIWasm [44]	WASI, runtime environment	lightweight, safe, compatible	wasmer

Zhao et al.'s runtime [152]	runtime environment	safe	WAMR
Ménétrey et al.'s runtime [96]	runtime environment	safe	TWINE
Sebrechts et al.'s runtime [119]	WASI	lightweight	wasmtime
Mäkitalo et al.'s runtime [91]	runtime environment, Wasm compiler(JIT)	efficient	wasmtime
WASP [94]	Wasm interpreter	safe	Wasm interpreter [32]
Gu et al.'s runtime [55]	Wasm compiler(JIT)	safe	wasmtime
Watt et al.'s runtime [139]	runtime environment	safe	V8 engine
Gobi [105] Wasm-precheck [53]	WASI runtime environment	safe safe	wasmtime wasmtime
Nießen et al.'s runtime [107]	Runtime environment	efficient	V8 engine
Kolosick et al.'s runtime [78]	Wasm compiler(AoT), runtime environment	safe, efficient	Lucet
LOWOW [155]	runtime environment	lightweight, efficient, compatible	WasmEdge
Nomad [108]	Wasm interpreter	compatible	wasm3
Abbadini et al.'s runtime [34]	WASI	safe	WasmEdge, wasmtime
WOW [49]	runtime environment	lightweight	wasmtime, wasmer, WAMR
Swivel [104]	Wasm compiler(AoT)	safe	Lucet
Szanto et al.'s runtime [125]	runtime environment	safe	JS VM
VMCANARY [151]	c safe	wasmtime	
WARDuino [56]	runtime environment	efficient	The runtime by Joel Martin [43]
Wasm/k [112]	Wasm compiler(JIT)	efficient	wasmtime
WOOD [42]	runtime environment	lightweight	WARDuino
Zaeske et al.'s runtime [147]	Wasm interpreter	compatible	Wasmi
Kjorveziroski et al.'s runtime [75]	runtime environment	efficient	wasmtime
EDGEDANCER [106]	runtime environment, WASI	efficient, lightweight	WAMR
Nakakaze et al.'s runtime [103]	Wasm interpreter	lightweight	wasm3, V8
CWASI [93]	runtime environment	safe	WasmEdge
Kreutzer et al.'s runtime [79]	runtime environment	safe	wasmer

*The second column refers to the components mainly modified in the runtime.

*The third column refers to the main contributed designed features.

6.2.1 Designing a complete Wasm runtime. To study the design of a complete Wasm runtime and what features of the Wasm runtime are primarily considered or utilized, this section is organized by the features mainly focused on.

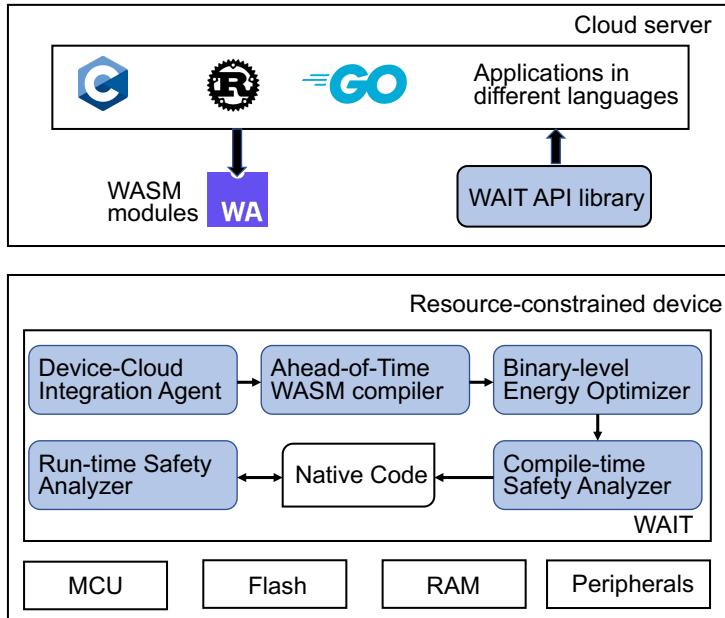


Fig. 15. The workflow of WAIT [84].

83.3%(10/12) of the completely designed Wasm runtimes mainly consider the safety feature. Bosamiya et al. design a safe Wasm runtime without sacrificing efficiency by guaranteeing the sandbox's safety with safe compilers through formal validation [40]. Li et al. propose the first Wasm runtime, WAIT, on resource-constrained devices, with safety, efficiency in memory, and energy as the design goal [84]. As shown in Figure 15, WAIT offers an API library consisting of IoT APIs for users to develop applications in different languages. Then, the applications are compiled into Wasm modules to be executed in WAIT. WAIT uses AoT compilation to ensure the fast speed of executing Wasm binaries and uses an energy optimizer to save energy. WAIT applies the compile-time and run-time analyzer to check the safety. Johnson et al. ensure the safety of the proposed Wasm runtime, WaVe, by contributing to the WASI component [68]. They systematically ensure that WASI, which interacts with the operating system, preserves memory safety and upholds the isolation of access to underlying resources. Peach et al. utilize the isolation of Wasm runtimes and design a Wasm runtime, eWasm, for microcontrollers [110]. Subramanian et al. design a Wasm runtime, PRIVATION, for general TEE architectures to utilize the safety feature [123].

33.3%(4/12) of the wholly designed Wasm runtimes mainly utilize the lightweight feature. Gade-palli design a complete runtime, Sledge, utilizing the lightweight Wasm runtime to execute multi-tenant serverless functions on resource-constrained edge systems [50]. Sledge uses AoT compiling

to convert Wasm binaries into LLVM IR by the Wasm compiler, which is implemented in more than 4,000 lines of Rust code inside Sledge and executes multiple untrusted modules in one process. Jacobsson et al. utilize the lightweight feature of Wasm runtime on wearable devices [64].

Half(6/12) of the entirely designed Wasm runtimes mainly consider the efficiency feature. Unlike other Wasm runtimes running on OSes, Wasmachine itself is an OS kernel that runs on bare-metal machines and could be faster than native code [141]. Wasmachine runs Wasm modules in the sandboxed kernel thread, invoking system calls as normal functions in Ring 0, which reduces the performance overhead caused by WASI in other Wasm runtimes. Salim et al. design a Wasm runtime, TruffleWasm, to execute Wasm binaries by the interpreter and JIT optimization, which is the first runtime to execute Wasm binaries in JVM [116]. Ménétréy et al. propose an efficient and safe Wasm runtime, WATZ for the TEE [98]. Li et al. propose an IoT OS, ThingSpire OS, based on an originally designed Wasm runtime with a contribution to the Wasm compiler(AoT) component to ensure the execution speed [83].

8.3%(1/12) of the fully completely designed Wasm runtimes primarily concentrate on the compatibility feature. To be compatible on different Andriod platforms, Wen et al. propose a new Wasm runtime, WasmAndriod [142]. Android developers could directly compile source code into Waasm binaries and execute them on various hardware platforms without configuration.

6.2.2 Adding features for a Wasm runtime. This section is organized by the components modified in the Wasm runtime. Figure 16 illustrates the foundational Wasm runtimes used for adding features in the 40 articles in *Category A.2*. Due to the consideration that multiple runtimes could be used in one article, these articles are included in the counts for different Wasm runtimes. As the Wasm runtime supported by the bytecode alliance, wasmtime [14] and WAMR [16] attract the attention of half of the researches in *Catgory A.2*. Moreover, wasmtime is the most widely used runtime. What's more, 82.5% of the researchers use standalone Wasm runtimes, including wasmtime, wasmer, WAMR, WasmEdge, wasm3, Lucet and Wasmi, the JS VM, such as V8 engine also attracts 15.0% of the researchers.

57.5%(23/40) of the articles in *Category A2* contribute to the runtime environment of a Wasm runtime. Goltzsche et al. design a Wasm runtime, AccTEE, for remote computation, based on the Node.js runtime [54]. As shown in Figure 17, AccTEE is applied inside the Intel SGX enclave, which is a kind of TEE (trusted execution environment), and utilizes the two-way sandbox of the Wasm runtime to protect the execution process and data due the untrust between infrastructure providers and workload providers. By mainly contributing to the runtime environment component, AccTEE computes the resource usage based on three factors: 1) the number of Wasm instructions, 2) the size of the allocated memory, and 3) the number of I/O operations. Liu et al. utilize the sandbox feature of the Wasm runtime, WAMR, which is used as a multi-tenant environment [89]. The optimization of Wasm introduces safety issues related to linear memory. Specifically, it introduces the possibilities by modifying metadata containing memory structure details. Malicious users could utilize heap memory overflow in executing Wasm applications to access arbitrary memory addresses and perform various malicious operations. To ensure the safety of executing Wasm binaries, Song et al. propose that the metaSafer shadow metadata from Wasm linear memory into JS virtual machines and perform metadata verification at a fast speed [120]. With the similar purpose of Song et al. to protect Wasm linear memory, Lei et al. propose the first Wasm runtime with memory isolation, PKUWA, based on wasmtime [14]. Unlike Song et al., who shadow the metadata in linear memory, PKUWA protects the linear memory at function-level with trivial memory cost and small execution speed cost [81]. Zhang et al. also focus on the safety of linear memory during Wasm execution [151]. They present a framework, VMCANARY, to protect the memory using the canary approach. Zhao et al. introduce WAMR [16] into serverless computing

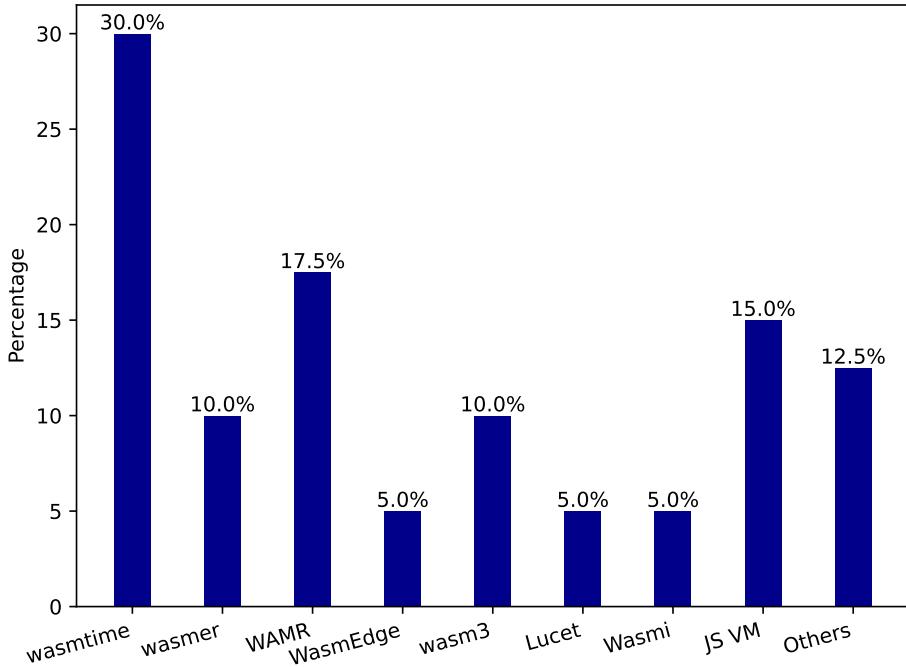


Fig. 16. The fundamental Wasm runtimes used to add features.

based on the isolated environment the runtime provided and modified the runtime library [152]. Ménétrey et al. update the Wasm runtime, TWINE, with the modification of two trusted primitives of attestation, including generate and verify, to extend TWINE into various TEEs [96]. Geller et al. design a Wasm runtime, Wasm-precheck, based on wasmtime [14]. They first extend the Wasm specification to a superset and then use Wasm-precheck to do additional static reasoning instead of the dynamic safety checks, which improve the performance of the Wasm runtime [53]. Nießen et al. design a Wasm runtime based on V8 engine [31] for the Node.js applications with the shared code cache among multi-process to improve the performance [107]. Zhu et al. propose a lightweight task scheduling framework for WasmEdge [11] to offload vital Wasm binaries to the edge, with the reduction of memory consume [155].

22.5%(9/40) of the articles in *Category A2* contribute to the WASI part of a Wasm runtime. As shown in Figure 18, Ménétrey et al. design TWINE, a lightweight Wasm runtime based on WAMR [16], with the modification of the original WASI component to translate the WASI operations to systems calls or TEE libraries [97, 100]. In order to protect the data being accessible outside the enclave, TWINE maps the file operating to the Intel-protected file system and utilizes the sandbox nested in the TEE. Kohn et al. utilize the safety feature provided by the Wasm runtime's sandbox to protect the host system from being influenced by smart contracts from malChadhaicous vendors. Moreover, they extend the API part to interact with the underlying storage system [36]. Chadha et al. propose the first Wasm runtime, MPIWasm, for HPC applications based on wasmer [12, 44]. MPIWasm extends the WASI part to provide the interaction layer between Wasm binaries and the underlying host1 MPI library. Sebrechts et al. extend the WASI component in wasmtime [14] to enable their runtime being used as Kubernetes controllers [119]. Abbadini et al. propose a permission-checking

framework with eBPF programs for the WASI part in different Wasm runtimes [34]. Wasm runtimes are generally expected to perform security checks to prevent accessing arbitrary locations through host calls in WASI. However, this approach is error-prone and is poor at granularity. Abbadini et al. replace the traditional checking with eBPF programs at fine-grained for each module [34].

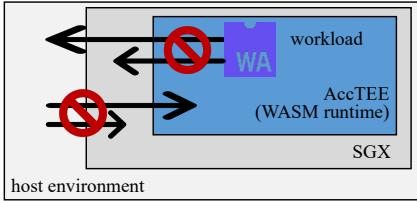


Fig. 17. Overview of AccTEE [54].

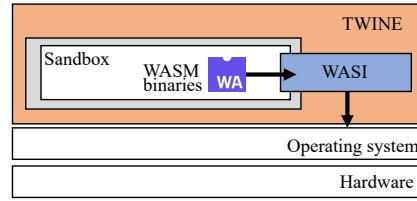


Fig. 18. The architecture of TWINE [97].

17.5%(7/40) of the articles in *Category A2* contribute to Wasm compiler of a Wasm runtime. In order to solve the two issues: 1) the code of the IoT applications cannot immigrate seamlessly in heterogeneous platforms, and 2) the code could not be fully portable, Li et al. utilize the compatibility feature of a designed Wasm runtime, WiProg. And they utilize the JIT compiler in wasmer and the interpreter in wasm3 [82] to execute Wasm binaries. Moron et al. introduce the JIT compiling into wasm3 [10] to alleviate the efficiency limit caused by interpreter [102]. Mäkitalo et al. optimize the implementation of dynamic linking Wasm modules based on wasmtime [14] to improve the startup speed [91]. Gu et al. provide a compiler verifier based on the JIT compiler in wasmtime [14, 55]. Kolosick et al. modify the Wasm compiler in Lucet [6] to remove the performance cost of the systems relying on sandboxing mechanism in Lucet [78]. Narayan et al. propose a compiler framework to protect Wasm from Spectre attacks based on Lucet [6, 104].

17.5%(7/40) of the articles in *Category A2* focus on the Wasm interpreter. In contrast to other runtimes like TruffleWasm [116], Wasmachine [141], and WAIT [84], Titzer et al. employ a space-efficient and rapid in-place Wasm interpreter to prioritize efficiency. This choice is made considering that for JIT compilers, both compilation time and memory usage profoundly impact application startup time [127]. Pop et al. extend the Wasm interpreter of WASMI by adding serialization, deserialization pausing, etc., for the secure enclave service migration [114]. Watt et al. propose a monadic fast Wasm interpreter based on wasmtime [14] to be used as a fuzzing oracle [140]. Marques et al. extend the Wasm interpreter [32] with symbolic facilities to analyze the execution of Wasm binaries [94]. Nurul-Hoque et al. develop an interpreter architecture for wasm3 [10] to separate the runtime state from the host environment, enabling wasm3 supporting live-migration [108].

Summary of answers to RQ2-1:

(1) More than half of the papers related to Wasm runtimes design a Wasm runtime entirely or based on existing Wasm runtimes, which shows that the design and utilization of Wasm runtimes currently attract the attention of the majority of researchers. Instead of designing an entire Wasm runtime, most researchers design a new runtime based on the existing Wasm runtimes, with wasmtime being the most used one.

(2) The majority of researchers primarily consider the runtime environment component when designing Wasm runtime, with the Wasm interpreter being the least considered component. This is mainly because the runtime environment component encompasses all functionalities besides Wasm instruction execution and the Wasm System Interface, such as memory allocation and resource usage calculation. Furthermore, only a minority of Wasm runtimes support execution through interpreters. Hence, the Wasm interpreter is the least considered component.

(3) The safety feature is researchers' most commonly utilized characteristic of Wasm runtimes, followed by the efficient feature. The potential of Wasm applications can be explored in various scenarios by designing Wasm runtime and ensuring its safety and efficiency.

6.3 RQ2-2:Wasm runtime testing

Instead of directly contributing to the design of the Wasm runtime itself, 33.7% (33/98) of researchers contribute to testing Wasm runtimes, promoting the development of the runtime by identifying shortcomings.

6.3.1 Performance testing. As shown in Figure 19, we summarize the performance and energy consumption testing workflow of Wasm runtimes. First, the researchers construct the benchmark in different languages and then compile the benchmark into Wasm binaries or native code. Second, the compiled benchmarks are deployed in different execution environments, including Wasm runtimes, JS engines, and native environments. For example, directly compiling high-level language benchmarks into native code and deploying them directly in a Linux environment. Finally, the researchers collect and analyze the execution result.

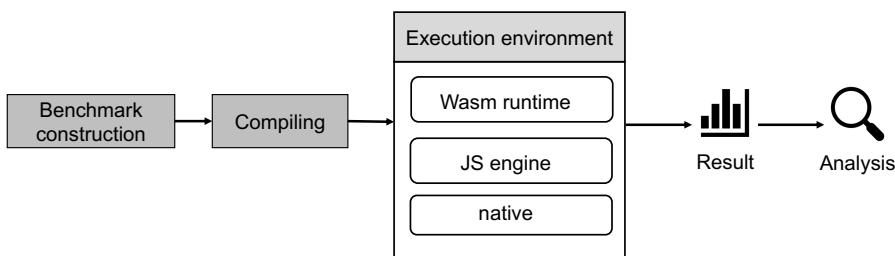


Fig. 19. The general workflow of performance and energy consumption testing.

Table 6. The benchmarks used when testing Wasm runtimes.

Benchmark	Usage	Description
-----------	-------	-------------

WABench [135]	performance testing	A benchmark contains 50 programs from JetStream2, MiBench, PolyBench, and the manually built part.
Ostrich Benchmark Suite [109, 129]	performance testing, energy consumption testing	A test suite comprises 12 algorithms with the same implementation in JS and C.
Wasm-MB and Wasm-BM [35]	performance testing	The WebAssembly Micro benchmarks (WASM-MB) and WebAssembly Benchmarks (WASM-BM) are often used in Wasm performance testing and cover various use cases.
SPEC benchmarks [62, 65]	performance testing	A test suite of micro-benchmarks and a macro-benchmark, both written in Rust.
Fibonacci benchmark [101]	performance testing	A program to calculate the Fibonacci of a number.
Prime benchmark [101]	performance testing	A program to calculate whether a given number is prime or not.
Stotoglou's benchmark [122]	performance testing	A test suite includes numerical computation, classification and image processing.
Benchmark.js [80]	performance testing	A benchmark library with high-resolution timer support, providing statistically significant results.
XR-related framework or library [88]	performance testing	A test suite consists of tasks of Web XR.
PolyBench C benchmark [121, 126, 134, 137]	performance testing	A test suite compiled from 30 widely-used C benchmarks.
Mendki's benchmark [95]	performance testing	A test suite includes different applications, such as compute-intensive tasks, memory-intensive tasks, file I/O intensive, and a simple image classification - machine learning application.
Part of the Computer Language Benchmarks Game (CLBG) [46, 132]	performance testing, energy consumption testing	A test suite implements ten computational problems in up to 26 languages. The authors use 3 of the 10.
Macedo's benchmark [46]	performance testing, energy consumption testing	A test suite mainly consists of sorting computation.
LLVM test suite [67].	performance testing	A test suite contains various programs to evaluate the performance of LLVM compilation.
The SuiteSparse Matrix Collection [117]	performance testing	A set of sparse matrix benchmarks from real life.
Yan's benchmarks [145]	performance testing	The test suite consists of three parts: 1)41 widely used C benchmarks, 2)9 hand-written JS programs, and 3)3 real-world programs in Wasm and JS.

Scheidl's benchmarks [118]	bench-	performance testing	A test suite consists of computing algorithms from two sources.
Zheng's benchmarks [153]	bench-	performance testing	A test suite contains smart contracts from the real world or manually written.
Kjorveziroski's benchmarks [74]	bench-	performance testing	A test suite from micro-benchmarks and real world programs.
Pham's benchmarks [111]	bench-	performance testing	A benchmark consists of compute-intensive workloads centered around computing Message Digest Method 5 (MD5) hashes.
Junior's benchmarks [70]	bench-	performance testing	A test suite contains digital image processing applications.
Macedo's benchmarks [47]	bench-	performance testing, energy consumption testing	Mirco benchmarks form C, and real-world benchmarks are built from Wasm-Boy and PSPDFKit.
Rosetta Code platform [113]	plat-	energy consumption testing	A website dedicated to programming exercises providing implementations of common algorithms in various programming languages.
WasmFuzzer's suite [66]	test	bug detection	Wasm programs generated automatically by WasmFuzzer.
WADIFF's suite [154]	test	bug detection	Wasm binaries generated based on the Wasm specification.
WRTester's suite [41]	test	bug detection	A test suite consists of Wasm programs by disassembling and assembling real-world Wasm binaries.

Benchmark. As shown in Table 6, we summarize the benchmarks used for testing Wasm runtimes.

Some articles apply existing benchmarks to test the performance of Wasm runtimes by compiling the high-level programs in existing benchmarks into Wasm binaries. Ostrich Benchmark Suite consists of 12 algorithms [109], such as computing the optimal alignment of two protein sequences, computing n queen problem, calculating particle potential and relocation within a 3D space, etc. Both the C and JS versions in the same implementation are provided. SPEC benchmarks were first proposed by Jangda et al. in 2019 [65] for the Web. Hockley et al. extended SPEC benchmarks to the non-Web environment in 2022 [62]. Stotoglou et al. build the benchmark based on similar works considering the availability and compatibility with three types of tasks [122]. Kyriakou et al. utilize Benchmark.js, a benchmarking library [80]. In order to meet the need of testing Web XR performance, Liu et al. introduce three XR-related tasks, including loading and rendering 3D content, sampling images, and tracking objects [88]. Wang et al. compile 30 C programs in the PolyBench C into Wasm binaries [134]. Moreover, Mäkitalo et al. and Wang et al. [137] also apply the PolyBench to compare the performance among different Wasm runtimes [91]. Wagner et al. introduce part of the computer language benchmarks as it has programs written in different languages with the same implementation [132]. Macedo et al. choose sorting programs from the programming chrestomathy repository, Rosetta Code [46]. Two intensive benchmarks compatible

with WebAssembly are incorporated, namely fannkuch-redux and fasta, sourced from the Computer Language Benchmarks Game (CLBG). Jiang et al. introduce the LLVM test suite [67]. They select 141 C/C++ programs from the various benchmarks for testing LLVM compilation. Sandhu et al. collect close to 2,000 test cases from the SuiteSparse Matrix Collection and classify the benchmarks into seven categories according to the number of non-zeros [117]. Besides, Pham's and Junior's benchmarks [70, 111] are also extracted from existing benchmarks.

The rest part of the articles use the benchmarks built by the authors. The Fibonacci benchmark contains the program of calculating Fibonacci sequences of a number [101]. The sequence is generated by adding the previous two numbers, which is a good choice for testing recursive calls. The prime benchmark is used in the same article by Tushar et al. The program needs to run a loop to judge whether the given number is prime, which is a good choice for testing iterative programs. Mendki develops a test suite that includes different applications written in Rust [95]. Inside, the compute-intensive tasks are derived from the wasmer test suite. Yan et al. build the benchmarks in three ways: hand-writing, real-world collection, and extracting from existing benchmarks. They write 9 JS programs and collect three Wasm binaries and JS applications from the real world [145]. Scheidl et al. build the benchmarks from two source [118]. Some benchmarks are randomly selected from PolyBench C, including nussinov, floydwarshall, jacobi1d, and correlation. The rest of the algorithm programs are written by themselves, including Fibonacci sequence computation, Mandelbrot set calculating, and the Pollard Rho integer factorization algorithm. Zheng et al. build 13 smart contracts manually, containing four kinds of operations: simple operations, arithmetic operations, block status-related operations, and hashing operations [153]. Moreover, they also collect 13 Solidity smart contracts from the real world. Besides, Kjorveziroski et al. also select benchmarks from the real world [74]. Macedo et al. build the benchmarks from C benchmarks and the real world, including the WasmBoy benchmark and PSPDFKit Benchmark [47].

Method. In this section, we compare the methods used in performance testing of Wasm runtimes. As shown in Figure 20, the methods the articles used are collected. The most widely used method is to compare the performance of Wasm, JS, and native code. As Sandhu et al.'s [117], Mohan et al.'s [101], Spies et al.'s [121] and Wang et al.'s [135] work cover two methods, they are calculated in both methods.

38.5% (10/26) of the articles in performance testing of Wasm runtimes focusing on comparing Wasm and JS. As Wasm was first proposed to improve the performance of JS, whether the execution of Wasm is indeed superior to JS remains the main focus of attention. Oliveira et al. explore Wasm's capability to enhance JS's performance [109]. They use the Ostrich Benchmark Suite, which provides programs with the same functionality in both C and JS. Thus, the Wasm counterpart could be generated from C. Finally, they compare the execution time of JS and Wasm in 8 of 12 algorithms in the benchmark, as the rest cannot be compiled into Wasm binaries due to compatibility. All the execution is on Raspberry Pi. In all cases, Wasm is faster than JS, especially in reducing battery-power consumption. And same result is found by Macedo et al [47]. Macedo et al. also find that Wasm performs better in the performance and energy consumption in most cases, and the larger the program input, the more pronounced the advantage of Wasm [46]. As sparse matrix-vector multiplication is a representative computation of compute-intensive programs, Sandhu et al. use sparse matrices to test the performance [117]. Yan et al. perform the testing based on comprehensive types of applications, including real-world programs, hand-written programs, and programs from other existing benchmarks [145]. Kyriakou et al. conducts stress tests to compare the performance of JS and Wasm [80]. In Web browsers, Wasm in a single thread performs 2-4x faster than the pure JS implementation. Tushar et al. compare the performance of Wasm and JS on Chrome and Firefox [101]. The Wasm binaries are compiled from the C, Rust, and Go versions of calculating the Fibonacci sequence of a given starting number. Wasm performs better than JS when the iterating

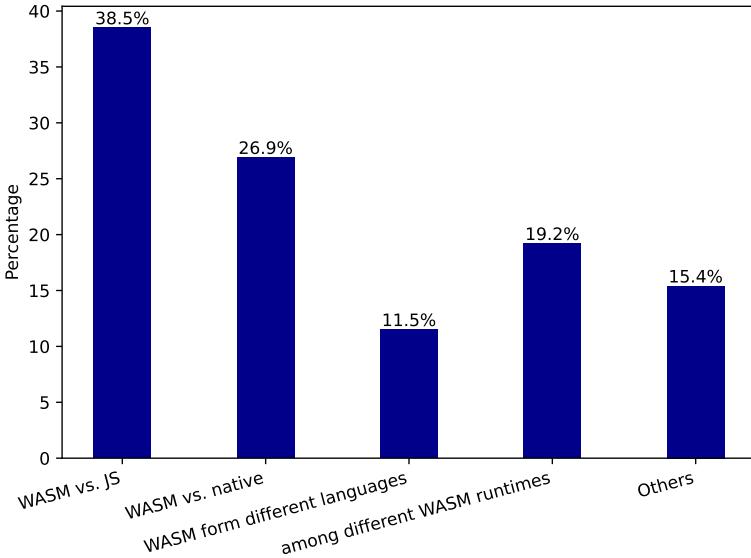


Fig. 20. The method percentage in performance testing of Wasm runtimes.

of Fibonacci becomes more extensive, and Wasm in Firefox is faster than in Chrome. Compared to the Wasm binaries compiled from Rust or Go, Wasm binaries from C/C++ perform better. Through comparing the execution of JS and Wasm browsers, Wang studied how the browsers optimize the execution process of Wasm binaries [134]. However, although the JIT optimization could accelerate JS significantly, it fails for Wasm. Moreover, Wasm consumes much more memory than JS, which is a good opportunity for the developers to improve. Stotoglou et al. test the performance of Wasm and JS in different browsers to compare the performance of different types of tasks across different browsers [122]. Wasm is found to be faster in all types of tasks. Performance largely depends on the input data, algorithms, hardware, and the browser's version. However, different browsers have their own optimization strategies; the V8 engine usually performs the best.

Moreover, 26.9% (7/26) of the articles compare Wasm and native code, marking it as the second significant focus. Mendki et al. find that Wasm runtime is not as fast as expected, hindering the application of Wasm in serverless computing [95]. And Szewczyk et al. investigate the performance impact arising from the bounds-checking mechanisms in WebAssembly [126]. Jangda et al. first propose the SPEC benchmarks by extending BROWSIX [65], ensuring the running of the Unix programs in Wasm without modifications. Second, they offer an automatically detailed timing collection and hardware performance counting tool, BROWSIX_Wasm, to conduct fine-grained performance testing. Third, they conduct the first comprehensive performance analysis of Wasm in 2019, finding a significant gap between Wasm and native code. The reasons are: 1) Wasm instructions entail a higher frequency of load and store operations; 2) Wasm instructions encompass more branches owing to safety checks; and 3) Wasm results in increased L1 instruction misses due to the higher volume of instructions. Junior et al. verify the promising future of Wasm as it performs similar performance of native code [70].

11.5% (3/26) of the articles focus on the Wasm binaries' performance compiled from different high-level languages. Aliyev et al. use the Web Assembly Micro benchmarks (Wasm-MB) and Web Assembly Benchmarks (Wasm-BM) benchmark suites to test the runtimes's performance [35]. Their

testing environment is oriented towards cloud providers, including Google Cloud Platform (GCP), Amazon Web Services (AWS), and Microsoft Azure. They believe that the execution time between 15ms and 20ms is a good performance, while the execution time between 20ms and 30ms is poor. Moreover, the Wasm runtime performs well in their view. Wagner et al. utilize three of the ten algorithms from the Computer Language Benchmarks to assess the impact of various high-level languages, including C, Rust, Go, and JS, on Wasm execution performance [132]. They find that C and Rust could be better choices to save energy in IoT devices, and different Wasm runtimes exhibit preferences for different high-level languages.

19.2% (5/26) articles dive into the performance in different Wasm runtimes. Hockley et al. compare the performance with different Wasm compilers in wasmer [12, 62]. They use wasmer to execute the SPEC benchmarks (micro and macro benchmarks written in Rust). After comparing the performance of three Wasm compilers (singlepass, cranelift, LLVM) used in wasmer, they find that cranelift and LLVM perform better, since basically, singlepass does not perform any optimization. Although LLVM performs better than cranelift in some test cases, LLVM costs a longer-time JIT process than cranelift with a more complex compilation. Jiang et al. first propose a testing approach, WarpDiff, based on the differential testing framework [67]. That is to say, for each test case, the execution ratio of different Wasm runtimes obeying a fixed value is considered to be normal, and the execution time that does not obey a fixed ratio is considered to be anomalous, which needs to be further analyzed. Second, they compare the performance of five standalone Wasm runtimes, including wasmer [12], wasmtime [14], WAMR [16], WasmEdge [11] and wasm3 [10] on more than 100 test cases extracted from the LLVM test suite. Third, seven performance issues are detected, all confirmed by the runtimes' developers. Scheidl et al. proposed an optimization through valent-blocks and getting performance improvement over other Wasm runtimes [118]. Kjorveziroski et al. compare the performance of wasmtime [14], wasmer [12] and WasmEdge [11] to judge which is the best in serverless computing [74].

The rest 4 articles use other methods. Zheng et al. compare the performance of smart contracts in Solidity and the counterpart in Wasm binaries, finding that the overhead introduced by the gas metering and switch of EEI (Ethereum environment interface) methods constrain the performance of Wasm smart contracts in eWasm engines [153]. Pham et al. commend Wasm runtimes as a resolution of IoT applications, especially Wasmedge [11, 111]. Sunarto et al. do not directly run Wasm binaries to compare the runtimes' performance. Instead, they compare the performance, memory usage, and energy usage between Wasm and JS of the website development [124]. They use the PRISMA methodology to systematically review the papers comparing Wasm and JS performance. After analyzing no more than 30 works, they find that, in energy consumption, Wasm performs better; however, in memory usage, JS performs better. As for performance, Wasm and JS perform differently in various tasks. Wasm shows faster speed in lightweight applications and numerical calculations, while JS shows rapid speed in heavy applications. Several reasons could influence these results, including devices, browsers, etc. Liu et al. explore how Wasm could improve Web XR [88]. Targeting the XR-related framework or library, they did the first systematic study comparing the performance improved by Wasm in Web XR on five browsers. Although a performance gap exists between Web XR improved by Wasm and the standalone XR, Wasm does accelerate XR in Web.

Result analysis. By summarizing the results of various papers, we find that Wasm outperforms JS regarding performance and energy consumption [46, 47, 80, 101, 109, 122]. However, the performance of Wasm compiled from different high-level languages varies [101, 132]. For instance, C/C++ performs better than Rust and Go. Additionally, Wasm tends to consume more memory than JS [134]. However, the performance comparison between Wasm and native is not quite satisfactory [65, 95]. This is mainly due to the increased number of load and store instructions and safety checks in Wasm [65]. Nonetheless, the performance gap between Wasm and native can still be

narrowed [70]. Different Wasm runtimes in browsers or standalone show different performances, such as different optimization strategies [122], and exception time consumption [67].

6.3.2 Energy consumption testing. For devices with limited battery capacity, such as smartphones, executing complex web applications can be challenging, making Wasm a potentially better choice. There are 6.1% (2/33) of the articles mainly test the energy consumption of executing Wasm in the runtimes.

Hasselt et al. first test the energy consumption in 2022 [129]. As shown in Table 6, they also use the Ostrich Benchmark Suite, the same as Oliveira et al. [109]. They compare the energy consumption between Wasm and JS on mobile devices with Firefox and Chrome. Wasm saves more energy than JS, and Firefox is better than Chrome. Pockstaller et al. also compare JS and Wasm in energy consumption in 2023 [113]. They select 19 algorithms from the Rosetta Code platform, each providing implementations in both C and JS. The results are similar to those of Hasselt et al., where JavaScript (JS) also performs better, with specific energy savings ranging from 20% to 30%.

6.3.3 Bug detection. The Wasm runtime significantly influences the correctness and security of Wasm binaries' execution. About 15.2% (5/33) of the articles on Wasm runtime testing focus on bug detection in Wasm runtime.

As shown in Figure 21, the researchers to detect bugs first generate seeds based on the Wasm specification or randomly and then mutate the seeds. Second, they deploy the generated Wasm binaries into different Wasm runtimes to execute. Finally, they compare the results from different Wasm runtimes and analyze or locate the bug.

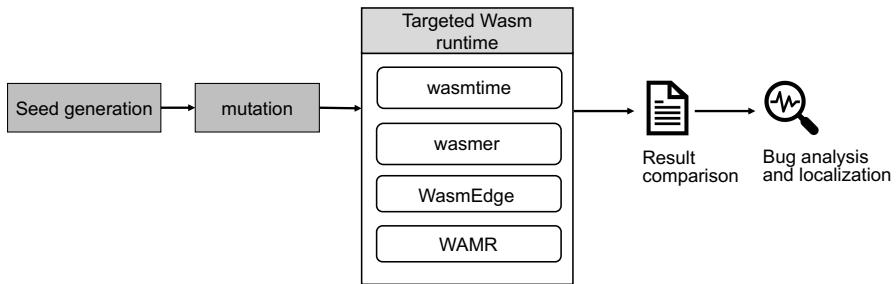


Fig. 21. The general workflow of bug detection.

Jiang et al. develop a fuzzing tool, WasmFuzzer, for Wasm runtimes [66] with automatically generated seeds and mutation operator at the binary level, performing better than AFL (American fuzzy lop) [1]. As different Wasm runtimes show different bug patterns and have various implementations, the mutation strategy in WasmFuzzer is adaptive to update the priorities of the mutation operators dynamically. Specifically, the mutation operators could be divided into two categories: 1) operators on Wasm instructions, including the instructions' insert, delete, and swap, and 2) other operators, including the add and delete of start, memory sections, etc. Finally, they apply WasmFuzzer and AFL in three Wasm runtimes: WAMR [16], WAVM [15] and EOS-VM [3], which are easier for code instrumentation and computing code coverage. WasmFuzzer performs better than AFL both on unique crashes and code coverage. Haßler et al. modify the Wasm runtime, WAVM [15], to calculate the coverage data for AFL++ Fuzzer and implement lightweight snapshots for the runtime to help detect bugs [60].

Unlike Jiang et al.'s code coverage guided method, Zhou et al. move their eyes on the Wasm specification [154]. They introduce the Wasm runtimes' differential testing framework, WADIFF,

to test seven Wasm runtimes and analyze the bugs' root causes. First, they convert the Wasm specification in natural language into structured DSL format. Second, they design a symbolic execution to generate Wasm binaries. Finally, they apply WADIFF into seven Wasm runtimes, including WAMR [16], wasmer [12], WasmEdge [11], etc. Twenty-one bugs are detected, with seven confirmed, resulting from various issues such as different feature support, undefined code, illegal alignment, etc.

Cao et al. devise a runtime-independent root cause identification algorithm that accurately pinpoints bugs using a static instrumentation approach to detect issues at both function and instruction levels [41]. Unlike WADIFF generating Wasm binaries based on Wasm specification, the differential testing framework WRTester Cao et al. used produces syntactically correct, semantically rich Wasm programs by disassembling and assembling real-world Wasm binaries. Finally, 33 unique bugs are detected with 25 confirmed, due to memory check, type conversion, etc.

Unlike the methods used in the above three articles, which can be applied in various runtimes and are regarded as general methods, VanHattum et al. narrow the focus to the lowering process of instructions within a Wasm compiler, cranelift [130]. They present a framework Crocus by adding concise semantics alongside definitions in ISLE to verify the lowering rules and find two unknown bugs and three found bugs, one of which is a CVE.

Summary of answers to RQ2-2:

- (1) Currently, testing of Wasm runtimes primarily focuses on performance, bugs, and energy consumption, with performance testing occupying the majority.
- (2) Researchers choose to build the benchmark themselves or compile the existing benchmarks into Wasm binaries for testing Wasm runtimes. These benchmarks include several kinds of algorithms, such as Fibonacci or some calculations from the real world.
- (3) Currently, the methods for testing the performance of Wasm runtime are relatively limited. They primarily involve comparisons between different Wasm runtimes, the execution of Wasm binaries compared to JS, and the execution of Wasm binaries compared to native code, among other methods. Despite outperforming JS in performance, Wasm's performance compared to native code often falls short than expected. Moreover, bug detection for Wasm runtime primarily relies on fuzzing and the principles of differential testing.

6.4 RQ2-3:Wasm runtime analysis

There are 13.3% (13/98) that analyze the security issues and the future directions or conduct an empirical study of Wasm runtimes.

6.4.1 Empirical study. 5.1% (5/98) of the articles apply empirical studies on the Wasm runtimes.

Two articles focus on the positive aspects of Wasm runtimes, including the promising features or usage cases. Titzer provide an empirical study of the single-pass compilers in Wasm runtimes [128]. Kjorveziroski et al. focus on the practicality of using Wasm runtimes in serverless by describing the current developments [76].

The other three articles focus on the issues that hinder the development of Wasm runtimes. Zhang et al. [150] and Wang et al. [136] provide empirical studies for bugs in Wasm runtimes, both constructing taxonomy for the bugs. Besides, Zhang et al. dive deep and summarize the fix strategies and propose a pattern-based bug detection framework based on the statistics of reported bugs and find 60 unknown bugs, with 13 confirmed and 9 fixed. By analyzing the reported issues from GitHub and Stack Overflow, Waseem et al. find that 28.16% of the issues that Wasm application developers encountered are introduced by Wasm runtimes [138].

6.4.2 Discussion about future directions. 6.1% (6/98) of the articles provide insights and discussions on the future development directions of Wasm runtimes. These papers are forward-looking studies in the application domains of Wasm runtimes, presenting ideas for various development directions.

Wasm runtimes provide a portable execution environment for the execution of Wasm binaries, which attracts the researchers' attention. Mäkitalo et al. proposed a vision using WebAssembly (Wasm) to implement lightweight containers for the IoT domain in 2021 [92]. This leverages the portability features of Wasm, allowing the code of IoT applications to be easily deployed and migrated across any IoT device in the system. Hoque et al. also utilize the compatibility and portability of Wasm and propose four proposals for using Wasm runtimes in Edge offloading computing [63]. Ménétréy et al. proposed using Wasm to solve the incompatibility between large data centers and user devices [99].

Besides, others focus on Wasm's high efficiency. Wallentowitz utilizes the high efficiency and isolation of the runtimes provided for embedded systems [133]. They demonstrate Wasm's viability and discuss its challenges, such as application management, minimum memory requests conflicting with embedded system resource constraints, etc. Joshi analyzes how to use Wasm outside the Web, including in docker, cloud, microservers, etc [69]. Kakati et al. regard the Wasm as an ideal resolution for applications in edge computing due to its compact format and the high-efficiency execution environment [71].

6.4.3 Security analysis. There are 2.0% (2/98) articles focus on the security issues in Wasm runtimes. Yang et al. analyze the security issues in the Wasm runtimes that execute smart contracts in Wasm by the tool Seraph [146]. Seraph develops an abstraction of the interaction layer to the underlying blockchain platform and describes the security issues by the symbolic semantic graph. Puddu et al. execute the confidential code as compiling to Wasm binaries in WAMR [16] in the TEE environment, finding that TEE leaks most IR instructions (Wasm binaries here) with the revealing of the confidential code, questing the commercial resolutions of TEEs+Wasm [115].

Summary of answers to RQ2-3:

- (1) Researchers primarily focus on both the promising aspects and the obstacles hindering the development of Wasm runtimes. This includes assessing the practicality of Wasm offered by these runtimes, identifying the challenges introduced by Wasm runtimes that Wasm application developers encounter, and categorizing the types of bugs present in Wasm runtimes.
- (2) The researchers also propose promising future applications, such as using Wasm runtimes as lightweight containers in IoT and applying them to address compatibility issues between large data centers and user devices.
- (3) Ensuring the security of Wasm runtime is crucial. Despite its widespread adoption, researchers have questioned existing solutions for Wasm runtime security by analyzing security issues in smart contracts and TEE environments.

Summary of answers to RQ2:

The existing articles on Wasm runtime research mainly focus on three aspects: the design of Wasm runtime, testing of Wasm runtime, and analysis of Wasm runtime, with the design of Wasm runtimes for various scenarios taking the main proportion.

7 FUTURE RESEARCH DIRECTIONS

This section discusses problems that exist in current articles and the future directions.

7.1 Wasm runtime design

53.1% of the articles explore the design of Wasm runtimes. Currently, the design of Wasm runtime has covered various fields and has gradually matured, but there are still areas for improvement.

As for the **Wasm runtime design for a specific scenario**, we provide the following advice:

1) As to applying Wasm into an embedded system, the minimal memory request of 64KB conflicts with the restrict resource limitation formulated in embedded systems [133]. The Wasm runtime designers and the specification formulator could provide the configurable requests of the linear memory to adapt to this scenario.

2) As to applying Wasm runtimes to executing smart contracts, especially in Ethereum, although the execution of instructions is high-performance, the interaction between Wasm binaries and the blockchain through the EEI methods hinders the high efficiency [153]. To improve the Wasm runtimes' performance, Wasm runtime designers could pay attention to the optimization of WASI and other counterparts, such as EEI methods, by trading off the performance and tedious security and sandbox checks.

What's more, as for the **Wasm runtimes in various environments**, almost all the articles need to design a Wasm runtime completely and modify the existing runtimes to adapt to a new environment [44, 78, 84, 110, 123]. That is mainly due to the fixed WASI specification and Wasm runtime architecture. We advise decoupling different components in Wasm runtimes thoroughly so that researchers can combine the components they need to form a new Wasm runtime rather than modifying more than one part of an existing Wasm runtime or designing a new one [44, 78, 84, 110, 123], such as design a standalone component to check the bounds of linear memory which could be applied to various Wasm runtimes as a hot plugging component.

7.2 Wasm runtime testing

Compared to the design and utilization of Wasm runtimes, testing for Wasm runtimes is not yet fully matured, thereby presenting numerous opportunities for improvement.

Although there are more than 20 articles focused on the performance testing of Wasm runtimes, there is still some space to improve, and we have some suggestions for **Wasm runtime performance testing** in the future:

1) *Methodology*. The existing forms of performance testing for Wasm runtimes are relatively monolithic, typically involving the design of benchmarks followed by running them under different runtimes and configurations to compare execution times [101, 109, 117, 122, 145]. Only Kyriakou et al. [80] considered stress testing. In the future, consider incorporating other methodologies that could help, such as concurrency testing, endurance testing, scalability testing, etc.

2) *Benchmark*. Many benchmarks for testing are pretty simple, with few test cases, and may not be representative [35, 109]. It might be beneficial to consider incorporating more representative Wasm benchmarks for performance testing in the future. For instance, the use of WasmBench [61], containing more than 8,000 Wasm programs from the real world, could be considered.

3) *High-level language*. The majority of Wasm test cases are compiled from C [47, 101, 109, 134, 145], and some are from Rust [65] and Go [101]. However, there is a lack of Wasm compiled from other high-level languages, such as Python and Java. The choice of languages may impact the execution performance of Wasm. Constructing datasets with equivalent functionality in different high-level languages is also challenging. Furthermore, while there are performance tests for Wasm compiled from various high-level languages, there is no fundamental analysis of why different

runtimes may exhibit preferences for Wasm programs compiled from different high-level languages. Additionally, there is a lack of analysis regarding the impact of different high-level languages on the size of Wasm files. Currently, there is a lack of a comprehensive test evaluating the influence of high-level languages on Wasm binaries and Wasm runtimes.

4) *Performance optimization.* While Jiang et al. have compared the performance of different runtimes, at most, it has only identified specific functions causing performance losses without analyzing the specific reasons or proposing optimization measures [67]. In the future, optimizing the performance of Wasm runtimes is a promising approach.

5) *Envirionment.* As Wasm's primary advantage of compatibility [26], it allows for varying CPU architectures(m), diverse OS platforms(n), and different runtimes(q), enabling the creation of $m \times n \times q$ types of testing platforms.

Currently, the **bug detection in Wasm runtimes** mainly uses black or grey box methods that could be applied to various runtimes [41, 66, 154]. The bug location by Cao et al. only pinpoint which line of the Wasm instructions cause the bug, rather than finding which line of source code in Wasm runtimes cause the bug. Introducing the white box method could detect and locate unknown bugs with domain knowledge in different Wasm runtimes, etc. Designing test generators for different components in the Wasm runtimes could help, such as proposing the test generator for WASI based on the WASI specification, which the current articles do not cover, etc.

7.3 Directions for different stakeholders

Besides designing or testing Wasm runtimes, there could be more directions to study for the stakeholders in the Wasm community.

For the **developers of the Wasm specification**, a more structured, decoupled Wasm specification would facilitate efficient development of Wasm runtimes. This would eliminate the need for extensive modifications at the source code level or create a runtime from scratch, promoting structured development and composition of Wasm runtimes. Additionally, as part of the Wasm specification, the imprecise description of the WASI specification has posed challenges to developing Wasm runtimes [150]. It would be better to clarify the WASI specification instead of relying solely on POSIX functions to represent the functionality of WASI functions, as is currently the case.

For the **Wasm runtime developers**, it is crucial to strictly adhere to the requirements of the Wasm specification in designing Wasm runtimes. Otherwise, executing the same program on different Wasm runtimes may yield different results, potentially causing issues such as calculation errors [41], inability to reach consensus in blockchain [153], and so forth. In addition, ensuring the security of Wasm runtimes is also crucial, which is fundamental to why many users choose Wasm runtimes. Therefore, Wasm runtime developers need to ensure the integrity of the Wasm runtime sandbox mechanism and the security of linear memory, especially in multi-tenant scenarios. Wasm runtime developers should also strive to optimize the runtime's performance, which is particularly important in resource-constrained environments such as IoT and edge computing.

For the **testers of Wasm runtimes**, many aspects can be improved, such as methods, benchmarks, and so on. Testing of Wasm runtimes is still immature. Only by clearly and thoroughly testing and analyzing the issues in existing Wasm runtimes can we properly guide the design direction of future Wasm runtimes.

For the **users of Wasm runtimes**, they have already deployed Wasm runtimes in a wide range of scenarios, fully leveraging its lightweight, secure, and other features. Users can consider various factors when selecting the appropriate Wasm runtime for their needs, such as performance, compatibility with hardware devices in the scenario, and memory consumption of the runtime. Although some Wasm runtimes claim to be compatible with multiple platforms, the reality may

differ. Users can refer to existing research results to choose the suitable platform rather than blindly trusting the official promotion of Wasm runtime.

8 RELATED WORK

There has been a large number of surveys focusing on different technology areas, such as serverless computing [48, 85, 143], machine learning [37, 131, 144, 148], software testing [38, 52, 86], etc. For example, Wen et al. presented a comprehensive overview of the existing literature to characterize the state of serverless computing [143]. They collected and analyzed the 164 articles on 17 research directions about serverless computing and derived the research trends and focus. Zhang et al. presented a comprehensive survey of the methodologies used to test machine learning (ML) systems based on the collected 144 papers. Moreover, they discussed the trends of datasets and research hotspots and proposed promising future directions in ML testing [148]. However, although Wasm runtime plays a significant role in the Wasm community, there is no survey about Wasm runtime research. We provide the first comprehensive survey of WSAM runtime research, following the methodologies used in the above surveys.

Currently, there are two surveys about Wasm, focusing on articles related to Wasm binaries [58, 73]. Harnew et al. analyze techniques for Wasm binaries [58], while Kim et al. analyze literature on methods used for studying Wasm binary security [73]. Our work presents the first comprehensive literature survey focusing on Wasm runtimes, aiming to contribute to the Wasm community.

9 THREATS TO VALIDITY

This comprehensive was conducted according to the established guidelines to alleviate potential threats to validity [59, 90, 143, 148]. However, there are still certain limitations, including the article searching and selection.

One primary threat is that the keywords used for paper searching may have limitations. For instance, some paper titles may not explicitly reflect any connection with Wasm but might involve Wasm runtimes. In our approach, there is a possibility of missing the collection of these articles. However, to gather papers as comprehensively as possible, the search keywords were jointly proposed by the first three authors, all of whom have over three years of research experience in the field of Wasm.

Another primary threat is the selection of articles. To filter out papers related to Wasm runtime from the initially collected 498 articles, we excluded a portion of them, which may have led to some papers being erroneously filtered out. To minimize this possibility and retain papers closely related to Wasm runtime as much as possible, we established strict criteria for paper selection. We only filtered out articles that were duplicates, unavailable or did not analyze or modify Wasm runtime in any way.

10 CONCLUSION

We provide a comprehensive outline and deeply analyze the research articles about Wasm runtimes. We first presented the definition, architecture, and current article state of Wasm runtimes. Second, we illustrated how the researchers designed a Wasm runtime entirely or added features based on existing Wasm runtimes. Third, we presented the benchmarks, methods, and results of testing Wasm runtimes and illustrated how researchers detect bugs in Wasm runtimes. Finally, we discussed the problems in current articles and proposed future research directions for Wasm runtimes and the whole Wasm community. We hope this survey can help Wasm runtime researchers, developers, users, and the developers of Wasm applications become familiar with the current development state and simultaneously provide research opportunities.

REFERENCES

- [1] 2014. American fuzzy lop. <https://github.com/google/AFL>.
- [2] 2018. Ethereum 2.0. <https://medium.com/rocket-pool/ethereum-2-0-76d0c8a76605>.
- [3] 2022. EOS VM - A Low-Latency, High Performance and Extensible WebAssembly Engine. <https://github.com/EOSIO/eos>.
- [4] 2022. hera - an ewasm (revision 4) virtual machine implemented in C++ conforming to EVMC ABIv9. <https://github.com/ewasm/hera>.
- [5] 2022. LLVM-a WASM compiler used in wasmer. <https://github.com/wasmerio/wasmer/tree/fc7c89fb1bfec332d9f26238740e14c1df605cd/lib/compiler-llvm>.
- [6] 2022. Lucet - a native WebAssembly compiler and runtime. <https://github.com/bytocodealliance/lucet>.
- [7] 2022. singlepass-a WASM compiler used in wasmer. <https://github.com/wasmerio/wasmer/tree/fc7c89fb1bfec332d9f26238740e14c1df605cd/lib/compiler-singlepass>.
- [8] 2022. WABT: The WebAssembly Binary Toolkit. <https://github.com/WebAssembly/wabt>.
- [9] 2022. WASM runtime definition. <https://medium.com/wasm/webassembly-wasm-runtimes-522bcc7478fd>.
- [10] 2022. wasm3 - The fastest WebAssembly interpreter, and the most universal runtime. <https://github.com/wasm3/wasm3>.
- [11] 2022. WasmEdge Runtime. <https://github.com/WasmEdge/WasmEdge>.
- [12] 2022. wasmer - a fast and secure WebAssembly runtime. <https://github.com/wasmerio/wasmer>.
- [13] 2022. wasmi - WebAssembly (Wasm) Interpreter. <https://github.com/paritytech/wasmi>.
- [14] 2022. wasmtime - A standalone runtime for WebAssembly. <https://github.com/bytocodealliance/wasmtime>.
- [15] 2022. WAVM - a WebAssembly virtual machine, designed for use in non-browser applications. <https://github.com/WAVM/WAVM>.
- [16] 2022. WebAssembly Micro Runtime. <https://github.com/bytocodealliance/wasm-micro-runtime>.
- [17] 2022. WebAssembly specification 2.0. <https://www.w3.org/TR/wasm-core-2/>.
- [18] 2022. WebAssembly System Interface. <https://wasi.dev/>.
- [19] 2022. WebAssembly system interface Doc. <https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/>.
- [20] 2023. arXiv. <https://arxiv.org/>.
- [21] 2023. dblp: computer science bibliography. <https://dblp.org/>.
- [22] 2023. Google Scholar. <https://scholar.google.com/>.
- [23] 2023. WASI preview1 documentation. <https://github.com/WebAssembly/WASI/blob/0ba0c5e2e37625ca5a6d3e4255a998dfa3efc52/phases/snapshot/docs.md>.
- [24] 2023. Wasm non web usage. <https://webassembly.org/docs/non-web/>.
- [25] 2023. Wat file. https://developer.mozilla.org/en-US/docs/WebAssembly/Text_format_to_wasm.
- [26] 2023. WebAssembly org. <https://webassembly.org/>.
- [27] 2024. cranelift-a WASM compiler used in wasmer. <https://github.com/wasmerio/wasmer/tree/fc7c89fb1bfec332d9f26238740e14c1df605cd/lib/compiler-cranelift>.
- [28] 2024. CSRankings: Computer Science Rankings. Retrieved Feb 21, 2024 from <https://csrankings.org/#/index?all&us>
- [29] 2024. Node.js. <https://nodejs.org/en/learn/getting-started/nodejs-with-webassembly>.
- [30] 2024. SpiderMonkey-The Firefox JavaScript engine. <https://spidermonkey.dev/>.
- [31] 2024. V8-The Chrome JavaScript engine. <https://github.com/v8/v8>.
- [32] 2024. WebAssembly interpreter. <https://github.com/WebAssembly/spec/tree/master/interpreter>.
- [33] 2024. WebKit-JavaScriptCore-The Safari JavaScript engine. <https://github.com/phoboslab/JavaScriptCore-iOS?tab-readme-ov-file>.
- [34] Marco Abbadini, Michele Beretta, Dario Facchinetti, Gianluca Oldani, Matthew Rossi, and Stefano Paraboschi. 2023. POSTER: Leveraging eBPF to enhance sandboxing of WebAssembly runtimes. (2023).
- [35] Elmir Aliyev. 2023. Analysis Performance of Web Assembly Applications on Cloud. In *1st INTERNATIONAL CONFERENCE ON THE 4th INDUSTRIAL REVOLUTION AND INFORMATION TECHNOLOGY*, Vol. 1. Azərbaycan Dövlət Neft və Sənaye Universiteti, 23–25.
- [36] Lennart Almstedt, Kai Blecke, Mohammad Mahhouk, Leander Jehl, Rüdiger Kapitza, and Lars Wolf. 2023. ContractBox: Realizing accountable data sharing on the edge using a small scale blockchain. *Computer Networks* 229 (2023), 109768.
- [37] Tadas Baltrušaitis, Chaitanya Ahuja, and Louis-Philippe Morency. 2018. Multimodal machine learning: A survey and taxonomy. *IEEE transactions on pattern analysis and machine intelligence* 41, 2 (2018), 423–443.
- [38] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2014. The oracle problem in software testing: A survey. *IEEE transactions on software engineering* 41, 5 (2014), 507–525.
- [39] Rafael Belchior, André Vasconcelos, Sérgio Guerreiro, and Miguel Correia. 2021. A survey on blockchain interoperability: Past, present, and future trends. *ACM Computing Surveys (CSUR)* 54, 8 (2021), 1–41.

- [40] Jay Bosamiya, Wen Shih Lim, and Bryan Parno. 2022. {Provably-Safe} multilingual software sandboxing using {WebAssembly}. In *31st USENIX Security Symposium (USENIX Security 22)*. 1975–1992.
- [41] Shangtong Cao, Ningyu He, Xinyu She, Yixuan Zhang, Mu Zhang, and Haoyu Wang. 2023. WRTester: Differential Testing of WebAssembly Runtimes via Semantic-aware Binary Generation. *arXiv preprint arXiv:2312.10456* (2023).
- [42] Carlos Rojas Castillo, Matteo Marra, Jim Bauwens, and Elisa Gonzalez Boix. 2021. WOOD: Extending a WebAssembly VM with Out-of-Place Debugging for IoT applications. (2021).
- [43] Walter Cazzola and Mehdi Jalili. 2016. Dodging unsafe update points in java dynamic software updating systems. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 332–341.
- [44] Mohak Chadha, Nils Krueger, Jophin John, Anshul Jindal, Michael Gerndt, and Shajulin Benedict. 2023. Exploring the Use of WebAssembly in HPC. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 92–106.
- [45] Zhenpeng Chen, Huihan Yao, Yiling Lou, Yanbin Cao, Yuanqiang Liu, Haoyu Wang, and Xuanzhe Liu. 2021. An empirical study on deployment faults of deep learning based mobile applications. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 674–685.
- [46] Joao De Macedo, Rui Abreu, Rui Pereira, and João Saraiva. 2021. On the runtime and energy performance of webassembly: Is webassembly superior to javascript yet?. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. IEEE, 255–262.
- [47] Joao De Macedo, Rui Abreu, Rui Pereira, and Joao Saraiva. 2022. WebAssembly versus JavaScript: Energy and Runtime Performance. In *2022 International Conference on ICT for Sustainability (ICT4S)*. IEEE, 24–34.
- [48] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L Abad, and Alexandru Iosup. 2021. The state of serverless applications: Collection, characterization, and community consensus. *IEEE Transactions on Software Engineering* 48, 10 (2021), 4152–4166.
- [49] Philipp Gackstatter, Pantelis A Frangoudis, and Schahram Dustdar. 2022. Pushing serverless to the edge with webassembly runtimes. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 140–149.
- [50] Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. 2020. Sledge: A serverless-first, light-weight wasm runtime for the edge. In *Proceedings of the 21st International Middleware Conference*. 265–279.
- [51] Phani Kishore Gadepalli, Gregor Peach, Ludmila Cherkasova, Rob Aitken, and Gabriel Parmer. 2019. Challenges and opportunities for efficient serverless computing at the edge. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 261–2615.
- [52] Antonio García de la Barrera, Ignacio García-Rodríguez de Guzmán, Macario Polo, and Mario Piattini. 2023. Quantum software testing: State of the art. *Journal of Software: Evolution and Process* 35, 4 (2023), e2419.
- [53] Adam T Geller, Justin Frank, and William J Bowman. 2024. Indexed Types for a Statically Safe WebAssembly. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 2395–2424.
- [54] David Goltzsche, Manuel Nieke, Thomas Knauth, and Rüdiger Kapitza. 2019. Acctee: A webassembly-based two-way sandbox for trusted resource accounting. In *Proceedings of the 20th International Middleware Conference*. 123–135.
- [55] Garrett Gu and Hovav Shacham. 2023. Constant-Time Wasmtime, for Real This Time: End-to-End Verified Zero-Overhead Constant-Time Programming for the Web and Beyond. *arXiv preprint arXiv:2311.14246* (2023).
- [56] Robbert Gurdeep Singh and Christophe Scholliers. 2019. WARDuino: a dynamic WebAssembly virtual machine for programming microcontrollers. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*. 27–36.
- [57] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 185–200.
- [58] Håkon Harnes and Donn Morrison. 2024. SoK: Analysis techniques for WebAssembly. *arXiv preprint arXiv:2401.05943* (2024).
- [59] Hassan B Hassan, Saman A Barakat, and Qusay I Sarhan. 2021. Survey on serverless computing. *Journal of Cloud Computing* 10, 1 (2021), 1–29.
- [60] Keno Haßler and Dominik Maier. 2021. Wafl: Binary-only webassembly fuzzing with fast snapshots. In *Reversing and Offensive-oriented Trends Symposium*. 23–30.
- [61] Aaron Hilbig, Daniel Lehmann, and Michael Pradel. 2021. An empirical study of real-world webassembly binaries: Security, languages, use cases. In *Proceedings of the web conference 2021*. 2696–2708.
- [62] Devon Hockley and Carey Williamson. 2022. Benchmarking runtime scripting performance in wasmer. In *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*. 97–104.
- [63] Mohammed Nurul Hoque and Khaled A Harras. 2022. WebAssembly for Edge Computing: Potential and Challenges. *IEEE Communications Standards Magazine* 6, 4 (2022), 68–73.

- [64] Martin Jacobsson and Jonas Willén. 2018. Virtual machine execution for wearables based on WebAssembly. In *EAI International Conference on Body Area Networks*. Springer, 381–389.
- [65] Abhinav Jangda, Bobby Powers, Emery D Berger, and Arjun Guha. 2019. Not so fast: Analyzing the performance of {WebAssembly} vs. native code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 107–120.
- [66] Bo Jiang, Zichao Li, Yuhe Huang, Zhenyu Zhang, and W Chan. 2022. Wasmfuzzer: A fuzzer for webassembly virtual machines. In *34th International Conference on Software Engineering and Knowledge Engineering, SEKE 2022*. KSI Research Inc., 537–542.
- [67] Shuyao Jiang, Ruiying Zeng, Zihao Rao, Jiazen Gu, Yangfan Zhou, and Michael R Lyu. 2023. Revealing Performance Issues in Server-side WebAssembly Runtimes via Differential Testing. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 661–672.
- [68] Evan Johnson, Evan Laufer, Zijie Zhao, Dan Gohman, Shravan Narayan, Stefan Savage, Deian Stefan, and Fraser Brown. 2023. WaVe: a verifiably secure WebAssembly sandboxing runtime. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2940–2955.
- [69] Harsh Joshi. 2022. ANALYSIS OF WEB ASSEMBLY TECHNOLOGY IN CLOUD AND BACKEND. (2022).
- [70] Joao Lourenço Souza Junior, Davi de Oliveira, Victor Praxedes, and Dennys Simiao. 2020. WebAssembly potentials: A performance analysis on desktop environment and opportunities for discussions to its application on CPS environment. In *Anais Estendidos do X Simpósio Brasileiro de Engenharia de Sistemas Computacionais*. SBC, 145–150.
- [71] Sangeeta Kakati and Mats Brorsson. 2023. WebAssembly Beyond the Web: A Review for the Edge-Cloud Continuum. In *2023 3rd International Conference on Intelligent Technologies (CONIT)*. IEEE, 1–8.
- [72] Staffs Keele et al. 2007. Guidelines for performing systematic literature reviews in software engineering.
- [73] Minseo Kim, Hyerean Jang, and Youngjoo Shin. 2022. Avengers, assemble! Survey of WebAssembly security solutions. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*. IEEE, 543–553.
- [74] Vojdan Kjorveziroski and Sonja Filiposka. 2023. WebAssembly as an Enabler for Next Generation Serverless Computing. *Journal of Grid Computing* 21, 3 (2023), 34.
- [75] Vojdan Kjorveziroski and Sonja Filiposka. 2023. WebAssembly Orchestration in the Context of Serverless Computing. *Journal of Network and Systems Management* 31, 3 (2023), 62.
- [76] Vojdan Kjorveziroski, Sonja Filiposka, and Anastas Mishev. 2022. Evaluating webassembly for orchestrated deployment of serverless functions. In *2022 30th Telecommunications Forum (TELFOR)*. IEEE, 1–4.
- [77] André Kohn, Dominik Moritz, Mark Raasveldt, Hannes Mühlleisen, and Thomas Neumann. 2022. DuckDB-wasm: fast analytical processing for the web. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3574–3577.
- [78] Matthew Kolosick, Shravan Narayan, Evan Johnson, Conrad Watt, Michael LeMay, Deepak Garg, Ranjit Jhala, and Deian Stefan. 2022. Isolation without taxation: near-zero-cost transitions for WebAssembly and SFI. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–30.
- [79] Marius Kreutzer, Maximilian Leonhard Seidler, Victor Pazmino Betancourt, and Jürgen Becker. 2023. Work-in-Progress: Integrating WebAssembly into Service-Oriented Architectures for Edge Systems. In *Proceedings of the International Conference on Embedded Software*. 17–18.
- [80] Kyriakos-Ioannis D Kyriakou and Nikolaos D Tselikas. 2022. Complementing JavaScript in High-Performance Node.js and Web Applications with Rust and WebAssembly. *Electronics* 11, 19 (2022), 3217.
- [81] Hanwen Lei, Ziqi Zhang, Shaokun Zhang, Peng Jiang, Zhineng Zhong, Ningyu He, Ding Li, Yao Guo, and Xiangqun Chen. 2023. Put Your Memory in Order: Efficient Domain-based Memory Isolation for WASM Applications. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 904–918.
- [82] Borui Li, Wei Dong, and Yi Gao. 2021. Wiprog: A webassembly-based approach to integrated iot programming. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 1–10.
- [83] Borui Li, Hongchang Fan, Yi Gao, and Wei Dong. 2021. ThingSpire OS: a WebAssembly-based IoT operating system for cloud-edge integration. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*. 487–488.
- [84] Borui Li, Hongchang Fan, Yi Gao, and Wei Dong. 2022. Bringing webassembly to resource-constrained iot devices for seamless device-cloud integration. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*. 261–272.
- [85] Yongkang Li, Yanying Lin, Yang Wang, Kejiang Ye, and Chengzhong Xu. 2022. Serverless computing: state-of-the-art, challenges and opportunities. *IEEE Transactions on Services Computing* 16, 2 (2022), 1522–1539.
- [86] Danilo Leandro Lima, Ronnie De Souza Santos, Guilherme Pires Garcia, Sildemir S Da Silva, Cesar França, and Luiz Fernando Capretz. 2023. Software testing and code refactoring: A survey with practitioners. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 500–507.
- [87] Bin Lin, Nathan Cassee, Alexander Serebrenik, Gabriele Bavota, Nicole Novielli, and Michele Lanza. 2022. Opinion mining for software development: a systematic literature review. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 3 (2022), 1–41.

- [88] Kaiyan Liu, Nan Wu, and Bo Han. 2023. Demystifying Web-based Mobile Extended Reality Accelerated by WebAssembly. In *Proceedings of the 2023 ACM on Internet Measurement Conference*. 145–153.
- [89] Renju Liu, Luis Garcia, and Mani Srivastava. 2021. Aerogel: Lightweight access control framework for webassembly-based bare-metal iot devices. In *2021 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 94–105.
- [90] Sin Kit Lo, Qinghua Lu, Chen Wang, Hye-Young Paik, and Liming Zhu. 2021. A systematic literature review on federated machine learning: From a software engineering perspective. *ACM Computing Surveys (CSUR)* 54, 5 (2021), 1–39.
- [91] Niko Mäkitalo, Victor Bankowski, Paulius Daubaris, Risto Mikkola, Oleg Beletski, and Tommi Mikkonen. 2021. Bringing webassembly up to speed with dynamic linking. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. 1727–1735.
- [92] Niko Mäkitalo, Tommi Mikkonen, Cesare Pautasso, Victor Bankowski, Paulius Daubaris, Risto Mikkola, and Oleg Beletski. 2021. WebAssembly modules as lightweight containers for liquid IoT applications. In *International Conference on Web Engineering*. Springer, 328–336.
- [93] Cynthia Marcelino and Stefan Nastic. 2023. CWASI: A WebAssembly Runtime Shim for Inter-function Communication in the Serverless Edge-Cloud Continuum. (2023).
- [94] Filipe Marques, José Fragoso Santos, Nuno Santos, and Pedro Adão. 2022. Concolic Execution for WebAssembly. In *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.
- [95] Pankaj Mendki. 2020. Evaluating webassembly enabled serverless approach for edge computing. In *2020 IEEE Cloud Summit*. IEEE, 161–166.
- [96] Jämes Ménétréy, Aeneas Grüter, Peterson Yuhala, Julius Oeftiger, Pascal Felber, Marcelo Pasin, and Valerio Schiavoni. 2024. A Holistic Approach for Trustworthy Distributed Systems with WebAssembly and TEEs. In *27th International Conference on Principles of Distributed Systems (OPODIS 2023)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik.
- [97] Jämes Ménétréy, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. 2021. Twine: An embedded trusted runtime for webassembly. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 205–216.
- [98] Jämes Ménétréy, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. 2022. Watz: a Trusted WebAssembly runtime environment with remote attestation for TrustZone. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1177–1189.
- [99] Jämes Ménétréy, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. 2022. WebAssembly as a Common Layer for the Cloud-edge Continuum. In *Proceedings of the 2nd Workshop on Flexible Resource and Application Management on the Edge*. 3–8.
- [100] Jämes Ménétréy, Marcelo Pasin, Pascal Felber, Valerio Schiavoni, Giovanni Mazzeo, Arne Hollum, and Darshan Vaydia. 2023. A Comprehensive Trusted Runtime for WebAssembly with Intel SGX. *IEEE Transactions on Dependable and Secure Computing* (2023).
- [101] Biju R Mohan et al. 2022. Comparative Analysis Of JavaScript And WebAssembly In The Browser Environment. In *2022 IEEE 10th Region 10 Humanitarian Technology Conference (R10-HTC)*. IEEE, 232–237.
- [102] Konrad Moron and Stefan Wallentowitz. 2023. Support for Just-in-Time Compilation of WebAssembly for Embedded Systems. In *2023 12th Mediterranean Conference on Embedded Computing (MECO)*. IEEE, 1–4.
- [103] Otoya Nakakaze, István Koren, Florian Brillowski, and Ralf Klamma. 2022. Retrofitting industrial machines with webassembly on the edge. In *International Conference on Web Information Systems Engineering*. Springer, 241–256.
- [104] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, et al. 2021. Swivel: Hardening {WebAssembly} against spectre. In *30th USENIX Security Symposium (USENIX Security 21)*. 1433–1450.
- [105] Shravan Narayan, Tal Garfinkel, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2019. Gobi: WebAssembly as a practical path to library sandboxing. *arXiv preprint arXiv:1912.02285* (2019).
- [106] Manuel Nieke, Lennart Almstedt, and Rüdiger Kapitza. 2021. Edgedancer: Secure mobile webassembly services on the edge. In *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*. 13–18.
- [107] Tobias Nießen, Michael Dawson, Panos Patros, and Kenneth B Kent. 2020. Insights into webassembly: compilation performance and shared code caching in node.js. In *EVOKE CASCON 2020*. ACM, 163–172.
- [108] Mohammed Nurul-Hoque and Khaled A Harras. 2021. Nomad: Cross-Platform Computational Offloading and Migration in Femtoclouds Using WebAssembly. In *2021 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 168–178.
- [109] Fernando Oliveira and Júlio Mattos. 2020. Analysis of WebAssembly as a strategy to improve JavaScript performance on IoT environments. In *Anais Estendidos do X Simpósio Brasileiro de Engenharia de Sistemas Computacionais*. SBC, 133–138.
- [110] Gregor Peach, Runyu Pan, Zhuoyi Wu, Gabriel Parmer, Christopher Haster, and Ludmila Cherkasova. 2020. ewasm: Practical software fault isolation for reliable embedded devices. *IEEE Transactions on Computer-Aided Design of*

- Integrated Circuits and Systems* 39, 11 (2020), 3492–3505.
- [111] Steven Pham, Kae Oliveira, and Chung-Horng Lung. 2023. WebAssembly Modules as Alternative to Docker Containers in IoT Application Development. In *2023 IEEE 3rd International Conference on Electronic Communications, Internet of Things and Big Data (ICEIB)*. IEEE, 519–524.
 - [112] Donald Pinckney, Arjun Guha, and Yuriy Brun. 2020. Wasm/k: delimited continuations for WebAssembly. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*. 16–28.
 - [113] Dennis Pockstaller, Stefan Huber, and Lukas Demetz. 2023. Comparing the Energy Consumption of WebAssembly and JavaScript in Mobile Browsers. (2023).
 - [114] Vasile Adrian Bogdan Pop, Arto Niemi, Valentin Manea, Antti Rusanen, and Jan-Erik Ekberg. 2022. Towards securely migrating webassembly enclaves. In *Proceedings of the 15th European Workshop on Systems Security*. 43–49.
 - [115] Ivan Puddu, Moritz Schneider, Daniele Lain, Stefano Boschetto, and Srdjan Čapkun. 2022. On (The Lack Of) code confidentiality in trusted execution environments. *arXiv preprint arXiv:2212.07899* (2022).
 - [116] Salim S Salim, Andy Nisbet, and Mikel Luján. 2020. Trufflewasm: a webassembly interpreter on graalvm. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 88–100.
 - [117] Prabhjot Sandhu, David Herrera, and Laurie Hendren. 2018. Sparse matrices on the web: Characterizing the performance and optimal format selection of sparse matrix-vector multiplication in JavaScript and WebAssembly. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*. 1–13.
 - [118] Fabian Scheidl. 2020. Valent-blocks: Scalable high-performance compilation of webassembly bytecode for embedded systems. In *2020 International Conference on Computing, Electronics & Communications Engineering (iCCECE)*. IEEE, 119–124.
 - [119] Merlijn Sebrechts, Tim Ramlot, Sander Borny, Tom Goethals, Bruno Volckaert, and Filip De Turck. 2022. Adapting Kubernetes controllers to the edge: on-demand control planes using Wasm and WASI. In *2022 IEEE 11th International Conference on Cloud Networking (CloudNet)*. IEEE, 195–202.
 - [120] Suhyeon Song, Seonghwan Park, and Donghyun Kwon. 2023. metaSafer: A Technique to detect heap metadata corruption in WebAssembly. *IEEE Access* (2023).
 - [121] Benedikt Spies and Markus Mock. 2021. An evaluation of WebAssembly in non-web environments. In *2021 XLVII Latin American Computing Conference (CLEI)*. IEEE, 1–10.
 - [122] Anastasios Stotoglou and Theodore H Kaskalis. 2023. Comparative Study of JavaScript and WebAssembly Derivatives in Browser Engines. In *2023 Intelligent Methods, Systems, and Applications (IMSA)*. IEEE, 476–483.
 - [123] Bala Subramanyan. 2023. PRIVATON-Privacy Preserving Automaton for Proof of Computations. *Cryptology ePrint Archive* (2023).
 - [124] Joshua Wenata Sunarto, Angelina Quincy, Fakhira Shafa Maheswari, Quesynovich Denis Al Hafizh, Melanie Gabriela Tjandrasubrata, and Mochammad Haldi Widianto. 2023. A Systematic Review of WebAssembly VS Javascript Performance Comparison. In *2023 International Conference on Information Management and Technology (ICIMTech)*. IEEE, 241–246.
 - [125] Aron Szanto, Timothy Tamm, and Artidoro Pagnoni. 2018. Taint tracking for webassembly. *arXiv preprint arXiv:1807.08349* (2018).
 - [126] Raven Szewczyk, Kimberley Stonehouse, Antonio Barbalace, and Tom Spink. 2022. Leaps and bounds: Analyzing WebAssembly’s performance with a focus on bounds checking. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 256–268.
 - [127] Ben L Titzer. 2022. A fast in-place interpreter for WebAssembly. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 646–672.
 - [128] Ben L Titzer. 2023. Whose Baseline (compiler) is it anyway? *arXiv preprint arXiv:2305.13241* (2023).
 - [129] Max Van Hasselt, Kevin Huijzendveld, Nienke Noort, Sasja De Ruijter, Tanjina Islam, and Ivano Malavolta. 2022. Comparing the energy efficiency of webassembly and javascript in web applications on android mobile devices. In *Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering*. 140–149.
 - [130] Alexa VanHattum, Monica Pardeshi, Chris Fallin, Adrian Sampson, and Fraser Brown. 2024. Lightweight, Modular Verification for WebAssembly-to-Native Instruction Selection. (2024).
 - [131] Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S Rellermeyer. 2020. A survey on distributed machine learning. *Acm computing surveys (csur)* 53, 2 (2020), 1–33.
 - [132] Linus Wagner, Maximilian Mayer, Andrea Marino, Alireza Soldani Nezhad, Hugo Zwaan, and Ivano Malavolta. 2023. On the energy consumption and performance of webassembly binaries across programming languages and runtimes in iot. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*. 72–82.
 - [133] Stefan Wallentowitz, Bastian Kersting, and Dan Mihai Dumitriu. 2022. Potential of WebAssembly for Embedded Systems. In *2022 11th Mediterranean Conference on Embedded Computing (MECO)*. IEEE, 1–4.
 - [134] Weihang Wang. 2021. Empowering web applications with WebAssembly: are we there yet?. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1301–1305.

- [135] Wenwen Wang. 2022. How Far We've Come—A Characterization Study of Standalone WebAssembly Runtimes. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 228–241.
- [136] Yue Wang, Zhide Zhou, Zhilei Ren, Dong Liu, and He Jiang. 2023. A Comprehensive Study of WebAssembly Runtime Bugs. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 355–366.
- [137] Zhen Wang, Jianda Wang, Zhendong Wang, and Yang Hu. 2021. Characterization and implication of edge WebAssembly runtimes. In *2021 IEEE 23rd Int Conf on High Performance Computing & Communications; 7th Int Conf on Data Science & Systems; 19th Int Conf on Smart City; 7th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*. IEEE, 71–80.
- [138] Muhammad Waseem, Teerath Das, Aakash Ahmad, Peng Liang, and Tommi Mikkonen. 2023. Understanding the Issues and Causes in WebAssembly Application Development: A Mining-based Study. *arXiv preprint arXiv:2311.00646* (2023).
- [139] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. 2019. Ct-wasm: type-driven secure cryptography for the web ecosystem. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.
- [140] Conrad Watt, Maja Trela, Peter Lammich, and Florian Märkl. 2023. WasmRef-Isabelle: A Verified Monadic Interpreter and Industrial Fuzzing Oracle for WebAssembly. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 100–123.
- [141] Elliott Wen and Gerald Weber. 2020. Wasmachine: Bring iot up to speed with a webassembly os. In *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE, 1–4.
- [142] Elliott Wen, Gerald Weber, and Suranga Nanayakkara. 2022. WasmAndroid: A Cross-Platform Runtime for Native Programming Languages on Android. *ACM Transactions on Embedded Computing Systems* 22, 1 (2022), 1–19.
- [143] Jinfeng Wen, Zhenpeng Chen, Xin Jin, and Xuanzhe Liu. 2023. Rise of the planet of serverless computing: A systematic review. *ACM Transactions on Software Engineering and Methodology* (2023).
- [144] Yutao Xie, Jiayi Lin, Hande Dong, Lei Zhang, and Zhonghai Wu. 2023. Survey of Code Search Based on Deep Learning. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023), 1–42.
- [145] Yutian Yan, Tengfei Tu, Lijian Zhao, Yuchen Zhou, and Weihang Wang. 2021. Understanding the performance of webassembly applications. In *Proceedings of the 21st ACM Internet Measurement Conference*. 533–549.
- [146] Zhiqiang Yang, Han Liu, Yue Li, Huixuan Zheng, Lei Wang, and Bangdao Chen. 2020. Seraph: enabling cross-platform security analysis for evm and wasm smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. 21–24.
- [147] Wanja Zaeske, Sven Friedrich, Tim Schubert, and Umut Durak. 2023. WebAssembly in Avionics: Decoupling Software from Hardware. In *2023 IEEE/AIAA 42nd Digital Avionics Systems Conference (DASC)*. IEEE, 1–10.
- [148] Jie M Zhang, Mark Harman, Lei Ma, and Yang Liu. 2020. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering* 48, 1 (2020), 1–36.
- [149] Xiuhong Zhang. 2020. *WebAssembly Principles and Core Technologies*. China Machine Press.
- [150] Yixuan Zhang, Shangtong Cao, Haoyu Wang, Zhenpeng Chen, Xiapu Luo, Mu Dongliang, Ma Yun, Huang Gang, and Liu Xuanzhe. 2023. Characterizing and Detecting WebAssembly Runtime Bugs. *ACM Transactions on Software Engineering and Methodology* (2023).
- [151] Ziyao Zhang, Wenlong Zheng, Baojian Hua, Qiliang Fan, and Zhizhong Pan. 2023. VMCanary: Effective Memory Protection for WebAssembly via Virtual Machine-assisted Approach. In *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*. IEEE, 662–671.
- [152] Shixuan Zhao, Pinshen Xu, Guoxing Chen, Mengya Zhang, Yingqian Zhang, and Zhiqiang Lin. 2023. Reusable enclaves for confidential serverless computing. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4015–4032.
- [153] Shuyu Zheng, Haoyu Wang, Lei Wu, Gang Huang, and Xuanzhe Liu. 2020. VM matters: a comparison of WASM VMS and EVMS in the performance of blockchain smart contracts. *arXiv preprint arXiv:2012.01032* (2020).
- [154] Shiyo Zhou, Muhui Jiang, Weimin Chen, Hao Zhou, Haoyu Wang, and Xiapu Luo. 2023. WADIFF: A Differential Testing Framework for WebAssembly Runtimes. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 939–950.
- [155] Shu Zhu, Bao Li, Yusong Tan, Xiaochuan Wang, and Jianfeng Zhang. 2022. LAWOW: Lightweight Android Workload Offloading Based on WebAssembly in Heterogeneous Edge Computing. In *2022 10th International Conference on Information Systems and Computing Technology (ISCTech)*. IEEE, 753–758.