future internet

MDPI

Article

An Overview of WebAssembly for IoT: Background, Tools, State-of-the-Art, Challenges, and Future Directions

Partha Pratim Ray

Department of Computer Applications, Sikkim University, Gangtok 737102, Sikkim, India; ppray@cus.ac.in

Abstract: This paper explores the relationship between two emerging technologies, WebAssembly (Wasm) and the Internet of Things (IoT). It examines the complementary roles of these technologies and their impact on modern web applications. First, it delves into the capabilities of Wasm as a high-performance binary format that allows developers to leverage low-level languages for computationally intensive tasks. Second, it seeks to explain why integration of IoT and Wasm is important. Third, it discusses the strengths and limitations of various tools and tool chains that are crucial for Wasm development and implementation, with a special focus on IoT. Fourth, it presents the state-of-the-art with regard to advances that combine both technologies. Fifth, it discusses key challenges and provides future directions. Lastly, it provides an in-depth elaboration of the future aspects of Wasm, with a strong focus on IoT, concluding that IoT and Wasm can provide developers with a versatile toolkit that enables them to balance productivity and performance in both web and non-web development scenarios. The collaborative use of these technologies opens up new possibilities for pushing the boundaries of web application development in terms of interactivity, security, portability, scalability, and efficient computational capabilities. As web and non-web embeddings continue to evolve, the integration of IoT and Wasm will play a crucial role in shaping the future of innovative application development. The key findings of this extensive review work suggest that existing tool sets can be easily conglomerated together to form a new era in WebAssembly–IoT infrastructure for low-power, energy-efficient, and secure edge–IoT ecosystems with near-native execution speed. Furthermore, the expansion of edge–IoT ecosystems can be augmented with prospective cloud-side deployments. However, there remains a strong need to more cohesively advance the amalgamation of Wasm and IoT technologies in the near future.

Keywords: WebAssembly; IoT; stack; virtual machine; binary format

Citation: Ray, P.P. An Overview of WebAssembly for IoT: Background, Tools, State-of-the-Art, Challenges, and Future Directions. *Future Internet* **2023**, *15*, 275. https://doi.org/10.3390/fi15080275

Academic Editor: Michael Sheng

Received: 13 July 2023
Revised: 6 August 2023
Accepted: 15 August 2023
Published: 18 August 2023

1. Introduction

In this section, a basic introductory overview of Wasm and IoT is provided as a foundation for understanding the remainder of this article. First, a holistic background is presented, followed by the motivation behind this study. The key objectives, contributions, and organization of this article are presented in the subsequent parts.

1.1. Background and Significance

The onset of the digital age has ushered in a plethora of technological advancements, among which web development has played a significant role. As the world becomes increasingly interconnected via the World Wide Web, the demand for more efficient, robust, and secure web applications is surging. JavaScript, while integral to web development, is showing its limitations as these applications grow more intricate and demanding. This trend has sparked the creation of WebAssembly (Wasm), an innovative binary instruction format engineered to complement JavaScript by providing a performance profile closer to machine code [1]. Wasm is designed with the purpose of allowing web pages to run high-performance applications efficiently, bringing a new level of dynamism and interactivity to the web [2,3].

However, the influence of Wasm extends far beyond traditional web applications [4]. A particularly promising avenue lies in the rapidly evolving field of the IoT [5,6]. IoT connects physical devices with the digital world, forming an interconnected network that allows data exchange and automated control. This technology is transforming many sectors, from home automation and healthcare to industrial systems and smart cities. As the IoT ecosystem expands it poses unique challenges, particularly concerning performance, security, and interoperability.

The unique characteristics of Wasm make it well-suited to addressing these challenges. Its compact binary format allows for quick transmission and loading, a critical requirement in IoT systems where devices may have limited computational power and network connectivity [7]. Wasm's near-native performance ensures efficient execution, enabling real-time processing of complex tasks. Moreover, its security features help to mitigate potential risks in IoT applications, where vulnerabilities can have serious implications [8–10].

The significance of Wasm, therefore, lies not only in its impact on web development but in its potential to revolutionize the IoT landscape. By understanding and leveraging Wasm, we can unlock new possibilities for IoT, making our interconnected world faster, safer, and more efficient.

1.2. Motivation

Wasm has been nothing short of revolutionary in the realm of web development. Originally designed as a high-performance target for languages such as C, C++, and Rust, it effectively bridges the divide between JavaScript's agility and the demand for rigorous low-level computation in web applications. Its attributes include the following. (1) Compact Binary Format: Wasm's concise binary representation not only ensures a reduced file size, optimizing loading times, it offers rapid parsing and execution, which is pivotal in environments such as IoT devices where resources are limited. (2) Near-Native Execution Speed: Wasm promises execution at near-native speed, a feat that ensures intensive computational tasks can be executed seamlessly, enhancing the user experience and ensuring that IoT devices can process information swiftly. (3) Language Interoperability: the ability to compile code from multiple languages into Wasm means that developers are not shackled to a single language. This adaptability is crucial, especially in the diverse ecosystem of IoT, where devices might have been developed using various programming paradigms. (4) Sandboxed Security Model: Wasm's execution within a sandboxed environment offers a robust security paradigm, ensuring code confidentiality and integrity. This is particularly vital in the IoT spectrum, where devices often lack in built-in security features and can be frequent targets for attacks.

As the realm of IoT burgeons with billions of interconnected devices, the quest for efficient, secure, and universally compatible technological foundations is paramount. The unprecedented scaling of IoT accentuates the need for solutions that are not just energy-efficient but compact, secure, and maintainable. Herein lies the potential of Wasm, with its unique strengths positioning it as an invaluable asset in the IoT toolkit. It is puzzling, considering its evident potential, that there remains a conspicuous absence of extensive literature focusing on the fusion of Wasm and IoT. While the intersection of these two burgeoning technologies possesses the potential to redefine our technological landscape, the lack of thorough research and understanding has left many nuances unexplored. Our motivation stems from this very gap. We embark on a journey to deeply understand, dissect, and then articulate the synergies between Wasm and IoT. This paper aims to be a beacon, shedding light on Wasm's architecture, its symbiotic relationship with IoT, the tools paving its developmental pathway, the intricacies of its security model, and the vistas it promises to unlock in the near future.

1.3. Objectives

The primary objectives of this review are as follows:

- To understand Wasm by providing a comprehensive understanding of the technical specifications, architectural design, and operational mechanics of Wasm.
- To explore the intersection of Wasm and IoT by investigating how the features of Wasm address the unique challenges of IoT environments and can improve the efficiency, security, and interoperability of IoT applications.
- To analyze Wasm tools and toolchains through in-depth examination of the tools and toolchains developers use to compile, debug, and optimize Wasm code, including their respective strengths and weaknesses.
- To highlight real-world applications by showcasing real-world applications of Wasm in the IoT sector, thereby demonstrating its capabilities and practical value.
- To discuss key challenges by analyzing the security model of Wasm, its benefits, and any potential vulnerabilities, particularly in the IoT context.
- Finally, speculating on future trends, to deal with the coming prospects and evolving trends of Wasm and its potential future impacts on the IoT.

1.4. Contributions

This comprehensive review makes several noteworthy contributions to the body of knowledge on Wasm and IoT:

- A wide-ranging review offering a broad and inclusive perspective on Wasm, encompassing its technical specifications, technological relevance, interplay with IoT, tools and toolchains, and future trends.
- An in-depth analysis of Wasm in IoT that fills an informational void by providing an exhaustive exploration of the role and potential of Wasm in the IoT sector.
- Comparison of tools through a comprehensive examination of various Wasm tools and toolchains, which can provide a valuable resource for developers.
- Inclusion of real-world examples, making this review a practical guide for developers seeking to leverage Wasm in IoT projects.
- Finally, the future perspectives provided in the forward-looking section of the review can provide strategic insights for those planning to engage with Wasm and IoT in the future.

1.5. Organization of the Paper

Figure 1 presents the organization of this paper. Section 2 deals with the overall understanding of Wasm. Section 3 aligns Wasm with IoT. Section 4 covers various tools for Wasm development. Section 5 presents a comprehensive state-of-the-art. Section 6 discusses about key challenges and future directions. Section 7 provides future aspects of Wasm. Section 8 concludes the review work.

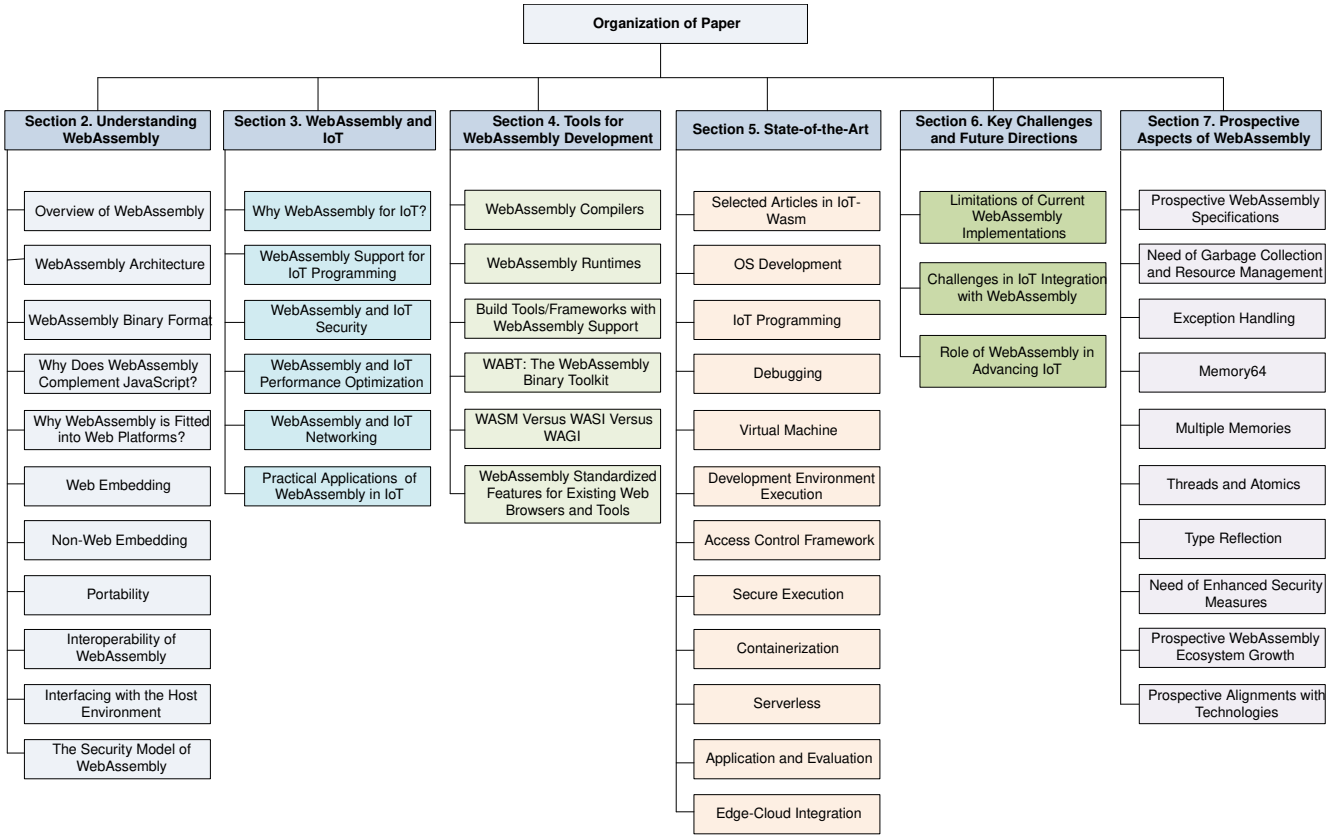

Figure 1. Organization of the paper.

2. Understanding WebAssembly

This section deals with an overview of Wasm, its usage in the IoT context, support system, security aspects, performance optimization, networking, and practical applications of Wasm in IoT.

2.1. Overview of WebAssembly

WebAssembly, colloquially known as Wasm, is a stack-based virtual machine [11]. It was designed as a portable target for the compilation of high-level languages such as C, C++, and Rust, enabling deployment on the web for client and server applications. Wasm provides a compact binary format that is designed to be fast to decode and execute. In addition, it is designed to be safe, operating inside a sandboxed environment separated from the host system. Wasm provides a compelling target for a range of programming languages. Wasm operates in conjunction with JavaScript, often as a complementary technology. While JavaScript does not "compile" to Wasm, it is important to understand that JavaScript interacts with Wasm modules in browsers, facilitating communication between the DOM and Wasm functions. Currently, it supports C, C++, Rust, Assembly Script, C#, F#, Dart, Go, Kotlin, Swift, D, Pascal, Zig, and Grain.

- **The Rationale Behind WebAssembly**
  The introduction of Wasm was in response to the need for an efficient low-level code representation that could be processed effectively within the stringent performance constraints of web environments. Over the course of their evolution, web applications have grown in complexity, requiring more computational resources and demanding superior performance from client-side scripts. JavaScript, while instrumental in the development and functionality of these applications, has limitations in terms of performance optimization that have become more evident with the increasing complexity of web tasks. Wasm was developed to address this performance gap, complementing JavaScript by running complex computational tasks at near-native speed. This

leap in performance opens up new possibilities for web-based applications, allowing them to handle more complex tasks that have traditionally been the domain of desktop applications.

- **Portability and Interoperability**
  A key attribute of Wasm is its design as a portable bytecode that can be executed across a variety of platforms. This universality makes it a versatile tool in heterogeneous environments, including the IoT, where devices of various types are interconnected. A Wasm module can run on any platform that hosts a compliant Wasm virtual machine, irrespective of the device's specific hardware or operating system. The interoperability of Wasm is another key aspect of its design. It is designed to work in conjunction with JavaScript, leveraging the existing web platform's infrastructure. This interoperability allows for a smoother transition to Wasm, as developers can gradually replace performance-critical sections of JavaScript code with more efficient Wasm modules without disrupting the application's overall functionality.
- **Safety and Security**
  Web technologies must prioritize safety and security, and Wasm is no exception. Wasm operates inside a secure sandbox environment separate from the host system, much as JavaScript does. This design restricts any potential malicious activity within the confines of the sandbox, providing a crucial layer of security. Even if an attacker manages to exploit a vulnerability within the Wasm code, they will be contained within the sandbox environment, mitigating the potential damage to the host system.
- **Evolution and Community Engagement**
  Wasm has received substantial interest and support from both the developer community and industry giants such as Google, Mozilla, Microsoft, and Apple. This widespread backing coupled with active community engagement has fostered rapid evolution. There have been numerous improvements and extensions to the Wasm specification since its initial release, highlighting the ongoing development of this powerful technology. The community engagement around Wasm drives its evolution and aids in exploring new possibilities for its application across various domains, including IoT. The continued research and development efforts around Wasm, including exploring potential use cases, improving security measures, and refining its performance, ensure its place at the forefront of web and IoT technologies.
- **High-Level Goals of WebAssembly**
  The primary objectives of Wasm can be summarized as follows:
  - Establish a binary format that is portable and efficient, serving as a target for compilation to achieve native execution speed on a wide range of platforms, including mobile and IoT devices.
  - Implement the standard in incremental stages, beginning with a Minimum Viable Product (MVP) that offers functionality comparable to asm.js, with a specific focus on C/C++ development. Additional features, such as threads, zero-cost exceptions, and Single Instruction and Multiple Data Stream (SIMD), including support for languages beyond C/C++, will be introduced based on feedback and priorities.
  - Ensure seamless integration with the existing web platform by aligning with its evolutionary nature, maintaining compatibility with JavaScript, enabling synchronous communication with JavaScript, adhering to security policies such as same-origin and permissions, accessing browser functionality through existing Web Application Programming Interfaces (APIs), and providing a human-readable text format that can be converted to and from the binary format, supporting the "View Source" functionality.
  - Support non-browser embeddings, enabling the usage of Wasm in diverse environments beyond the web.
  - Enhance the platform by developing a new LLVM backend and accompanying clang port specifically for Wasm, encouraging the adoption of other compilers

and tools targeting Wasm and fostering the development of additional useful tooling to enrich the Wasm ecosystem.

2.2. WebAssembly Architecture

Wasm's architecture is one of its key strengths, contributing significantly to its speed, security, and versatility. It operates through a stack-based virtual machine. This means that unlike register-based code, which requires different instructions to execute on various physical machine architectures, Wasm code can be run consistently across different platforms. Wasm modules declare the types, functions, tables, memories, and globals that they use, and these elements interact in well-defined ways. Furthermore, the architecture includes features for direct memory access and control flow, providing for efficient execution. Figure 2 presents the data flow architecture of Wasm.

- **Overview**
  Wasm is a stack-based virtual machine, which has significant implications for its design and operation. Stack-based machines use a stack to manage computations, which means that operands are pushed onto the stack, operations are performed on the top elements of the stack, and results are stored on the stack. This design makes the architecture of the virtual machine simple and efficient.
- **Stack-Based Design**
  The stack-based design of Wasm is a significant factor contributing to its portability and efficiency. Stack machines are inherently simpler than register machines, as they have no need to manage and allocate registers. Furthermore, stack machines are agnostic to the specifics of the host's physical machine architecture. This means that the same Wasm code can be run consistently across different platforms, enhancing its portability. Wasm's design is based on a stack machine model. This means that it performs computations using a stack data structure where operands are pushed onto the stack; operators pop operands off the stack and push the results back onto it. This design enhances the portability of the Wasm binary format and helps to simplify the process of validation and execution. In addition, the stack machine model results in highly compact bytecode. In a register-based machine model it is necessary to specify the register for each operand and result. In contrast, a stack-based machine model implicitly uses the top of the stack, eliminating the need for such specifications and yielding a smaller binary size.
- **WebAssembly Modules**
  Wasm code is organized into modules, which are the unit of deployment, loading, and compilation. These modules are stateless and can be statically analyzed, which contributes to both the safety and efficiency of Wasm. Modules declare the types, functions, tables, memories, and globals that they use, and these declarations form the basis of Wasm's structured control flow. In Wasm, a module is a binary format file that contains a program's compiled code along with all the data and information necessary to execute it. Each module includes a series of sections that specify types, functions, imports, exports, memory, data, tables, elements, globals, and more. Wasm modules are stateless and side-effect-free, meaning that they do not have access to any state or perform any operations unless they explicitly receive access from the host environment. This characteristic ensures isolation between modules and their host environment, improving security.
- **Linear Memory Model**
  Wasm uses a linear memory model. Each Wasm module has access to a single resizable array of bytes, known as its linear memory. This memory is sandboxed from the host environment and other instances, and is accessed using integer indices. This design simplifies memory management and improves safety by isolating the memory of different instances. This memory is initialized during instantiation of the module, and can be accessed and manipulated through memory instructions provided by Wasm. While this model may seem limiting, it is important to note that each Wasm module can

have its own memory, allowing for isolation of different Wasm programs. Additionally, the host environment can create and provide shared memory to Wasm modules, enabling communication and shared memory concurrency between Wasm threads.

- **Direct Memory Access**
  Wasm operates a linear memory model that is directly accessible by the program through load and store operations. This direct memory access is a fundamental aspect of Wasm's design and performance efficiency. Memory in Wasm is a resizable array buffer, and every load or store operation accesses this buffer directly. Load and store instructions specify an immediate offset as well as the alignment for effective memory access. This means that programs can read and write data to memory one byte at a time if needed, or in larger chunks when it is more efficient. Importantly, this direct memory access does not compromise safety. Even though the program can read and write directly from and to memory, every access is checked to ensure that it falls within the bounds of the Wasm module's memory. If an out-of-bounds memory access is attempted, a runtime error is triggered, halting execution and preventing any data corruption or security breach.
- **Control Flow**
  Control flow in Wasm revolves around structured constructs that facilitate the design of a wide array of programs and algorithms. These constructs include common elements such as loops and conditionals along with function calls and returns, forming the building blocks of Wasm's control flow. Wasm employs a structured control flow, meaning that control flow constructs such as loops and conditionals have clearly defined entry and exit points. This structure significantly simplifies the validation, compilation, and optimization of Wasm programs. Moreover, control flow in Wasm is designed to be robust and resilient, limiting potential security vulnerabilities. This is particularly evident in how the language prevents unbounded jumps or "gotos," limiting possible control flow graphs to structured and predictable patterns. As a result, Wasm's control flow enables the construction of complex programs while ensuring that their execution remains safe and secure.
- **Interacting Elements**
  Wasm has been designed with deep interaction and interoperability in mind. This extends not only to the interaction between Wasm modules and the hosting environment, but between multiple Wasm modules and between Wasm and JavaScript. The elements within a Wasm module, such as functions, globals, tables, and memories, interact in well-defined ways. Functions can be called with parameters and return results, globals can be read from and written to, tables allow for dynamic function dispatch, and memories provide the raw storage for application data. These interactions are carefully controlled to ensure the safety and reliability of Wasm applications.
  - **WebAssembly and JavaScript**
    One of the key features of Wasm is its ability to work seamlessly with JavaScript. Wasm and JavaScript can interact through a set of APIs that allows them to share data and functions. JavaScript can create a Wasm module instance and call its exported functions, while Wasm can call into JavaScript through imported functions. They can share data via Wasm's linear memory or through JavaScript objects.
  - **WebAssembly and the Host Environment**
    Wasm modules interact with their host environment through a system of imports and exports. A module can export functions, memory, tables, and global variables that can be used by the host or by other modules. Conversely, a module can import functions and variables from the host, allowing it to interact with the outside world. This makes Wasm a highly adaptable technology that can be integrated into a wide variety of host environments.
  - **Multiple WebAssembly Modules**
    Multiple Wasm modules can interact and work together within a single application. This is done by instantiating the modules with appropriate imports from

other modules, allowing them to share functions, memory, tables, and globals. This modular design supports the creation of large, complex applications built from smaller, reusable components. These interacting elements contribute significantly to the versatility and power of Wasm, enabling it to perform tasks from high-performance computing to dynamic generation of web content. This makes Wasm an essential part of the modern web platform and a valuable tool for the IoT space.

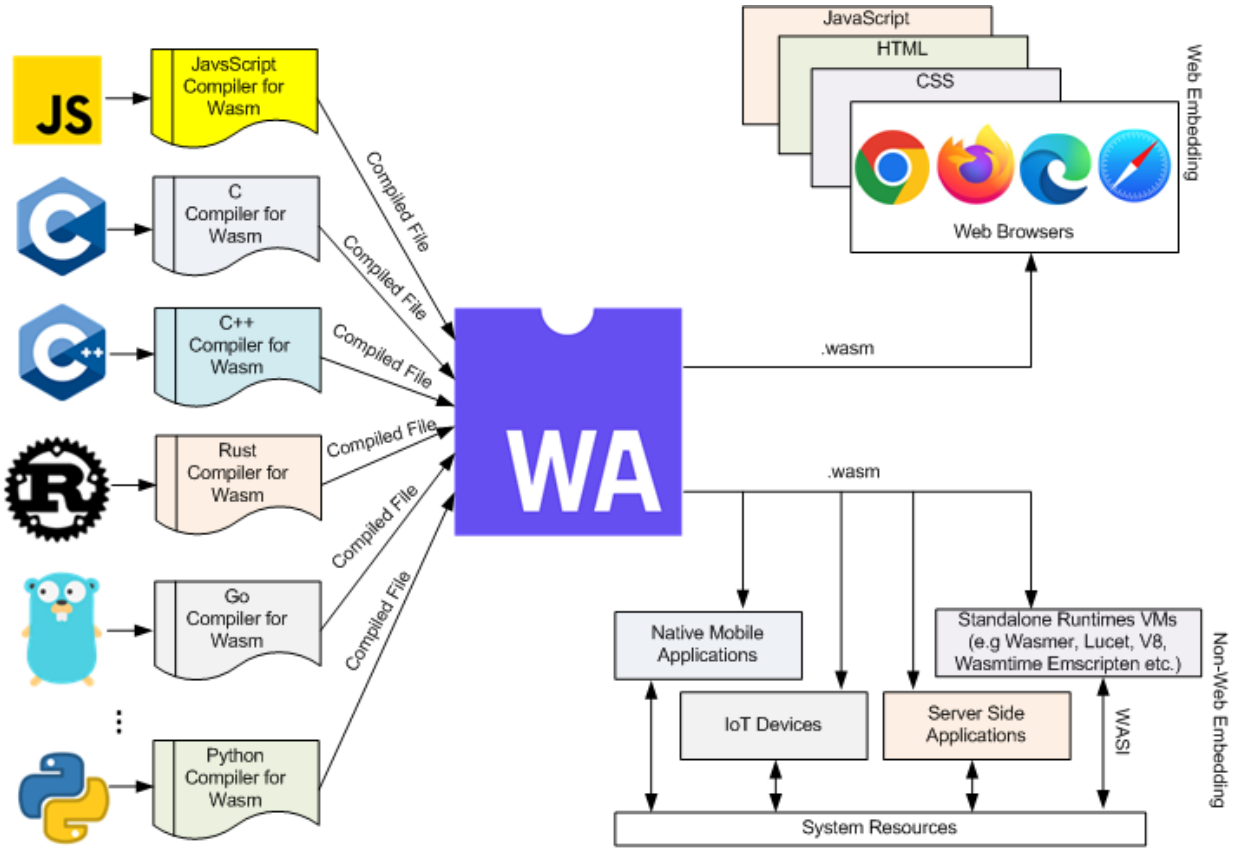

Figure 2. WebAssembly data flow architecture.

2.3. WebAssembly Binary Format

Wasm's binary format embodies a compact low-level representation of Wasm code structured to allow rapid decoding and execution [11]. Engineered as the machine-readable blueprint of Wasm programs, this format is crucial for Wasm runtimes when they aim for efficient execution, as highlighted in Figure 3. At its core, the binary format is segmented into multiple sections, each catering to a distinct aspect of the Wasm module. First, the "Version" section explicitly determines the version of the format, such as version 1. Diving deeper, the "Type Section" encapsulates function signatures used within the module, defining both parameters and return values to ensure smooth interaction with other modules and the runtime. To maintain extensibility, the "Import Section" enumerates external functionalities and global variables imported from distinct modules. Following suit, the "Function Section" categorizes the module's functions, correlating them with respective function signatures from the "Type Section". To foster efficient indirect function calls and dynamic dispatch, the "Table Section" stipulates table types, sizes, and element types. Meanwhile, the "Memory Section" delineates the linear addressable memory parameters of the module. The "Global Section" holds the global variables, defining their characteristics and initial values. Ensuring accessibility to other modules, the "Export Section" enlists the module's exportable entities. If initialization is necessitated upon module load, the "Start Section" pinpoints the relevant start function. For modules leveraging tables, the "Element Section"

lays out the initialization criteria. The crux of the module's functionality is enshrined in the "Code Section", which comprises the compiled code of all functions. To establish a foundation of initial data, the "Data Section" manages memory initialization specifics. Lastly, the "Custom Sections" offer an avenue for integrating optional data pertinent to the module's broader context.

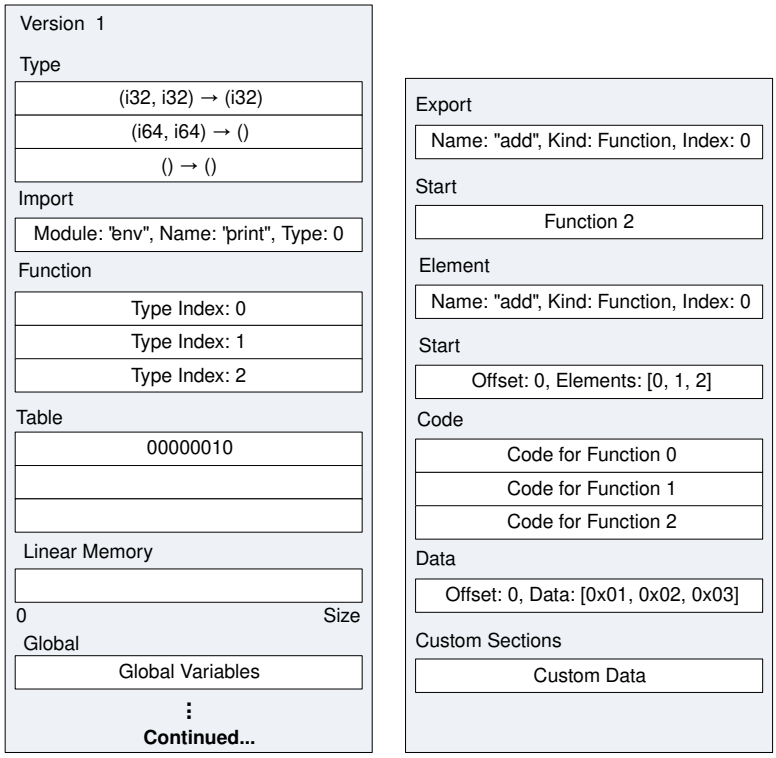

Figure 3. Wasm binary data format.

2.4. How Does WebAssembly Complement JavaScript?

JavaScript and Wasm are two powerful technologies that, when used together, can complement each other and provide significant benefits in web development [12]. We next explore how JavaScript and Wasm work together and how they can enhance each other's capabilities.

- **Performance Enhancement:** JavaScript is the traditional language used in web development, and is known for its versatility and ease of use. However, JavaScript is an interpreted language, which can sometimes result in slower execution compared to lower-level languages such as C and C++. By leveraging Wasm, developers can offload computationally intensive tasks to the Wasm module, resulting in improved performance and responsiveness.
- **Language Compatibility:** JavaScript has a vast ecosystem of libraries, frameworks, and tools that have been developed over the years. It is the language of the web, and many developers are already familiar with it. Wasm, on the other hand, supports multiple programming languages, including C/C++, Rust, and more. This means that developers can utilize their existing codebase written in different languages and compile it to Wasm to run in the browser. This enables developers to reuse and integrate existing code with JavaScript seamlessly.
- **Code Portability:** JavaScript is supported by all modern web browsers, making it highly portable. However, there are cases in which certain algorithms or complex computations are more efficiently implemented in lower-level languages. With Wasm, developers can compile code from these languages into Wasm modules, which can then be executed in any web browser that supports Wasm. This porta-

bility allows developers to target multiple platforms without the need for language-specific implementations.

- **Security and Isolation:** JavaScript executes within the browser's sandbox environment, which provides a level of security by preventing direct access to the underlying system resources. Wasm runs within a sandboxed environment as well; however, it provides additional security features such as memory isolation and fine-grained control over resource access. By utilizing Wasm, developers can ensure a higher level of security when executing potentially untrusted code.
- **Extending JavaScript's Capabilities:** Wasm can be used as a complementary technology to extend JavaScript's capabilities. Developers can leverage Wasm modules to perform computationally intensive tasks, data processing, or even run existing software libraries. JavaScript can act as a glue language, interacting with Wasm modules and providing a higher-level interface or handling UI-related tasks.
- **Improved Developer Experience:** Wasm and JavaScript work hand in hand to enhance the developer experience. Developers can utilize their preferred language for different parts of the application, choosing JavaScript for UI interactions, DOM manipulation, and event handling while offloading performance-critical or complex computations to Wasm modules. This division of labor allows developers to write cleaner and more maintainable code and to optimize performance where it matters most.

2.5. Why WebAssembly Is a Good Fit for Web Platforms

Wasm is a valuable addition to the web platform, working alongside JavaScript to enhance its capabilities and address certain limitations. The web platform consists of a virtual machine (VM) that executes web app code and a collection of web APIs for controlling browser/device functionality [13]. Traditionally, the VM can only handle JavaScript, which, while is powerful, may face performance challenges in intensive use cases such as 3D games, augmented reality, and image editing. Moreover, downloading and parsing large JavaScript applications can be time-consuming, especially on resource-constrained devices. Wasm serves as a complementary language to JavaScript rather than a replacement. JavaScript excels as a high-level language for web applications, offering flexibility, expressiveness, and an extensive ecosystem of frameworks and libraries. On the other hand, Wasm is a low-level language with a compact binary format that delivers near-native performance. It serves as a compilation target for languages such as C++ and Rust, providing them with a web-compatible execution environment. While JavaScript is dynamically typed and does not require a compilation step, Wasm supports low-level memory models and aims to support garbage-collected languages in the future. With the integration of Wasm into browsers, the VM can now load and execute both JavaScript and Wasm code. These two code types can interact with each other seamlessly. The Wasm JavaScript API allows JavaScript to call exported Wasm code, while Wasm code can import and call JavaScript functions synchronously. Wasm modules, the basic units of Wasm code, exhibit similarities to ES modules in their structure and functionality. Wasm encompasses several key concepts that are essential to understanding its operation within the browser.

Several of these concepts are directly reflected in the Wasm JavaScript API, as mentioned below:

- **Module**
  Represents a Wasm binary that has been compiled by the browser into executable machine code. Similar to a Blob, a Module is stateless and can be shared between windows and workers explicitly through methods such as postMessage(). It declares imports and exports, similar to an ES module.
- **Memory**
  A resizable ArrayBuffer that holds the linear array of bytes accessed by Wasm's low-level memory instructions for reading and writing.
- **Table**

A resizable typed array that stores references, for instance function references, which cannot be stored directly as raw bytes in Memory due to safety and/or portability considerations.

- **Instance**
  A combination of a Module and its associated runtime state, including a Memory, Table, and imported values. An Instance is akin to an ES module loaded into a specific global context with specific imports.

The Wasm JavaScript API empowers developers to create modules, memories, tables, and instances. With a Wasm instance, JavaScript code can synchronously invoke its exports, which are exposed as regular JavaScript functions. Conversely, Wasm code can synchronously call arbitrary JavaScript functions by passing them as imports to a Wasm instance. Considering that JavaScript has control over the downloading, compilation, and execution of Wasm code, JavaScript developers can view Wasm as a JavaScript feature designed to generate highly efficient and high-performance functions. In the future, Wasm modules will be loadable in a similar fashion to ES modules, allowing JavaScript to fetch, compile, and import Wasm modules with the same ease as ES modules using the <script type = 'module'> syntax.

2.6. Web Embedding

It comes as no surprise that Wasm is designed to be utilized on the web, primarily within web browsers. However, its purpose is not limited to the web alone. The integration of Wasm with the web platform involves various aspects, such as leveraging existing web APIs, adhering to the web's security model, ensuring portability, and allowing for evolutionary development. These objectives align closely with the overarching goals of Wasm. Notably, the Wasm Minimum Viable Product (MVP) aims to maintain a security level no less stringent than that of JavaScript modules. To facilitate this integration, several key points of interaction between Wasm and the web platform have been considered:

- **JavaScript API**
  A dedicated JavaScript API enables developers to compile Wasm modules, perform limited reflection on compiled modules, store and retrieve compiled modules from offline storage, instantiate modules with JavaScript imports, invoke exported functions from instantiated modules, and establish memory aliases, among other functionalities. It is important to note that these APIs may not be available in non-web environments.
- **Developer-facing display conventions**
  Similar to how browsers and JavaScript engines handle JavaScript artifacts and language constructs, Wasm adopts conventions for representing its constructs in a developer-friendly manner. For example, locations in Wasm binaries can be displayed similarly to JavaScript source locations, ensuring consistency across different contexts.
- **Modules**
  Wasm modules seamlessly integrate with the ES6 module system, facilitating interoperability between the two.
- **Names**
  Wasm modules utilize UTF-8 byte sequences to identify imports and exports. To align with web conventions, a mapping of export names to exports is represented as a JavaScript object, where each export is a property with a UTF-16 encoded name. On the web platform, successful transcoding of names to UTF-16 is a validation requirement for Wasm modules.
- **Security**
  Wasm's security model aligns with the web's same-origin policy, employing mechanisms such as cross-origin resource sharing (CORS) and subresource integrity to enable distribution through content delivery networks and support dynamic linking.
- **Future Features**
  When features such as Single Instruction, Multiple Data (SIMD), and garbage collection become supported, Wasm will adhere to established conventions, leveraging

specifications from existing standards such as SIMD.js and TC39 for SIMD operations and reusing backend implementations. With garbage collection support, Wasm code will be capable of referencing and accessing JavaScript objects, the DOM, and other WebIDL-defined objects.

By taking these considerations into account, Wasm integrates seamlessly into the web platform, making use of existing infrastructure while enhancing web applications with improved capabilities and performance.

2.7. Non-Web Embeddings

Although Wasm is primarily designed for web usage, it is important for it to be compatible with other environments beyond the web. These environments can vary from simple shells used for testing purposes to fully-fledged application environments found in data centers, IoT devices, and desktop or mobile apps. There may be a need to incorporate Wasm within larger software programs as well. Non-web environments may provide different sets of APIs compared to web environments, which can be easily discovered and utilized through features such as testing and dynamic linking. While JavaScript virtual machines such as node.js can support Wasm, the design of Wasm aims to ensure that it can function independently without relying on a JavaScript VM. The Wasm specification itself does not attempt to define a comprehensive portable library similar to libc. However, certain core features in Wasm semantics that resemble functions found in native libc are included as primitive operators in the core Wasm specification. For example, the "grow_memory" operator, similar to the "sbrk" function, is part of the core specification. In the future, additional operators similar to "dlopen" may be included. In cases where there is overlap between web and non-web environments, shared specifications can be proposed; however, these would be separate from the Wasm specification. An example of this is the Loader specification, which is being developed for both web and node.js environments and is distinct from the JavaScript specification. To achieve source code-level portability, it is expected that communities will create libraries that map source-level interfaces to the capabilities provided by the host environment. These libraries can be developed during either the build-time or runtime process. Wasm provides essential components such as feature testing, built-in modules, and dynamic loading to facilitate the creation of these libraries. Examples of anticipated libraries include Portable Operating System Interface (POSIX) and Specification and Description Language (SDL). Overall, by ensuring that non-web implementations do not rely on web APIs, Wasm can be used as a portable binary format across various platforms. This offers advantages in terms of portability, tooling, and language compatibility, as Wasm supports semantics at the level of languages such as C and C++.

2.8. Portability

Wasm's binary format is designed for efficient execution across various operating systems and instruction set architectures, both within and beyond the web environment. Efficient execution in different environments may require certain characteristics. Even if an execution environment lacks these characteristics, it may be able to execute Wasm modules by emulating the unsupported behaviors, although this could result in suboptimal performance. As the standardization of Wasm progresses, these requirements and adaptations for new platforms will be formalized. The portability of Wasm relies on execution environments that provide the following features:

- Eight-bit byte representation.
- Byte-level memory addressing.
- Support for unaligned memory accesses or reliable trapping mechanisms to emulate them.
- Support for 32-bit two-complement signed integers, with the option for 64-bit support.
- Compliance with the IEEE 754-2008 standard for 32-bit and 64-bit floating-point numbers, with a few exceptions.

- Little-endian byte ordering.
- Efficient addressing of memory regions using 32-bit pointers or indices. The wasm64 extension allows for linear memory larger than 4 GiB with 64-bit pointers or indices.
- Secure isolation between Wasm modules and other modules or processes running on the same machine.
- Guarantee of forward progress for all execution threads, even in non-parallel execution scenarios.
- Availability of lock-free atomic memory operators for 8-, 16-, and 32-bit accesses, including an atomic compare-and-exchange operator. The wasm64 extension requires lock-free atomic memory operators for 64-bit accesses.
- Wasm does not define specific APIs or system calls. Instead, it relies on an import mechanism in which the available imports are determined by the host environment. In web environments, functionality is accessed through the Web APIs provided by the web platform. Non-web environments have the flexibility to implement standard Web APIs, standard non-web APIs (such as POSIX), or develop their own custom APIs.

2.9. Interoperability of Wasm

The portable nature of Wasm's binary format enables it to run across a variety of platforms and environments. This cross-platform compatibility is central to Wasm's design philosophy, allowing developers to write code once and run it anywhere, whether in a desktop browser, a mobile browser, a server environment, or an IoT device. Wasm provides a foreign function interface (FFI) for interoperation with other languages. This means that Wasm can be used to build web applications as well as to provide high-performance components for Python, Ruby, PHP, Java, and .NET applications, among others. Wasm's design supports interoperability, ensuring that it can function seamlessly alongside traditional web technologies such as JavaScript and can access various web APIs. This section further explores the various aspects of Wasm's interoperability.

- **Interaction with JavaScript**
  Wasm is designed to complement and work side-by-side with JavaScript. Wasm modules can be loaded and executed from JavaScript, allowing developers to run complex computations using Wasm while leveraging JavaScript for less performance-sensitive tasks. This interaction between JavaScript and Wasm is facilitated by the Wasm JavaScript API, which includes functions to compile and instantiate Wasm modules. Furthermore, Wasm and JavaScript can share data using Wasm's linear memory, a contiguous and resizable array of bytes that JavaScript can read from and write to. This shared memory model allows for efficient communication between Wasm and JavaScript, enabling complex applications that utilize the strengths of both languages.
- **FFI**
  The FFI is a mechanism through which a program written in one programming language can call routines or make use of services written in another [14,15]. FFI is used when two languages do not share a common interface or when they cannot interact directly with each other. Wasm utilizes an FFI to enable interoperability with JavaScript and other programming languages. Through the FFI, Wasm can call functions written in JavaScript and vice versa. This is critical because it allows Wasm to leverage the extensive ecosystem of JavaScript libraries and the broader web platform. The FFI plays a significant role in Wasm's integration with other languages in non-web contexts as well. Languages such as Rust, Go, and C/C++ can compile to Wasm and interact with other systems or libraries through the FFI. This interaction is critical for many use cases, especially when it comes to system-level programming in fields such as IoT. However, working with the FFI can be quite complex due to the difference in memory models between Wasm and other languages. Wasm's linear memory model is different from the models used by most programming languages, and translating between the two can be challenging. Solutions such as the Wasm Interface Types

proposal are aimed at making this easier, and interoperation between Wasm and other languages is expected to become more seamless as these improvements are adopted.

- **Access to Web APIs**
  Wasm can indirectly access various web APIs through JavaScript. While Wasm cannot directly call web APIs, it can call JavaScript functions that can interact with these APIs. This model provides a layer of abstraction that enhances security while enabling the rich functionality that web APIs offer. In the future, with the integration of the Wasm Interface Types proposal, Wasm modules will be able to directly interact with web APIs, enhancing its interoperability and functionality.
- **Compatibility Across Browsers**
  Wasm modules are designed to be portable, meaning that they can run on any platform that provides a compliant Wasm virtual machine. This includes all modern web browsers (Google Chrome, Mozilla Firefox, Safari, and Edge). This cross-browser compatibility ensures that a Wasm module developed on one platform can run seamlessly on another, enhancing the reach and usability of web applications built with Wasm.
- **Use in Non-Web Environments**
  Although Wasm was designed for the web, properties such as its compact binary format, sandboxed execution environment, and deterministic behavior make it an attractive option for non-web environments as well, including IoT devices, edge computing, and serverless computing. Wasm modules can be run outside of a web browser using a runtime such as Wasm System Interface (WASI), further enhancing its interoperability. This broad interoperability of Wasm in terms of working with other web technologies and compatibility across a wide range of platforms is one of its key strengths. As Wasm continues to evolve its interoperability is likely to expand, further increasing its potential use cases and influence in the web development and IoT ecosystems.

2.10. Interfacing with the Host Environment

- **How WebAssembly Interacts with the Host**
  Wasm modules are designed to be embedded within a host environment, which in most cases is a web browser [16]. A Wasm module cannot interact with the host environment directly; instead, it must interact through JavaScript using a set of explicit imports and exports. This mechanism allows Wasm to interact with JavaScript APIs, access the Document Object Model (DOM), and more, all while maintaining a clear boundary between the Wasm module and the host environment.
- **Web APIs and JavaScript Glue Code**
  To interact with web APIs, Wasm relies on JavaScript "glue" code that serves as an intermediary. This glue code can call web APIs and pass the results back to the Wasm module. This mechanism allows Wasm to take advantage of the wide range of APIs available on the web, such as those for accessing hardware capabilities, networking, storage, and more.
- **WebAssembly JavaScript API**
  Wasm provides its own JavaScript API, which is used to load, compile, and instantiate Wasm modules. This API allows developers to interact with Wasm directly from their JavaScript code, providing a high degree of control over the lifecycle of Wasm modules.
- **Direct DOM Access**
  While Wasm cannot currently access the DOM directly, and must rely on JavaScript, there is an ongoing proposal to add direct DOM access to Wasm. This would allow Wasm to manipulate the DOM without having to go through JavaScript, potentially leading to performance improvements for certain types of applications.
- **Beyond the Browser**
  While the primary host environment for Wasm is the web browser, it is not limited to this context. Wasm can be hosted in non-web environments, for instance, standalone runtimes such as Wasmer and Wasmtime, or embedded in other applications. This

flexibility makes Wasm an appealing choice for a variety of use cases, from server-side applications to IoT devices and more.

2.11. The Security Model of WebAssembly

Wasm incorporates a security model with two primary goals: (1) protecting users from potentially malicious or faulty modules, and (2) providing developers with effective tools and measures to create secure applications within the given constraints [17].

- **User Protection**
  Wasm modules operate within a secure sandbox environment that uses fault isolation techniques to separate them from the host runtime. This ensures the following:
  - Applications run independently within the sandbox, and cannot escape without using appropriate APIs.
  - Deterministic execution is generally maintained, with limited exceptions.
  - Modules adhere to the security policies of their embedding. In web browsers, this includes restrictions imposed by the same-origin policy, while other platforms may have their own security models.
- **Developer Support**
  Wasm's design focuses on supporting the development of secure programs by eliminating dangerous features from its execution semantics while remaining compatible with C/C++ code.
  - Modules must declare all accessible functions and associated types during load time, even when utilizing dynamic linking. This enables implicit control-flow integrity through structured control flow. Immutable compiled code and lack of runtime observability protect Wasm programs from control flow hijacking attacks.
  - Function calls within Wasm must specify valid target indices that correspond to entries in the function index space or table index space.
  - Indirect function calls undergo runtime type signature checks, ensuring that the selected function matches the expected signature.
  - The use of a protected call stack prevents buffer overflows in the module heap and ensures safe function returns.
  - Branches within functions must point to valid destinations within the same function.
  - Variables in C/C++ are represented using primitives in Wasm, depending on their scope. Local variables with fixed scope and global variables are stored as fixed-type values indexed by their respective indices. Local variables are initialized to zero and stored in the protected call stack, while global variables reside in the global index space and can be imported from external modules. Local variables with uncertain static scope are stored in a separate user-addressable stack in linear memory during compilation. Bounds checking is performed at the region level to prevent out-of-bounds accesses. Future enhancements may include support for multiple memory sections and more advanced memory operations.
  - Traps are used to immediately terminate execution and signal abnormal behavior. They are triggered by operations such as invalid index access, mismatched type signature in indirect function calls, exceeding the maximum size of the protected call stack, out-of-bounds linear memory access, or illegal arithmetic operations.
- **Memory Safety**
  Wasm's execution semantics mitigate certain memory safety issues compared to traditional C/C++ programs. Local and global variables stored in the index space are protected from buffer overflows due to their fixed sizes and indexed addressing. Linear memory accesses are checked at the region level, preventing out-of-bounds access. However, control-flow integrity and the protected call stack prevent direct code injection attacks. Hence, mitigations such as data execution prevention and stack smashing protection are unnecessary for Wasm programs. However, certain types of bugs are not obviated by Wasm's semantics. Control flow hijacking attacks can manipulate module

control flow using code reuse attacks against indirect calls. Nevertheless, common techniques such as return-oriented programming are not possible in Wasm due to enforced control-flow integrity. Additionally, race conditions, side channel attacks, and time of check to time of use vulnerabilities can occur. Future enhancements may provide additional protections such as code diversification, memory randomization, or bounded pointers.

- **Control-Flow Integrity**
  Control-flow integrity is crucial for measuring security effectiveness. Wasm provides implicit guarantees for direct function calls through function section indexes and for returns through the protected call stack. Indirect function calls undergo runtime type checks to ensure coarse-grained control-flow integrity. Fine-grained control-flow integrity can be achieved through the use of the Clang/low level virtual machine (LLVM) compiler infrastructure, which offers built-in support for Wasm.
- **Clang/LLVM CFI**
  Enabling fine-grained control-flow integrity (CFI) using the Clang/LLVM compiler has several advantages. It provides enhanced defense against code reuse attacks involving indirect function calls, and performs function signature checks at the C/C++ type level. Although enabling this feature incurs a slight performance cost for each indirect call, future updates will optimize it by leveraging built-in support for multiple indirect tables.
- **Code Validation and Verification**
  Wasm implements a two-step validation process to ensure the safety of its modules. The first step involves structural validation to check whether the module follows the binary format specification. The second step includes control flow and type checking to ensure that the code is well-structured and does not perform illegal operations.
- **Safe Interoperability with JavaScript**
  Considering that Wasm is designed to work alongside JavaScript in the same web environment, it is crucial to ensure safe interoperability between the two. Wasm modules can interact with JavaScript through a set of explicit exports and imports, allowing for controlled interaction between the two. By strictly defining these interfaces, Wasm ensures that the security of the web platform is preserved.

3. WebAssembly and IoT

Wasm is proving to be a game-changer in the field of IoT. Its performance efficiency, security, and cross-platform compatibility make it an ideal choice for IoT applications. In this section, we delve deeper into the reasons behind this fit and the practical applications of Wasm in IoT.

3.1. Why WebAssembly for IoT?

Wasm offers features and benefits that are notably impactful in the IoT ecosystem. These facets, outlined below, enhance its applicability across a range of IoT devices, ranging from low-power sensors to more complex edge computing systems.

- **Performance Efficiency**
  Wasm, designed as a low-level binary format, delivers near-native performance, an essential consideration for IoT devices as they are often constrained by limited resources. The fast downloading, efficient parsing, and swift execution times offered by Wasm contribute to reduced latency, a significant requirement in real-time IoT applications. The IoT landscape is characterized by a multitude of interconnected devices, and these are often constrained by computational resources while needing to perform tasks efficiently and quickly. Wasm's performance efficiency presents a significant advantage in this context.
  - **Near-Native Speed**
    At its core, Wasm is a low-level binary format designed to be decoded and executed at near-native speed. Unlike traditional web scripting languages such

as JavaScript, which are parsed and executed at runtime, Wasm is a compiled binary, allowing it to run much faster. This speed is of paramount importance in IoT environments where real-time processing is often required.

  - **Resource Optimization**
    Wasm's binary format is designed to be both small in size and fast to parse, leading to a reduction in the required amount of resources. This makes it particularly suitable for IoT devices, which often have limited memory and processing power. Reduced binary size means less storage space and quicker download times, while faster parsing leads to improved execution speed, both of which are crucial to the performance optimization of IoT applications.
  - **Concurrent and Parallel Processing**
    Wasm is designed to support concurrent and parallel processing. It enables multi-threading capabilities using Web Workers in the web environment. This means that Wasm can execute instructions on multiple cores simultaneously, making it even more efficient for computationally intensive tasks. This feature can be highly beneficial for IoT environments in which tasks need to be performed in parallel, and can improve the overall performance of the system.
  - **Stream Compilation and Tiered Compilation**
    Wasm benefits from stream compilation, meaning it can be compiled and optimized while being downloaded. This leads to faster startup times, and becomes crucial in IoT scenarios, where quick application startup can be critical. In addition, modern Wasm engines use tiered compilation, a technique where the code is first compiled and optimized for speed, then re-optimized during execution for improved overall performance.
  - **Optimized Execution Environment**
    Wasm runs within a highly optimized execution environment which is designed to utilize the capabilities of modern CPUs to the fullest extent. It can take advantage of common hardware capabilities and processor features, further boosting execution speed. This is particularly useful for IoT devices, where the computational capabilities of the device need to be used efficiently.

- **Flexibility**
  Wasm's platform-agnostic nature adds flexibility to IoT development, enabling the same code to run across multiple devices and platforms. This universal compatibility eliminates the need for platform-specific adaptations, reducing development time and effort, which is critical in fast-evolving IoT ecosystems. Wasm's design as a platform-agnostic bytecode provides a high degree of portability and flexibility, particularly beneficial in the diverse landscape of IoT. This section delves into the aspects that make Wasm's portability and flexibility a critical advantage for IoT applications.

  - **Platform-Agnostic Nature**
    Wasm is designed to be platform-agnostic. It can be run on any device with a compliant Wasm runtime, irrespective of the underlying hardware or operating system. This portability is especially critical in IoT, where a wide range of devices need to be programmed, from small sensors to large industrial machines. Developers can write code once and run it on any IoT device, reducing the time, effort, and cost of development.
  - **Language Independence**
    Wasm allows developers to work in multiple high-level programming languages. At present, languages such as C/C++, Rust, and AssemblyScript can be compiled to Wasm. This means developers can choose the language that best suits their use case or expertise, adding to the flexibility of the development process.
  - **Interoperability with JavaScript**
    Despite being an independent format, Wasm is designed to be highly interoperable with JavaScript. It can leverage the existing web platform and interact seamlessly with JavaScript APIs. This attribute is essential in IoT contexts, where

Wasm and JavaScript can be used interchangeably or together within the same application, enabling a highly flexible and adaptive development approach.

- **Resource Efficiency**
  The binary nature of Wasm enables smaller file sizes, leading to faster downloads and reduced network overhead. This is particularly beneficial in IoT environments, where network bandwidth can be constrained and devices have limited storage and processing power.
- **Evolving Ecosystem**
  The ecosystem around Wasm, including tools, libraries, and runtime environments, is continuously evolving and improving. This evolving ecosystem enables developers to leverage new features and enhancements over time, adding to the flexibility and adaptability of Wasm in the IoT context.

- **Compactness and Energy Efficiency**
  Wasm's compact binary format allows smaller payloads over network communication, making it ideal for IoT networks, where bandwidth may be limited. Furthermore, the efficiency of Wasm execution can contribute to lower energy consumption in battery-powered IoT devices. Wasm's compact design and energy-efficient execution make it a particularly attractive technology for IoT applications, which often involve resource-constrained devices operating in environments where power supply may be limited or intermittent. This section, explores how Wasm's compactness and energy efficiency contribute to its suitability for IoT.

  - **Compactness and Efficient Execution**
    Wasm, as a low-level binary format, is far more compact than traditional text-based languages. This compactness has several implications:
    * First, smaller binary sizes mean faster transmission over networks, an essential attribute for IoT devices that often rely on low-bandwidth networks.
    * Second, small binary size leads to less memory usage on devices, an important aspect considering many IoT devices operate with limited memory.
    * Lastly, because Wasm is a low-level bytecode, it requires fewer processing cycles to execute than high-level languages, resulting in more efficient execution. These characteristics collectively enable Wasm applications to be faster and more responsive, which is especially important in real-time IoT applications.
  - **Energy Efficiency**
    Energy efficiency is a critical factor in the IoT landscape, where devices are often battery-powered or need to operate in energy-efficient modes. The execution efficiency of Wasm contributes to reduced power consumption. As a binary instruction set, Wasm is processed directly by the device, eliminating the need for resource-intensive operations such as parsing or bytecode interpretation. This efficiency results in lower CPU usage, which in turn leads to reduced energy consumption.
  - **Reducing Network Load**
    The compactness of Wasm results in less network load, as smaller file sizes require less bandwidth to transmit. This is a significant advantage in IoT environments, where network bandwidth may be a scarce resource. By reducing the amount of data that needs to be transmitted, Wasm can contribute to overall energy savings in IoT networks.
  - **Support for Energy-Efficient Programming Paradigms**
    Wasm's support for languages such as Rust, which emphasizes zero-cost abstractions and fine-grained control over system resources, allows for energy-efficient programming paradigms. Developers can optimize their code to manage resources effectively and minimize energy consumption, which is crucial for power-constrained IoT devices.

3.2. WebAssembly Support for IoT Programming

In the realm of IoT, programming languages play a crucial role in determining the performance, compatibility, and ease of development of applications. Wasm is language-agnostic by its very nature, and currently supports multiple programming languages, enabling developers to select the one that best suits their needs. In this section, we explore the languages that can be compiled to Wasm and their potential implications for IoT development. Wasm's support for multiple languages (C/C++, Rust, AssemblyScript, Go) opens the door for developers to utilize their preferred programming languages for IoT application development. This section covers the various languages that can be compiled to Wasm and their relative advantages for IoT applications. Table 1 presents a comparison of key supported languages in the context of Wasm for IoT.

Table 1. Comparison of key supported languages in the context of WebAssembly for IoT.

Features\Languages	C/C++	Rust	AssemblyScript	Go
Performance Efficiency	High, as they provide low-level hardware access and enable fine control over system resources.	High, it combines low-level control over system resources with a high-level syntax.	Comparable to JavaScript; suitable for less resource-intensive applications.	Generally slower than C/C++, but easier to write and maintain.
Memory Safety	Manual memory management can lead to errors and vulnerabilities.	Rust has built-in memory safety without garbage collection, effectively preventing common memory errors.	Memory management is similar to JavaScript and is garbage collected.	Go has a garbage collector which can impact performance but increases safety and ease of use.
Concurrency	Manual management, complex to handle but allows fine-grained control.	Advanced concurrency support with ownership and lifetime concepts.	Not inherently built-in, relies on the concurrency model of Wasm.	Native support for concurrent programming using goroutines.
Interoperability with Web Tech	Not designed with web technologies in mind, but can be used effectively with Wasm.	Compatible with web technologies through Wasm.	Being a variant of TypeScript, it has excellent compatibility with web technologies.	Fully compatible with Wasm, but not specifically designed for web technologies.
Learning Curve	Steep, especially for developers unfamiliar with low-level programming.	Moderately steep, but offers more safety guarantees than C/C++.	Easy for JavaScript/TypeScript developers.	Easier compared to C/C++ and Rust.
Ecosystem and Community	Mature and extensive libraries and tools. Large community.	Growing rapidly with increasingly robust libraries and tools.	Smaller community and less mature ecosystem, but growing.	Large community, mature ecosystem but Wasm support is still experimental.
Use Cases in IoT	System-level programming, performance-critical applications.	Safety-critical systems, applications where memory safety is paramount.	Applications with a strong web-based component, less resource-intensive applications.	Networking applications, distributed systems, less performance-critical applications.

3.3. WebAssembly and IoT Security

Security is a significant concern in IoT due to the distributed nature of devices and their exposure to various networks. This section delves deeper into the mechanisms through which Wasm enhances security in IoT applications. It discusses sandboxing, the features of Wasm, and how these mechanisms can help to mitigate common security risks in IoT. Considering the interconnectivity of devices and the often sensitive nature of the data being processed, ensuring that robust security measures are in place is crucial. Wasm's design inherently addresses several significant security concerns, making it a compelling choice for IoT applications. In this section, we explore various aspects of Wasm that can enhance security in IoT environments.

- **Secure Sandboxing**
  One of the central features of Wasm is that it runs in a secure sandbox environment [18]. This means that the Wasm code execution is isolated from the rest of the system, preventing it from engaging in uncontrolled interactions with other parts of the system. This feature is critical in the IoT context, where devices are often connected to

public networks, where they are vulnerable to various kinds of attacks. By confining each Wasm module to its sandbox, the damage an attacker can cause is limited.

- **Controlled Interactions with the Host Environment**
  In Wasm, all interactions with the host environment must be explicitly defined through imports and exports. This means that a Wasm module can only access the host resources that it has been explicitly granted access to. This capability provides a strong isolation between the Wasm module and the host system, limiting the potential attack surface and reducing the risk of unauthorized access to sensitive resources, a particularly desirable trait for IoT applications.
- **Verification and Validation**
  Wasm modules undergo a validation process before execution. This validation ensures that the module adheres to the Wasm specification and is free of certain types of errors, further enhancing its security. Moreover, certain Wasm compilers, such as those used in the Rust ecosystem, offer even more robust validation and verification processes, further enhancing the security of IoT applications.
- **Secure IoT Applications with WebAssembly**
  The security features offered by Wasm are not merely theoretical; they are being actively leveraged in real-world IoT applications. For instance, edge computing applications, which require processing data close to the source, often utilize Wasm for its secure execution environment. Similarly, industrial IoT solutions benefit from the security and isolation features of Wasm when dealing with critical infrastructure components.
- **Future Directions in WebAssembly and IoT Security**
  While Wasm already brings significant security benefits to IoT, the technology continues to evolve, and future enhancements may further bolster these security credentials. Initiatives such WASI aim to define a secure and portable interface for Wasm modules, broadening their applicability beyond the web browser while maintaining robust security guarantees. As IoT continues to expand and evolve, the security capabilities of technologies such as Wasm can be expected to evolve alongside.

3.4. WebAssembly and IoT Performance Optimization

Performance optimization is a critical aspect of IoT application development. IoT devices often operate on limited resources and have power efficiency requirements, making it essential to maximize computational efficiency and minimize resource usage. With its high-performance characteristics, Wasm serves as an ideal choice for optimizing IoT applications. This section delves deeper into how Wasm contributes to performance optimization in IoT, and explores how the performance benefits of Wasm can be leveraged to optimize IoT applications. It discusses aspects such as lower resource usage, reduced latency, and faster execution times, which are crucial for the smooth functioning of IoT applications.

- **Fast Execution Speed**
  Wasm is designed to offer near-native performance, which is crucial for IoT devices that often have to process complex tasks within strict time constraints. With its efficient binary format, Wasm instructions can be executed quickly and effectively, which results in faster application performance. This speed is particularly valuable in IoT environments, where real-time data processing can be critical.
- **Efficient Use of Memory**
  Wasm implements a linear memory model, which means all memory is managed in a single, contiguous block. This approach reduces memory fragmentation and allows more efficient memory usage. Furthermore, Wasm's compact binary format requires less memory for code storage. These memory management practices can greatly benefit IoT devices, which often operate with limited memory.
- **Streamlined Parsing and Compilation**
  Wasm's binary format is designed to be fast to parse and compile. Compared to traditional text-based languages, binary formats are significantly quicker to process. This feature can greatly speed up the time from fetching the code to executing it, an

important factor in IoT devices, as they often need to start quickly and perform their tasks in real-time.

- **Parallelism and Concurrency**
  Wasm is designed with parallelism and concurrency in mind. With the proposal of Wasm threads, developers can leverage multicore processors more effectively by parallelizing their computations. This feature can be especially beneficial in the IoT context, where devices equipped with multi-core processors can utilize Wasm to run computationally intensive tasks more efficiently.
- **Performance in Real-world IoT Applications**
  Wasm's performance features are not just theoretical; they have practical implications in the world of IoT. For instance, in edge computing, where latency is a critical factor, Wasm's fast execution speed and efficient memory usage can greatly improve application responsiveness. Similarly, in the realm of smart devices and wearables, where battery life and responsiveness are crucial, the efficient use of resources that Wasm allows can lead to significant improvements.

3.5. WebAssembly and IoT Networking

IoT systems are often characterized by networks of devices communicating with each other. Wasm's compact binary format can reduce the load on these networks and increase their efficiency. This section discusses the role of Wasm in IoT networking and how it can support efficient communication in IoT systems. With its cross-platform compatibility, security features, and performance efficiency, Wasm is well-positioned to support networking in IoT applications. This section discusses the role and implications of Wasm in IoT networking.

- **Cross-platform Communication**
  One of the challenges in IoT networking is the heterogeneous nature of IoT devices. They can have different operating systems, architectures, and capabilities. Wasm, with its cross-platform nature, can help to bridge these differences. Regardless of the device's specific hardware or operating system, a Wasm module can run on any device with a compliant Wasm runtime. This provides a common platform that can facilitate communication and data exchange across diverse devices.
- **Efficient Data Transfer**
  Wasm's compact binary format facilitates efficient execution while contributing to efficient data transfer. Compared to traditional text-based code, Wasm's binary format is smaller and quicker to load, which can result in less network usage and faster transfer speeds. This is particularly important in IoT networks, which can involve the transfer of large amounts of data.
- **Secure Communication**
  Security is a paramount concern in IoT networks due to the potentially sensitive nature of the data being transferred. Wasm contributes to secure communication in number of ways:
  - Wasm operates within a sandboxed environment, meaning that it is isolated from the rest of the system, helping to contain any potential security threats.
  - Wasm is designed to be integrated with existing web platform security mechanisms, meaning that it can leverage features such as HTTPS for secure data transfer.
- **Real-time Communication**
  Many IoT applications require real-time or near-real-time communication. Whether a security system detecting an intruder or a health monitor tracking a patient's vitals, quick and responsive communication can be critical. Wasm's efficient execution and fast parsing can help to facilitate this kind of real-time communication.
- **WebAssembly Networking in Real-world IoT Applications**
  Wasm's networking features are being leveraged in various IoT applications. For instance, in the realm of smart home devices Wasm can facilitate efficient and secure communication between devices, including lighting systems, thermostats, and security

cameras. In industrial IoT, Wasm can enable real-time monitoring and control of equipment across a network.

3.6. Practical Applications of WebAssembly in IoT

Wasm's compatibility with IoT is not merely theoretical, and it is already being utilized across numerous IoT applications. This section presents an array of these practical applications in which Wasm is proving its mettle.

- **Edge Computing**
  Edge computing refers to the paradigm of processing and analyzing data at or near the edge devices in an IoT network, rather than relying solely on centralized cloud infrastructure. By moving computational tasks closer to the data source, edge computing enables real-time processing, reduced latency, improved privacy, and bandwidth optimization. Wasm has emerged as a promising technology for practical applications in IoT edge computing environments.
  - One of the key advantages of Wasm in the context of IoT is its ability to execute efficiently on resource-constrained edge devices. IoT devices often have limited computational power, memory, and battery life, making it challenging to run complex applications directly on the devices. Wasm's compact binary format and efficient execution enable the deployment of lightweight and high-performance applications on IoT devices. This allows for the offloading of computational tasks from cloud servers to the edge, reducing network traffic and enabling faster response times.
  - Wasm's security features play a significant role in IoT edge computing. With the sandboxed execution environment provided by Wasm, the risk of malicious code execution or unauthorized access to sensitive data is mitigated. This is crucial in the context of IoT, where devices may be vulnerable to security threats. Wasm's security model ensures that only authorized and verified code can be executed on edge devices, enhancing the overall security posture of the IoT ecosystem.
  - In practical applications, Wasm in IoT edge computing can enable a wide range of use cases. For instance, it can facilitate real-time analytics and decision-making at the edge by running machine learning algorithms for sensor data processing, anomaly detection, and predictive maintenance. Wasm's low latency execution allows for immediate response to critical events and minimizes the need for constant communication with the cloud.
  - Additionally, Wasm can enable edge devices to interact with existing web technologies and APIs. This allows developers to leverage the vast ecosystem of web-based tools, libraries, and frameworks for building IoT applications. Wasm's interoperability with JavaScript and other web languages further simplifies the integration of edge devices into larger IoT systems.
  - Moreover, the flexibility of Wasm makes it suitable for dynamic edge computing environments. The ability to dynamically load and update Wasm modules on edge devices enables the deployment of over-the-air updates, application customization, and remote management of IoT deployments. This dynamic nature of Wasm empowers IoT systems with greater adaptability and scalability.
- **Smart Home Devices**
  Smart home devices in which various devices are interconnected and communicate with each other to create intelligent and automated living spaces play a crucial role in the IoT ecosystem. Wasm technology offers practical applications in enhancing the capabilities and performance of smart home devices within the IoT framework. There are a number of practical benefits to using Wasm in smart home devices:
  - **Efficient and Fast Execution**
    Wasm allows smart home devices to execute code with near-native performance. By compiling code into a compact and efficient binary format, Wasm enables

smart home devices to perform complex tasks and computations quickly, resulting in improved responsiveness and overall user experience.

- **Cross-platform Compatibility**
  Wasm provides a platform-independent execution environment, enabling smart home devices to run the same code across various operating systems and hardware architectures. This compatibility allows developers to build applications and services that work seamlessly on different devices, reducing development efforts and ensuring broader device support.
- **Enhanced Security**
  Security is a critical aspect of smart home devices, and Wasm contributes to strengthening device security. With its sandboxed execution environment, Wasm ensures that malicious code cannot access or manipulate sensitive data within the smart home device. This protection layer helps to prevent unauthorized access and safeguard user privacy.
- **Extensibility and Flexibility**
  Wasm allows developers to extend the functionalities of smart home devices by easily integrating third-party libraries and modules. This extensibility empowers developers to leverage existing libraries and tools to enhance the capabilities of smart home devices without having to rewrite the entire codebase, and promotes the creation of a vibrant ecosystem of reusable components and modules for smart home applications.
- **Offline and Edge Computing**
  Smart home devices often operate in environments with intermittent or limited internet connectivity. Wasm enables the execution of code offline and supports edge computing scenarios, where computation is performed locally on the device rather than relying solely on cloud services. This capability ensures that smart home devices can continue functioning even in situations where internet access is unavailable or unreliable.
- **User Interface and Interactivity**
  Wasm allows developers to create rich and interactive user interfaces for smart home devices. By leveraging web technologies such as HTML, CSS, and JavaScript, developers can design intuitive and visually appealing interfaces that enable users to seamlessly control and monitor their smart home devices. This flexibility in user interface design enhances the overall user experience and makes smart home devices more accessible to a wider range of users.
- **Over-the-Air Updates**
  Wasm facilitates over-the-air updates for smart home devices, allowing manufacturers to efficiently deliver software updates and bug fixes. With Wasm, updates can be delivered in a modular and incremental manner, reducing the impact on device performance and minimizing downtime. This capability ensures that smart home devices can benefit from the latest features, security patches, and improvements without requiring manual intervention.

- **Industrial IoT**
  The Industrial Internet of Things (IIoT) refers to the use of interconnected devices and systems in industrial environments, such as manufacturing plants, factories, and industrial facilities. Wasm technology has practical applications in enhancing the capabilities and efficiency of IIoT systems. There are several ways in which Wasm can be applied in the IIoT context:

  - **Real-time Data Processing**
    IIoT systems generate a vast amount of sensor data in real time. Wasm enables efficient and high-speed data processing by executing code closer to the edge. With its near-native performance, Wasm can process and analyze data streams rapidly, facilitating real-time decision-making and automation in industrial processes.
  - **IIoT Edge Computing**

In IIoT applications, low-latency processing is essential for time-critical operations. Wasm enables edge computing by allowing computations to be performed directly on edge devices or gateways. This reduces the dependency on cloud services and enables faster response times, making IIoT systems more resilient and efficient.

- **Compatibility and Interoperability**
  Wasm's platform-independent nature makes it an ideal technology for achieving compatibility and interoperability in IIoT environments. It enables seamless execution of the same code across different devices, operating systems, and hardware architectures, facilitating the integration of diverse systems and components in industrial settings.
- **Security and Safety**
  Industrial environments require robust security measures to protect critical infrastructure and sensitive data. Wasm enhances the security of IIoT systems by providing a sandboxed execution environment that isolates code execution and prevents unauthorized access or tampering. This helps in safeguarding industrial processes, ensuring data integrity, and mitigating potential cybersecurity threats.
- **Legacy System Integration**
  Many industrial facilities rely on legacy systems and equipment that may not natively support modern technologies. Wasm enables the integration of legacy systems by providing a bridge between old and new technologies. By compiling legacy code into Wasm modules, industrial organizations can leverage the benefits of IIoT without the need for a complete system overhaul.
- **Remote Monitoring and Control**
  IIoT systems often involve remote monitoring and control of industrial processes. Wasm enables the development of lightweight and efficient web-based user interfaces that can be accessed from anywhere using standard web browsers. This allows remote operators and engineers to monitor real-time data, control industrial equipment, and make informed decisions remotely, improving operational efficiency and reducing downtime.
- **Offline Operation and Fault Tolerance**
  Industrial environments may experience network disruptions or operate in remote areas with limited connectivity. Wasm enables IIoT systems to operate offline or with intermittent connectivity by executing code locally on edge devices. This ensures uninterrupted operation, data logging, and fault tolerance even in challenging network conditions.
- **Customization and Modularity**
  Wasm's modular architecture enables the creation of customized IIoT applications by combining pre-built modules and components. This promotes code reusability, accelerates development cycles, and allows for the creation of tailored solutions that address specific industrial requirements. It facilitates easy maintenance and updates by enabling modular replacement or addition of functionality.

- **Wearable Devices**
  Wearable devices have gained significant popularity in recent years, offering various functionalities and improving the way we monitor and interact with our health, fitness, and daily activities. With the emergence of Wasm, these wearable devices can benefit from its practical applications in the IoT ecosystem. There are a number of ways in which Wasm can be applied in the context of wearable devices:

  - **Enhanced Performance**
    Wasm enables wearable devices to efficiently execute computationally intensive tasks. By leveraging near-native performance, Wasm can handle complex algorithms and data processing tasks, providing a seamless user experiences without compromising battery life or device performance. This allows wearable devices to

perform tasks such as real-time health monitoring, fitness tracking, and advanced data analysis without significant lag or slowdown.

- **Cross-platform Compatibility**
  Wasm's platform-independent nature makes it an ideal technology for developing wearable applications that can run across different operating systems and hardware platforms. This cross-platform compatibility ensures that wearable devices can offer consistent functionality and user experiences regardless of the device or operating system being used. Developers can write code once and deploy it on multiple wearable platforms, saving time and effort in application development.
- **Offline Functionality**
  Wearable devices often operate in situations where internet connectivity may be limited or unreliable. Wasm enables offline functionality by allowing key application components to run locally on the wearable device. This ensures that critical functionalities such as heart rate monitoring or step counting can continue even when the device is not connected to the internet or a smartphone. Offline functionality enhances the usability and reliability of wearable devices in various scenarios.
- **Customizable User Interfaces**
  Wasm empowers developers to create customized user interfaces for wearable devices. With its ability to integrate with web technologies, Wasm allows for the development of interactive and dynamic user interfaces that can adapt to different screen sizes and input methods. This enables wearable devices to provide intuitive and personalized user experiences, enhancing user engagement and satisfaction.
- **Secure Data Processing**
  Data security and privacy are crucial considerations in wearable devices, especially when handling personal health and fitness information. Wasm's sandboxed execution environment provides an additional layer of security by isolating code execution from the underlying system. This helps to protect sensitive data and prevent unauthorized access or tampering, ensuring the confidentiality and integrity of user information.
- **Connectivity and Interoperability**
  Wearable devices often interact with other IoT devices, smartphones, or cloud services to provide a comprehensive user experience. Wasm facilitates seamless connectivity and interoperability by allowing wearable devices to communicate with other devices through standardized web APIs. This enables data sharing, synchronization, and integration with companion apps or cloud platforms, enhancing the functionality and versatility of wearable devices.
- **Efficient Application Updates**
  Wasm simplifies the process of updating wearable device applications. With Wasm, developers can deliver updates by sending optimized binary modules that can be quickly downloaded and executed on the device. This eliminates the need for time-consuming app store updates and enables wearable devices to receive the latest features and bug fixes promptly.
- **Extendable Functionality**
  Wearable devices often have limited resources and storage capacity. Wasm's modular architecture allows developers to create extensible applications by leveraging pre-built modules and components. This enables wearable device manufacturers to provide additional functionalities through modular upgrades, such as adding new sensors or integrating with third-party services, without the need for significant hardware changes or device replacements.

- **IoT Gateways**
  IoT gateways play a critical role in the practical implementation of Wasm in the IoT ecosystem. These gateways act as intermediaries between IoT devices and the cloud

or edge infrastructure, facilitating seamless communication, data processing, and management. There are a number of ways in which Wasm can be applied in the context of IoT gateways:

- **Protocol Translation**
  IoT gateways often need to support multiple communication protocols to interact with a diverse range of IoT devices. Wasm can be leveraged to implement protocol translation layers within the gateway. The gateway can efficiently convert data between different protocols using Wasm modules, ensuring compatibility and interoperability among various IoT devices and systems. This enables seamless integration of devices that use different communication standards, such as MQTT, CoAP, Zigbee, and Bluetooth.
- **Edge Computing and Data Processing**
  IoT gateways act as the edge computing nodes, bringing computational capabilities closer to the IoT devices and reducing latency. Wasm can be utilized within the gateway to execute compute-intensive tasks and perform data processing at the edge. By running Wasm modules on the gateway, critical data analysis, filtering, or aggregation can be performed locally, reducing the need to transmit large volumes of raw data to the cloud. This improves response time while minimizing bandwidth consumption and cloud infrastructure costs.
- **Security and Access Control**
  Wasm enhances the security of IoT gateways by providing a sandboxed execution environment for executing untrusted code. This allows the gateway to execute Wasm modules securely, isolating them from the underlying operating system and protecting the gateway from potential vulnerabilities or malicious code. Additionally, Wasm enables the implementation of access control policies, ensuring that only authorized modules can be executed on the gateway and granting specific permissions based on the module's trust level.
- **Gateway Management and Orchestration**
  Wasm can be leveraged in the management and orchestration of IoT gateways. Thanks to its modular and lightweight nature, Wasm modules can be used to implement gateway management functionalities such as configuration management, firmware updates, and remote monitoring. These modules can be dynamically loaded, enabling flexible and efficient gateway management without the need for complex software updates or downtime.
- **Offline Operation and Local Decision Making**
  IoT gateways often operate in environments with intermittent or unreliable internet connectivity. Wasm enables the gateway to operate offline by executing critical logic and decision-making processes locally. By running Wasm modules on the gateway, it can continue to function autonomously even when disconnected from the cloud or experiencing network disruptions. This ensures that essential operations such as real-time control, event processing, and local data analytics can be performed without relying on constant internet connectivity.
- **Intelligent Edge Analytics**
  Wasm allows IoT gateways to perform advanced analytics and machine learning tasks at the edge. By executing Wasm modules that incorporate pre-trained models or algorithms, the gateway can analyze sensor data in real time, identify patterns, detect anomalies, and derive valuable insights locally. This enables timely and intelligent decision-making at the edge, reducing the need to transmit raw data to the cloud for analysis and enabling faster response times for critical applications.
- **Scalability and Flexibility**
  Wasm offers scalability and flexibility in IoT gateway deployments. The modular nature of Wasm allows developers to build and deploy custom functionalities and services as lightweight modules. These modules can be dynamically loaded or

unloaded on the gateway, enabling easy customization and scalability based on specific use cases or changing requirements. Wasm's ability to run across different hardware architectures and operating systems further enhances the flexibility and compatibility of IoT gateway deployments.

– **Over-the-Air Updates**
Wasm simplifies the process of updating IoT gateway applications and functionalities. By delivering optimized Wasm binary modules, developers can perform over-the-air updates on the gateway, allowing seamless deployment of new features, bug fixes, and security patches without interrupting device operation. This ensures that the gateway remains up-to-date with the latest capabilities and improvements, enhancing the overall performance and security of the IoT ecosystem. Table 2 presents a comparison of practical applications in the context of Wasm in IoT.

Table 2. Comparison of practical applications in the context of WebAssembly in IoT.

IoT Application	Performance Efficiency	Security	Portability	Noteworthy Use Case
Edge Computing	Wasm's near-native performance enables real-time data processing on edge devices, reducing latency.	The sandboxed execution model isolates applications, enhancing security on edge devices that are often exposed to the network.	The same Wasm codebase can run across a variety of edge devices regardless of their hardware or operating system.	Wasm allows sophisticated computation close to the data source, minimizing bandwidth usage and latency.
Smart Home Devices	Efficient use of resources, enabling high-speed operation on devices with limited resources.	Sandboxed execution enhances the security of personal data often handled by these devices.	Cross-platform nature allows for a variety of smart devices to run the same Wasm applications.	Powers applications on devices like smart thermostats, home security systems, etc., enabling quick responses and reliable operation.
Industrial IoT	Ability to process complex tasks quickly is critical in industrial scenarios, e.g., real-time monitoring of industrial processes.	Strong isolation between applications minimizes the impact of security breaches in an industry setting.	The same code can run on a variety of industrial devices, reducing development efforts.	Wasm can be used for predictive maintenance, automation of tasks, etc., improving operational efficiency.
Wearable Devices	Wasm's efficient use of resources enables applications to run smoothly even on wearables with limited resources.	The sandboxed environment ensures user data on the devices remains secure.	The device-agnostic nature of Wasm means applications can run on various types of wearables.	Powers applications on devices like smartwatches, fitness bands, enabling efficient operation and real-time responses.
IoT Gateway	Enhanced due to edge computing, protocol translation, and efficient data processing using Wasm.	Fortified with Wasm's sandboxed execution for untrusted code and robust access control policies.	High, thanks to Wasm's compatibility across various hardware architectures and operating systems.	Protocol translation layers for seamless integration of diverse IoT devices and performing advanced analytics at the edge using pre-trained models.

4. Tools for WebAssembly Development

Several tools exist that can be used for Wasm programming and system development. This section discusses compilers, runtimes, build tools/frameworks, and the Wasm Binary Toolkit (WABT). Table 3 shows a comparison of Wasm compilers, Table 4 presents a comparison of Wasm runtimes, Table 5 presents a comparison of build tools/frameworks with Wasm support, and Table 6 presents a comparison of tools belonging to the WABT.

Table 3. Comparison of WebAssembly compilers.

Compiler	Language Support	Optimization	Community Support	Ecosystem	Performance	IoT Support	Development Complexity	Documentation
Emscripten	C/C++	High	High	Wide	Moderate	Yes	Moderate	Extensive
tinygo	Go	Moderate	Moderate	Growing	High	Yes	Low	Moderate
WARDuino	C	Low	Low	Arduino	Low	Yes	Low	Limited
wasm3	C/C++	Low	Moderate	Limited	Moderate	Yes	Low	Limited
AssemblyScript	TypeScript-like	Moderate	High	Growing	High	Yes	Moderate	Extensive
wasmino-core	C/C++	Low	Low	Limited	Low	Yes	Low	Limited
Binaryen	Multiple	High	Moderate	Wide	High	Yes	Moderate	Extensive
rustc (Rust)	Rust	High	High	Growing	High	Yes	Moderate	Extensive
Zigwasm	Zig	Moderate	Low	Growing	High	Yes	Moderate	Limited
fable-compiler	F#	Moderate	Moderate	Limited	Moderate	No	Moderate	Extensive
Pyodide	Python	Low	Low	Limited	Low	No	Low	Limited

Table 4. Comparison of WebAssembly runtimes.

Runtime	Performance	Compatibility	Language Support	Ecosystem	Security	IoT Support	Development Complexity	Documentation
Wasmer	High	High	Multiple	Growing	Moderate	Yes	Moderate	Extensive
WAMR	Moderate	Moderate	C/C++	Limited	High	Yes	Low	Limited
Node.js	Moderate	High	JavaScript	Wide	Moderate	Yes	Moderate	Extensive
Wasmtime	High	High	Multiple	Growing	High	Yes	Moderate	Extensive
WAVM	High	High	Multiple	Limited	High	Yes	Moderate	Limited
Deno	Moderate	High	JavaScript	Limited	High	Yes	Moderate	Extensive
Lucet	High	High	C/C++	Limited	High	Yes	High	Limited
wascc	Moderate	High	Multiple	Limited	High	Yes	Moderate	Limited
Kotlin/JS	Moderate	High	Kotlin	Growing	Moderate	Yes	Moderate	Extensive
WasmEdge	High	High	Multiple	Extensible	High	Yes	Moderate	Good

Table 5. Comparison of build tools/frameworks with WebAssembly support.

Tool/Framework	Language Support	Ecosystem	Performance	Build Optimization	Learning Curve	Browser Compatibility	Development Experience	IoT Support
CheerpX	C/C++	Limited	High	High	Moderate	Moderate	C/C++ Development	Limited
Go	Go	Moderate	High	High	Low	High	Go Development	Moderate
Webpack	JavaScript	Wide	Moderate	High	High	High	JavaScript Development	Limited
Rollup	JavaScript	Moderate	Moderate	High	Moderate	High	JavaScript Development	Limited
Blazor	C#	Wide	Moderate	Moderate	Moderate	Moderate	.NET Development	Limited
wasm-bindgen	Rust	Moderate	High	Moderate	Moderate	High	Rust and JS/Wasm	Limited

Table 6. Comparison of tools belonging to WABT.

Tool	Description	Purpose	Features	Performance	Usability
wat2wasm	Translate from Wasm text format to binary format	Conversion, compilation	Control over formatting, comments, labels	Fast and efficient	Easy
wasm2wat	Translate from binary format back to text format	Inspection, understanding of module structure	Control over output format	Fast and efficient	Easy
wasm-objdump	Print information about a Wasm binary file	Analysis, debugging	Control over output format	Fast and efficient	Moderate
wasm-interp	Decode and run a Wasm binary file using interpreter	Testing, debugging, analyzing behavior	Tracing function calls, step-by-step execution	Moderate performance	Easy
wasm-decompile	Decompile a Wasm binary into readable C-like syntax	Decompilation, code analysis	Control over decompilation settings	Varies depending on complexity	Moderate
wat-desugar	Parse .wat text form and print "canonical" flat format	Parsing, transformation	Support for s-expressions, flat syntax	Varies depending on input size	Moderate
wasm2c	Convert a Wasm binary file to C source and header	Porting, integration with existing codebases	Control over C code generation	Varies depending on input size	Moderate
wasm-strip	Remove sections of a Wasm binary file	File size reduction, optimization	Control over stripping options	Fast and efficient	Easy
wasm-validate	Validate a file in Wasm binary format	Verification, security	Detection of malformed or invalid modules	Fast and efficient	Easy
wast2json	Convert a file in wasm spec test format to JSON	Test conversion, JSON output	Associated wasm binary files generation	Varies depending on input size	Easy
wasm-opcodecnt	Count opcode usage for instructions	Instruction analysis	Detailed opcode statistics	Fast and efficient	Easy
spectest-interp	Read a Spectest JSON file and run tests in interpreter	Testing, validation	Compliance with Spectest suite	Varies depending on test complexity	Easy

4.1. WebAssembly Compilers

- **Emscripten**
  Emscripten is a popular compiler toolchain that enables the translation of C/C++ code into Wasm, allowing developers to run code written in these languages on the web [19].
  - Pros
    - Wide adoption and active community support
    - Provides high compatibility with existing C/C++ codebases
    - Offers integration with popular web frameworks and libraries and provides advanced optimization options
  - Cons
    - Requires a more substantial toolchain setup and configuration
    - Compilation process can be slower compared to other compilers
    - Code output can be larger in size, impacting load times
- **TinyGo**
  The TinyGo is a compiler is specifically designed to compile Go code into Wasm [20]. It aims to provide a minimal and efficient Go runtime for running Go applications in resource-constrained environments.
  - Pros
    - Produces small Wasm binaries optimized for size and speed
    - Provides good compatibility with the Go programming language

    - Supports direct interaction with JavaScript APIs, offering a lightweight and fast compilation process
  - Cons
    - Limited support for certain Go language features and standard library packages
    - Less mature compared to other Go compilers
    - Smaller ecosystem and community compared to mainstream Go

- **WARDuino**
  WARDuino is a compiler that targets the Arduino platform, allowing developers to write programs in a subset of the C language and compile them to run on Arduino devices [21].
  - Pros
    - Enables developers to write Arduino programs using a higher-level language
    - Provides abstraction and simplification for Arduino programming Offers compatibility with Arduino libraries and hardware
  - Cons
    - Limited to Arduino-specific use cases
    - Less support for advanced features compared to general-purpose compilers
    - Requires familiarity with the Arduino platform

- **wasm3**
  Wasm3 is a lightweight Wasm runtime and interpreter that allows Wasm modules to be run efficiently [22]. It is designed for resource-constrained environments and has a small footprint.
  - Pros
    - Lightweight and efficient runtime suitable for embedded systems and IoT devices
    - Provides fast startup and execution times
    - Supports multiple Wasm language bindings
  - Cons
    - Limited support for some advanced Wasm features
    - Lacks extensive tooling and development ecosystem compared to larger runtimes
    - Limited support for interacting with host environments

- **AssemblyScript**
  AssemblyScript is a programming language similar to TypeScript that is designed to compile to Wasm [23]. It allows developers to write high-level code that compiles to efficient Wasm modules.
  - Pros
    - Familiar syntax and tooling for TypeScript developers
    - Provides type safety and high-level abstractions
    - Offers seamless integration with JavaScript code and libraries
    - Generates efficient Wasm modules
  - Cons
    - Limited support for advanced JavaScript features
    - Less mature compared to mainstream programming languages
    - Smaller community and ecosystem compared to other languages

- **wasmino-core**
  Wasmino-core is a compiler and runtime for Wasm modules specifically designed for running Wasm on resource-constrained microcontrollers [24].
  - Pros

    - Optimized for microcontrollers and embedded systems
    - Provides a lightweight runtime with a small memory footprint
    - Supports C/C++ programming languages
  - Cons
    - Limited to microcontroller-specific use cases
    - Less extensive tooling and community support compared to mainstream compilers
    - Limited support for advanced Wasm features

- **Binaryen**
  Binaryen is a compiler infrastructure project that provides a suite of tools and libraries for working with Wasm [25]. It includes a Wasm optimizer, a binary format parser, and various other utilities.
  - Pros
    - Offers powerful optimization capabilities for Wasm modules
    - Provides a flexible and modular toolset for working with Wasm binaries
    - Supports multiple input languages, including C/C++ and Rust
  - Cons
    - Primarily a development library and toolset, not a standalone compiler
    - Requires integration into custom build systems or toolchains
    - Advanced features may require more expertise to utilize effectively

- **rustc (Rust)**
  Rustc is the official compiler for the Rust programming language [26]. It can compile Rust code to various target platforms, including Wasm.
  - Pros
    - Offers a modern and expressive programming language with strong memory safety guarantees
    - Provides efficient code generation and optimization for Wasm
    - Offers seamless interoperability with JavaScript and other Wasm modules
    - Benefits from the Rust ecosystem and active community support
  - Cons
    - Requires familiarity with the Rust programming language
    - Compilation times can be slower compared to other compilers, and it may have a learning curve for developers new to Rust.

- **Zigwasm**
  Zigwasm is a compiler that enables developers to write code in the Zig programming language and compile it to Wasm [27].
  - Pros
    - Offers a modern, expressive, and safe programming language with a focus on performance
    - Provides seamless interoperability with C and other Wasm modules
    - Generates efficient Wasm code with low overhead
  - Cons
    - Smaller community and ecosystem compared to mainstream programming languages
    - Limited tooling and library support compared to more established languages
    - May require familiarity with the Zig programming language

- **fable-compiler**
  Fable-compiler is a compiler specifically designed for the F# programming language, targeting JavaScript and Wasm as output formats [28].

  - Pros
    - Allows developers to write code in F# and compile it to Wasm
    - Offers seamless integration with JavaScript and popular web frameworks
    - Provides interoperability with JavaScript libraries
  - Cons
    - Limited to F# programming language use cases
    - Smaller community and ecosystem compared to mainstream languages
    - Less mature compared to other compilers

- **Pyodide**
  Pyodide is a project that aims to bring the Python programming language to the web by compiling it to Wasm. It provides a way to run Python code directly in the browser [29].
  - Pros
    - Enables running Python code in the browser without the need for a Python interpreter
    - Provides access to a wide range of Python libraries and packages
    - Allows for interactive Python programming on the web
  - Cons
    - Limited to Python-specific use cases
    - Python runtime in the browser may have performance limitations compared to native Python execution
    - Smaller ecosystem and community compared to mainstream Python

4.2. WebAssembly Runtimes

- **Wasmer**
  Wasmer is a standalone runtime for executing Wasm modules. It supports multiple programming languages and provides a secure and efficient runtime environment [30].
  - Pros: Fast startup time, low memory footprint, supports multiple languages, supports ahead-of-time (AOT) and just-in-time (JIT) compilation, extensible with plugins.
  - Cons: Limited ecosystem compared to Node.js, relatively new compared to other runtimes.

- **WAMR**
  WebAssembly Micro Runtime (WAMR) is a lightweight and efficient runtime designed for resource-constrained devices [31]. It focuses on low memory usage and fast startup time.
  - Pros: Lightweight and efficient, designed for resource-constrained devices, supports AOT and interpreter execution modes, platform-specific optimizations available.
  - Cons: Limited language support compared to other runtimes, limited ecosystem and community support.

- **Node.js**
  Node.js is a popular JavaScript runtime that supports Wasm. It provides a rich ecosystem and extensive libraries and tools [32].
  - Pros: Vast ecosystem and community support, mature and stable runtime, easy integration with JavaScript code, wide range of libraries and tools available.
  - Cons: Relatively slower startup time and higher memory usage compared to some other runtimes, single-threaded by default.

- **Wasmtime**
  Wasmtime is a standalone runtime developed by the Bytecode Alliance. It focuses on security, compatibility, and performance [33].

  - Pros: High performance, supports AOT and JIT compilation, integration with the Cranelift code generator, compatible with multiple platforms.
  - Cons: Limited language support compared to Node.js, less mature compared to some other runtimes.

- **WAVM**
  Wasm Virtual Machine (WAVM) is a standalone Wasm runtime designed for high-performance applications. It focuses on providing low-latency execution [34].
  - Pros: High performance, low-latency execution, supports AOT and JIT compilation, compatible with multiple platforms.
  - Cons: Limited language support compared to Node.js, less mature compared to some other runtimes.

- **Deno**
  Deno is a secure JavaScript and TypeScript runtime built on the V8 engine. It provides first-class support for Wasm [35].
  - Pros: Secure by default, built-in module system, supports TypeScript natively, allows fine-grained control over permissions.
  - Cons: Relatively new and less mature compared to Node.js, limited ecosystem and library support.

- **Lucet**
  Lucet is a native Wasm compiler and runtime developed by Fastly. It is designed for high-performance serverless applications [36].
  - Pros: High-performance execution, optimized for serverless use cases, low startup time, efficient memory management.
  - Cons: Limited language support compared to Node.js, specialized for serverless environments.

- **wascc**
  Wasm Secure Capabilities Connector (wascc) is a lightweight Wasm runtime focused on secure and isolated execution of capabilities-based components [37].
  - Pros: Secure and isolated execution, supports capabilities-based security model, lightweight and efficient, extensible with plug-ins.
  - Cons: Limited language support compared to Node.js, specialized for capabilities-based component execution.

- **Kotlin/JS**
  Kotlin/JS is a language and runtime combination that allows writing Kotlin code that can be compiled to JavaScript or Wasm [38].
  - Pros: Seamless integration with Kotlin language and tooling, support for multi-platform development, interoperability with JavaScript and Wasm.
  - Cons: Limited language support compared to Node.js, less mature Wasm support compared to other runtimes.

- **WasmEdge**
  WasmEdge is a compact, efficient, and flexible runtime environment for Wasm that is designed to support a range of applications, including cloud-native, edge computing, and decentralized systems [39]. It provides the necessary infrastructure to run serverless applications, embedded functions, microservices, smart contracts, and IoT devices. WasmEdge offers high performance and can be easily extended to meet the specific needs of different use cases.
  - Pros: WasmEdge provides a secure execution environment, ensuring isolation and protection for system resources and memory space. It is recognized for its lightweight design and high-performance capabilities, making it suitable for serverless computing and resource-constrained environments.

  - Cons: WasmEdge is currently not thread-safe, which may restrict its usability in certain multi-threaded applications. While gaining popularity, its community support may not be as extensive as other well-established runtime environments, potentially impacting the availability of resources and community-driven libraries.

4.3. Build Tools/Frameworks with WebAssembly Support

- **CheerpX**
  CheerpX is a Wasm compiler that allows developers to compile C/C++ code into Wasm modules. It provides high-performance execution of C/C++ code in web applications, enabling developers to leverage existing codebases. CheerpX offers build optimizations to minimize code size and improve runtime performance [40]. However, it has a limited ecosystem and community support compared to other tools, and the learning curve can be moderate, especially for developers who are not familiar with C/C++.
  - Pros
    * Provides high performance by compiling C/C++ code to Wasm
    * Offers build optimizations to minimize code size and improve runtime performance
    * Suitable for leveraging existing C/C++ codebases in web applications
  - Cons
    * Limited ecosystem and community support compared to other tools/frameworks
    * Moderately steep learning curve, especially for developers not familiar with C/C++
- **Go**
  Go is a programming language that provides native support for Wasm [41]. With Go, developers can write web applications and compile them into Wasm modules. It offers high-performance execution, a rich ecosystem, and easy integration with existing Go projects. The learning curve for Go is relatively low, making it accessible to developers. However, the IoT-specific libraries and frameworks for Go may be limited compared to other languages, requiring additional effort for integrating with IoT-specific hardware.
  - Pros
    * Provides high-performance execution of Go code using Wasm
    * Offers a rich ecosystem with various libraries and frameworks for Go development
    * Easy integration with existing Go projects
  - Cons
    * Limited IoT-specific libraries and frameworks compared to other languages
    * May require additional effort for integrating Go with IoT-specific hardware
- **Webpack**
  Webpack is a popular build tool for JavaScript applications, including those targeting Wasm [42]. It offers extensive configuration options and advanced build optimization features such as code splitting and bundling. Webpack has a wide ecosystem and community support, making it a widely adopted choice. However, the learning curve for Webpack can be high due to its configuration complexity. Additionally, while it is a versatile build tool, out-of-the-box support for IoT-specific development may be limited.
  - Pros
    * Widely adopted build tool with extensive community support
    * Offers advanced build optimization features like code splitting and bundling
    * Provides a wide range of plugins and loaders for customizing the build process

  - Cons
    - Higher learning curve due to its extensive configuration options
    - Limited support for IoT-specific development out of the box

- **Rollup**
  Rollup is a JavaScript module bundler that provides efficient code bundling and tree-shaking capabilities. It is optimized for creating smaller bundles and better performance [43]. Rollup is well-suited for building libraries or smaller web applications where size and performance optimizations are crucial. However, more complex projects may require additional configuration. Similar to Webpack, Rollup has limited out-of-the-box IoT-specific support.
  - Pros
    - Optimized for creating smaller bundles and better performance
    - Offers tree-shaking capabilities to remove unused code
    - Well-suited for building libraries or smaller web applications
  - Cons
    - Requires additional configuration for more complex projects
    - Limited out-of-the-box IoT support

- **Blazor**
  Blazor is a web framework developed by Microsoft that enables developers to build interactive web applications using C# and .NET. It allows code sharing between server-side and client-side, providing a familiar development experience for .NET developers [44]. Blazor leverages Wasm to execute .NET code in the browser. While it offers the benefits of using the C# and .NET ecosystem, the performance of Blazor applications may be moderate compared to native JavaScript-based web applications. Additionally, IoT-specific libraries and frameworks for Blazor may be limited compared to JavaScript.
  - Pros
    - Allows developers to build web applications using C# and .NET ecosystem
    - Enables code sharing between server-side and client-side
    - Offers familiar development experience for .NET developers
  - Cons
    - Moderate performance compared to native JavaScript-based web applications
    - Limited IoT-specific libraries and frameworks compared to JavaScript

- **wasm-bindgen**
  Wasm-bindgen is a Rust library that facilitates seamless integration between Rust and JavaScript/Wasm. It enables efficient communication between Rust and JavaScript code, allowing developers to leverage Rust's performance and safety features in web applications [45]. Wasm-bindgen simplifies the interaction between Rust and JavaScript by automatically generating JavaScript bindings for Rust functions and data structures. However, using wasm-bindgen requires knowledge of both Rust and JavaScript, and while it offers high performance, out-of-the-box support for IoT-specific development may be limited.
  - Pros
    - Provides seamless integration between Rust and JavaScript/Wasm
    - Offers efficient communication between Rust and JavaScript code
    - Allows leveraging Rust's performance and safety features in web applications
  - Cons
    - Requires knowledge of both Rust and JavaScript for effective use
    - Limited out-of-the-box IoT-specific support

4.4. WABT: The WebAssembly Binary Toolkit

WABT is a powerful set of tools and libraries designed to work with Wasm binaries [46]. It provides developers with a comprehensive suite of utilities for manipulating, analyzing, and optimizing Wasm modules. The WABT toolkit is an open-source project that offers a wide range of functionalities to assist developers in working with Wasm. One of the key features of WABT is its ability to parse Wasm binary files and generate a structured representation of the module, allowing developers to inspect and analyze the contents of a Wasm module. This includes information about functions, imports, exports, memory layout, and more. By providing this low-level access to the internals of a Wasm module, WABT enables developers to gain insights into the module's structure and behavior, facilitating tasks such as debugging and optimization. WABT offers a set of powerful command-line tools that simplify the process of working with Wasm binaries. The toolkit includes tools such as "wasm2wat" and "wat2wasm" that allow conversion between the binary format and the human-readable Wasm Text Format (Wat). This is particularly useful when inspecting or modifying the code of a Wasm module, as the Text Format provides a more readable representation of the module's instructions and structure. Additionally, WABT provides tools for validating Wasm modules, ensuring that they conform to the Wasm specification. The "wasm-validate" tool can check the correctness and integrity of a WebAssembly module, helping developers to catch errors and potential security vulnerabilities early in the development process. This validation step is crucial for ensuring the safe execution of WebAssembly code. Another noteworthy component of WABT is the "wasm-opt" tool, which performs a variety of optimizations on WebAssembly modules. These optimizations can significantly improve the performance and efficiency of WebAssembly code, making it run faster and consume fewer resources. The "wasm-opt" tool applies transformations such as dead code elimination, function inlining, constant folding, and more, resulting in optimized Wasm modules that deliver better execution times and reduced memory footprint. In addition to the command-line tools, WABT provides a C and C++ API that allows developers to incorporate Wasm manipulation and analysis capabilities directly into their own applications. This API provides a convenient interface for working with Wasm modules programmatically, enabling tasks such as dynamic module loading, code generation, and custom analysis. The WABT provides a rich set of toolsets that assist developers in working with Wasm binaries. Among the key tools offered by WABT are:

- **wat2wasm**
  This tool enables the translation of Wasm text format (also known as .wat files) into the Wasm binary format. It allows developers to write and edit Wasm modules in a human-readable text format.
- **wasm2wat**
  The inverse of wat2wasm, wasm2wat converts Wasm binary files back to the text format. It provides a way to inspect and understand the contents of a Wasm module in a readable and understandable form.
- **wasm-objdump**
  This tool provides detailed information about a Wasm binary file. Similar to the traditional objdump tool used for object files, wasm-objdump can display various aspects of a Wasm module, such as sections, function names, and control flow structures.
- **wasm-interp**
  Using a stack-based interpreter, wasm-interp allows the decoding and execution of Wasm binary files. It provides a runtime environment to run and test Wasm modules without the need for a browser or dedicated runtime.
- **wasm-decompile**
  This tool decompiles Wasm binaries into a C-like syntax, making the code more readable and understandable. It can assist in analyzing and reverse-engineering Wasm modules.
- **wat-desugar**

Wat-desugar parses Wasm text files in various supported formats (such as s-expressions, flat syntax, or mixed) and prints them in a canonical flat format. It helps in standardizing and normalizing Wasm source code.

- **wasm2c**
  Wasm2c converts Wasm binary files to C source code and header files. This allows developers to integrate Wasm modules into existing C projects or embed them in C-based applications.
- **wasm-strip**
  This tool removes unnecessary sections from a Wasm binary file, reducing its size and removing any debug or non-essential information.
- **wasm-validate**
  Wasm-validate verifies the validity and correctness of a Wasm binary file. It performs a series of checks to ensure that the module adheres to the Wasm specification.
- **wast2json**
  Wast2json converts files in the Wasm spec test format to JSON files and generates associated Wasm binary files. This tool aids in running and analyzing the official Wasm specification tests.
- **wasm-opcodecnt**
  Wasm-opcodecnt counts the usage of individual Wasm opcodes or instructions within a binary file. It provides insights into the composition and distribution of instructions in a module.
- **spectest-interp**
  Spectest-interp reads a Spectest JSON file and executes its tests using the Wasm interpreter. It helps ensure compliance with the official Spectest suite for Wasm.

4.5. WASM Versus WASI Versus WAGI

Wasm, WASI, and WAGI are three important technologies that contribute to the advancement of Wasm and its applications in web and system development. This section delves deeper into each of these technologies and explores their unique characteristics and contexts. Table 7 presents a comparison of WASM, WASI, and WAGI.

- **Wasm**
  - Purpose: Wasm is designed to deliver high-performance code execution in web browsers and other environments. It aims to bridge the gap between high-level programming languages and the performance available with lower-level languages, allowing developers to run computationally intensive tasks on the web.
  - Use Cases: Wasm finds applications in various areas, including web applications that require near-native performance, porting existing applications to the web, gaming, multimedia, virtual reality, and augmented reality.
  - Advantages:
    * Performance: Wasm offers excellent performance due to its compact binary format, which allows for efficient parsing and execution.
    * Language-Agnostic: Wasm supports multiple programming languages, enabling developers to write code in their preferred language and compile it to Wasm.
    * Portability: Wasm is designed to be platform-independent, enabling code to run consistently across different systems and devices.
    * Web Ecosystem Integration: Wasm integrates seamlessly with the web platform, allowing access to web APIs and facilitating interactions with JavaScript.
  - Limitations:
    * Limited System Access: Wasm operates within a sandboxed environment, providing security by restricting direct access to system resources.

    - Lack of Standardized System Interface: Wasm does not define a standard system interface, which limits its direct interaction with the underlying operating system.

- **WASI**
  - Purpose: WASI extends the capabilities of Wasm by providing a standardized system interface [47,48]. It enables Wasm modules to interact with the host operating system in a secure and platform-agnostic manner.
  - Use Cases: WASI is particularly useful for system-level programming within the Wasm environment, where access to system resources such as file I/O, network communication, and system libraries is required.
  - Advantages:
    - Standardized System Interface: WASI defines a set of system calls and APIs that facilitate interactions with the host operating system in a consistent and portable manner.
    - Sandboxed Environment: WASI ensures secure and isolated execution of Wasm modules by enforcing restrictions on system-level interactions.
    - Portability: WASI enables Wasm modules to run on different platforms without modification, as it abstracts the underlying host operating system.
    - Compatibility: WASI allows developers to write code that can be executed on various WASI-compliant runtimes and environments.
  - Limitations:
    - Restricted System Access: Although WASI provides a standardized system interface, it imposes certain restrictions to maintain security and prevent unauthorized access to system resources.
    - Limited to Wasm Execution Environment: WASI is primarily focused on providing system-level interactions within the Wasm execution environment and does not extend to other technologies or platforms.

- **WAGI**
  - Purpose: WAGI is a specification that defines a standard interface for running Wasm modules as HTTP serverless functions [49,50]. It simplifies the deployment and scaling of serverless applications by leveraging the versatility of Wasm modules.
  - Use Cases: WAGI is particularly useful for building lightweight serverless applications, microservices, and APIs using Wasm modules.
  - Advantages:
    - Serverless Architecture: WAGI leverages the serverless paradigm, allowing developers to focus on writing business logic in Wasm modules while abstracting away server management and scalability concerns.
    - Language-Agnostic: WAGI supports serverless functions written in different programming languages that can be compiled to Wasm, providing flexibility for developers.
    - Scalability and Portability: WAGI simplifies the deployment and scaling of serverless applications by providing a standardized interface that can be used across multiple cloud providers and platforms.
    - Efficient Execution: Wasm's compact binary format and optimized execution enable fast and efficient execution of serverless functions.
  - Limitations:
    - Limited System Access: Similar to Wasm and WASI, WAGI operates within a sandboxed environment, preventing direct access to system-level resources.
    - Focused on HTTP Serverless Functions: WAGI is primarily designed for building HTTP-based serverless functions, and may not be suitable for all types of applications or use cases.

Table 7. Comparison of WASM, WASI, and WAGI.

Features	Wasm	WASI	WAGI
Purpose	High-performance code execution in web browsers and other environments	Standardized system interface for Wasm modules	Running Wasm modules as HTTP serverless functions
Use Cases	Web applications, gaming, multimedia, VR/AR, porting applications to the web	System-level programming within Wasm, secure sandboxed execution	Serverless applications, microservices, APIs
Portability	Platform-independent	Platform-independent	Platform-independent
Language Agnostic	Supports multiple programming languages	Supports multiple programming languages	Supports multiple programming languages
Integration with Web Platform	Seamless integration with web APIs and JavaScript	Interaction with host operating system in a secure and platform-agnostic manner	Leveraging Wasm modules for serverless functions
System Access	Restricted access to system resources	Standardized system interface for controlled system access	Restricted access to system resources
Scalability	Suitable for various scales of applications	NA	Scalable deployment and management of serverless functions
Security	Sandboxed environment with limited system access	Secure execution within controlled boundaries	Sandboxed environment with limited system access
Interoperability	Compatible with multiple runtimes and environments	Compatible with multiple runtimes and environments	Compatible with multiple cloud providers and platforms
Performance	Near-native execution speed	NA	High-performance execution for serverless functions
Ecosystem Support	Large and growing ecosystem of tools, libraries, and frameworks	NA	Growing ecosystem of WAGI-compatible tools and integrations
Community Engagement	Active community contributions and collaboration	NA	Active community involvement and contributions
IoT Applications	Efficient execution on resource-constrained IoT devices	Standardized system interface for secure interaction with IoT devices	Integration of Wasm modules for IoT gateway applications
Device Compatibility	Works on a wide range of IoT devices with Wasm support	Depends on the availability of a WASI-compliant runtime on the device	Works on IoT gateways capable of running WAGI-compatible servers
Low Power Consumption	Can be optimized for low power consumption in IoT scenarios	NA	Optimized for low power consumption in IoT gateway applications
Real-Time Processing	Supports real-time processing requirements in IoT applications	Depends on the capabilities of the underlying WASI runtime	Real-time processing capabilities for IoT gateway applications

NA represents no data availability.

4.6. WebAssembly Standardized Features for Existing Web Browsers and Tools

- **JS BigInt to Wasm i64 Integration**
  The initiation for Wasm *BigInt<->i64* conversion in the JS API has been incorporated into the main specification. As a result, BigInts now have the ability to convert back and forth between 64-bit integer Wasm values. This conversion can be applied to parameters and return values of exported Wasm functions, as well as parameters and return values of host functions. Additionally, *BigInts* can be used with imported and exported globals. Reading from or writing to Wasm memory using *BigInt64Array* or *BigUint64Array*, as introduced in the BigInt proposal, can be done without requiring any additional support.
- **Bulk Memory Operations**
  The introduction of Bulk Memory Operations and Conditional Segment Initialization in Wasm addresses specific needs and optimizations related to memory operations and segment initialization.

Bulk Memory Operations (BMO) were introduced in response to observations that functions such as *memcpy* and *memset* are frequently used and can significantly impact performance in Wasm benchmarks. This proposal aims to provide more efficient memory copying and filling operations. A prototype implementation of the *memory.copy* instruction was created, comparing it to an existing implementation generated by emscripten. The benchmark tests the performance of copying size bytes of data from one address to another without overlapping, and repeats this process N times. The results show potential for significant performance improvements with optimized bulk memory operations.

Conditional Segment Initialization (CSI) is motivated by the need to share Wasm modules among multiple agents. Under the current threading proposal, if a module is shared, it needs to be instantiated multiple times, potentially resulting in multiple initializations of linear memory and overwriting of stores. CSI introduces the concept of passive segments that are not automatically copied into memory or tables during instantiation. Instead, they require manual application using the new *memory.init* and *table.init* instructions. By distinguishing between active and passive segments, finer control over memory initialization is achieved, ensuring proper sharing of modules without data corruption.

The design of BMO involves new instructions such as *memory.copy*, *table.copy*, and *memory.fill*. These instructions enable efficient copying and filling of memory regions and tables. The binary format for the data section includes segments with a memory index, an initializer expression for offset, and raw data. The proposal repurposes the memory index field to serve as a flags field, allowing for additional meanings attached to nonzero values. Passive segments, indicated by the low bit of the flags field, are not automatically copied on instantiation, and require manual application using *memory.init* and *table.init* instructions. Segments can be shrunk to size zero using the data.drop and elem.drop instructions. As with passive segments, active segments have an implicit initialization and drop process as part of the module's start function. Furthermore, the reference-types proposal introduces the notion of function references, and allows element segments to be used for forward-declaring functions with addresses that will be taken. The proposal introduces the bulk instructions *table.fill* and *table.grow*, which take a function reference as an initializer argument.

The validation process and initialization behavior are updated with the bulk memory scheme. Active segments are initialized in module-definition order, with out-of-bounds checks to prevent instantiation failures. Data that have already been written for previous in-bounds segments remain unchanged.

Instructions such as *memory.init*, *data.drop*, *memory.copy*, *memory.fill*, *table.init*, *elem.drop*, and *table.copy* enable efficient segment initialization, copying, and filling operations in Wasm. The parameters and signatures of these instructions determine the source and destination addresses, offsets, and sizes, ensuring that bounds checks are performed to prevent memory access violations.

The introduction of the *DataCount* section addresses the issue of *memory.init* and *data.drop* instructions breaking the guarantee of single-pass validation. This new section provides information about the number of data segments defined in the Data section, enabling proper validation during the parsing of the Code section.

Overall, these proposals enhance the memory management capabilities of Wasm, allowing for more efficient memory operations, controlled initialization of segments, and optimized sharing of modules among multiple agents.

- **Extended Constant Expressions**

  This approach aims to expand the capabilities of constant expressions in Wasm. The current specification for constant expressions is limited, and was always intended to be extended. This proposal introduces new instructions that can be used in constant expressions, enhancing the flexibility and functionality of this feature.

The main motivation behind this approach comes from LLVM, where the use of integer addition in global initializers and segment initializers could provide significant benefits. These use cases are particularly relevant for dynamic linking, which is currently an experimental feature.
In dynamic linking scenarios, data segments are relative to a global import named *"__memory_base,"* which is supplied by the dynamic linker. However, the current limitations prevent the expression *"__memory_base + CONST_OFFSET"* from being used in segment initializers. As a result, all memory segments need to be combined into one. Additionally, the linker has to generate dynamic relocations for certain Wasm globals due to the inability to initialize a global with the value of *"__memory_base + CONST_OFFSET"*. This situation arises when the static linker determines that a symbol is local to the module, resulting in the creation of a new global that points to a data segment, i.e., its value is an offset from *"__memory_base"* or *"__table_base,"* which are themselves imported.
This concept introduces several new instructions that are considered valid constant instructions. These instructions include integer addition, subtraction, and multiplication for both 32-bit and 64-bit integers. The new instructions provide more flexibility and enable complex constant expressions to be used in Wasm programs.
The concept has made progress in terms of implementation across various platforms and tools. The *spec* interpreter, Firefox, Chrome/v8, wabt, LLVM, binaryen, and emscripten have all implemented support for the extended constant instructions. However, the implementation status in Safari is currently unknown.
This technqiue expands the capabilities of constant expressions in Wasm, addressing the limitations of the current specification. By introducing new instructions and enabling more complex expressions, developers can leverage enhanced functionality in their Wasm programs.

- **Multi-Value**
  The Multi-Value Extension approach aims to enhance the functionality of Wasm by allowing functions, instructions, and blocks to consume multiple operands and produce multiple results. While the current specification limits functions and instructions to at most one result, this proposal seeks to generalize them to accept and produce multiple values. The concept of multi-value semantics is well-understood and has been implemented in platforms such as V8.
  The motivation behind this concept is to address the asymmetries in the current Wasm design, which restrict functions and instructions to a single result. By enabling multiple return values for functions and instructions, several benefits can be achieved. This includes the ability to unbox tuples or structs returned by value, efficient compilation of multiple return values, and support for instructions that produce several results, such as divmod or arithmetic operations with carry.
  Additionally, allowing inputs to blocks opens up new possibilities. Loop labels can have arguments, enabling the representation of phis on backward edges and facilitating the future implementation of a pick operator that can cross block boundaries. The macro-definability of instructions with inputs is enabled, allowing for concise and expressive code patterns.
  This technique provides examples to illustrate the usage of multiple return values, instructions with multiple results, and blocks with inputs. These examples showcase how the extension enhances the flexibility and expressiveness of Wasm programs. For instance, a swap function can be defined to exchange two values, an addition function can produce an additional carry bit, and instructions such as divrem and add_carry can produce multiple results.
  This changes to the Wasm specification in this approach mainly involve the generalization of type syntax. The *resulttype* is extended from *[valtype?]* to *[valtype*]*, allowing for multiple values. Block types in block, loop, and if instructions are generalized

from *resulttype* to *functype*. Validation rules are adjusted accordingly, removing *arity* restrictions and replacing "?" with "*" in type occurrences.
The execution of multiple results involves replacing "?" with "*" and handling the entry of blocks with operands by popping the values from the stack and pushing them back after the label.
The binary format requires an extension to support function types as block types, allowing references to function types. The text format undergoes minimal changes, primarily in the syntax for block types, which is generalized to accommodate multiple values.
The scheme addresses the typing of administrative instructions and provides a formulation of formal reduction rules. Possible alternatives and extensions are under discussion, including more flexible block and function types using references to the type section or unifying the encoding of block types with function types.
An open question is whether locals are sufficient for destructuring or reshuffling multiple values, or if additional instructions such as pick or let should be introduced to enhance these capabilities.
In summary, the Multi-Value Extension expands the capabilities of Wasm by enabling functions, instructions, and blocks to consume and produce multiple operands and results. This extension enhances expressiveness, facilitates efficient compilation, and opens up new possibilities for code patterns and optimizations.

- **Mutable Globals**
  Import/Export Mutable Globals aims to address the inconvenience of providing mutable thread-local values in Wasm that can be dynamically linked. The proposal focuses on the ability to import and export mutable globals, specifically using the C++ stack pointer (SP) as a motivating example. While the examples revolve around the SP, the rationale can be extended to other thread-local variables or the TLS pointer itself. The motivation behind this concept stems from the limitation of not being able to import or export mutable globals in the MVP of Wasm. To illustrate the need for mutable globals, consider the scenario of dynamically linking two modules, m1 and m2, both utilizing the C++ stack pointer. In the MVP, this cannot be achieved directly. In a single-agent program, a workaround involves storing the stack pointer in linear memory. However, this solution fails when linear memory is shared, as each agent requires a different memory location for its stack pointer. The scheme presents several solutions to address the challenge of importing and exporting mutable globals.
  - Solution 1
    Use Per-Module Immutable Global: one approach is to utilize a per-module immutable global to store the location of the SP in linear memory. This can be extended to other thread-local variables as well. However, this solution has drawbacks, such as the need to read the global every time the SP is accessed and the SP in linear memory being vulnerable to to interference from other agents.
  - Solution 2
    Use Internal Mutable Global with Shadow in Linear Memory: a mutable global can be employed to optimize SP access. In this approach, the SP is stored in a mutable global, and when crossing module boundaries it is spilled to linear memory in the caller and loaded in the callee. Although this solution could be further optimized, such as spilling SP only when necessary, it incurs additional overhead for every function call and requires spilling and loading of other thread-local values in the same manner. Moreover, the SP remains susceptible to interference from other agents.
  - Solution 3
    Modify Function Signature to Pass SP as Parameter: instead of spilling the SP to linear memory, it can be passed as a parameter in function signatures. However, because it is not known whether an imported function will use the SP, all exported functions must be modified accordingly. The SP value is ultimately saved in a mutable global and loaded from the parameter at function entry points.

This approach provides an optimization, as passing the SP to all functions is unnecessary and it is only required for exported functions.
  - New Solution for Globals using Wasm.Global:
    the new solution suggests loosening the restriction on importing and exporting mutable globals to provide a better solution for thread-local values. This allows for the usage of mutable globals as thread-local storage. In the web binding, exported globals are of the Wasm.Global type rather than being converted to JavaScript Numbers. A Wasm.Global object comprises a single global value that can be simultaneously referenced by multiple Instance objects. Each Global object has two internal slots.

The *Import/Export Mutable Globals* scheme aims to enhance Wasm's capabilities by allowing the import and export of mutable globals. This enables the provision of mutable thread-local values that can be dynamically linked, addressing scenarios such as utilization of the C++ stack pointer. This proposal presents multiple solutions, highlighting the benefits and drawbacks of each approach. By allowing the import and export of mutable globals, Wasm gains greater flexibility in handling thread-local variables and opens up new possibilities for dynamic linking.

- **Reference types**
The Reference Types for Wasm proposal aims to introduce new types and instructions to enhance interoperation with the host environment and provide more flexibility within Wasm. The motivation behind this proposal includes easier and more efficient interoperability, manipulation of tables (paving the way for future extensions), and smoother transition paths to features such as garbage collection.
The proposal introduces the following key additions.
  - **New Types**
    The schem adds the type "externref," which can be used as both a value type and a table element type. Additionally, "funcref" is introduced as a value type.
    * **Table Instructions**
      Instructions are provided to get and set table slots, ensuring that basic manipulation of tables is possible within Wasm. Furthermore, missing instructions such as table size, grow, and fill are added.
    * **Multiple Tables**
      The scheme allows for the use of multiple tables, increasing flexibility and enabling more complex data structures.
  - **Possible Future Extensions**
    The technique mentions potential future extensions that could be explored.
    * **Subtyping**
      The introduction of a simple subtype relation between reference types to enable various extensions.
    * **Equality on References**
      Allows references to be compared by identity while ensuring implementation details are not observable in a non-deterministic manner.
    * **Typed Function References**
      Enhances the representation of function pointers and enables the easy passing of functions between modules.
    * **Down Casts**
      Introduces a cast instruction to enable the implementation of generics using anyref as a top type. It is worth noting that several of the future extensions mentioned here are covered by separate proposals, such as the Typed Function References proposal and the garbage collection proposal.

The *Reference Types* proposal for Wasm suggests the introduction of new types, instructions, and features to improve interoperability, table manipulation, and flexibility

within Wasm. These additions aim to enhance the overall capabilities of Wasm while setting the stage for potential future extensions.

- **Non-Trapping float-to-int Conversions**
  The Non-Trapping Float-to-Int Conversions technique aims to address the undefined behavior of float-to-int conversions in LLVM and establish a convention for saturating operations. The primary motivations behind this proposal are to improve the consistency of float-to-int conversions and provide a convention that can be shared with SIMD operations to avoid trapping.
  The motivation for this scheme stems from LLVM's undefined result for float-to-int conversions and the desire for SIMD operations to behave more like SIMD hardware without trapping. The proposal aims to establish a convention for saturating operations that can be used by SIMD operations, ultimately avoiding trapping. It is important to note that this proposal is driven by the desire for consistent and non-trapping conversions, not performance concerns.
  The issue of non-trapping float-to-int conversions has been discussed in various contexts, including GitHub issues and CG meetings. The discussions have involved considerations of performance effects and encoding strategies. While initial performance concerns were addressed through implementation fixes, no real-world performance problems related to this issue have been reported.
  The scheme introduces eight new instructions that provide saturating versions of float-to-int conversions for both signed and unsigned integers. The semantics of these instructions are the same as the corresponding non-saturating instructions, with the following differences:
  - Instead of trapping on positive or negative overflow, the saturating instructions return the maximum or minimum integer value, respectively, without trapping.
  - Instead of trapping on NaN, the saturating instructions return 0 without trapping.

  To accommodate the new instructions, a new prefix byte is introduced, labeled *"misc"*, which is intended to be used for future miscellaneous operations. The encodings for the new instructions utilize this prefix, allowing for clear differentiation from other instructions. The Non-Trapping Float-to-Int Conversions proposal presents a solution to the undefined behavior of float-to-int conversions in LLVM and introduces saturating versions of the instructions. By providing consistent and non-trapping conversions, this proposal aims to enhance the reliability and predictability of float-to-int conversions in Wasm.
- **Sign-extension Operations**
  The Sign-Extension Operators proposal focuses on introducing new sign-extension operators as a post-MVP feature for Wasm. This proposal aims to add support for sign-extending *8-bit*, *16-bit*, and *32-bit* values with the introduction of five new integer instructions.
  To facilitate sign-extension, five new sign-extension operators are proposed:
  - *i32.extend8_s: sign-extend a signed 8-bit integer to a 32-bit integer.*
  - *i32.extend16_s: sign-extend a signed 16-bit integer to a 32-bit integer.*
  - *i64.extend8_s: sign-extend a signed 8-bit integer to a 64-bit integer.*
  - *i64.extend16_s: sign-extend a signed 16-bit integer to a 64-bit integer.*
  - *i64.extend32_s: sign-extend a signed 32-bit integer to a 64-bit integer.*

  It was later added for consistency, as its behavior differs from *i64.extend_s/i32*.
  The instruction syntax and binary format are modified to accommodate the new sign-extension operators. The instruction syntax is expanded to include the new operators, while the binary format assigns specific values to each operator for encoding purposes. The Sign-Extension Operators proposal offers an enhancement to Wasm by introducing new instructions that enable sign-extension for various integer sizes. By incorporating these sign-extension operators, Wasm gains increased flexibility and functionality in working with signed integer values.

- **Fixed-Width SIMD**
  This specification presents an extension to Wasm that introduces a 128-bit packed Single Instruction Multiple Data (SIMD) feature. This extension is designed to efficiently leverage the capabilities of current instruction set architectures commonly used in hardware.
  The primary motivation behind this proposal is to enable Wasm to utilize the SIMD instructions available in hardware, thereby achieving performance levels close to native execution speed. SIMD instructions in hardware allow simultaneous computations on packed data within a single instruction, often employed to enhance multimedia application performance. This proposal defines a portable subset of operations that closely aligns with commonly used SIMD instructions found in modern hardware.
  The Wasm extension introduces a new value type called *v128* that is specifically tailored for SIMD operations. The v128 type corresponds to a 128-bit representation with bits numbered from 0 to 127. It serves as a mapping to a vector register in SIMD instruction set architectures. The interpretation of the 128 bits within the vector register is determined by individual instructions. When representing a v128 value as 16 bytes, bits 0-7 are stored in the first byte, with bit 0 as the least significant bit (LSB); bits *8-15* are stored in the second byte, and so on.
  The v128 SIMD type is versatile and capable of representing various types of packed data. For example, it can represent four 32-bit floating-point values, eight 16-bit signed or unsigned integer values, and more.
  Instructions in this specification follow a naming convention: *interpretation.operation*. The *interpretation* prefix indicates how the bytes of the *v128* type are interpreted by the *operation*. For instance, the *f32x4.extract_lane* and *i64x2.extract_lane* instructions perform the same operation of extracting the scalar value of a vector lane. However, the *f32x4.extract_lane* instruction returns a 32-bit floating-point value, while the *i64x2.extract_lane* instruction returns a 64-bit integer value.
  The v128 vector type interpretation treats the vector as a collection of bits. The *vlane_widthxn* interpretations (e.g., v32x4) view the vector as n lanes, each consisting of *lane_width* bits. The *tlane_widthxn* interpretations (e.g., i32x4 or f32x4) treat the vector as n lanes, with each lane having a type of *tlane_width*.
  Attempting to access SIMD types in Wasm module imports or exports from JavaScript results in an error.
  If an imported function in JavaScript expects or returns a *v128*-type argument or result, invoking that function immediately throws a *TypeError*.
  When instantiating a Wasm module using a *moduleObject*, an exception of type *LinkError* is thrown if a global value has the type v128 and the imported object is not a *Wasm.Global*.
- **Tail Calls**
  This scheme aims to enable tail call optimizations in Wasm, which are currently prohibited. Tail call elimination is crucial for correct and efficient implementations of languages that rely on this optimization, as well as for certain compilation and optimization techniques such as dynamic recompilation, tracing, and CPS. By introducing tail call support, Wasm can better handle control constructs such as co-routines, continuations, and finite state machines.
  Tail calls involve unwinding the current call frame before performing the call, regardless of any differences between the caller and callee functions. This proposal introduces explicit tail call instructions distinct from the regular call instructions that explicitly disallow TCE. Tail calls behave as a combination of a return instruction followed by a call instruction, unwinding the operand stack similar to a return and keeping only the necessary call arguments. While tail calls to host functions cannot guarantee tail behavior (beyond the scope of the specification), tail calls across Wasm module boundaries do guarantee tail behavior.

Tail calls are performed using separate call instructions designed explicitly for tail calls. This proposal introduces a tail version of each existing call instruction. An alternative approach involving a single instruction prefix applicable to every call instruction was considered and rejected by the *WebAssembly Community Group*. Although Wasm may include additional call instructions in the future, such as call_ref, the use of instruction prefixes as modifiers is not employed elsewhere in the Wasm specification.

Tail calls behave as a combination of return and call instructions, as they unwind the operand stack similar to a return instruction. They only retain the necessary call arguments. It is important to note that tail calls to host functions cannot guarantee tail behavior, whereas tail calls across Wasm module boundaries do ensure tail behavior. Tail call instructions are stack-polymorphic due to their combining call and return operations. Previously, there was a question of whether tail calls should induce different function types. Four possibilities were considered: distinguishing tail-callees by type, distinguishing tail-callers by type, both, or neither. After careful consideration, the fourth option was chosen as the simplest option conceptually. Experimental validation showed no significant performance benefit to the first option, and bifurcating the function space with the third option could lead to difficulties with function tables and dynamic indirection.

Validating the new instructions combines the typing rules for returns and the existing call instructions, resulting in stack-polymorphic behavior. The type of a *return_call* instruction with function index x is derived from the type of the function referred to by x. If x has a function type *[t1*] -> [t2*]*, then return_call x has the type *[t3* t1*] -> [t4*]*, where t3* and t4* can be any types provided that the current function has a return type of [t2*]. The same principle applies to *return_call_indirect* instructions.

The execution semantics of the new instructions involve popping the call operands, clearing and popping the topmost stack frame like a return instruction, pushing back the operands, and then delegating to the semantics of the corresponding plain call instructions.

The reserved opcodes following the existing call instructions are used for the new instructions in the binary format; specifically, *return_call* is represented by *0x12* and *return_call_indirect* is represented by 0x13.

Table 8 presents a comparison of existing WebAssembly standardized features in various web browsers and tools.

Table 8. Comparison of existing WebAssembly standardized features in various web browsers and tools.

Standardized Features	Chrome	Firefox	Safari	Wasmtime	Wasmer	Node.js	Deno	wasm2c
JS BigInt to Wasm i64 integration	85	78	Supported in desktop Safari since 14.1 and iOS Safari since 14.5	N/A	N/A	15	1.1.2	N/A
Bulk memory operations	75	79	15	0.2	1	12.5	0.4	1.0.30
Extended constant expressions	114	Enabled in Nightly, unavailable in Beta/Release	N/A	N/A	N/A	Requires flag –experimental-wasm-extended-const	Requires flag –v8-flags=–experimental-wasm-extended-const	N/A
Multi-value	85	78	Yes	0.17	1	15	1.3.2	1.0.24
Mutable globals	74	61	Yes	Yes	0.7	12	0.1	1.0.1
Reference types	96	79	15	0.2	2	17.2	1.16	1.0.31

Table 8. *Cont.*

Standardized Features	Chrome	Firefox	Safari	Wasmtime	Wasmer	Node.js	Deno	wasm2c
Non-trapping float-to-int conversions	75	64	15	Yes	Yes	12.5	0.4	1.0.24
Sign-extension operations	74	62	Supported in desktop Safari since 14.1 and iOS Safari since 14.5	Yes	Yes	12	0.1	1.0.24
Fixed-width SIMD	91	89	16.4	0.33	2	16.4	1.9	N/A
Tail calls	112	N/A	N/A	N/A	N/A	Requires flag –experimental-wasm-return-call	Requires flag –v8-flags=–experimental-wasm-return-call	N/A

Number in each cell represents the version number as on July 2023.

5. State-of-the-Art

In this section, existing research articles and review papers are presented. First, selected review articles are presented. Later, the rest of the literature is segregated into the categories of operating system development, IoT programming, virtual machine debugging, development environment, access control network, secure execution, containerization, serverless aspects, evaluation, and edge–cloud integration.

5.1. Selected Articles on IoT-Wasm

This subsection includes the articles that have discussed the use of Wasm in IoT and edge ecosystems in the recently published literature.

In [51], the authors introduced the use of Wasm as an application virtual machine in embedded systems, which are prevalent in IoT deployments. The work provides an overview of Wasm's basic concepts, the current runtime environments, and challenges involved in its efficient implementation in embedded systems. The paper concludes with a case study illustrating Wasm's practical applications and a discussion of unresolved issues and future work. Another study [52] explored the potential of Wasm for the development of comprehensive IoT applications. It posits that Wasm can run identical application code on a diverse range of devices in a headless mode outside of the web browser. To mitigate the issues of IoT edge devices related to latency, privacy, compatibility, and migratability, ref. [53] addresses the emerging interest, fueled by latency and privacy concerns, in computational offloading to the edge, along with the proliferation of smart and IoT devices. It focuses on code compatibility as a core challenge in achieving a stable edge-offloading platform due to the varied nature of edge devices. Furthermore, the article suggests leveraging Wasm for edge computing and compares its performance and potential to other solutions. Several methods to achieve migratability with Wasm are proposed, alongside an analysis of their trade-offs and deployment costs. In [54], the authors provided an overview of the evolution of virtualization technologies culminating in the advent of Wasm. It outlines how the focus has shifted from virtual machines to Linux containers and now to Wasm, a computation unit within a Linux process. Wasm, as an industry-wide collaborative effort by major tech players, offers improved security, speed, and portability for a range of applications, including serverless workloads. The paper discusses the application of Wasm in IoT devices, its potential in serverless deployments, and its built-in security capabilities. Other topics discussed include the stack-based virtual machine operation of Wasm runtimes, its universal bytecode format, and future proposals for additional data types in Wasm. The document introduces the WAS, which allows interoperability across different Wasm runtimes. The thesis in [55] presented a systematic literature review involving various use cases, key aspects, possibilities, and challenges in

the Wasm-enabled IoT ecosystem. Lastly, ref. [56] provides a practical guide to embedded programming in book form, focusing on low-powered devices and complex IoT systems using TinyGo and Wasm. It offers hands-on DIY projects to demonstrate how to build embedded applications and program sensors, and to work with microcontrollers such as Arduino UNO and Arduino Nano IoT 33. The book aim to educate its readers on the use of TinyGo to program and interact with various sensors, hardware devices, and network protocols, and guides the integration of TinyGo in modern browsers using Wasm.

5.2. OS Development

The operating system is a key component for managing the activities of Wasm. Wasmachine, an operating system designed to securely and efficiently run Wasm applications in IoT and Fog devices with constrained resources, is presented in [57]. Instead of the conventional approach, Wasmachine compiles Wasm ahead-of-time (AOT) into native binary, executing it in kernel mode to enable zero-cost system calls. To ensure memory safety, Wasmachine's OS kernel is implemented in Rust. By utilizing Wasm's sandboxing features, Wasmachine achieves high security. Benchmark tests have demonstrated Wasmachine's performance, revealing up to an 11% speed advantage over Linux. Adoption of Wasm for non-web environments is tricky due to its high portability and security. Another similar study [58] acknowledges the performance gap of Wasm with native code, primarily due to the conventional Wasm runtimes' lack of complex code optimization and the high overhead of system calls. It uses Wasmachine with an improved performance evaluations, with results indicating that Wasm applications running on Wasmachine are up to 21% faster than native Linux applications. Another work [59] introduces ThingSpire OS, an IoT operating system that integrates cloud–edge computing through Wasm. Wasm is central to ThingSpire OS's design, which uses it to ensure consistent execution across IoT devices, edges, and the cloud. ThingSpire OS employs a Wasm runtime based on Ahead-of-Time (AoT) compilation to facilitate efficient execution on resource-constrained devices. Other notable features include seamless inter-module communication regardless of the modules' locations and optimizations such as lightweight pre-emptible invocation for memory isolation and control-flow integrity. The abstract concludes with a brief mention of preliminary evaluations on the prototype of ThingSpire OS's intermodule communication performance.

5.3. IoT Programming

WiProg is an integrated IoT application programming solution based on Wasm, addressing the challenges of platform dependency and inefficient migration across device, edge, and cloud environments [60]. By enabling edge-centric programming through peripheral-accessing SDKs and computation placement annotations, WiProg optimizes for efficiency in the IoT application lifecycle. Through dynamic code offloading and compact memory snapshotting, WiProg ensures reduced energy consumption and execution time. Developers are provided with interfaces for customization of offloading policies. Real-world application and computation benchmark results demonstrate an average reduction in energy consumption and execution time between 18.7–54.3% and 20.1–57.6%, respectively. In [61], the authors introduced an automated static program analysis designed to enhance the security of Wasm applications. Noting the security challenges that have exposed users of Wasm websites to malicious code, they proposed a compositional analysis focused on information flow. For every Wasm function, a summary is computed to soundly depict where information from its parameters and the global program state can flow. The summaries can then be applied during subsequent function call analyses. The approximation of the information flow in the Wasm program is achieved through a classical fixed-point formulation. The proposed method proved effective on a set of 34 benchmark programs, with at least 64% of function summaries precisely computed in less than a minute. A sucre programming approach was discussed in [62] to investigate the feasibility of obfuscation techniques for Wasm programs. The authors explored numerous obfuscation techniques applied to both benign Wasm-based web applications and cryptojacking malware instances.

In [63] the authors used a WebAssembly interpreter (WAMR) embedded in the DoRIoT software stack to address issues of security, scheduling, and coordination. This approach offers a user-friendly programming environment that does not limit developers to hardware-oriented paradigms and languages, supporting standard building blocks and language choices by introducing a Wasm toolchain and architecture that supports the orchestration of multiple parallel requests.

5.4. Debugging

Traditional offline debugging techniques such as logs and dumps are unsuitable for IoT due to their overhead on devices and their inability to capture contextual information to identify the source of bugs. In [64], the authors discussed the extension of the WARDuino IoT platform with an online debugging tool called WOOD, designed to overcome the challenges of debugging IoT applications. Online debugging seems more fitting for IoT, as it allows remote debugging; however, it faces issues such as the probe effect, non-reproducibility, and high latency. The same work explores an out-of-place debugging technique that brings the state of a running application to the developer's machine, ensuring low latency remote debugging. It discusses features of WOOD such as capturing, moving and reconstructing debugging sessions, and updating live code. Similar work on "out-of-things" debugging is demonstrated in [65] to recognize the debugging as a major challenge for IoT developers due to unique properties such as non-deterministic data and hardware restrictions. Out-of-things debugging is always-on and works by transferring the state of a failing application to the developer's machine. The developer can then locally debug the application, apply necessary operations, and commit bug fixes to the device. This approach provides a flexible interface for accessing remote resources and mitigates debugging interference by eliminating network delays during debugging operations. The above work implemented the debugger on a Wasm Virtual Machine and demonstrated its suitability for IoT systems based on various metrics. A systematic study was conducted in [66] to understand and detect bugs in Wasm runtimes, a platform often used for various software applications including mobile and desktop apps. Cnosidering that bugs in Wasm runtimes can lead to application crashes, the researchers collected and studied 311 real-world bugs from GitHub posts and categorized them into 31 distinct bug types, providing common fix strategies for each. They developed a pattern-based bug detection framework to automatically identify bugs in Wasm runtimes and successfully discovered 53 previously unreported bugs in five popular Wasm runtimes. Of these, fourteen were confirmed and six were fixed by the runtime developers.

5.5. Virtual Machine

The VM is core to the Wasm environment. To investigate the potential of using VM approaches, ref. [67] used two minimal VMs, extended Berkeley Packet Filters (eBPF) and Wasm, on low-power microcontroller-based IoT devices. The authors designed rBPF, a register-based VM derived from eBPF, and compared it to a stack-based VM based on Wasm adapted for embedded systems. Each VM was implemented in the RIOT IoT operating system, with measurements performed on commercial IoT hardware. As expected, both Wasm and rBPF VMs introduced execution time and memory overhead compared to not using a VM. Interestingly, while the Wasm VM doubled the memory budget for a simple networked IoT application, the rBPF VM only added negligible memory overhead, making it a promising choice for hosting small updatable software modules on low-power networks. In [68], the authors proposed the use of Wasm, a VM standard, to simplify the programming of wearable sensor systems, which currently require different programming languages for different system components. The researchers demonstrated the possibility of implementing a Wasm interpreter for embedded systems such as the Texas Instruments CC2652R system-on-chip, enabling the same code to run across all system parts. Their proof-of-concept implementation used Bluetooth low energy for communication, meaning

that smartphones can program the device without special hardware. This suggests that Wasm could potentially streamline development processes for wearable sensor systems.

5.6. Development Environment Execution

A proof-of-concept integrated development environment (IDE) for executing Wasm modules on microcontrollers was demonstrated in [69], featuring a built-in web server that provides a browser-based IDE for direct AssemblyScript code development, compilation, and flashing to a device. This approach could help create a simplified and efficient operationalization context for data streaming and process adaptations in industrial devices. In [70], the authors introduced WaTZ, a system combining a secure runtime for trusted Wasm code execution in Arm's TrustZone Trusted Execution Environments (TEEs) and a lightweight remote attestation system optimized for Wasm applications. While TEEs protect software assets, they do not guarantee code trustworthiness or tamper-proofing. Remote attestation is used to assess the code pre-execution. WaTZ, which has been formally verified and evaluated using synthetic and real-world IoT tasks, provides comparable execution speeds to standard Wasm runtimes and about half the speed of native execution. The trade-off is enhanced security and interoperability offered by Wasm. A study by [71] delved into the potential of Rust, a promising young programming language, as a platform for IoT and ubiquitous computing. This work introduces these two concepts before moving on to discuss the current Rust ecosystem. It then examines how effectively the Rust ecosystem can meet the requirements of these domains, emphasizing the need for secure, high-performance, and efficient software writing with minimized potential for human error. In [72], the authors explored Wasm's performance in a desktop environment, providing empirical comparisons between programs compiled in native machine code and those compiled in Wasm. The main objective of the work was to examine Wasm's flexibility in compiling code written in different languages for web applications while maintaining performance comparable to native applications. The work highlights the potential for Wasm to serve as a semantic abstraction layer for embedded devices in Cyber–Physical Systems (CPS) development, despite existing issues related to semantic heterogeneity, maintainability, and development. Another work [73] presented a comprehensive study of standalone Wasm runtimes, aiming to elucidate their characteristics, which have been unclear due to limited academic research. This study covered five major standalone Wasm runtimes, and included the construction of a benchmark suite, WABench, used for testing. The study revealed that all studied runtimes introduce a performance slowdown when running Wasm binaries compared to native executions, indicating the need for effective and low-cost dynamic optimization. The same work observed architectural impacts such as increased branch prediction misses and higher cache pressure, providing insights for future Wasm deployment in non-web domains. Another study [74] evaluated the performance and portability of these technologies, demonstrating that Wasm generally outperforms JavaScript and can in certain cases match native code performance. Despite its limitations, Wasm shows potential benefits across various environments, with prospects for further growth outside the web. In [75], the implications of Wasm for browser-based computation were shown, noting its potential to enable universal applications that can run on any machine with a web browser. The paper proposed a design to enhance the performance of web applications using Wasm.

5.7. Access Control Framework

In the wake of growing demands for decentralized computing due to application latency requirements, privacy, and security, ref. [76] introduced Aerogel, an access control framework designed to bridge the security gap between bare-metal IoT devices and the Wasm execution environment, particularly regarding access control for sensors, actuators, processor energy usage, and memory usage. The framework views the runtime as a multi-tenant environment in which each Wasm-based application is a tenant. It utilizes the intrinsic sandboxing mechanisms of Wasm to enforce access control policies without

trusting the bare-metal operating system. Evaluation on a cortexM4 based development board (nRF52840) demonstrated that Aerogel can effectively enforce compute resource and peripheral access control policies with minimal runtime overhead and without additional energy consumption. Another study [77] introduced CAPLets, an authorization mechanism that extends capability-based security to provide fine-grained access control for multi-scale IoT deployments. CAPLets uses a robust cryptographic structure to ensure integrity while maintaining computational efficiency for resource-constrained systems. The system enhances capabilities with dynamic user-defined constraints to describe arbitrary access control policies. The same paper introduced CapVM, a Turing-complete virtual machine, along with eBPF and Wasm, to describe constraints. Empirical evaluation of CAPLets shows that it can express permissions and requirements at a granular level, aiding in constructing complex access control policies. The results demonstrate that CAPLets is significantly faster and more energy-efficient than current IoT authorization systems.

5.8. Secure Execution

In [78], the authors introduced a portable and efficient implementation of a Crystals–Kyber post-quantum key encapsulation mechanism (KEM) algorithm based on Wasm. The overall software is written in JavaScript with core performance components in Wasm. This software was shown to significantly outperform other JavaScript-based Kyber implementations in terms of key generation, encapsulation, and decapsulation. In [79], an identity-based cryptography library called CryptID was designed for efficiency and portability across various platforms, including desktop, mobile, and IoT. Enabled by Wasm, CryptID can operate both client-side and server-side. One unique aspect of CryptID is the use of structured public keys that can contain arbitrary metadata, allowing for a wide range of domain-specific use cases. In [80], a comprehensive review of recent malware and security threats related to IoT systems is provided. This review proposes a solution that allows secure applications to run on IoT devices in a secure execution framework by using verified bytecode in an isolated environment. This ensures that security is enforced by software and does not rely on the security features provided by the hardware components. Another survey paper [81] studied various techniques and methods for Wasm binary security and proposed future research directions regarding the current lack of Wasm binary security research. Memory safety aspect is very important to Wasm environment. In [82], the authors introduced MS-Wasm, an extension to Wasm that allows developers to capture low-level C/C++ memory semantics at compile time. At deployment time, Wasm compilers can use these added semantics to enforce different models of memory safety depending on user preferences and the hardware available on the target platform. MS-Wasm offers a range of security–performance trade-offs, and allows users to upgrade to stronger models of memory safety as hardware evolves. In [83], the security implications of cross-compiling C programs to Wasm are investigated. This work found that the execution of certain binaries produced different results across platforms for various reasons, including the use of different standard library implementations, lack of security measures in Wasm, and differing semantics of the execution environments. These findings suggest that porting existing C programs to Wasm may require source code adaptations to maintain security. Swivel was proposed in [84]. It is a compiler framework designed to harden Wasm against Spectre attacks, which can bypass Wasm's isolation guarantees. Swivel ensures that potentially malicious code cannot exploit Spectre attacks to break out of the Wasm sandbox or force other Wasm clients to leak secret data. The work presents two designs, a software-only approach and a hardware-assisted approach, and demonstrates that Swivel's overhead is much lower than existing defenses. Finally, a multi-layer secure framework dedicated to hardware-constrained devices, which is increasingly important considering the proliferation of IoT devices, was developed in [85].

5.9. Containerization

A lightweight container using Wasm that offers flexible deployment and live code migration in isomorphic IoT systems was proposed in [86] with the goal of supporting developers in creating liquid IoT applications that enable seamless and hassle-free use of multiple devices. Another study [87] investigated the use of Wasm for containers in IoT devices and compared it with Docker, the current leader in container technology. The findings showed that while Docker is more efficient in most cases, Wasm excels in scenarios involving sporadic execution of simple programs, suggesting its suitability for serverless computing at the edge. Another study found that performance of Wasm runtimes (e.g., Wasmer and Wasmtime) in terms of execution speed, memory footprint, and maturity, could outperform similar Docker solutions in certain instances, especially for short tasks on IoT devices [88]. In [89,90], attempts were made to replace Docker with Wasm. The results indicated that Wasm containers can achieve faster cold starts and higher throughput, suggesting their suitability for low-latency serverless edge computing with improved security and fault tolerance. An isomorphic microservice deployment architecture for IoT-based systems was proposed in [91]. It uses Wasm to achieve a uniform computing environment to simplify software deployment on future heterogeneous devices. Another study introduced WASMICO, a full-cycle management solution for containers built on the Wasm3 interpreter and the FreeRTOS operating system [92]. The platform allows developers to write programs in various programming languages, compile them to Wasm, and remotely manage tasks on low-end resource-limited IoT devices through an HTTP API and command line interface. It enables access to device capabilities through ready-to-use abstractions in higher-level specifications. The authors found that WASMICO performed better than other solutions in terms of efficiency, computation, and memory usage. By facilitating firmware development similar to software development, it can bridge the gap between these two computing paradigms.

5.10. Serverless

In [93], an examinination was conducted into a serverless edge computing architecture through a Wasm-based approach, comparing its performance to native and container-based applications using the ARM architecture toolchain. The benchmarks for comparison included several types of applications, including compute-intensive, memory-intensive, and file I/O-intensive applications, along with a basic image classification machine learning application. The paper presented the experimental setup, performance results, and conclusion. Sledge was proposed in [94], which is a Wasm-based serverless framework for edge computing. Sledge is designed for the unique needs of serverless workloads, including high-density multi-tenancy, low startup time, and short-lived computations. Using optimized scheduling policies and efficient work distribution, Sledge offers a lightweight function isolation model implemented with Wasm-based software fault isolation infrastructure. Compared to Nuclio, an open-source serverless framework, Sledge achieved up to four times higher throughput and four times lower latency. Another article [95] examined the use of Wasm as a lightweight alternative to container technologies such as Docker for serverless functions, with a focus on edge computing settings. The paper presented a Wasm-based runtime environment for serverless edge computing, called WOW, which significantly reduced cold-start latency, improved memory consumption, and enhanced function execution throughput on low-end edge computing equipment compared to Docker-based container runtime. Another work [96] discussed the recent developments that enable Wasm to be used for server-side applications and serverless functions. The goal of this work was to integrate Wasm with existing cloud and edge infrastructure. While Wasm has been identified as a solution to the cold start problem in serverless platforms, this study concluded that further work is needed and suggested the support of multiple runtime environments in addition to Wasm. Another study [97] investigated the potential of Wasm to transform serverless computing. Wasm's strengths, such as reduced cold start times, efficiency, easy portability, and compatibility with popular programming languages,

were evaluated using a benchmarking suite comprised of thirteen different functions. An initial step towards integrating Wasm runtimes with the APIs and command line interfaces of popular container runtimes was discussed. In the results the Wasm runtimes outperformed containered runtimes on ten out of thirteen tests, with Wasmtime being the fastest Wasm runtime. In [98], the authors introduced a nomenclature for characterizing serverless function access patterns to derive the basic requirements of a serverless computing runtime. Their paper proposed the use of Wasm as an alternative method for running serverless applications. They showed that a Wasm-based serverless platform can provide the same isolation and performance guarantees as container-based platforms while having reduced application start times and requiring fewer resources. A white paper expresses the security aspects of Wasm in the *libc* implementation [99]. This extension can help Wasm compilers and JITs to enforce different models of memory safety depending on user preferences and available hardware on the target platform. MS-Wasm offers a range of security–performance trade-offs, and allows users to move to progressively stronger models of memory safety as hardware evolves. The comprehensive survey in [100] discusses the suitability of serverless computing for IoT data processing at the edge in consideration of the limitations of existing solutions based on VMs and containers. It introduces a Wasm-based serverless framework, aWsm, to manage efficient serverless operations at the edge, with opportunities for function profiling and SLO-driven performance management. Initial assessments show aWsm performing efficiently, with an average startup time of 12 to 30 microseconds and an economical memory footprint ranging from 10 to 100s per kB for a subset of MiBench microbenchmarks used as functions.

5.11. Applications and Evaluation

In [101], the authors evaluated the use of Wasm to enhance the performance of JavaScript applications in IoT environments. Tests were conducted on a Raspberry Pi using the Ostrich Benchmark Suite. The study found that when JavaScript was executed in Wasm, performance improved by 39.81% in terms of execution time, there was a slight increase in memory usage, and battery consumption decreased by 39.86%. The authors of [102] delved into Wasm as a potential solution for performance issues in JavaScript applications, especially highly user-interactive websites and browser-based games. In a continuation of our previous research on the GraalVM ecosystem, we examined TruffleWasm, a Wasm based system hosted on GraalVM and Truffle Java framework, outlining the architecture and performance of TruffleWasm within the GraalVM-based ecosystem based on tests conducted in our academic environment. Another work investigated the correlation between energy consumption and the use of Wasm versus JavaScript [103]. This research showed that Wasm consumes less energy than JavaScript. It was found that Firefox has significantly lower energy consumption compared to Chrome for both Wasm and JavaScript. These findings indicate that utilizing Wasm for web application development can reduce energy consumption and improve Android device battery life. In another article [104], the authors sought to evaluate advances in JavaScript and Wasm execution environments across a variety of devices. Using the Ostrich benchmark suite, they assessed the improvements in JavaScript-based browser engines, a performance comparison of JavaScript and Wasm, the performance of portable versus vendor-specific browsers, and server-side versus client-side JavaScript/Wasm, with the aim of determining the best performing browser/language and device. The thesis in [105] provides a general introduction to Wasm and its potential to revolutionize web application development. After a review of articles and lectures, Wasm's applications and capabilities were evaluated and it was found to have the potential to change web and other applications in the future. This thesis demonstrates how to create a simple Wasm module with the Rust language to decode QR code data from images. In [106], the authors examined the performance of numerical computing on the web, including both JavaScript and Wasm, across a variety of devices. They created a new benchmarking approach for centralized benchmarking on mobile and IoT devices, and conducted four performance studies using the Ostrich benchmark suite,

then analyzed the performance evolution of JavaScript, the relative performance of Wasm, and the performance of server-side Node.js while providing a comprehensive performance analysis for a wide range of devices. The need for isomorphic software architectures in IoT development was investigated in [107]. Such architectures can allow for consistency in programming across various devices, enabling applications and their components to be deployed or migrated dynamically without any alterations to their original format. Another study [108] explored the use of Wasm beyond the browser environment. Wasm is a relatively new technology designed to provide a compilation target for numerous programming languages, with the aim of deploying both client and server applications on the web. This study particularly focused on partitioning Wasm applications into modules and linking them during execution, potentially reducing memory usage, binary size, and compilation and startup time. Another paper proposed a method for slicing Wasm programs to aid applications in reverse engineering, code comprehension, and security, among others [109]. Program slicing creates a minimal version of the original program that maintains the behavior at a specific location. The proposed approach, which focuses on Wasm's specific dependencies, faces the challenge of performing dependence analysis at the binary level. The approach was implemented and evaluated on a suite of real-world Wasm binaries, and its efficiency was compared to previous works. In [110], the authors addressed the limitations of existing container-based serverless computing systems, such as cold-start issues and complex architecture for stateful services and multi-tenancy. The proposed solution involves positioning serverless functions according to data locality and executing them as a web-assembly sandbox. This approach results in improved execution latency and reduced network usage compared to existing architectures. The serverless runtime designed for this solution provides resource isolation (CPU, memory, file-system) and supports multi-tenancy executions. The architecture's effectiveness was assessed using IoT workloads with various performance metrics.

5.12. Edge–Cloud Integration

Studies have discussed the increasing integration of IoT devices and cloud servers to enhance the efficiency of IoT applications [111,112]. While Wasm is considered a potential solution to overcome the heterogeneity between devices and servers, it presents challenges for resource-constrained devices. The authors of the aforementioned studies proposed WAIT, a lightweight Wasm runtime for such devices, which optimizes memory usage and ensures safe sandbox execution while reducing energy consumption. Several articles have discussed the challenges and potential of harnessing data from previously untapped sources in the manufacturing industry [113,114], including a concept and prototype for retrofitting aged machines using Wasm on edge devices. This concept provides a uniform development environment from the cloud to the edge, achieving near-native performance with modularity similar to container-based service architectures. Another work looked into the interest in edge computing and computational offloading due to latency and privacy concerns anlong with the spread of IoT devices. The authors assessed how popular technologies achieve portability and migratability, and highlighted the benefits of Wasm as a platform for femtocloud offloading. The same paper introduced Nomad, a Wasm environment capable of live-migrating across different systems and hardware architectures [115]. A thesis [116] investigated the use of Wasm for computational offloading at the edge in the context of edge computing and 5G. The aim was to improve program performance by reducing the execution time and energy consumption on the end device. A proof-of-concept offloading system was presented and evaluated through several use cases. Another work discussed the high demand for computing power due to the rapid growth of IoT and AI and how edge computing can meet these demands [117]. Existing offloading approaches based on virtual machines and containers are criticized for their slow operation and large memory footprint. This work proposes LAWOW, a lightweight and high-performance framework that utilizes the Wasm runtime wasmedge for task offloading, providing a cross-platform solution to reduce computation costs and memory footprint.

Identified Gaps:

- Analyzing the provided articles, the key research gaps in the field of Wasm, especially when applied to edge computing, IoT devices, and serverless architectures, appear to be as follows:
- **Efficient Execution on Resource-Constrained Devices:** [111] discusses the problem of executing Wasm on resource-constrained IoT devices. It appears that further research is needed to optimize the execution of Wasm applications on devices with limited memory, processing power, or energy resources.
- **Migration and Portability:** [108,109,115] highlight the need for more research into the migration and portability of Wasm applications. There is a need for frameworks or solutions that allow Wasm applications to be dynamically deployed, migrated, or partitioned without changing their form while working across different operating systems and hardware architectures.
- **Performance Analysis and Optimization:** while many of the discussed works [111, 113,114,116] involve efforts to optimize the performance of Wasm applications, it is clear that more research is needed in order to understand the performance implications of Wasm in different contexts, e.g., edge computing, IoT, offloading computational tasks, etc., and to further improve it.
- **Secure and Multi-Tenancy Executions:** [110] hints at the challenge of ensuring secure and isolated execution environments for Wasm in serverless computing contexts. Further research into ways of ensuring security and facilitating multi-tenancy executions using Wasm might be needed.
- **Applications in Industrial Retrofitting:** two articles [113,114] indicate a gap in the application of Wasm in retrofitted industrial equipment. More studies are needed in order to test and evaluate how Wasm can be implemented in real-world industrial settings.
- **Improving Energy Efficiency:** a few articles [111,116] touch on energy efficiency in running Wasm on IoT devices and computational offloading, respectively. However, more research could be carried out on ways of optimizing energy consumption when executing Wasm in different scenarios.
- **Generalization of Deployments:** multiple works [108,112] suggest a need for broader application of Wasm outside browser environments and across different hardware devices and software environments. This indicates a gap in how Wasm can be generalized and deployed beyond the specific use cases studied to date.

Lessons Learned:

- **Wasm is Versatile:** Wasm offers a highly flexible approach for executing code across a variety of environments, from IoT devices to cloud servers and from in-browser to edge computing contexts.
- **Performance Gains:** when correctly optimized, Wasm can yield substantial performance improvements in certain contexts even on resource-constrained devices. For example, [111] demonstrates how a correctly optimized Wasm runtime can achieve substantial reductions in RAM usage and energy consumption.
- **Benefit for IoT and Edge Computing:** Wasm can be beneficial in the contexts of edge computing and IoT. It can aid in overcoming issues of device heterogeneity, resource constraints, and the need for close-to-source data processing. In addition, it supports the idea of computational offloading, enabling more efficient execution of tasks.
- **Potential for Industrial Applications:** there are potential benefits of using Wasm in retrofitting industrial machinery for smart manufacturing (Industry 4.0), as suggested by [113,114]. It can help to modernize old equipment without substantial upfront investment, making it a cost-effective solution for businesses.
- **Migration and Portability Challenges:** despite its versatility, Wasm faces challenges regarding the portability and migration of applications. Efforts are being made to address these challenges, such as the development of solutions that allow Wasm applications to be dynamically deployed, migrated, and partitioned.

- **Security and Multi-tenancy:** while implementing secure and isolated execution environments for Wasm in serverless computing contexts can be complex, it is crucial in order to ensure the safety of operations and data.
- **Energy Efficiency:** while there is ongoing research into ways of optimizing energy consumption when executing Wasm, it is clear that energy efficiency is an important factor, especially when deploying on IoT devices or in computational offloading scenarios.
- **Non-Browser Deployments:** Wasm's use is not limited to the browser environment; it shows promise in various other environments, including the cloud–edge continuum, servers, and IoT devices. Its capabilities in these environments are currently being explored and better understood.

6. Key Challenges and Future Directions

Wasm and IoT integration is not without challenges. In this section, a comprehensive discussion of the current issues around integrating Wasm to IoT is presented. Later, an in-depth subsection deals with the future directions in this regard moving forward.

6.1. Limitations of Current WebAssembly Implementations

- **Lack of Garbage Collection**
  One of the major hurdles when dealing with Wasm in the context of IoT is the lack of native support for garbage collection. Garbage collection is a form of automatic memory management that can greatly simplify programming, and is a critical feature in many high-level programming languages. Wasm does not directly support garbage collection, which can complicate things for developers.
  - **Understanding the Problem**
    Garbage collection is the process of identifying and freeing up memory that is no longer in use by the program. It is used in many modern programming languages, including JavaScript, Python, Java, and Go. The absence of native garbage collection support means languages that rely on it cannot be fully or effectively compiled to Wasm.
    In IoT scenarios, where devices often have constrained memory resources, efficient memory management is essential. Without garbage collection developers have to manually manage memory, which is more challenging and error-prone as well as time-consuming.
  - **Current Solutions and Workarounds**
    For languages such as C and C++, which do not use garbage collection and allow developers to manually manage memory, this is not an issue. However, for languages that depend on garbage collection there are a few solutions and workarounds.
    - **Compile the Garbage Collector**
      One approach is to compile the garbage collector of the source language into Wasm along with the program itself. This can make the resulting Wasm module self-contained and capable of doing its own garbage collection. However, this can significantly increase the size of the resulting module, which might not be suitable for IoT devices with limited memory.
    - **Use WASI**
      The WASI provides a capability-oriented system interface that allows Wasm modules to interact with system resources, including memory management.
    - **Use Linear Memory**
      Wasm's linear memory can be manipulated manually from within Wasm code (or from JavaScript, if running in a browser environment). However, this places the burden of memory management onto the programmer, increasing complexity.

- **Interoperability Challenges**
  Interoperability refers to the ability of different systems or applications to communicate, exchange, and utilize data effectively. It is a critical aspect of modern technology stacks, in which multiple languages, libraries, and services often need to work together seamlessly. There are certain interoperability challenges that emerge when considering Wasm in the context of IoT, predominantly due to the varying nature of IoT devices and Wasm's unique operational dynamics.
  - **Communication between JavaScript and WebAssembly**
    While Wasm was designed to coexist with JavaScript in a web environment, communication between the two is not always straightforward. Currently, interaction between JavaScript and Wasm is mainly restricted to numeric data types. This constraint means that transferring more complex data types such as strings or objects between JavaScript and Wasm requires them to be broken down into simple numeric components or transferred using shared memory. The process of data conversion adds an extra layer of complexity to application development, and can introduce a performance overhead. It requires careful management to ensure data integrity, and can lead to errors if not properly implemented. While proposals such as Wasm Interface Types aim to improve this, the problem remains a significant challenge for developers.
  - **Compatibility With Various Devices**
    IoT devices come in various forms and use a range of software and hardware configurations. Certain IoT devices might use different architectures and operating systems, creating a need for additional layers of abstraction in the application code. Wasm's goal of "write once, run anywhere" promises to ease this problem by providing a universal binary format. However, the Wasm runtime needs to be ported to each new system architecture, which can require considerable effort. Even then, the device's resource constraints, such as memory, CPU power, or battery life, can limit the complexity or the number of Wasm modules that can be used.
  - **Interactions with IoT Protocols**
    IoT applications often communicate using IoT-specific protocols such as MQTT, CoAP, and DDS. Currently, Wasm cannot directly interact with these protocols because it has no direct access to the underlying system or network I/O operations. Therefore, developers need to rely on JavaScript or the host environment to manage these interactions, which may not always be feasible or efficient in an IoT context. The existing interoperability issues pose hurdles for developers, and addressing these challenges is a critical area of focus in the ongoing evolution of Wasm. With further development and enhancements, Wasm's impact on the IoT world could be revolutionary, turning these challenges into opportunities for creating more robust, efficient, and secure IoT applications.
- **Memory Management**
  The manual memory management required by Wasm's design can pose challenges for developers used to languages with automatic memory management. This can lead to a steep learning curve and potential memory management issues. In the context of Wasm and IoT, managing memory efficiently presents a distinct set of challenges. The uniqueness of the Wasm memory model and the often resource-constrained nature of IoT devices amplify these challenges.
  - **Wasm's Linear Memory Model**
    Wasm employs a linear memory model in which all data (both code and state) are stored in a large contiguous block of memory. This model is simple and efficient, making it easy for the Wasm virtual machine to manage memory. However, it means that the application must manually manage memory allocation and

deallocation. This is a complex task and prone to human error, with potentially serious consequences such as memory leaks or buffer overflows.

  - **Lack of Automatic Garbage Collection**
    Wasm does not yet have support for automatic garbage collection, which is a feature found in many high-level languages that automatically reclaims memory no longer in use. Although there is ongoing work to add garbage collection support to Wasm, the lack of it currently means that developers must manually manage memory, increasing the complexity of developing Wasm applications. This issue is particularly important in the context of IoT, where devices are often constrained in terms of memory resources. An IoT application with inefficient memory usage can easily consume too much memory, potentially causing the device to crash or behave unpredictably.
  - **Limited Memory Resources on IoT Devices**
    IoT devices range from powerful edge servers to tiny sensors with very limited processing power and memory. For devices at the lower end of this spectrum, the relatively large size of the Wasm runtime and the memory consumed by Wasm applications can be a significant problem. Developers must carefully optimize their applications to reduce memory usage, a task that is made more difficult by the current limitations of Wasm's memory management.
  - **Difficulty Debugging Memory Issues**
    Wasm's binary nature and the lack of a built-in debugger make debugging memory-related issues challenging. While developers can use tools such as source maps to map the Wasm code back to the original source code, these tools might not always provide enough information to diagnose a memory problem. Additionally, the linear memory model of Wasm is very low-level, making it difficult to understand how memory is being used and where potential problems might lie.

- **Limited Access to Web APIs**
  Wasm modules do not have direct access to web APIs and need to go through JavaScript, which can result in performance bottlenecks and increased complexity. Wasm's integration into the web ecosystem presents unique challenges, particularly in the context of access to web APIs. While Wasm has been designed to work alongside JavaScript, its ability to directly access web APIs has been a significant concern, especially for IoT applications.

  - **Restricted Direct Access to Web APIs**
    Wasm cannot directly access the web APIs that JavaScript can. Instead, in order to use these APIs Wasm needs to call out to JavaScript, which can be a cumbersome and inefficient process. This is due to the fact that Wasm runs in a sandboxed environment to ensure security, which inherently limits its ability to directly interact with the host environment. For IoT applications, this limitation can be particularly restrictive. Many IoT applications require real-time interaction with various system APIs for tasks such as device control, data collection, and communication with other devices or services. The current inability of Wasm to directly access these APIs adds an extra layer of complexity to building these applications.
  - **WASI and Interface Types**
    To address these limitations, two proposals are being developed in the Wasm community: the WASI and Interface Types. The WASI aims to provide a system interface for Wasm that abstracts away the details of the host system. This allows Wasm applications to interact with the system without compromising security. The WASI, however, is mainly designed for server-side or standalone applications, and does not provide access to most web-specific APIs. Interface Types, on the other hand, aim to define a way for Wasm modules to communicate with each other and with the host environment using high-level data types. This would allow Wasm to interact with web APIs in a more direct and efficient way

without needing to go through JavaScript. These two proposals, if accepted and implemented, would go a long way in addressing the current limitations of Wasm in terms of API access.

  - **Implications for IoT**
    The current state of Wasm's limited access to web APIs presents a challenge for IoT development. However, the advancements in the form of WASI and Interface Types bring a promising future. Access to web APIs in a more direct and efficient manner would enable developers to build more sophisticated, high-performing, and secure IoT applications using Wasm.

- **Debugging and Testing Challenges**
  In light of its binary nature, debugging and testing Wasm applications can be more challenging compared to JavaScript. While a number of tools are available, they often provide less visibility and convenience than developers are accustomed to. As Wasm finds its footing in the landscape of IoT development, the challenges associated with debugging and testing of Wasm modules are becoming increasingly evident. These challenges stem from a few aspects of Wasm's design and the current state of tooling.

  - **Lack of Mature Debugging Tools**
    While Wasm's design is optimized for efficient execution, its binary format makes debugging a complex task. Compared to traditional programming languages, debugging tools for Wasm are in their nascent stages. While there are debugging protocols available, such as Dwarf for Wasm, they may not be as mature or fully featured as those available for other languages.
  - **Source Map Limitations**
    Source maps are used to map the compiled Wasm code back to the original source code, making debugging easier. However, the effectiveness of source maps can be limited by the fact that they only provide information about where the code originated, not how the compiled code is behaving. This can make it harder to debug complex issues that involve the runtime environment or interactions between different parts of the code.
  - **Interoperability Issues**
    Considering the interoperability of Wasm with JavaScript and other web technologies, debugging can be complicated by issues that span multiple languages and technologies. For instance, an issue might involve both JavaScript and Wasm code, requiring developers to use different tools and strategies to debug different parts of the application.
  - **Testing Challenges**
    Similar to debugging, testing Wasm modules, especially in the context of IoT applications, presents its own set of challenges. Considering the wide variety of devices and environments in IoT, it can be difficult to effectively test Wasm applications in order to ensure they work correctly in all potential scenarios.

- **Restricted Networking Capabilities**
  In its current state, Wasm lacks direct access to networking APIs, limiting its capabilities for networking tasks without interaction with JavaScript, which can introduce complexity and performance drawbacks. As Wasm's use expands in the world of IoT, it encounters unique challenges, one of which is its restricted networking capabilities. This section highlights these challenges and discusses their impact on the use of Wasm in IoT applications.

  - **Wasm's Networking Model**
    Wasm currently operates within the constraints of the same-origin policy, meaning that it can only make network requests to the same source from which the Wasm module was loaded. While this policy is a crucial security feature in web environments, it can limit the direct networking capabilities of Wasm, particularly

in the context of IoT, where devices often need to communicate with different servers and devices.

- **Lack of Direct Access to Networking APIs**
  As of now, Wasm does not have direct access to networking APIs. This means that Wasm applications must rely on the host environment, typically a JavaScript environment in a web browser, to make network requests on their behalf. This lack of direct access can lead to inefficiencies and makes it more difficult to implement networking functionality in a straightforward manner.
- **Implications for IoT**
  These networking restrictions can be particularly problematic for IoT applications. IoT devices often need to communicate with various servers, devices, and services, often across different origins. The lack of direct access to networking APIs presents a challenge, as it means that all networking operations must go through the host environment, which could potentially impact performance and introduce additional complexity.
- **Potential Solutions**
  Proposed solutions to this challenge involve extending Wasm's capabilities either through new Wasm APIs or via the WASI. In particular, the latter aims to provide Wasm applications with more direct access to system capabilities, including networking. This evolution could significantly enhance the utility and power of Wasm in the context of IoT.

6.2. Challenges in IoT Integration with WebAssembly

- **IoT Device Garbage Collection**
  Garbage collection is a critical feature in many programming languages, helping to manage memory automatically by detecting and freeing up memory that is no longer needed and thereby reducing the risk of memory leaks. While JavaScript has built-in garbage collection, Wasm currently lacks this feature. However, there is a promising proposal on the horizon to introduce garbage collection to Wasm, which could have significant implications for its application in IoT contexts.
  The proposal for garbage collection in Wasm aims to offer the same level of memory management that high-level languages such as JavaScript, Python, and Ruby provide, with the goal of enabling these and other languages to target Wasm more efficiently. The integration of garbage collection into Wasm could vastly simplify the development process, improve performance, and make Wasm a more attractive and feasible choice for a wider range of applications, including IoT.
  By enabling managed languages to compile to Wasm efficiently, it would become much easier to port existing libraries, tools, and applications to run in the Wasm environment without the need for complex workarounds or compromises. This could lead to a significant increase in the ecosystem of available software and libraries, which would in turn boost the utility and versatility of Wasm.
  Moreover, the addition of garbage collection could reduce the size of Wasm modules by allowing them to leverage the host environment's garbage collector instead of having to include their own memory management code. This would make the modules more lightweight and faster to load, which is an important consideration in the bandwidth-constrained environments typical of IoT.
  However, introducing garbage collection into Wasm is a complex task, and there are several challenges that need to be addressed. For example, there needs to be a balance between the increased overhead of garbage collection and its benefits. In addition, it is important to ensure that the garbage collector is efficient and does not introduce significant performance overhead.
  Overall, the proposal of garbage collection for Wasm represents an exciting direction for the future. The successful implementation of this feature could unlock a new level of functionality and efficiency, paving the way for a broader application of Wasm

in IoT and other domains. As such, it will be crucial to monitor the progress of this proposal and its potential impact on the Wasm ecosystem.

- **Enhancing the WebAssembly Ecosystem**
  Efforts are ongoing to further develop and refine the tooling and libraries around Wasm with the aim of addressing current challenges and making the technology more approachable and easier to use.
  A robust and well-supported ecosystem is vital for the success and wider adoption of any technology. As of now, the Wasm ecosystem is burgeoning, propelled by its promising capabilities and the support of industry stalwarts such as Google, Mozilla, Microsoft, and Apple. However, there is much to be done in order to fully realize its potential, especially in the IoT domain. Here, we discuss key future directions for enhancing the Wasm ecosystem.
  - **Improved Tooling**
    Improving the toolchain is crucial for the broader adoption of Wasm. This includes refining existing tools and developing new ones that cater specifically to the needs of Wasm developers. For example, creating user-friendly debuggers, profilers, and performance analyzers designed for Wasm can significantly ease the development process and foster its adoption.
  - **Broadening Language Support**
    While Wasm currently supports a variety of languages, broadening this support is essential. Efforts should be aimed at making the compilation to Wasm seamless and efficient for a wider array of languages. This would enable developers of different backgrounds and expertise to leverage the benefits of Wasm, promoting its use in diverse applications, including IoT.
  - **Encouraging Community Engagement**
    One of the most effective ways to enhance the Wasm ecosystem is to foster a strong and active community of developers and users. This not only fuels the evolution of the technology through a constant influx of ideas and improvements, it helps to uncover new possibilities for its application. In the context of IoT, this could mean finding innovative ways to use Wasm in edge computing, wearable technology, smart homes, and more.
  - **Establishing Standards and Best Practices**
    As Wasm matures, it is crucial to establish clear standards and best practices for its use. This includes defining guidelines for security, performance optimization, interoperability, and more. Establishing these standards can streamline the development process and ensure that Wasm applications are robust, secure, and efficient.
  - **Developing IoT-Specific Libraries and Frameworks**
    In light of the unique constraints and requirements of IoT applications, developing libraries and frameworks specifically designed for IoT could significantly accelerate the adoption of Wasm in this domain. These could provide out-of-the-box solutions for common IoT tasks such as networking, data processing, and device management, easing the development process and improving efficiency.
- **Enhanced Security Features**
  Security is a priority for future Wasm updates, with expectations for enhanced memory protection, more granular control over system resources, and improvements to the existing sandboxing mechanisms to further secure Wasm execution. In an era where cyberthreats are increasing in both number and sophistication, security is a paramount concern for any computing system, and the IoT landscape is no exception. In fact, the distributed nature of IoT networks and the sensitive data they often handle makes security even more critical. In this section, we discuss potential future directions for enhancing the security features of Wasm in IoT applications.

  - **WebAssembly and Secure Execution Environments**
    In the future, there could be an increased focus on integrating Wasm with secure execution environments (SEEs) such as Intel SGX, ARM TrustZone, or even custom-built SEEs. This could enable sensitive computations to be isolated at a hardware level, providing a higher degree of security than software-level isolation mechanisms alone. In this scenario, Wasm could serve as a portable and efficient execution format for code running inside these secure environments.
  - **Formal Verification of Wasm Code**
    Another direction could be the application of formal methods to verify the security properties of Wasm code. Formal methods involve mathematically proving that a piece of software satisfies certain properties, and are typically used in high-assurance systems where security is critical. By developing tools and methodologies for formally verifying Wasm code, it could be possible to provide strong mathematically-backed guarantees about the security of Wasm applications.
  - **Fine-Grained Sandboxing**
    While Wasm already operates in a sandboxed environment, future developments could see a move towards more fine-grained sandboxing mechanisms. This could involve isolating different components of a Wasm application from each other, thereby limiting the potential impact of a security vulnerability in one component.
  - **Enhanced Cryptographic Capabilities**
    Wasm could be extended with more advanced cryptographic capabilities, such as support for secure multi-party computation (SMPC), homomorphic encryption, or quantum-resistant cryptography. These enhancements could allow Wasm to securely handle sensitive data in a wider range of scenarios, and could be particularly beneficial in IoT networks, where sensitive data is often transmitted between devices.
  - **Improved Auditability and Transparency**
    With the potential integration of blockchain technology into IoT, the aspect of auditability and transparency of code execution becomes crucial. Wasm, with its human-readable text format (WAT), can play a significant role here. Enhancements can be made to the Wasm ecosystem to facilitate the verification and auditing of code.

- **Improved Networking Capabilities**
  Efforts are underway to provide Wasm with direct access to networking APIs in order to eliminate the need for JavaScript intermediation and unlock the potential for more powerful and efficient networked applications. Networking is a crucial aspect of any IoT application. Devices in an IoT network communicate with both each other and with cloud-based services, necessitating reliable, efficient, and secure networking capabilities. However, Wasm's initial design does not have inherent networking capabilities, instead relying on the host environment for this. In the future, there are several directions Wasm could take to improve networking capabilities in the context of IoT.

  - **Networking APIs in WebAssembly**
    One of the most promising developments in Wasm is the proposal for adding networking APIs directly to the WASI. This would allow Wasm applications to establish network connections, send and receive data, and perform other networking operations without relying on JavaScript or other host environment features. This could significantly improve the efficiency of networking in Wasm-based IoT applications.
  - **Peer-to-Peer Networking**
    Peer-to-peer networking is an important feature for many IoT applications, especially those that involve direct device-to-device communication. Future versions of Wasm could include support for peer-to-peer networking protocols, allow-

ing devices to establish direct connections with each other without needing a central server.

- **Secure Networking**
  Security is paramount in IoT networking due to the sensitive nature of the data being transferred. Future developments could focus on enhancing the security of Wasm's networking capabilities, for instance by integrating secure networking protocols directly into Wasm or providing APIs for cryptographic operations.
- **Network Performance Optimization**
  Efficient use of network resources is important in IoT, especially in applications where bandwidth is limited or expensive. Future iterations of Wasm could include features for optimizing network performance, such as support for data compression or efficient binary protocols.
- **Advanced Networking Features**
  There are numerous other advanced networking features that could be beneficial in an IoT context. For example, support for multicast or broadcast communication could allow a single Wasm application to efficiently send data to multiple devices. Similarly, support for real-time communication protocols could enable use cases such as VoIP or live video streaming.

6.3. Role of WebAssembly in Advancing IoT

- **Performance Optimization in IoT Devices**
  Wasm plays a crucial role in advancing the performance optimization of IoT devices thanks to its compact binary format, efficient execution, and ability to work seamlessly with other web technologies. Here, we delve deeper into the elements that make Wasm a key player in performance optimization for IoT.

  - **Efficient Binary Format**
    Wasm's binary format is both compact and designed for fast parsing, two factors which greatly benefit performance. On resource-constrained IoT devices, these qualities make it possible to execute complex tasks with less memory and processing power than traditional interpreted languages. Furthermore, its compact size allows for faster download times, a critical advantage for devices operating on networks with limited bandwidth.
  - **Near-Native Execution Speed**
    Wasm code is compiled to a form that is very close to machine code, enabling it to execute at near-native speed. This speed advantage is particularly beneficial in the IoT space, where devices must often process data and make decisions in real time.
  - **Interoperability with JavaScript and Web APIs**
    Wasm is designed to interoperate seamlessly with JavaScript and existing web APIs, enabling it to take advantage of the broad ecosystem of web technologies. This interoperability can lead to performance improvements, as developers can use each technology for what it does best, such as using Wasm for compute-intensive tasks and JavaScript for interaction with web APIs.
  - **Thread and Memory Management**
    Wasm's linear memory model and its proposal for threading capabilities play a significant role in performance optimization. For devices handling complex operations or managing multiple processes simultaneously, Wasm's efficient memory management and potential multithreading support can boost performance significantly.
  - **Edge Computing Applications**
    Wasm's performance characteristics make it particularly well-suited to edge computing applications, which aim to reduce latency by performing data processing at the edge of the network closer to the source of data. Wasm's high execution speed and low resource requirements enable it to run complex algorithms on edge devices, which are often resource-constrained.

- **Strengthening Security in IoT**
  Wasm's sandboxing and planned security enhancements can significantly bolster IoT security. As a result, it could serve as a powerful tool to mitigate common security risks such as buffer overflow and injection attacks. The role of Wasm in enhancing the security of IoT ecosystems is of immense value. Due to their pervasive nature and the sensitivity of the data they often handle, IoT devices are attractive targets for attackers. Wasm, with its security-focused design, offers several key elements that can help to strengthen the security of IoT deployments.
  - **Strong Isolation**
    Each Wasm module operates within a well-defined sandbox environment. This means that code executing in one Wasm module is isolated from both the host system and other Wasm modules, preventing any malicious activity from affecting other parts of the system. This isolation makes it difficult for attackers to exploit vulnerabilities in one part of the system to compromise others.
  - **Managed Memory**
    Wasm's memory model offers security advantages as well. Wasm programs manipulate a linear memory array, which is isolated from the host memory and checked for out-of-bounds accesses. This model helps to prevent common vulnerabilities such as buffer overflows that attackers could otherwise exploit.
  - **Secure Interactions**
    Interactions between Wasm and the host environment (typically JavaScript in a browser context) are strictly controlled, with a clear interface for calling functions and transferring data. This clear interface makes it harder for an attacker to trick the program into executing malicious code or accessing sensitive data.
  - **Futuristic Scope**
    Proposed enhancements to Wasm, such as the addition of garbage collection and the WASI, could offer additional security benefits. Garbage collection can help prevent memory leaks, which can sometimes be exploited to carry out attacks, while WASI could provide a more secure way for Wasm modules to interact with system resources.
- **Enabling Advanced Features in IoT**
  Wasm's efficient execution can enable more advanced features in IoT devices, including edge computing, real-time analytics, and machine learning. As Wasm continues to evolve and improve, it can be expected that these capabilities will be further enhanced. Wasm's potential extends beyond the immediate benefits of performance and security. Its flexibility and portability open doors to new possibilities for the advancement of IoT applications. This section explores how Wasm can enable more advanced features in the IoT realm.
  - **Edge AI and Machine Learning**
    The high performance of Wasm enables the deployment of Artificial Intelligence (AI) and Machine Learning (ML) models directly onto edge devices. This capability allows IoT devices to make intelligent decisions in real time without relying on a cloud-based service, thereby reducing latency and bandwidth usage. Additionally, running AI/ML models on-device can offer privacy benefits, as sensitive data no longer need to be transmitted over the network.
  - **Digital Twins**
    Wasm's cross-platform compatibility can play a significant role in the implementation of digital twins, a concept that involves creating a digital replica of a physical object or system. Wasm allows the same digital twin code to run on various platforms, from small IoT devices to cloud servers, enabling real-time synchronization between the physical entity and its digital replica.

  - **Advanced Augmented Reality (AR)**
    Wasm can power more advanced AR applications in wearable IoT devices such as smart glasses. Its efficiency and performance can enable complex AR tasks such as object recognition or positional tracking to be performed directly on the device, offering a more responsive and immersive user experience.
  - **Ubiquitous Computing**
    Wasm's portability and efficient performance make it a promising technology for ubiquitous computing, a concept in which computing is embedded in everyday objects and activities. With Wasm, developers can write code once and run it on a multitude of devices, facilitating the integration of computing into peoples' daily lives.

- **Standardization of IoT Development**
  The language-neutral and platform-agnostic nature of Wasm could contribute to standardization in IoT development, reducing fragmentation and simplifying the development process across various platforms and devices. The IoT landscape is inherently diverse and fragmented, with a multitude of different devices, platforms, and technologies. This fragmentation can often be a barrier to the widespread adoption and development of IoT applications. Here, Wasm's platform-agnostic nature and standardization can provide a unifying layer to facilitate the standardization of IoT development.

  - **Unifying the Development Process**
    Wasm offers a single uniform target for all IoT developers. Its universal bytecode can be executed on any device that has a Wasm runtime, regardless of the underlying hardware or operating system. This universality simplifies the development process, as developers no longer need to build different versions of the same application for different platforms; instead, they can focus on building one version that can run across all platforms.
  - **Facilitating Code Reuse**
    The ability to write code once and run it anywhere encourages code reuse. In traditional IoT development, code often needs to be rewritten when moving from one platform to another due to differences in APIs, operating systems, or hardware capabilities. Wasm mitigates this issue, as the same Wasm code can be used across different devices and platforms.
  - **Ensuring Predictable Performance**
    Wasm's low-level nature ensures that it offers predictable performance across platforms. Its compact binary format is designed for fast download, parsing, and execution. This predictability allows developers to create applications with consistent performance regardless of the platform on which they are running.
  - **Promoting Interoperability**
    Wasm promotes interoperability among IoT devices. It can serve as a universal runtime for executing IoT applications, allowing devices from different manufacturers and platforms to run the same application code and communicate more seamlessly with each other.

- **Resource Management in IoT**
  Wasm's efficient memory and resource utilization could play a critical role in managing the constrained resources of IoT devices, thereby enhancing their performance and capabilities. In IoT systems, resource management is a critical consideration. IoT devices, ranging from small sensors to more complex machines, usually operate under tight resource constraints, including limited processing power, memory, storage, and energy. Wasm, with its compact, low-level format and efficient execution model, offers several advantages for resource management in IoT.

- **Efficient Use of Processing Power**
  Wasm's compact binary format and efficient execution model ensure that it uses processing power optimally. Unlike interpreted languages, which require parsing and interpreting at runtime, Wasm's bytecode is pre-compiled and can be executed quickly by the Wasm virtual machine. This reduces the processing overhead, making it ideal for resource-constrained IoT devices.
- **Compact Memory Footprint**
  Wasm has a compact memory footprint. It uses a linear memory model, which provides a single, contiguous array of bytes representing the application's memory. In combination with Wasm's efficient use of memory, this model reduces the amount of memory needed to run applications, an important advantage for memory-limited IoT devices.
- **Energy Efficiency**
  Efficient use of processing power and memory translates to energy efficiency, a critical consideration for IoT devices, many of which are battery-powered. By minimizing the computational resources needed to run applications, Wasm helps to reduce the energy consumption of IoT devices, extending their battery life and reducing the overall energy footprint of the IoT system.
- **Granular Control over Resources**
  Wasm's low-level nature provides developers with granular control over system resources. For example, developers can control memory allocation and deallocation explicitly in Wasm, which is often not possible in high-level languages. This control allows for more efficient use of resources and can lead to better performance and reduced resource consumption.
- **Future Developments**
  Proposed enhancements to Wasm, such as the introduction of garbage collection and improved resource management primitives, are expected to further enhance Wasm's resource management capabilities for IoT. These developments can help to address the current limitations and make Wasm an even more attractive option for resource-constrained IoT devices.

7. Prospective Aspects of WebAssembly

The prospective aspects of Wasm hold significant promise, as numerous improvements and new specifications are on the horizon. These specifications aim to expand the functionality and applicability of Wasm across different domains. For instance, the proposal to add garbage collection to Wasm would revolutionize how memory management is carried out, making it easier to interface with host languages and enhancing performance in memory-intensive applications. Another critical aspect is the WASI, which seeks to standardize how Wasm modules interact with the operating system, thereby opening doors for universal applications that can run anywhere regardless of the system architecture. In addition, enhanced debugging support is in the pipeline, which will make development and troubleshooting processes more efficient and accessible. These advancements signify the commitment of the Wasm community towards continuous innovation, paving the way for a bright and transformative future for Wasm.

7.1. Prospective WebAssembly Specifications

Wasm, as a crucial aspect of modern web technology, continues to evolve and expand in scope and functionality. As it has been adopted in diverse fields such as IoT, the development community and browser vendors are keen to continue rolling out new features and specifications that bolster the language's power and flexibility. This section explores several of the potential future specifications and enhancements to Wasm that could have profound implications for its usage in IoT and beyond.

- **WASI**
  The WASI is another promising future specification. It seeks to standardize the way in which Wasm modules interact with the underlying operating system to facilitate consistent and secure system calls. With WASI, developers can create universal applications that are platform-independent, enhancing the portability of Wasm. For IoT, this could mean that a single Wasm module could run on a myriad of devices, irrespective of the device's specific operating system or architecture.
- **Multi-Threading and SIMD Support**
  Future specifications of Wasm include the introduction of multi-threading and SIMD support. Multi-threading would allow Wasm applications to execute multiple threads simultaneously, significantly increasing computational speed and efficiency for multi-core processors. SIMD, on the other hand, would enable a single operation to be performed on multiple data points at once, which is crucial for heavy computational tasks such as graphics rendering, audio processing, or machine learning algorithms.
- **Enhanced Debugging Support**
  Another area of focus for future Wasm specifications is enhanced debugging support. At present, debugging Wasm code can be challenging, especially when compared to more traditional languages such as JavaScript. Future enhancements promise a more accessible and efficient debugging process, enabling developers to easily track and fix bugs within their Wasm code. This would make the development process more streamlined while resulting in more stable and reliable Wasm applications.
- **Module Types** The Module Types proposal is a future specification that aims to provide a robust mechanism for defining the interface of a Wasm module. This includes the functions it exports, the types of these functions, the memories it uses, and more. With this feature, developers could build more complex modular applications with clearly defined interactions between different modules.

7.2. Need for Garbage Collection and Resource Management

One of the significant future directions for Wasm involves integrating garbage collection and advancing its resource management capabilities. The absence of automatic garbage collection is a known limitation in the current Wasm specification, and addressing it will mark a significant milestone.

- **The Impact of Garbage Collection**
  Garbage collection has a profound impact on software development. The automatic memory management provided by garbage collection removes the burden of manually tracking and freeing up unused memory from developers, prevents memory leaks, and reduces the likelihood of certain types of bugs.
  In the context of Wasm, the addition of garbage collection support could drastically simplify the process of compiling high-level garbage-collected languages to Wasm. Languages such as JavaScript, Python, and many others use garbage collection, and their direct compilation to Wasm would be more straightforward with the introduction of garbage collection support.
- **Resource Management Advancements**
  In addition to garbage collection, future improvements to Wasm's resource management capabilities are expected to evolve. Currently, Wasm operates with a linear memory model, which offers simplicity and performance advantages but presents limitations. As IoT devices continue to grow in complexity and capability, more sophisticated memory management techniques may be needed.
  Future developments could see the emergence of more advanced memory models in Wasm, including support for shared memory between modules or even between threads. This would allow for more complex multi-threaded applications and for modules that can communicate and share data more efficiently.

- **Garbage Collection and IoT**
  In the specific context of IoT, the addition of garbage collection to Wasm could provide significant advantages. IoT devices often need to run for extended periods without manual intervention, and can ill afford memory leaks or other resource management issues that can disrupt their operation. Thus, the inclusion of automatic garbage collection would enhance the reliability and robustness of Wasm-based IoT applications.

7.3. Exception Handling

This section discusses the future deployment aspects of exception handling in Wasm. The proposal for exception handling reflects the current version agreed upon by the Wasm Community Group; however, it is important to note that the specifics may evolve as the proposal progresses. The inclusion of exception handling in Wasm opens up new possibilities for handling errors and exceptional situations in future deployments. With exception handling, developers can break the control flow when an exception is thrown, allowing for more robust error handling and recovery mechanisms. In future deployments, exception handling can enhance the reliability and stability of Wasm applications. It enables developers to handle various types of exceptions, both known and unknown, ensuring that critical errors are properly handled and the application can gracefully recover from exceptional situations. The introduction of a new section for declaring exceptions provides a standardized way to define and manage exceptions in Wasm modules. This allows developers to precisely specify the types of exceptions that can be thrown and caught within their applications. One of the future directions for exception handling in Wasm is the potential expansion of the tag section. Currently, the attribute value for a tag can only specify that it is for an exception. However, the proposal allows for future extension by defining a more general format tag that can accommodate other types of typed tags. This opens up possibilities for handling different types of exceptional situations beyond traditional exceptions. As exception handling evolves, it is expected that the Wasm ecosystem will provide tools, libraries, and frameworks to facilitate the development and deployment of exception-aware applications. These tools will help developers write robust code that effectively handles exceptions and ensures the overall stability of Wasm applications. The future deployment of exception handling in Wasm holds promise for improving the reliability and error handling capabilities of applications. By providing a standardized mechanism for handling exceptions, Wasm enables developers to build more resilient and fault-tolerant applications that can recover from exceptional situations. As the Wasm ecosystem matures, further advancements and tooling support are expected to enhance the exception handling capabilities of Wasm applications.

7.4. Memory64

Wasm utilizes a page-based measurement system for linear memory objects, where each page consists of 65536 (2^{16}) bytes. In the current version of Wasm (version 1), the maximum number of pages allowed is 65536, resulting in a total memory size of 2^{32} bytes (4 gibibytes). This limitation extends to memory instructions, which currently employ the i32 type as a memory index. Consequently, these instructions can address a maximum of 2^{32} bytes of memory. For most applications, the 4 gibibyte memory limit proves to be sufficient. In such cases, utilizing 32-bit memory indexes offers the advantage of smaller pointers in the producer language, leading to memory savings. However, certain applications may require more memory than the current limit permits. Regrettably, the existing Wasm feature set lacks straightforward solutions to address this challenge. However, introducing the capability for a Wasm module to select between 32-bit and 64-bit memory indexes effectively resolves both concerns. Furthermore, considering that Wasm serves as a Virtual Instruction Set Architecture (ISA), hosts may prefer to employ the Wasm binary format as a portable executable format while supporting other non-virtual ISAs. With the widespread adoption of 64-bit memory addresses in most ISAs, hosts may be reluctant to maintain support for 32-bit memory addresses in their Application Binary Interface (ABI).

7.5. Multiple Memories

Developers are aiming to extend the capabilities of Wasm in the future by allowing the use of multiple memories within a single module. While it is currently possible to create multiple memories in a Wasm application, each memory is isolated within its own module, and a module or function cannot access multiple memories simultaneously. This limitation prevents efficient data transfer between memories, as it requires individual function calls into different modules for each value. The motivation behind supporting multiple memories in a single module is driven by several use case scenarios. First, for security purposes, a module may want to separate public memory which is shared with the outside from private memory that is encapsulated within the module. By isolating these memories, data exchange can be controlled and managed effectively. Second, even within a single module it is beneficial to have separate memories for shared memory used by multiple threads and memory used in a single-threaded manner. This isolation ensures better memory management and avoids potential conflicts or data corruption. Another use case is the need for persistence. An application may want to preserve certain memory states between runs by storing them in a file. However, not all memory needs to be persistent, and separating memories allows for greater flexibility in managing memory lifetimes. This proposal addresses the issue of linking multiple Wasm modules into one. Many tools exist that can merge multiple modules through static linking. However, if the modules define more than one memory, this merging process becomes challenging. Allowing multiple memories in a single module resolves this limitation and enables seamless merging of modules. Additionally, the ability to use multiple memories is crucial for scaling purposes. With the current 32-bit memory address space limitation, applications requiring more than 4 GB of memory face efficiency challenges. Multiple memories provide an efficient workaround until 64-bit memories become available. Furthermore, the proposal supports polyfilling, which emulates certain features such as garbage collection or interface types in existing Wasm. The ability to add auxiliary memories distinct from the module's address space facilitates the implementation of these features within the current Wasm infrastructure. The design of this extension aligns with the original Wasm design and introduces minimal changes. It allows for multiple memory imports and definitions in a single module, adds memory index parameters to memory-related instructions, and extends validation and execution semantics accordingly. The binary and text formats are updated to accommodate these changes. Implementations of Wasm engines already handle multiple memories, although code within a module has been limited to accessing a single memory. Reserving a register for the base address of the main memory has been a common technique. With multiple memories, engines may require an extra level of indirection to access the desired memory. In summary, this proposal enhances the capabilities of Wasm by enabling the use of multiple memories within a single module. It addresses various use case scenarios, improves memory management, facilitates data exchange, and aligns with the existing design of Wasm. By allowing the explicit definition and access of multiple memories, Wasm becomes more flexible and versatile in its memory handling capabilities.

7.6. Threads and Atomics

This section focuses on the introduction of threading and atomics features in Wasm. The proposal suggests the addition of shared linear memory type and new operations for atomic memory access. The responsibility for creating and joining threads is left to the embedder. Agents, which are execution contexts for Wasm modules, are introduced, and can be seen as threads within the web embedding context. Agents belong to an agent cluster, which in the case of web embedding is the ECMAScript agent cluster. Shared linear memory allows memory to be shared among all agents in an agent cluster. While it can be imported or defined within a module, it must be specified in the module's memory import if it is shared. Modification of shared memory by one agent can be observed by other agents in the same cluster. The resizing of shared linear memory requires the specification of a maximum memory size. The future design of Wasm will

address instantiation, initialization, and memory access operations. When a module with imported memory is instantiated, its data segments are copied into the memory. The initialization of data segments follows a specific order and granularity. The proposal allows for memory to be initialized only once by placing all data segments in a separate module that is instantiated once and then shared with other modules. The proposal introduces atomic memory accesses, which can be performed on both shared and unshared linear memories. These accesses include load/store operations, read–modify–write operators, and compare–exchange operators. Atomic memory accesses have sequentially consistent ordering, and misaligned accesses in atomic operations result in traps. Additionally, the proposal introduces wait and notify operators, which are optimizations for value change detection. The wait operator waits for a value to change, and the notify operator wakes up waiters. Alignment requirements and validation rules are defined for these operators. The fence operator is included to preserve synchronization guarantees in higher-level languages. Overall, this proposal aims to enhance Wasm by introducing threading capabilities and providing atomic memory access operations, allowing for more efficient and synchronized execution in multi-threaded scenarios.

7.7. Type Reflection

Type reflection in Wasm is the focus of this proposal, aiming to provide improved access to type information for Wasm modules and objects through the JavaScript API. Wasm is a typed language, and its types contain valuable details such as import and export specifications, memory and table size limits, and global mutability. The need to query this information from JavaScript has been raised in various contexts, including the development of a JS-hosted linker or an adapter mechanism for modules. Therefore, this proposal introduces new functionality to the JS API in a systematic manner. The future design will consist of three main components. First, it defines a representation of Wasm types as JavaScript objects. This representation allows for a direct and extensible mapping of Wasm types to JSON-style JS objects. Second, the API classes are extended with a type method that enables the retrieval of the underlying Wasm object's type. Finally, a new class called Wasm.Function is introduced, which subclasses JavaScript's Function and specifically represents exported functions in Wasm. This class provides a constructor to create Wasm exported functions from regular JS functions, enabling the usage of JS functions in tables, which was not previously possible. To achieve type representation, a simple grammar is used to define all Wasm types, which can then be mapped to JSON-style JS objects. The existing descriptor interfaces in the API are repurposed as types, with slight renaming and restructuring. Furthermore, the proposal introduces additional methods to query types, such as type() for Memory, Table, and Global interfaces. The constructors for Memory and Table are adjusted to accept the updated type parameters. Additionally, the Wasm.Function class is introduced, allowing exported functions to have a distinct type attribute and supporting explicit construction of Wasm exported functions. By implementing this proposal, developers gain enhanced access to type information in Wasm through the JavaScript API. They can retrieve and manipulate type details for imports, exports, memories, tables, globals, and exported functions. The proposal offers improved compatibility, flexibility, and extensibility for working with Wasm modules in JavaScript, enabling more sophisticated tools and mechanisms to interact with Wasm code.

7.8. Need for Enhanced Security Measures

Security is a paramount concern in the IoT landscape in light of the critical nature of many IoT applications and the potentially sensitive data they handle. While the secure-by-default nature of Wasm is one of its strongest advantages, further enhancements in this area expected as the technology matures and its use in IoT grows.

- **Expanding Sandbox Capabilities**
  One of the key security features of Wasm is its sandboxing mechanism, which isolates Wasm modules from each other and from the rest of the system. This containment

prevents a module from affecting other parts of the system in case it becomes compromised. Future developments could further enhance this sandboxing mechanism to provide even stronger isolation and control over module interactions, as well as to provide more granular permissions and capabilities to each module.

- **Advanced Cryptographic Features**
  Cryptography is a cornerstone of secure communications, and as Wasm is increasingly used for IoT applications we may see the introduction of advanced cryptographic features directly into the Wasm standard. This could include possibilities such as built-in support for encryption and decryption functions, digital signatures, secure random number generation, and other cryptographic primitives. These features would make it easier for developers to build secure IoT applications with Wasm while potentially improving performance by providing these capabilities at a low level.
- **Hardware Security Integration**
  Another promising direction for enhancing Wasm's security is deeper integration with hardware security features. Many modern processors offer hardware-level security features such as secure enclaves, which provide a protected area of memory where sensitive data can be processed securely. Through integration with these features, Wasm could offer an even higher level of security for IoT applications.
- **Fine-Grained Access Control**
  As Wasm evolves, we may see the introduction of more fine-grained access control mechanisms. These could allow for detailed control over what resources a Wasm module can access, such as specific devices, network interfaces, or files. This would provide an additional layer of security by ensuring that a module can only access the resources it needs to function and nothing more.
- **Security in a Multi-Tenant Environment**
  With the rise of edge computing and the proliferation of IoT devices, there may be situations in which multiple Wasm applications coexist on the same device. In such scenarios, ensuring isolation and security in a multi-tenant environment would be crucial. Future Wasm specifications could cater to such use cases, providing robust security mechanisms to ensure data privacy and system stability.

7.9. Prospective WebAssembly Ecosystem Growth

The growth of the Wasm ecosystem is an integral part of the language's future, particularly in the realm of IoT. As more developers adopt Wasm for their IoT solutions, there is an increasing need for comprehensive tools, libraries, and community resources to support this growth. This section explores the potential future development of the Wasm ecosystem.

- **Expansion of Development Tools**
  The future will likely see a proliferation of development tools tailored for Wasm and IoT. This could include more comprehensive debuggers, performance profilers, module optimizers, and improved IDE support for languages compiling to Wasm, with the goal of making the development process as seamless as possible from writing code to debugging and deployment.
- **Growth of Libraries and Frameworks**
  Libraries and frameworks play a crucial role in easing the development process by providing pre-written code for common tasks. As the Wasm ecosystem matures, the number of available libraries and frameworks can be expected to increase. These could range from low-level libraries for things such as network communication and hardware interaction to high-level frameworks that provide entire boilerplate IoT applications.
- **Community Expansion**
  The Wasm community's expansion, including developers, users, and researchers, is crucial for its growth. A larger community means more collaboration, which leads to more tools, more libraries, and more shared knowledge. This could lead to increased contributions to the Wasm specification itself, thereby driving the language's evolution.

- **Increased Industry Adoption**
  Increased industry adoption of Wasm is another key aspect of the ecosystem's growth. As more businesses realize the benefits of using Wasm for their IoT solutions, there are likely to be more industry-sponsored projects, job opportunities for Wasm developers, and overall investment in the ecosystem.
- **Education and Training**
  As the demand for Wasm in IoT applications grows, so will the need for education and training in this area. This could result in more online courses, workshops, and resources for learning Wasm and how to use it in the context of IoT.

7.10. Prospective Alignments with Technologies

As Wasm continues to evolve and mature, it is set to revolutionize various aspects of IoT applications and usher in a new era of innovation. The flexible, efficient, and secure characteristics of Wasm make it highly suitable for futuristic IoT use cases. This section delves deeper into the potential future applications of Wasm in the IoT landscape.

- **Dew Computing**
  Dew computing, an emerging concept in the realm of IoT, envisions a paradigm in which computing and data processing capabilities are distributed across the network's edge, closer to the IoT devices and sensors themselves. In the context of futuristic use cases in IoT, Wasm can play a crucial role in enabling dew computing architectures. By leveraging Wasm, IoT devices can execute lightweight and efficient code at the edge, reducing reliance on cloud infrastructure and minimizing latency. Wasm's ability to run on resource-constrained devices makes it an ideal technology for enabling intelligent edge analytics and decision-making. With Wasm, IoT devices can perform real-time data processing, advanced machine learning tasks, and local decision-making all while maintaining low power consumption and ensuring fast response times. This empowers IoT deployments with the ability to derive valuable insights, make autonomous decisions, and respond to critical events in a timely manner even in scenarios with limited connectivity or stringent privacy requirements. Wasm's versatility and compatibility make it a promising technology for realizing the potential of dew computing in futuristic IoT use cases.
- **WebAssembly and AI**
  AI has become a fundamental component in a variety of technology sectors, bringing automation and intelligence to tasks traditionally requiring human intervention. As we anticipate the future of Wasm, its role in enhancing and accelerating AI applications is an intriguing subject. This section delves into the potential intersections between Wasm and AI.
  - **Why Wasm for AI?**
    The use of Wasm in AI can be attributed to its core strengths of performance, portability, and security. These attributes can significantly influence the development, execution, and scalability of AI applications.
  - **Wasm and AI Models**
    Many AI models, especially those involving deep learning, require substantial computational power. Currently, these models are typically trained on powerful servers and executed on the edge, for instance, in a user's browser or on an IoT device. This execution often relies on JavaScript, which, while highly portable, does not deliver the performance of a compiled language. Wasm, with its near-native execution speed, offers a compelling alternative for executing AI models on the edge, potentially improving performance and responsiveness.
  - **Wasm and On-Device AI**
    Wasm can enhance on-device AI capabilities. In many scenarios, running AI models directly on end-user devices such as smartphones or IoT devices can provide benefits in terms of latency, privacy, and bandwidth. However, these devices often have diverse architectures and capabilities. Wasm's portability makes it an

excellent choice for running AI workloads in such a heterogeneous environment, allowing developers to write their code once and run it on various devices.

  - **Wasm and AI in the Browser**
    Wasm can accelerate AI in the browser context as well. It opens up possibilities for running complex AI models directly in the browser, allowing for real-time AI applications such as image recognition, natural language processing, and predictive modeling to run efficiently without the need for server-side computation.

- **Decentralized Computing**
  With the advent of blockchain technologies and decentralized computing, there is a growing need for efficient, secure, and platform-independent computation models. Wasm's platform-agnostic nature in combination with its sandboxed execution environment make it an ideal candidate for this use case. Wasm could potentially enable smart contracts execution or complex computations on decentralized platforms, opening up new possibilities for IoT applications in a decentralized context.
- **Heterogeneous Computing**
  In the future, IoT devices will be increasingly heterogeneous, involving a mix of CPUs, GPUs, and other specialized hardware. The need for a common runtime that can efficiently use these diverse resources will be more critical than ever. Wasm, with its portable and efficient nature, could provide a unified platform for executing code across this heterogeneous hardware, unlocking new possibilities for IoT applications.
- **Advanced Robotics**
  Wasm's high performance and real-time capabilities could make it suitable for advanced robotics applications. Whether controlling robotic movements with precision, processing sensor data in real time, or running complex navigation algorithms, Wasm could be instrumental in pushing the boundaries of what is possible in robotics.

8. Conclusions

The integration of Wasm and IoT offers significant advantages for IoT development. Wasm provides a portable and efficient runtime environment for executing code on resource-constrained IoT devices. By enabling developers to write code in high-level languages such as C/C++ and Rust, Wasm simplifies the development process and allows for greater code reuse across different IoT platforms. Wasm's small footprint and low overhead make it well suited for IoT applications, where memory and processing power are typically limited. It enables the deployment of complex applications and algorithms on edge devices, reducing the need for data transmission to the cloud and improving real-time responsiveness. Additionally, Wasm's sandboxed execution model enhances security by isolating code execution and preventing unauthorized access to sensitive resources. This is particularly crucial in IoT deployments, where security vulnerabilities can have severe consequences. The integration of Wasm and IoT opens up new possibilities for developing innovative IoT solutions, such as intelligent edge devices, real-time analytics, and machine learning at the edge. It empowers developers to leverage the rich ecosystem of Wasm libraries and frameworks, thereby accelerating the development of IoT applications and fostering interoperability. Existing toolsets can be a good starting point to implement and design IoT-based architectures and applications; however, thorough research and further investigations are required in order to efficiently deal with the compiler design process. As there exist many types of IoT devices and toolchains, at present, Wasm toolsets are not enough to cover all the long list of IoT devices and related peripherals. A more holistic Wasm compiler design with augmentation of VM-based approach should be put in place in near future in order to realize the real potential of the Wasm–IoT juxtaposition. Overall, the integration of Wasm and IoT presents a powerful combination that enhances the capabilities, performance, and security of IoT systems, driving the advancement of the Internet of Things and enabling the next generation of connected devices and applications.

Funding: This research received no external funding.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The author declares no conflict of interest.

References

1. Haas, A.; Rossberg, A.; Schuff, D.L.; Titzer, B.L.; Holman, M.; Gohman, D.; Wagner, L.; Zakai, A.; Bastien, J.F. Bringing the web up to speed with WebAssembly. In Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, Barcelona, Spain, 18–23 June 2017; pp. 185–200.
2. Lehmann, D.; Kinder, J.; Pradel, M. Everything old is new again: Binary security of WebAssembly. In Proceedings of the 29th USENIX Security Symposium (USENIX Security 20), Boston, MA, USA, 12–14 August 2020; pp. 217–234.
3. Yan, Y.; Tu, T.; Zhao, L.; Zhou, Y.; Wang, W. Understanding the performance of webassembly applications. In Proceedings of the 21st ACM Internet Measurement Conference, Virtual Event, 2–4 November 2021; pp. 533–549.
4. Watt, C. Mechanising and verifying the WebAssembly specification. In Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, Los Angeles, CA, USA, 8–9 January 2018; pp. 53–65.
5. Ray, P.P. A survey on Internet of Things architectures. *J. King Saud Univ.-Comput. Inf. Sci.* **2018**, *30*, 291–319.
6. Ray, P.P. A survey of IoT cloud platforms. *Future Comput. Inform. J.* **2016**, *1*, 35–46. [CrossRef]
7. Rossberg, A.; Titzer, B.L.; Haas, A.; Schuff, D.L.; Gohman, D.; Wagner, L.; Zakai, A.; Bastien, J.F.; Holman, M. Bringing the web up to speed with webassembly. *Commun. ACM* **2018**, *61*, 107–115. [CrossRef]
8. Musch, M.; Wressnegger, C.; Johns, M.; Rieck, K. New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, 19–20 June 2019*; Springer International Publishing: Cham, Switzerland, 2019; pp. 23–42.
9. Wang, W. Empowering web applications with WebAssembly: Are we there yet? In Proceedings of the 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), Melbourne, Australia, 15–19 November 2021; pp. 1301–1305.
10. Dejaeghere, J.; Gbadamosi, B.; Pulls, T.; Rochet, F. Comparing Security in eBPF and WebAssembly. In Proceedings of the ACM SIGCOMM 1st Workshop on eBPF and Kernel Extensions, New York City, NY, USA, 10 September 2023.
11. WebAssembly Website. Available online: https://webassembly.org/ (accessed on 1 July 2023).
12. WebAssembly Mozilla. Available online: https://developer.mozilla.org/en-US/docs/WebAssembly/ (accessed on 2 July 2023).
13. Why WebAssembly Is Needed. Available online: https://thenewstack.io/webassembly/what-is-webassembly-and-why-do-you-need-it/ (accessed on 1 July 2023).
14. Foreign Function Interface. Available online: https://hyperledger.github.io/iroha-2-docs/api/ffi.html/ (accessed on 1 July 2023).
15. FFI Github. Available online: https://github.com/DeMille/wasm-ffi/ (accessed on 2 July 2023).
16. How WebAssembly Modules. Available online: https://training.linuxfoundation.org/blog/how-and-why-to-link-webassembly-modules/ (accessed on 1 July 2023).
17. WebAssembly Security. Available online: https://webassembly.org/docs/security/ (accessed on 1 July 2023).
18. WebAssembly Sandboxing. Available online: https://thenewstack.io/how-webassembly-offers-secure-development-through-sandboxing/ (accessed on 1 July 2023).
19. Emscripten. Available online: https://emscripten.org/ (accessed on 1 July 2023).
20. TinyGo. Available online: https://tinygo.org/ (accessed on 1 July 2023).
21. WARDuino. Available online: https://github.com/TOPLLab/WARDuino (accessed on 1 July 2023).
22. Wasm3. Available online: https://github.com/wasm3/wasm3 (accessed on 1 July 2023).
23. AssemblyScript. Available online: https://www.assemblyscript.org/ (accessed on 1 July 2023).
24. Wasmino-Core. Available online: https://github.com/wasmino/wasmino-core (accessed on 1 July 2023).
25. Binaryen. Available online: https://github.com/WebAssembly/binaryen (accessed on 1 July 2023).
26. Rustc. Available online: https://doc.rust-lang.org/rustc/what-is-rustc.html (accessed on 1 July 2023).
27. Zigwasm. Available online: https://www.fermyon.com/wasm-languages/zig (accessed on 2 July 2023).
28. Fable-Compiler. Available online: https://github.com/fable-compiler/Fable (accessed on 2 July 2023).
29. Pyodide. Available online: https://pyodide.org/ (accessed on 2 July 2023).
30. Wasmer. Available online: https://wasmer.io/ (accessed on 2 July 2023).
31. WAMR. Available online: https://github.com/bytecodealliance/wasm-micro-runtime (accessed on 1 July 2023).
32. Node.js. Available online: https://nodejs.org/ (accessed on 1 July 2023).
33. Wasmtime. Available online: https://wasmtime.dev/ (accessed on 1 July 2023).
34. WAVM. Available online: https://github.com/WAVM/WAVM (accessed on 1 July 2023).
35. Deno. Available online: https://deno.land/ (accessed on 1 July 2023).
36. Lucet. Available online: https://github.com/bytecodealliance/lucet (accessed on 1 July 2023).
37. Wascc. Available online: https://github.com/wasmCloud/wascc-actor (accessed on 1 July 2023).
38. Kotlin Wasm. Available online: https://kotlinlang.org/docs/wasm-overview.html (accessed on 2 July 2023).
39. WasmEdge. Available online: https://github.com/WasmEdge/WasmEdge (accessed on 2 July 2023).
40. CheerpX. Available online: https://leaningtech.com/cheerpx-for-flash/ (accessed on 2 July 2023).

41. Go. Available online: https://golangbot.com/webassembly-using-go/ (accessed on 2 July 2023).
42. Webpack. Available online: https://webpack.js.org/configuration/experiments/ (accessed on 2 July 2023).
43. Rollup. Available online: https://www.npmjs.com/package/@rollup/plugin-wasm (accessed on 2 July 2023).
44. Blazor. Available online: https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor (accessed on 2 July 2023).
45. wasm-bindgen. Available online: https://rustwasm.github.io/wasm-bindgen/#:~:text=The%20wasm%2Dbindgen%20tool%20and,can%20find%20that%20documentation%20here (accessed on 2 July 2023).
46. WABT. Available online: https://github.com/WebAssembly/wabt (accessed on 2 July 2023).
47. WASI. Available online: https://wasi.dev/ (accessed on 2 July 2023).
48. WASI Integration. Available online: https://github.com/WebAssembly/WASI (accessed on 2 July 2023).
49. WAGI. Available online: https://github.com/deislabs/wagi (accessed on 2 July 2023).
50. WASI vs WAGI. Available online: https://medium.com/@shyamsundarb/wasm-wasi-wagi-web-assembly-modules-in-rust-af7335e80160 (accessed on 2 July 2023).
51. Wallentowitz, S.; Kersting, B.; Dumitriu, D.M. Potential of WebAssembly for Embedded Systems. In Proceedings of the 2022 11th Mediterranean Conference on Embedded Computing (MECO), Budva, Montenegro, 7–10 June 2022; pp. 1–4.
52. Kotilainen, P.; Järvinen, V.; Tarkkanen, J.; Autto, T.; Das, T.; Waseem, M.; Mikkonen, T. WebAssembly in IoT: Beyond Toy Examples. In *International Conference on Web Engineering*; Springer Nature: Cham, Switzerland, 2023; pp. 93–100.
53. Hoque, M.N.; Harras, K.A. WebAssembly for Edge Computing: Potential and Challenges. *IEEE Commun. Stand. Mag.* **2022**, *6*, 68–73. [CrossRef]
54. Jain, S.M. *WebAssembly Introduction. WebAssembly for Cloud: A Basic Guide for Wasm-Based Cloud Apps*; Apress: Berkeley, CA, USA, 2022; pp. 1–11.
55. Zhamashev, Y. WebAssembly in Building Internet of Things Systems. Systematic Literature Review of Use Cases, Characteristics, Opportunities, and Threats. Available online: https://aaltodoc.aalto.fi/handle/123456789/121767 (accessed on 2 July 2023).
56. Theel, T. *Creative DIY Microcontroller Projects with TinyGo and WebAssembly: A Practical Guide to Building Embedded Applications for Low-powered Devices, IoT, and Home Automation*; Packt Publishing Limited: Birmingham, UK, 2021; ISBN 9781800560208.
57. Wen, E.; Weber, G. Wasmachine: Bring IOT up to speed with a webassembly OS. In Proceedings of the 2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops), Austin, TX, USA, 23–27 March 2020; pp. 1–4.
58. Wen, E.; Weber, G. Wasmachine: Bring the edge up to speed with a webassembly OS. In Proceedings of the 2020 IEEE 13th International Conference on Cloud Computing (CLOUD), Beijing, China, 19–23 October 2020; pp. 353–360.
59. Li, B.; Fan, H.; Gao, Y.; Dong, W. 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, Virtual Event, 24 June–2 July 2021; pp. 487–488.
60. Li, B.; Dong, W.; Gao, Y. Wiprog: A webassembly-based approach to integrated iot programming. In Proceedings of the IEEE INFOCOM 2021-IEEE Conference on Computer Communications, Vancouver, BC, Canada, 10–13 May 2021; pp. 1–10.
61. Stiévenart, Q.; De Roover, C. Compositional information flow analysis for WebAssembly programs. In Proceedings of the 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM), Adelaide, SA, Australia, 28 September–2 October 2020; pp. 13–24.
62. Bhansali, S.; Aris, A.; Acar, A.; Oz, H.; Uluagac, A.S. A first look at code obfuscation for webassembly. In Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks, San Antonio, TX, USA, 16–19 May 2022; pp. 140–145.
63. Fessel, K.; Dietrich, A.; Zug, S. Programming IoT applications across paradigms based on WebAssembly. In Proceedings of the Workshop on Tools and Concepts for Communication and Networked Systems, Karlsruhe, Germany, 28 September–2 October 2020; pp. 1247–1256.
64. Castillo, C.R.; Marra, M.; Bauwens, J.; Boix, E.G. WOOD: Extending a WebAssembly VM with Out-of-Place Debugging for IoT applications. In Proceedings of the Workshop on Virtual Machines and Language Implementations (VMIL '21), Chicago, IL, USA, 17 October 2021.
65. Castillo, C.R.; Marra, M.; Bauwens, J.; Boix, E.G. Out-of-Things Debugging: A Live Debugging Approach for Internet of Things. *arXiv* **2022**, arXiv:2211.01679.
66. Zhang, Y.; Cao, S.; Wang, H.; Chen, Z.; Luo, X.; Mu, D.; Ma, Y.; Liu, X.; Huang, G. Characterizing and Detecting WebAssembly Runtime Bugs. *arXiv* **2023**, arXiv:2301.12102.
67. Zandberg, K.; Baccelli, E. Minimal virtual machines on IoT microcontrollers: The case of berkeley packet filters with rBPF. In Proceedings of the 2020 9th IFIP International Conference on Performance Evaluation and Modeling in Wireless Networks (PEMWN), Berlin, Germany, 1–3 December 2020; pp. 1–6.
68. Jacobsson, M.; Willén, J. Virtual machine execution for wearables based on WebAssembly. In *EAI International Conference on Body Area Networks*; Springer International Publishing: Cham, Switzerland, 2018; pp. 381–389.
69. Koren, I. A standalone webassembly development environment for the internet of things. In *International Conference on Web Engineering*; Springer International Publishing: Cham, Switzerland, 2021; pp. 353–360.
70. Ménétrey, J.; Pasin, M.; Felber, P.; Schiavoni, V. Watz: A Trusted WebAssembly runtime environment with remote attestation for TrustZone. In Proceedings of the 2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS), Bologna, Italy, 10 July 2022; pp. 1177–1189.

71. Sander, Y. Rust as a platform for IoT. Available online: https://blog.ysndr.de/posts/essays/2021-12-12-rust-for-iot/ (accessed on 1 July 2023).
72. Junior, J.L.S.; de Oliveira, D.; Praxedes, V.; Simiao, D. WebAssembly potentials: A performance analysis on desktop environment and opportunities for discussions to its application on CPS environment. In Proceedings of the 2020 the Anais Estendidos do X Simpósio Brasileiro de Engenharia de Sistemas Computacionais, SBC, Brasilia, Brazil, 23–27 November 2020; pp. 145–150.
73. Wang, W. How Far We've Come–A Characterization Study of Standalone WebAssembly Runtimes. In Proceedings of the 2022 IEEE International Symposium on Workload Characterization (IISWC), Austin, TX, USA, 6–8 November 2022; pp. 228–241.
74. Spies, B.; Mock, M. An evaluation of WebAssembly in non-web environments. In Proceedings of the 2021 XLVII Latin American Computing Conference (CLEI), Cartago, Costa Rica, 25–29 October 2021; pp. 1–10.
75. Alamari, J.; Chow, C.E. Computation at the Edge with WebAssembly. In *ITNG 2021 18th International Conference on Information Technology-New Generations*; Springer International Publishing: Cham, Switzerland, 2021; pp. 229–238.
76. Liu, R.; Garcia, L.; Srivastava, M. Aerogel: Lightweight access control framework for webassembly-based bare-metal iot devices. In Proceedings of the 2021 IEEE/ACM Symposium on Edge Computing (SEC), San Jose, CA, USA, 14–17 December 2021; pp. 94–105.
77. Bakir, F.; Krintz, C.; Wolski, R. Caplets: Resource aware, capability-based access control for IoT. In Proceedings of the 2021 IEEE/ACM Symposium on Edge Computing (SEC), San Jose, CA, USA, 14–17 December 2021; pp. 106–120.
78. Seo, S.C.; Kim, H. Portable and Efficient Implementation of CRYSTALS-Kyber Based on WebAssembly. *Comput. Syst. Sci. Eng.* **2023**, *46*, 2091–2107.
79. Vécsi, Á.; Bagossy, A.; Pethő, A. Cross-platform identity-based cryptography using WebAssembly. *Infocommun. J.* **2019**, *11*, 3–38. [CrossRef]
80. Radovici, A.; Cristian, R.U.S.U.; Şerban, R. A survey of iot security threats and solutions. In Proceedings of the 2018 17th RoEduNet Conference: Networking in Education and Research (RoEduNet), Cluj-Napoca, Romania, 6–8 September 2018; pp. 1–5.
81. Kim, M.; Jang, H.; Shin, Y. Avengers, assemble! Survey of WebAssembly security solutions. In Proceedings of the 2022 IEEE 15th International Conference on Cloud Computing (CLOUD), Barcelona, Spain, 10–16 July 2022; pp. 543–553.
82. Disselkoen, C.; Renner, J.; Watt, C.; Garfinkel, T.; Levy, A.; Stefan, D. Position paper: Progressive memory safety for webassembly. In Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy, Phoenix, AZ, USA, 23 June 2019; p. 18.
83. Stiévenart, Q.; De Roover, C.; Ghafari, M. Security risks of porting c programs to WebAssembly. In Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing, Brno, Czech Republic, 25–29 April 2022; pp. 171–1722.
84. Narayan, S.; Disselkoen, C.; Moghimi, D.; Cauligi, S.; Johnson, E.; Gang, Z.; Vahldiek-Oberwagner, A.; Sahita, R.; Shacham, H.; Tullsen, D.; et al. Swivel: Hardening WebAssembly against spectre. In Proceedings of the 30th USENIX Security Symposium (USENIX Security 21), Virtual, 11–13 August 2021; pp. 1433–1450.
85. Vochescu, A.; Culic, I.; Radovici, A. Multi-Layer Security Framework for IoT Devices. In Proceedings of the 2020 19th RoEduNet Conference: Networking in Education and Research (RoEduNet), Bucharest, Romania, 11–12 December 2020; pp. 1–5.
86. Mäkitalo, N.; Mikkonen, T.; Pautasso, C.; Bankowski, V.; Daubaris, P.; Mikkola, R.; Beletski, O. WebAssembly modules as lightweight containers for liquid IoT applications. In *International Conference on Web Engineering*; Springer International Publishing: Cham, Switzerlan, 2021; pp. 328–336.
87. Napieralla, J. Considering Webassembly Containers for Edge Computing on Hardware-Constrained Iot Devices. Available online: https://www.diva-portal.org/smash/get/diva2:1451494/FULLTEXT02 (accessed on 1 July 2023).
88. Eriksson, F.; Grunditz, S. Containerizing WebAssembly: Considering WebAssembly Containers on IoT Devices as Edge Solution. Available online: https://www.diva-portal.org/smash/get/diva2:1575228/FULLTEXT01.pdf (accessed on 1 July 2023).
89. Putra, R.P. Implementation and Evaluation of WebAssembly Modules on Embedded System-Based Basic Biomedical Sensors. Available online: https://www.diva-portal.org/smash/get/diva2:1360063/FULLTEXT01.pdf (accessed on 1 July 2023).
90. Pham, S.; Oliveira, K.; Lung, C.H. WebAssembly Modules as Alternative to Docker Containers in IoT Application Development. In Proceedings of the 2023 IEEE 3rd International Conference on Electronic Communications, Internet of Things and Big Data (ICEIB), Taichung, Taiwan, 14–16 April 2023; pp. 519–524.
91. Kotilainen, P.; Autto, T.; Järvinen, V.; Das, T.; Tarkkanen, J. Proposing isomorphic microservices based architecture for heterogeneous IoT environments. In *International Conference on Product-Focused Software Process Improvement*; Springer International Publishing: Cham, Switzerland, 2022; pp. 621–627.
92. Ribeiro, E.C. Micro-Containerization in Microcontrollers for the IoT. Available online: https://repositorio-aberto.up.pt/bitstream/10216/142728/2/572043.pdf (accessed on 1 July 2023).
93. Mendki, P. Evaluating webassembly enabled serverless approach for edge computing. In Proceedings of the 2020 IEEE Cloud Summit, Harrisburg, PA, USA, 21–22 October 2020; pp. 161–166.
94. Gadepalli, P.K.; McBride, S.; Peach, G.; Cherkasova, L.; Parmer, G. Sledge: A serverless-first, light-weight wasm runtime for the edge. In Proceedings of the 21st International Middleware Conference, Delft, The Netherlands, 7–11 December 2020; pp. 265–279.
95. Gackstatter, P.; Frangoudis, P.A.; Dustdar, S. Pushing serverless to the edge with webassembly runtimes. In Proceedings of the 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid), Taormina, Italy, 16–19 May 2022; pp. 140–149.

96. Kjorveziroski, V.; Filiposka, S.; Mishev, A. Evaluating webassembly for orchestrated deployment of serverless functions. In Proceedings of the 2022 30th Telecommunications Forum (TELFOR), Belgrade, Serbia, 15–16 November 2022; pp. 1–4.
97. Kjorveziroski, V.; Filiposka, S. WebAssembly as an Enabler for Next Generation Serverless Computing. *J. Grid Comput.* **2023**, *21*, 34. [CrossRef]
98. Hall, A.; Ramachandran, U. An execution model for serverless functions at the edge. In Proceedings of the International Conference on Internet of Things Design and Implementation, Montreal, QC, Canada, 12–18 April 2019; pp. 225–236.
99. McFadden, B.; Lukasiewicz, T.; Dileo, J.; Engler, J. Security Chasms of WASM. NCC Group Whitepaper. Available online: https://git.edik.cn/book/awesome-wasm-zh/raw/commit/e046f91804fb5deb95affb52d6348de92c5bd99c/spec/us-18-Lukasiewicz-WebAssembly-A-New-World-of-Native_Exploits-On-The-Web-wp.pdf (accessed on 1 July 2023).
100. Gadepalli, P.K.; Peach, G.; Cherkasova, L.; Aitken, R.; Parmer, G. Challenges and opportunities for efficient serverless computing at the edge. In Proceedings of the 2019 38th Symposium on Reliable Distributed Systems (SRDS), Lyon, France, 1–4 October 2019; pp. 261–2615.
101. Oliveira, F.; Mattos, J. Analysis of WebAssembly as a strategy to improve JavaScript performance on IoT environments. In Proceedings of the Anais Estendidos do X Simpósio Brasileiro de Engenharia de Sistemas Computacionais, SBC, Brasilia, Brazil, 18 November 2020; pp. 133–138.
102. Šipek, M.; Muharemagić, D.; Mihaljević, B.; Radovan, A. Next-generation Web Applications with WebAssembly and TruffleWasm. In Proceedings of the 2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO), Opatija, Croatia, 27 September–1 October 2021; pp. 1695–1700.
103. Van Hasselt, M.; Huijzendveld, K.; Noort, N.; De Ruijter, S.; Islam, T.; Malavolta, I. 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, Gothenburg, Sweden, 13–15 June 2022; pp. 140–149.
104. Herrera, D.; Chen, H.; Lavoie, E.; Hendren, L. *WebAssembly and JavaScript Challenge: Numerical Program Performance Using Modern Browser Technologies and Devices*; Technical Report SABLE-TR-2018-2; University of McGill: Montreal, QC, Canada, 2018.
105. Niemelä, V.P. WebAssembly, Fourth Language in the Web. Available online: https://www.theseus.fi/bitstream/handle/10024/507127/Niemela_Vili-Petteri.pdf?sequence=2 (accessed on 1 July 2023).
106. Herrera, D.; Chen, H.; Lavoie, E.; Hendren, L. Numerical computing on the web: Benchmarking for the future. In Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages, Boston, MA, USA, 6 October 2018; pp. 88–100.
107. Mikkonen, T.; Pautasso, C.; Taivalsaari, A. Isomorphic internet of things architectures with web technologies. *Computer* **2021**, *54*, 69–78. [CrossRef]
108. Mäkitalo, N.; Bankowski, V.; Daubaris, P.; Mikkola, R.; Beletski, O.; Mikkonen, T. Bringing webassembly up to speed with dynamic linking. In Proceedings of the 36th Annual ACM Symposium on Applied Computing, Virtual, 22–26 March 2021; pp. 1727–1735.
109. Stiévenart, Q.; Binkley, D.W.; De Roover, C. Static stack-preserving intra-procedural slicing of webassembly binaries. In Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 25–27 May 2022; pp. 2031–2042.
110. Tiwary, M.; Mishra, P.; Jain, S.; Puthal, D. Data aware Web-assembly function placement. In Proceedings of the Companion Proceedings of the Web Conference 2020, Taipei, Taiwan, 20–24 April 2020; pp. 4–5.
111. Li, B.; Fan, H.; Gao, Y.; Dong, W. 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, Portland, OR, USA, 27 June–1 July 2022; pp. 261–272.
112. Ménétrey, J.; Pasin, M.; Felber, P.; Schiavoni, V. 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, Minneapolis, MN, USA, 1 July 2022; pp. 3–8.
113. Nakakaze, O.; Koren, I.; Brillowski, F.; Klamma, R. Retrofitting industrial machines with webassembly on the edge. In *International Conference on Web Information Systems Engineering*; Springer International Publishing: Cham, Switzerland, 2022; pp. 241–256.
114. Watt, C.; Rossberg, A.; Pichon-Pharabod, J. Weakening webassembly. *Proc. ACM Program. Lang. (OOPSLA)* **2019**, *3*, 1–28. [CrossRef]
115. Nurul-Hoque, M.; Harras, K.A. Nomad: Cross-Platform Computational Offloading and Migration in Femtoclouds Using WebAssembly. In Proceedings of the 2021 IEEE International Conference on Cloud Engineering (IC2E), San Francisco, CA, USA, 4–8 October 2021; pp. 168–178.
116. Hansson, G. Computation Offloading of 5G Devices at the Edge Using WebAssembly. Available online: https://www.diva-portal.org/smash/get/diva2:1571440/FULLTEXT03 (accessed on 1 July 2023).
117. Zhu, S.; Li, B.; Tan, Y.; Wang, X.; Zhang, J. LAWOW: Lightweight Android Workload Offloading Based on WebAssembly in Heterogeneous Edge Computing. In Proceedings of the 2022 10th International Conference on Information Systems and Computing Technology (ISCTech), Guilin, China, 28–30 December 2022; pp. 753–758.