The Fast and the Vulnerable: A Systematization of WebAssembly Security

Sahar Zamanian[1], Sara Ramezanian[2], Maria Kihl[3]
[1,3]*Lund University, Lund, Sweden*, [2]*Karlstad University, Karlstad, Sweden*
sahar.zamanian@eit.lth.se, sara.ramezanian@kau.se, maria.kihl@eit.lth.se

***Abstract*—WebAssembly (Wasm) has evolved from a performance-oriented web technology into a portable execution platform for cloud and edge computing, and embedded systems. Despite its sandboxed execution model, Wasm remains vulnerable to weaknesses in memory-unsafe programs, runtimes and compilers, host interfaces, and shared hardware resources. This paper presents a Systematization of Knowledge (SoK) on Wasm security based on 135 peer-reviewed studies published since 2017. We classify the literature by vulnerable components, attack and exploitation classes, adversarial models, application domains, detection and mitigation techniques, and security frameworks and tools. Linear memory is the most consistently reported attack surface, highlighting persistent concerns over memory-safety violations despite Wasm's sandboxing and isolation. Other major threats include code-execution and control-flow attacks, side channels, resource abuse, sandbox and host-isolation violations, and malware. Longitudinal analysis shows a shift from runtime correctness and microarchitectural leakage toward cryptojacking and memory exploitation, and more recently toward vulnerabilities in runtimes, compilers, foreign-function interfaces, and host APIs. We also systematize static, dynamic, hybrid, and testing-based detection techniques and isolation, memory-safety, hardware-assisted, and other mitigations. Finally, we identify open challenges in scalable memory safety, cross-language analysis, multi-tenant isolation, malware detection, standardized evaluation datasets, and security support for emerging Wasm features and domains. To the best of our knowledge, this is the first SoK integrating evidence on Wasm vulnerabilities, component-specific attack surfaces, adversarial models, detection and mitigation techniques, and application-domain risks.**

***Index Terms*—WebAssembly security, vulnerabilities, cloud security, security solutions, application domains**

I. Introduction

WebAssembly (Wasm) is a sandboxed, low-level bytecode designed as a portable compilation target that enables near-native execution across different platforms. It was introduced to meet the growing performance demands of web applications like games, 3D graphics, and multimedia. JavaScript could not always handle computationally intensive or low-level tasks efficiently, so Wasm was designed as a fast, safe, and portable execution format independent of language and hardware [1]. It operates as a stack-based virtual machine (VM) that compiles bytecode before execution at near-native speed. Its execution environment is sandboxed, so external resources are accessible only through explicitly defined imports and exports. The main deployment unit is the Module, which contains opcodes and sections for types, functions, tables, memory, globals, imports, and exports [2].

Wasm is now used beyond web browsers, including games, 3D graphics, and multimedia applications [3], as well as cloud, serverless, and edge computing for microservices, lightweight containers, and Function-as-a-Service (FaaS) workloads [4]. Despite its performance, portability, and sandboxing, Wasm still presents security challenges. It can improve system security but may also be abused to evade detection or perform malicious actions. Vulnerabilities from memory-unsafe languages such as C and C++ can persist in Wasm binaries, while traditional native protections may be less effective [5]. Minification and obfuscation can also make malicious behavior harder to detect [6].

As Wasm expands across web and non-web environments, understanding its security landscape has become increasingly important. Its diverse components, deployment settings, and use cases introduce many vulnerabilities and attack vectors. Although existing studies examine specific attacks, vulnerabilities, detection methods, and mitigations, the literature remains fragmented across different components, threat models, and application domains. A systematic synthesis is therefore needed to understand where vulnerabilities arise, how they are exploited, which detection and mitigation techniques have been proposed, and how the research landscape has evolved over time. To address this gap, we conduct a Systematization of Knowledge (SoK) of Wasm security, consolidating findings from 135 peer-reviewed studies published since 2017.

The main contributions of this work are:

1) A systematic analysis of 135 peer-reviewed studies on Wasm security published since 2017.
2) A taxonomy of Wasm vulnerabilities, attacks, adversarial models, and application domains.
3) A systematization of existing detection and mitigation techniques, frameworks, and tools.

The authors at Lund University are part of the Excellence Center at Linköping-Lund on Information Technology (ELLIIT) strategic research area. The work was partially supported by the SSF project SEC4FACTORY under grant no. SSF RIT17-0032. Maria Kihl is partially funded by the competence centre NextG2Com in the VINNOVA program for Advanced Digitalisation with grant number 2023-00541, and she is partially funded by the Wallenberg AI, Autonomous Systems and Software Program (WASP). Sara Ramezanian is partially funded by the Swedish Knowledge Foundation (KKS). This article has received funding from the Smart Networks and Services Joint Undertaking (SNS JU) under the European Union's Horizon Europe research and innovation program under Grant Agreement No 101139067. Views and opinions expressed are, however, those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.

4) An analysis of the evolution of Wasm security research and emerging attack trends.
5) An identification of key research gaps and future challenges in Wasm security.

The remainder of this paper is organized as follows: Sections II and III describe related work and the methodology, respectively; Sections IV and V present the Wasm security taxonomy and application domains, respectively; Section VI presents the cross-analysis; Section VII discusses detection and mitigation techniques; Section VIII covers security frameworks and tools; Section IX identifies research gaps and future directions; and Section X concludes the paper.

II. Related Work

In this section, we review prior SoKs, literature reviews, and surveys on Wasm security, focusing on vulnerabilities, attacks, security mechanisms, and analysis techniques. Kim et al. [7] published the first survey of Wasm security, examining binary protection techniques and vulnerabilities in Wasm-based applications. Harnes et al. [8] later systematized Wasm analysis techniques, with a focus on vulnerability analysis, cryptojacking detection, and smart-contract security. Orlando et al. [9] surveyed Wasm adoption in embedded systems, providing useful context on the security implications of deploying Wasm in resource-constrained environments. More recently, Perrone et al. [10] presented the most closely related study, organizing Wasm security research into seven categories and providing a broad overview of vulnerabilities, attacks, and empirical studies. While valuable, their work remains primarily descriptive. In contrast, our SoK develops a comprehensive taxonomy spanning Wasm components, attacks, vulnerabilities, use cases, and security mechanisms. We further cross-analyze these dimensions to reveal relationships between components, deployment contexts, and threats, identifying the most vulnerable components and severe attack classes. To the best of our knowledge, this is the first multi-dimensional systematization of Wasm security.

III. Methodology

This section describes the methodology used to conduct our SoK on Wasm security. We follow the systematic review process proposed by Kitchenham et al. [11], consisting of three stages: *planning*, *conducting*, and *reporting*. We further use a PRISMA-inspired approach to structure and transparently report the study-selection process, comprising *identification*, *screening*, *eligibility assessment*, and *inclusion* [12].

A. Planning the Systematic Review

1) Objectives and Scope: The primary objective of this study is to systematically characterize Wasm security research. Specifically, we identify and organize: (i) Wasm application domains and use cases; (ii) vulnerabilities; (iii) attacks and exploitation techniques; (iv) adversarial models; (v) threats across application domains and Wasm components; (vi) detection and mitigation techniques; (vii) security tools and frameworks; and (viii) open problems and research gaps.

We consider security research published from 2017 onward, corresponding to the initial release of Wasm. Studies focused exclusively on non-security aspects, such as performance or general language/runtime optimization, are outside the scope unless they make a substantive security contribution.

2) Research Questions (RQs): Based on these objectives, we formulate the following research questions:

RQ1: Application Landscape. Which application domains and use cases are represented in WebAssembly security research?

RQ2: Threat and Vulnerability Landscape. What security threats and vulnerabilities have been investigated in WebAssembly?

- **RQ2.1** What types of attacks have been demonstrated or analyzed?
- **RQ2.2** Which WebAssembly components or layers are affected by these vulnerabilities?
- **RQ2.3** What adversarial models and attacker capabilities are assumed in the included studies?
- **RQ2.4** How is the severity or impact of the identified attacks assessed?

RQ3: Domain-Specific Security Characteristics. How do threats and vulnerable WebAssembly components vary across application domains and use cases?

- **RQ3.1** Which attacks are associated with each WebAssembly application domain?
- **RQ3.2** Which WebAssembly components are most frequently implicated in vulnerabilities within each domain?

RQ4: Detection and Mitigation. What techniques have been proposed or evaluated to detect vulnerabilities and mitigate attacks in WebAssembly?

- **RQ4.1** How do detection and mitigation techniques differ across application domains and WebAssembly components?
- **RQ4.2** What security frameworks and tools have been introduced to support Wasm security?

RQ5: Research Gaps and Future Directions. What limitations, research gaps, and future research directions emerge from the existing Wasm security literature?

3) Identification: We selected Scopus[1] as the primary bibliographic database because of its broad coverage of computer science and engineering literature and its extensive indexing of peer-reviewed publications. The search string was derived from the RQs and eligibility criteria:

```
TITLE-ABS-KEY ( WebAssembly ) AND TITLE-ABS-KEY ( security
     OR secure OR vulnerability OR vulnerabilities OR
     attack OR attacks OR malware OR sandbox OR sandboxing
     OR isolation OR verification OR fuzzing OR detection
     OR monitoring OR intrusion OR threat OR threats ) AND
     ( LIMIT-TO ( SUBJAREA , "COMP" ) OR LIMIT-TO (
     SUBJAREA , "ENGI" ) OR LIMIT-TO ( SUBJAREA , "MATH" )
     OR LIMIT-TO ( SUBJAREA , "DECI" ) OR LIMIT-TO (
     SUBJAREA , "MULT" ) ) AND ( LIMIT-TO ( LANGUAGE ,
     "English" ) )
```

[1] https://www.scopus.com/pages/search/publications?type=advanced

The search was executed on March 2026, and the corpus includes publications available in Scopus up to that date. The search returned 401 records. Scopus was selected for its broad coverage of peer-reviewed computer science and engineering literature and major WebAssembly security venues. To mitigate indexing gaps, backward and forward snowballing was conducted in May 2026 during the inclusion phase.

4) Screening: Two co-authors independently and blindly screened the retrieved records. After removing duplicates, 376 unique records remained. Each reviewer classified papers as *included*, *excluded*, or *maybe*; the latter was used when the title and abstract were insufficient to determine relevance.

A study was considered relevant when its primary focus involved Wasm security, including vulnerabilities, attacks, defenses, or detection. Studies addressing Wasm without a substantive security contribution were excluded.

5) Eligibility Assessment: Potentially relevant studies were then assessed in greater detail against the following criteria: (i) published in 2017 or later; (ii) written in English; (iii) peer-reviewed; and (iv) containing a substantive security contribution related to Wasm. Studies not satisfying these criteria or outside the scope of the RQs were excluded.

6) Inclusion and Conflict Resolution: The two reviewers compared their decisions and resolved disagreements through discussion and consensus. This resulted in 128 papers. Then, the co-authors validate the query by performing backward and forward snowballing, and found additional 7 relevant studies. The final corpus contained 135 papers, which formed the basis of the subsequent analysis. The bibliography also contains background/supporting references that are not included in the 135-study analytical corpus.

B. Conducting the Systematic Review

We conducted systematic data extraction and synthesis using a predefined data-extraction spreadsheet, which is available in our GitHub repository [2]. The extracted information was organized according to the RQs to establish relationships among Wasm components, application domains, vulnerabilities, attacks, detection techniques, and mitigations.

C. Reporting and Synthesis

We synthesize the 135 included studies through a structured taxonomy, cross-analysis, and research-gap analysis. First, we construct a taxonomy of Wasm security threats based on the affected components such as binaries and runtimes. Second, we analyze Wasm security from an application-domain perspective, identifying the threats, affected components, attacks, and defenses investigated in each use case. This enables comparison of domain-specific threats with security problems recurring across multiple environments. Third, we cross-analyze attacks, vulnerabilities, Wasm components, application domains, adversarial models, and defensive techniques to identify recurring attack classes, component-specific weaknesses, and gaps between demonstrated attacks and available defenses. Fourth, we synthesize detection and mitigation techniques according to the security problems they address and the Wasm components at which they operate. This highlights both well-studied defenses and threats for which effective defenses remain limited. Finally, we identify research gaps.

[2]https://github.com/saharzamanian/WebAssembly-Security-SoK

IV. Taxonomy of WebAssembly Security

This section presents a taxonomy of Wasm security that addresses RQ2 and RQ3. The taxonomy, summarized in Figure 1, organizes the main security concerns, including vulnerabilities and attacks identified across the reviewed studies, and provides a structured view of the Wasm security landscape. It also incorporates the adversarial models considered in each study.

A. Vulnerabilities

1) Linear Memory: Linear memory is the most frequently identified vulnerable component. Wasm bounds-checks memory accesses against the size of the module's linear memory, preventing ordinary Wasm instructions from directly accessing memory outside that region. However, this mechanism provides *memory isolation*, not complete *memory safety* within the linear memory itself [13].

This distinction is crucial for C and C++ programs, where stack and heap objects, global data, pointers, and allocator metadata share Wasm's linear memory. Accesses may therefore remain within valid memory bounds while violating object boundaries, preserving vulnerabilities such as buffer overflows, use-after-free, double-free, heap-metadata corruption, invalid pointer accesses, and memory leaks [14]. Although Wasm prevents direct corruption of native executable memory, overflows can still corrupt pointers, function-table indices, or allocator metadata within linear memory. Thus, Wasm's bounds checks limit memory corruption but do not eliminate inherited memory-safety vulnerabilities.

Several studies also identify the *unmanaged stack* as a vulnerable region. This is different from the Wasm runtime's internal operand and call stacks. Compilers such as LLVM may place a C or C++ code stack inside linear memory and maintain its pointer using a mutable global variable. Since this stack is ordinary writable linear memory, stack-based buffer overflows and corruption remain possible [15], [16].

2) Runtime: The Wasm runtime forms a critical part of the sandbox boundary. RQ2 identifies standalone runtimes and virtual machines such as Wasmtime, Wasmer, WAMR, WAVM, and EOS-VM, as well as browser engines such as V8, SpiderMonkey, and JavaScriptCore [17].

Each stage introduces implementation-specific attack surfaces: decoders may misparse malformed instructions, validators may accept invalid modules, interpreters may incorrectly implement opcodes, and JIT/AOT backends may generate incorrect native code [18]. Accordingly, runtime studies report defects in decoding, validation, instruction execution, memory management, error handling, SIMD, garbage collection, exception handling, and other execution features.

Runtime vulnerabilities are particularly important because the runtime enforces Wasm's security guarantees. For example,

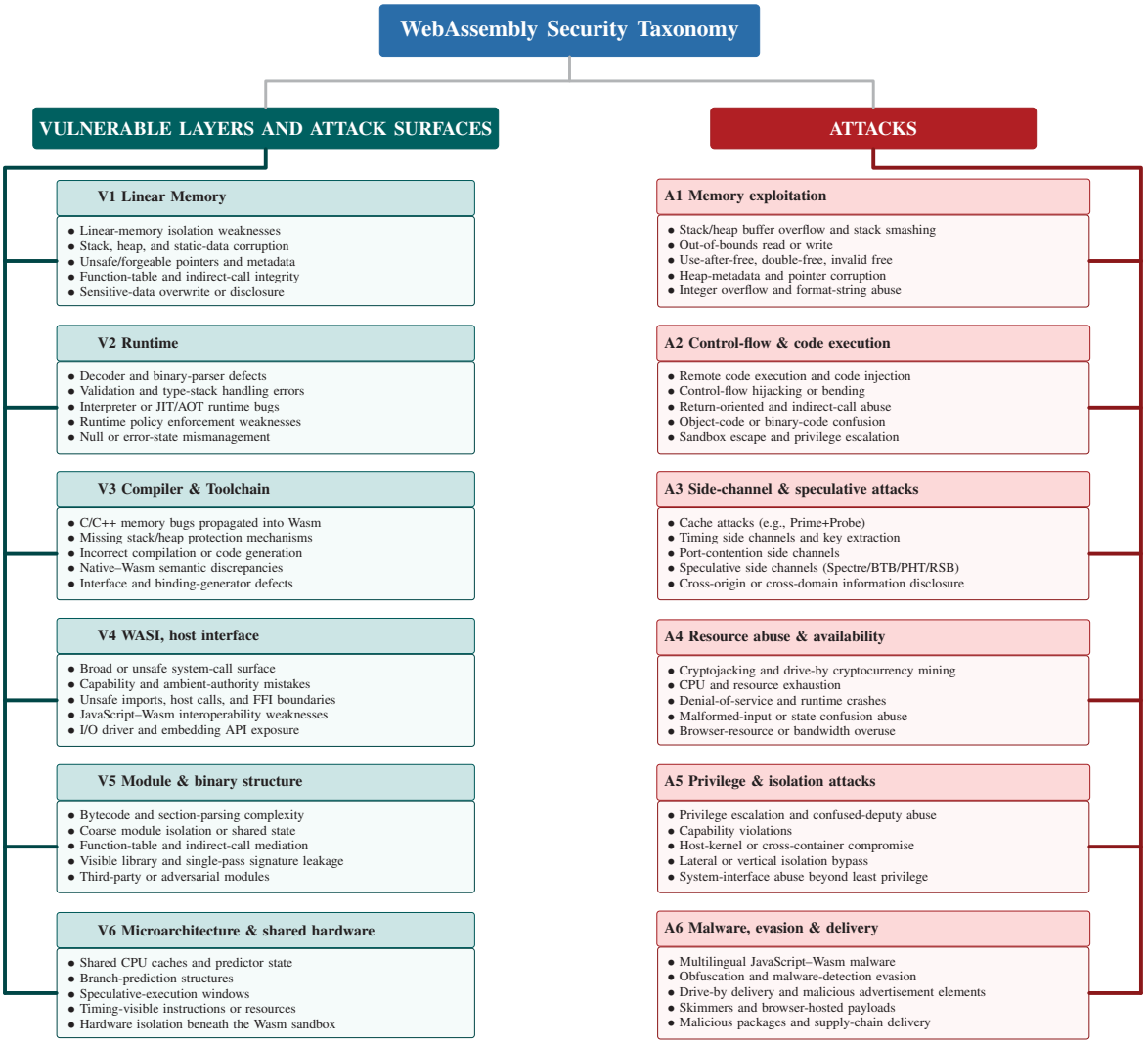


Fig. 1. Taxonomy of WebAssembly security vulnerabilities and attacks

Wasm requires linear-memory accesses to be bounds checked. If a JIT compiler incorrectly removes or generates a bounds check, the generated native code may violate an isolation property that is guaranteed at the Wasm specification level [19]. Severe runtime defects can result in crashes, memory corruption, incorrect execution, or sandbox escape [20], [21].

3) Compiler and Toolchain: Compiler-related vulnerabilities occur both when source programs are translated into Wasm and when Wasm is translated into native machine code. At the source-to-Wasm level, toolchains such as Clang/LLVM, GCC, Emscripten, and WASI SDK determine the layout of the stack, heap, global data, and other program structures. Several reviewed studies identify missing or weaker protection mechanisms, such as absent stack-protection instrumentation, as a reason why vulnerabilities from native C/C++ programs may remain exploitable after compilation to Wasm [22].

At the Wasm-to-native level, JIT and AOT compilers must correctly translate Wasm's safety checks into machine instructions. Bugs in optimization, instruction selection, register handling, or code generation can therefore introduce behavior that does not correspond to the original Wasm semantics [23], [24]. The reviewed studies report compiler bugs, miscompilation, instruction-selection errors, and incorrect code generation as security-relevant implementation weaknesses [25].

4) WASI, Host Interfaces: Core Wasm does not directly provide unrestricted access to operating-system resources. Instead, a module interacts with its environment through imported functions and interfaces such as WASI [26]. These interfaces therefore represent an important security boundary.

When a Wasm module invokes a host function, trusted host code interprets parameters such as memory offsets, lengths, file descriptors, handles, and pointers. Incorrect validation can allow an untrusted module to access resources outside its linear memory. Vulnerable components identified in the reviewed studies include WASI implementations, host-call handlers, system-call wrappers, imported functions, JavaScript–Wasm interfaces, FFIs, and other language bindings [27].

Key risks include overly broad capabilities, unsafe host

functions, inadequate parameter validation, filesystem and path-handling errors, symlink and TOCTOU attacks, and misuse of operating-system services. Thus, even when the core Wasm sandbox operates correctly, flaws in the host interface can expose protected resources to the module [28].

Cross-language communication introduces a related problem. JavaScript–Wasm interactions, native FFIs, WIT-generated bindings, and imported/exported functions must correctly translate data, memory references, types, and resource lifetimes across environments. Errors at this boundary can therefore bypass assumptions within the Wasm module.

5) Module and Binary Structure: Wasm uses structured control-flow instructions such as `block`, `loop`, `br`, and `br_if`, preventing arbitrary jumps to instruction addresses. Indirect calls use `call_indirect`, which selects a function from a table and performs bounds and type checks before the call [2]. These mechanisms make traditional native-code control-flow attacks more difficult, but do not eliminate control-flow manipulation [29], [30].

An issue arises when the *index* for an indirect call originates from corruptible application memory. C/C++ function pointers, for example, are commonly represented as table indexes. If memory corruption modifies the index, `call_indirect` may invoke a different function if the target passes Wasm's type check. Thus, type checking restricts possible targets but does not ensure the intended function is called [31]. The reviewed studies identify function tables, indirect-call indexes, jump tables, mutable stack pointers, and branch-dependent values as security-sensitive control-flow data that can enable control-flow hijacking when combined with memory corruption or implementation weaknesses [32], [33].

A Wasm module is divided into structured sections containing types, imports, functions, tables, memories, globals, exports, code, data, and optional custom sections. Parsers, validators, analysis tools, and malware detectors must correctly interpret this representation. Malformed or unusual bytecode can expose bugs in runtime decoders and validators [34].

In addition, attackers can manipulate custom sections, instruction sequences, constants, function bodies, or data sections without necessarily changing the observable behavior of the program. Such transformations have been investigated for obfuscation and malware-detection evasion [35], [36]. Therefore, binary-level vulnerabilities appear in two forms: implementation bugs triggered while parsing or validating malicious modules, and deliberate manipulation of valid Wasm binaries to conceal malicious behavior or evade static and learning-based detection systems [37], [38].

6) Microarchitecture and Shared Hardware: Several reviewed studies identify hardware components below the Wasm abstraction as additional attack surfaces. These include CPU caches, branch predictors, speculative-execution mechanisms, simultaneous multithreading (SMT), execution ports, and high-resolution timing mechanisms such as `SharedArrayBuffer` [39], [40].

These attacks differ from ordinary memory corruption because they do not necessarily violate Wasm's architectural memory bounds. Instead, sensitive information is inferred from microarchitectural state. For example, a victim's memory access may modify the CPU cache, and an attacker can infer this access by measuring differences in subsequent access times [41]. Similarly, speculative execution can transiently execute operations that affect cache state even when their architectural results are later discarded [42].

Consequently, a correctly implemented Wasm sandbox does not by itself prevent cache-based, speculative-execution, branch-prediction, or port-contention side channels [43]. Protection against these attacks also depends on the browser or runtime and on the underlying processor architecture.

B. Attacks and Exploitation Techniques

Attackers can exploit weaknesses in compiled programs, runtimes, browser interfaces, or underlying hardware, and can use Wasm to deliver or execute malicious code. The identified attacks fall into six main classes, described below.

1) Memory Exploitation: Memory exploitation targets errors in Wasm linear-memory management. While Wasm enforces runtime bounds checks, programs compiled from memory-unsafe languages such as C and C++ may retain vulnerabilities from the source program [14].

Common examples include buffer overflows, out-of-bounds reads and writes, use-after-free, double-free, stack smashing, heap corruption, null-pointer de-references, and integer overflows that lead to incorrect memory allocation. A *buffer overflow* occurs when data is written beyond the allocated boundary of a buffer, while *use-after-free* refers to accessing memory after it has already been released [44].

These vulnerabilities are severe because corruption can occur *within* valid Wasm linear memory without violating the outer sandbox boundary. For example, an overflow may overwrite adjacent objects, function-related data, heap metadata, or security-sensitive values. Memory corruption can therefore enable data modification, information disclosure, control-flow manipulation, or subsequent code execution [13].

2) Code Execution: Code-execution attacks aim to make a program perform operations outside its intended control flow. Reported attacks include remote code execution (RCE), code injection, indirect-call redirection, control-flow hijacking, code-reuse attacks, and arbitrary code execution.

Remote code execution allows attacker-controlled code or operations to execute on a target system. *Control-flow hijacking* redirects execution from its intended path to an attacker-selected target [45]. In Wasm, this may involve corrupting data used by indirect calls, exploiting runtime or compiler bugs, or combining memory corruption with imported host functionality [42]. The impact depends on where execution is obtained: within a Wasm module, it may modify application state or sensitive data; if it reaches privileged host functions, JavaScript interfaces, system calls, or a vulnerable runtime, it can extend beyond the module to host-level compromise.

3) Side-Channel Attacks: Side-channel attacks recover information from observable execution effects rather than directly reading protected memory. Reported techniques in-

clude timing attacks, cache attacks, port-contention attacks, speculative-execution attacks, memory-deduplication attacks, and DRAM-based attacks [46].

A *timing attack* infers information from differences in execution time. Cache attacks such as `Prime+Probe` infer activity by observing changes in cache sets, whereas `Flush+Reload` determines whether shared cache lines have been accessed by measuring reload latency. Port-contention attacks infer another computation's activity by measuring competition for shared CPU execution units [47].

Speculative-execution attacks such as `Spectre` exploit instructions that execute transiently following an incorrect processor prediction. Variants target structures such as the Pattern History Table (PHT), Branch Target Buffer (BTB), and Return Stack Buffer (RSB). Although the transient instructions are later discarded architecturally, their effects can remain observable in caches or other processor state [48].

These attacks are severe because they can disclose cryptographic keys, passwords, memory-access patterns, or other secrets without an ordinary out-of-bounds read [49]. Wasm's memory bounds therefore provide logical memory isolation but do not isolate shared microarchitectural resources such as caches, branch predictors, execution ports, and DRAM [50].

4) Resource Abuse: Resource-abuse attacks unauthorizedly consume computational or system resources or deny them to legitimate workloads. Cryptojacking is the most common example, in which a malicious Wasm module mines cryptocurrency using the victim's CPU [51], [52]. Other forms include CPU monopolization, memory exhaustion, bandwidth consumption, battery draining, file-system exhaustion, entropy depletion, and denial-of-service (DoS). A *DoS attack* reduces or prevents service access by exhausting resources or forcing the target into failure. Wasm enables abuse because CPU-intensive operations run efficiently in browsers, edge devices, and servers.

Resource abuse does not necessarily require memory corruption or sandbox escape; a module may operate within its granted permissions while deliberately consuming excessive CPU time, memory, storage, network capacity, or energy. The consequences include performance degradation, service unavailability, increased cloud costs, and battery depletion on constrained devices [39], [40].

5) Isolation Attacks: Isolation attacks target the boundaries intended to separate Wasm modules from the host system, other modules, or other tenants. Examples include sandbox escape, file-system escape, privilege escalation, unauthorized host-resource access, cross-module interference, capability violations, and host-kernel compromise. A *sandbox escape* occurs when untrusted code crosses the restrictions of its execution environment and gains access to resources outside that environment. Such attacks may originate from runtime bugs, compiler defects, incorrectly implemented bounds checks, unsafe host interfaces, excessive WASI permissions, vulnerable system calls, or errors at the Wasm–host boundary [21].

Isolation attacks are particularly severe because sandboxing is a primary security boundary for untrusted Wasm. Once

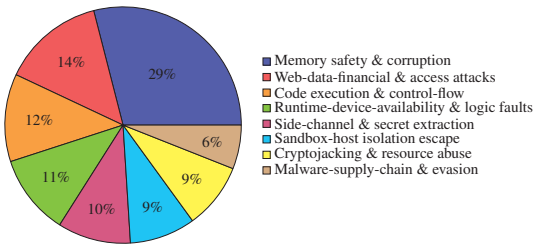


Fig. 2. Distribution of attacks considered most severe

crossed, an attacker may access host files, invoke privileged operations, interfere with other modules, compromise neighboring workloads, or execute code with the hosting process's privileges. A sandbox escape can therefore turn a module-level vulnerability into a system-level compromise [53].

6) Malware: Wasm can also be used to implement, conceal, or deliver malicious functionality. Reported examples include ransomware, cryptominers, phishing components, web keyloggers, JavaScript skimmers, drive-by downloads, malicious advertisements, and JavaScript–Wasm malware [54].

A major concern is *obfuscation*, which makes code harder to analyze while preserving functionality [36]. Wasm malware can use string and function-name obfuscation, control-flow transformations, binary diversification, and semantics-preserving code mutations [38]. *Binary diversification* generates functionally equivalent binaries with different low-level representations, reducing the effectiveness of detectors relying on fixed signatures or structural patterns [55], [56].

Wasm can also split malicious behavior between Wasm and JavaScript. Computational or concealed logic can remain in the Wasm module, while imported JavaScript functions access browser APIs, networking, storage, the DOM, or other host capabilities. This cross-language execution can hinder reconstruction of malicious behavior when security tools analyze only one side of the JavaScript–Wasm boundary [57].

Malware is severe when Wasm is used not only to execute the payload but also to evade detection. Ransomware can use Wasm for high-performance encryption, cryptominers can conceal resource theft, and diversified or obfuscated binaries can bypass signature-based detection. Supply-chain attacks further extend this threat by introducing malicious Wasm modules through dependencies that are treated as trusted components.

7) Which Attacks Are Considered the Most Severe and Why: Of the 135 papers examined, 35 did not specify a most-severe attack. Thus, the distribution shown in Figure 2 is based on the remaining 100 papers. To construct Figure 2, we identified the attack each study explicitly described as its primary or most severe threat. Attacks were grouped by their dominant security objective and exploitation mechanism (e.g., memory corruption and code execution). For studies reporting multiple attacks, only the principal or most severe attack was counted. Studies without an identified primary attack or focused primarily on defenses, tools, methodologies, or general security analyses were excluded. Thus, Figure 2 shows the distribution of reported primary attack concerns among

studies explicitly identifying a most severe attack, rather than across the entire corpus.

Memory-safety and corruption attacks form the largest category (29%), including buffer overflows, out-of-bounds accesses, use-after-free, double-free, and stack corruption. Web, data, financial, and access-related attacks account for 14%, involving unauthorized access, authentication or authorization failures, XSS, phishing, and blockchain attacks, with impacts including data disclosure, manipulation, account compromise, and financial loss.

Code-execution and control-flow attacks represent 12%, covering code injection, remote or arbitrary code execution, and control-flow manipulation that redirects execution to unintended or attacker-controlled code. Runtime, device, availability, and logic-related attacks account for 11%, including denial of service, crashes, resource exhaustion, runtime and compiler failures, device faults, and logic flaws. Side-channel and secret-extraction attacks account for 10%, including Spectre, cache, timing, and port-contention attacks that can expose sensitive information without violating memory boundaries.

Sandbox and host-isolation attacks account for 9%, targeting Wasm's isolation boundary to compromise hosts or protected domains. Cryptojacking and resource-abuse attacks also account for 9%, exploiting computational, memory, energy, or cloud resources. The remaining 6% comprises malware, supply-chain, and detection-evasion attacks, including malicious dependencies and techniques that conceal Wasm code from static or behavioral detection.

C. Adversarial Models

We classify adversarial assumptions as honest, semi-honest, malicious, or unspecified. Of the 135 reviewed papers, 112 (83.0%) assume malicious adversaries, 2 (1.5%) honest adversaries, and 1 (0.7%) semi-honest adversaries, while 20 (14.8%) leave the model unspecified. Among studies defining an adversarial model, 97.4% assume malicious adversaries.

The malicious model is frequently used because Wasm is widely accessible while increasingly deployed in critical infrastructures handling sensitive data and resources. Researchers therefore consider worst-case scenarios where users, inputs, or other components may be intentionally malicious, testing whether Wasm's sandboxing, memory isolation, runtime, compiler, and other security mechanisms withstand active attacks. The very small number of honest and semi-honest models indicates that Wasm security research primarily focuses on active adversarial threats rather than *accidental faults* or *honest but curious* participants.

V. Security in WebAssembly Application Domains

The security implications of Wasm depend on its deployment environment. Across the reviewed literature, attack surfaces extend beyond Wasm linear memory to runtime implementations, host interfaces, shared hardware, peripheral access, and application-specific logic. Table I presents the most frequently studied Wasm application domains in the reviewed literature, their vulnerable components, consequent exploits, and corresponding mitigations. The following discussion summarizes the principal vulnerabilities and attacks in the most frequently studied domains, and addresses RQ1 and RQ4.

A. Domain-Specific Vulnerabilities and Attacks

1) Web and Browser Applications: Browser-based Wasm is exposed through linear memory, browser engines, JavaScript–Wasm interfaces, browser APIs, and native libraries compiled to Wasm. Threats include memory corruption, cryptojacking, XSS, ransomware, malicious modules and extensions, and Spectre-style timing or cache side channels [48], [58], [59]. Memory vulnerabilities inherited from C/C++ libraries can also remain exploitable in Wasm linear memory [60].

2) IoT and Embedded Systems: In IoT deployments, primary attack surfaces include linear memory, embedded runtimes, peripheral interfaces, and resource-management mechanisms [61]. Reported attacks include buffer overflows, out-of-bounds accesses, use-after-free, resource exhaustion, control-flow violations, and unauthorized sensor or actuator access [62]. These risks are particularly significant on constrained devices with limited or absent hardware memory protection.

3) Cloud and Edge Computing: Cloud and edge environments primarily expose Wasm through runtime sandboxing, WASI/WASIX interfaces, host system calls, shared memory, and processor resources. Runtime or WASI flaws may enable sandbox escapes or unauthorized host access, while Spectre, cache attacks, port contention, covert channels, and RowHammer exploit shared hardware. Resource exhaustion and malicious modules further threaten availability and tenant isolation [19], [21].

4) Industrial Control Systems: In industrial systems, vulnerable components include memory-mapped I/O registers, interrupt interfaces, runtime isolation, and third-party modules [63]. The reviewed studies report remote code execution, register manipulation, malicious modules, memory corruption, and priority or interrupt interference [64]. Unlike conventional web environments, exploitation can therefore affect not only software state but also physical control and real-time behavior.

5) Wasm-Based Smart Contracts: Smart-contract security involves Wasm-level memory safety and blockchain-specific logic. Memory errors such as stack overflows and allocator vulnerabilities may propagate from unsafe source languages, while fake transfers, forged notifications, missing authorization checks, reentrancy, rollback, and integer overflow arise from contract logic [65]. Predictable blockchain state and vulnerabilities in blockchain-specific Wasm engines introduce additional risks [66]. Consequently, analysis must consider both linear-memory behavior and persistent blockchain state.

6) Other Application Domains: Additional attack surfaces arise in server-side applications, software supply chains, cryptographic libraries, machine-learning systems, and storage applications. These include unsafe WASI interfaces, malicious packages, compiler or binding-generation defects, cryptographic timing leakage, model extraction, malware-detector evasion, and ransomware. Such cases demonstrate that Wasm

security also depends on the surrounding toolchain, host environment, and application-specific components.

B. Domain-Specific Mitigation

1) Web and Browser Applications: Browser-oriented defenses combine malicious-module detection, memory isolation, and side-channel protection. Static, dynamic, and ML-based systems such as MINOS, MineSweeper, SEISMIC, and OutGuard detect malicious or cryptojacking modules, while stack canaries and fine-grained library isolation limit memory corruption [84], [85]. Constant-time programming mitigates timing leakage, and browser-API monitoring can restrict suspicious file or host operations.

2) IoT and Embedded Systems: Mitigation focuses on lightweight isolation and resource control. Trusted Execution Environments (TEEs), such as Arm TrustZone, isolate sensitive Wasm workloads [86], while capability-based permissions restrict peripheral access. Bounds checking, stack canaries, memory tagging, execution timeouts, and resource quotas address memory corruption and denial-of-service attacks. Runtime anomaly detection provides additional protection against malicious module behavior.

3) Cloud and Edge Computing: Cloud and edge defenses primarily strengthen tenant and host isolation. Restricted WASI interfaces and syscall mediation limit host access, while capability-based memory protection, Intel MPK, and memory tagging provide finer-grained isolation [41], [87]. TEEs and remote attestation protect workloads from untrusted hosts, whereas resource quotas and side-channel defenses address denial-of-service and shared-hardware leakage.

4) Industrial Control Systems: Industrial defenses emphasize isolation while preserving real-time requirements. Software Fault Isolation (SFI) constrains memory and control transfers, while driver isolation separates potentially faulty components. Wasm-IO additionally protects hardware registers and interrupt handling from unauthorized modification and timing interference [73]. Runtime monitoring and policy enforcement can further detect abnormal control behavior.

5) Wasm-Based Smart Contracts: Smart-contract defenses combine program analysis with blockchain-aware validation. Static and symbolic analysis identify unsafe memory operations, authorization errors, and vulnerable execution paths, while state-aware analysis tracks persistent state across calls. Explicit authorization checks mitigate contract-logic attacks, and stack canaries provide basic protection against inherited memory errors. Formal verification can further establish that contract execution follows intended semantics [88], [89].

6) Other Application Domains: For other Wasm deployments, mitigation is largely application-specific. Supply-chain uses fuzzing, differential testing, static analysis, and reproducible builds; cryptographic applications require constant-time implementations; server-side deployments use restricted WASI interfaces and privilege separation; and untrusted binaries are analyzed through static analysis, instrumentation, symbolic or concolic execution, and fuzzing [90], [91].



Fig. 3. Annual trends in WebAssembly security research; 2026 represents a partial year.

VI. Cross-Analysis of WebAssembly Security

For the cross-analysis, we identified a recurring pattern in which certain vulnerabilities and their exploitations were more common in specific years, followed by their mitigation. We therefore analyzed the vulnerabilities discussed most frequently each year, the attacks exploiting them, and the corresponding mitigation techniques. Figure 3 summarizes the evolution of the main Wasm security topics from 2017 to May of 2026[3]. Values indicate the number of reviewed papers addressing each category and thus represent research attention rather than real-world attack frequency.

In 2017–2018, research attention increasingly shifted toward *Wasm runtime correctness* and *microarchitectural leakage*, with the main demonstrated threat being a cache-based side-channel exploiting shared processor resources rather than linear-memory corruption. Defenses included formal validation, isolation, and trusted cryptographic implementations.

During 2019–2020, *cryptojacking* became a major investigated attack. In this case, attackers primarily abused Wasm's computational efficiency rather than exploiting a memory vulnerability. Dynamic monitoring and resource-usage analysis were therefore common mitigation approaches. At the same time, research increasingly examined linear-memory weaknesses and indirect calls as possible sources of memory corruption and control-flow attacks.

From 2021 to 2024, *memory exploitation* became the most frequently represented in our corpus attack category. Buffer overflows, use-after-free, out-of-bounds accesses, and heap or stack corruption were mainly associated with Wasm's linear memory and memory-unsafe source languages such as C and C++. Corruption of function tables or indirect-call state could further enable control-flow hijacking or remote code execution (RCE) [22]. Consequently, the main defenses included static

[3]The main database search was conducted in March 2026, followed by the snowballing search in May 2026.

TABLE I. OVERVIEW OF WASM APPLICATION DOMAINS WITH THEIR SECURITY RELATED ISSUES

Application domains	Includes	Vulnerable Components	Exploitations	Mitigation	References
Web, Browser, Client-side web	Browser-based Applications	Linear/shared memory, browser Wasm engines (V8, JavaScriptCore), JS–Wasm boundary, embedding/host functions, GC/SIMD, data/code sections	Memory corruption, XSS, Spectre/cache/timing side channels, cryptojacking, malware evasion/obfuscation, crashes	Sandboxing and fine-grained library isolation; linear-memory bounds checking; constant-time execution and branch linearization against timing attacks; strict CSP/COOP/COEP policies; filesystem-access interception; W^X and runtime hardening.	[19], [46], [49], [54], [56]–[59], [67]
Cloud and Edge Computing	FaaS, Serverless, 5G/6G	Runtime sandbox, intra-process isolation, WASI/WASIX interfaces, host syscall ABI, virtual memory, filesystem/memory-mapping interfaces	Sandbox escape, tenant-isolation failure, Spectre/isolation bypass, resource exhaustion, scalability issues,Container escape, executable-memory abuse	TCB minimization; TEEs (SGX/TrustZone) and remote attestation; Memory migration/encryption; syscall/host-interface isolation; CPU, memory and filesystem resource quotas; timeouts; W^X/HFI-based runtime isolation.	[7], [68]
IoT and Embedded Systems	CPS, ICS, Automation	Peripherals, MMIO registers, ECUs/MCUs, vehicle-signal interfaces, interrupt lines, industrial data layers, untrusted modules, software stack, Energy and resource exhaustion	Unauthorized peripheral access, Exacerbated by absence of MMU, Remote code injection, Backdoors, Unauthorized hardware and data access, cascading outages, Isolation failure, unauthorized access/interference between applications	Software fault isolation (SFI); TrustZone/TEE isolation; capability-based peripheral access; counting semaphores and strict energy allotments; secure boot and remote attestation; isolated device drivers; memory bounds/address-translation protection.	[41], [61], [62], [64], [69]–[76]
Blockchain	EOSIO, Cryptocurrency applications, Smart Contracts	Linear memory, contract execution state, runtime/VM, authorization logic, RNG, persistent storage	Buffer/stack overflow, authorization flaws, forged notifications, weak randomness, state-manipulation bugs, runtime/VM vulnerabilities	Sandboxed contract execution; stack canaries, memory hardening; authorization and address verification; token issuer/recipient validation; sensitive-data-flow tracking, symbolic analysis; runtime validation	[28], [52], [65], [77]–[80]
Infrastructure, Server-side	Node.js, HTTP servers, Isolation/sandboxing of native/COTS libraries	Wasm code and host environment, imported functions/runtime	Code injection, memory corruption, remote code execution (RCE)	Native/COTS libraries; Seccomp/syscall filtering; host-function restriction and capability-based permissions; memory bounds checking and stack-smashing protection; W^X enforcement; pre-deployment testing and runtime validation.	[19], [43], [81]
WASM Core and Runtimes	Wasmtime, WAMR, wasm3, wasm-interp, EOS-VM, browser engines, WASI, FFI	Module-to-module interfaces and shared mutable memory, Filesystem APIs, pre-opened directories, FSA API, host syscall interfaces	Malicious module interaction, cross-module corruption, Sandbox escape, path/filesystem violations, unauthorized local file access	Module-boundary and encapsulation verification; centralized hostcall safety policies; capability-based filesystem/WASI access; linear-memory bounds checking and fine-grained memory isolation; Stack protection; systematic runtime testing.	[20], [21], [63], [82], [83]

analysis, fuzzing, stack canaries, shadow memory, bounds checking, memory isolation, and formal verification.

In 2025–2026/05/01, memory exploitation remained important, but research expanded to runtimes, compilers, WASI, FFIs, and host APIs. Excessive host permissions or unsafe imports could enable sandbox escape, filesystem abuse, or privilege escalation despite intact linear-memory isolation. Mitigations emphasize least-privilege access, stronger WASI isolation, runtime/compiler fuzzing, memory hardening, and hardware-assisted isolation [87].

VII. DETECTION TECHNIQUES AND MITIGATION

A. Detection Techniques

The reviewed studies employ three main classes of Wasm security detection techniques: static, dynamic, and hybrid (RQ4).

1) Static Analysis: Static analysis examines Wasm binaries without execution using techniques such as abstract interpretation, control-flow and reachability analysis, information-flow and taint analysis, and binary-level vulnerability scanning [92]–[94]. These methods are primarily used to identify memory-safety violations, unsafe function interactions, malicious data flows, and vulnerabilities in Wasm smart contracts [95].

2) Dynamic Analysis: Dynamic analysis observes program behavior at runtime through instrumentation, runtime checks, resource monitoring, and profiling. Key indicators include CPU usage, instruction distributions, memory-access patterns, and timing behavior, especially for detecting cryptojacking, anomalous execution, and side-channel activity.

3) Hybrid Analysis: Hybrid approaches combine static program properties with runtime observations, enabling detection of vulnerabilities and anomalous behaviors that static or dynamic analysis alone may miss [95].

B. Mitigation Techniques

The reviewed mitigation mechanisms can be grouped into five main categories: isolation and memory protection, code hardening, privilege restriction, compiler and runtime hardening, and resource protection.

1) Isolation and Memory Protection: Sandboxing and isolation are among the most common defenses. These include Wasm linear-memory sandboxing, SFI, process and tenant

isolation, and separation between Wasm modules and host components. They are often combined with dynamic bounds checking to restrict memory accesses to valid regions [55], [96], [97].

Memory-safety mechanisms further include memory-safe Wasm extensions, segmented memory, unforgeable handles or capabilities, memory tagging, canaries, shadow memory, and redzones [98]–[101]. Hardware-assisted mechanisms such as Memory Protection Keys (MPK), TEEs, and TrustZone provide stronger memory separation [63]. Control-Flow Integrity (CFI), stack-smashing protection, and Write-XOR-Execute (W⊕X) policies additionally mitigate control-flow and memory-corruption attacks.

2) Binary and Code Hardening: Code-level defenses include binary rewriting, diversification, obfuscation, code encryption, and instruction or memory-layout transformations [16], [30]. These mechanisms aim to reduce exploitability and hinder reverse engineering, malware analysis, and some side-channel attacks [102].

3) Privilege and Interface Restriction: System-level defenses apply least-privilege execution, privilege separation, capability-based access control, RPC-based process separation, and mediation of WASI calls. These mechanisms limit the access of untrusted Wasm modules to host resources and sensitive system services [97].

4) Compiler and Runtime Hardening: Compiler and runtime defenses include hardened JIT and verified compilation, secure or measured boot, remote attestation, and runtime sanity checks [22], [103]. Testing techniques such as structure-aware, greybox, and binary-only fuzzing, differential testing, symbolic execution, and concolic execution identify defects in Wasm runtimes, interpreters, JIT compilers, and smart contracts before deployment. Sanitizers and canary-based instrumentation further detect stack and heap corruption [99].

5) Resource and Application-Level Protection: Resource monitoring, throttling, and rate limiting mitigate cryptojacking and resource-exhaustion attacks, while trusted execution and access control protect Wasm deployments in edge, cloud, IoT, and ICS environments.

6) Anomaly-Based Detection: Several studies detect threats by identifying deviations from expected Wasm execution behavior using runtime features or behavioral profiles [104]–[106]. These approaches are particularly useful when explicit vulnerability signatures are unavailable.

VIII. Security Frameworks and Tools

The reviewed studies cover static and information-flow analysis, symbolic and concolic execution, dynamic analysis and fuzzing, memory isolation, side-channel protection, malware detection, smart-contract security, and formal verification [89], [107].

A. Static Analysis and Information-Flow Security

Static tools detect vulnerabilities without executing Wasm. *WasmChecker* [108] uses abstract interpretation to detect memory and program errors; *Wasmati* [109] builds a Code Property Graph (CPG) with the Wasmati Query Language (WQL) for vulnerability queries; and *WasmA* [93] generates call, control-, and data-flow graphs for analysis and malware fingerprinting. *WAPPLER* [95] performs Horn-clause reachability analysis, *Wassail* [8] supports compositional and binary-slicing analysis, and *WaLi* [78] identifies security-critical paths.

For multilingual applications, *JWBinder* [67] reconstructs JavaScript–Wasm dependencies using an Inter-language Program Dependency Graph, while *PowerPoly* [110] uses Wasm as a common Intermediate Representation. For information-flow security, *SecWasm* [45] combines static Information-Flow Control typing with dynamic memory checks, while *Wanilla* [111] combines taint tracking, abstract interpretation, and relational reasoning to verify noninterference.

B. Dynamic Analysis, Instrumentation, and Fuzzing

Dynamic frameworks inspect runtime behavior. *Wasabi* [96] instruments Wasm binaries for profiling, coverage, call-graph extraction, memory tracing, and taint analysis [92], while *WASMDYPA* [112] combines instrumentation with runtime plugins for overflow and memory analysis. Other frameworks include *JASMaint* [92], *Oron* [113], and the *Wizard Research Engine* [91].

Memory detectors include *WBSan* [114], *VMCANARY* [99], *WASHADOW* [100], and *metaSafer* [115]. WBSan combines dependency analysis, instrumentation, shadow memory, and red zones; VMCANARY validates stack canaries; WASHADOW checks allocation and memory accesses using shadow memory; and metaSafer protects heap metadata outside linear memory.

Fuzzers include *WasmFuzzer* [116], *SwFuzz* [117], *WALTZZ* [103], *ESFuzzer* [118], *WAFL* [119], *WasmCFuzz* [25], *WAGEN* [20], and *Wapplique* [120], using coverage, grammar, structure, grey-box, mutation, and differential-testing techniques [120]. *LWDIFF* [121] adds Large Language Model generated mutations; WAGEN and Wapplique perform differential testing; *WasmDiff* [31] compares native and Wasm executions; *Atlas* [27] tests WebAssembly Interface Type (WIT) bindings; and *WEST* [89] derives tests from mechanised specifications.

C. Symbolic and Concolic Vulnerability Detection

SeeWasm [33], *Owi* [122], *Eunomia* [34], *WANA* [123], and *FVPS* [124] use symbolic execution and Satisfiability Modulo Theories (SMT) solvers such as Z3 to determine whether security-sensitive states are reachable. Concolic execution combines concrete and symbolic execution: *WASP* [125] instruments the reference interpreter; *WASIF* [65] adds taint-based information-flow analysis and function-invocation sequences; and *WASDAM* [80] combines concolic fuzzing with a Data Access Model (DAM), sensitive-code tracing, test oracles, and path optimisation [126]. These approaches reduce path imprecision associated with purely static analysis [107].

D. Memory Safety, Sandboxing, and Privilege Isolation

The following preventive frameworks target memory safety and isolation. *WASMSEPA* [127] isolates unsafe libraries in

separate processes via Remote Procedure Calls, preventing direct access to Wasm linear memory [97]. *MSWasm* [98] uses segmented memory and unforgeable handles for spatial and temporal safety. *Cage* [40] combines Arm Memory Tagging Extension and Pointer Authentication; *PKUWA* [14] uses Intel Memory Protection Keys and Domain Isolated Linear Memory; *WARD* [76] uses red zones and randomised address translation; and *Bento* [87] isolates memory regions through static binary rewriting [128].

At the system level, *WaSC* [129] separates WASI using a daemon and syscall barriers, while *WASMBOX* [70], *WAIT* [72], *Aerogel* [71], and *Silverline* [62] provide sandboxing, access control, resource isolation, or syscall monitoring. *WasDom* [24] protects JIT code, while stack protection uses canaries and VM-assisted validation.

TEEs provide stronger isolation: *WATZ* [86] uses Arm TrustZone with remote attestation; *TWINE* [63] combines Wasm with Intel SGX, encrypted storage, and attestation; and *Edgedancer* [68] uses SGX for trusted edge execution. *WaVe* [28] instead formally verifies WASI host-call safety contracts.

E. Side-Channel and Speculative-Execution Security

Swivel [48] mitigates Spectre-style attacks using SFI, Intel Control-flow Enforcement Technology (CET), MPK, linear blocks, branch-target-buffer flushing, separated return stacks, and register interlocking. *WaSCR* [59] detects secret-dependent branches through taint analysis over a Program Dependency Graph (PDG) and rewrites them into constant-time forms, while *VIVIENNE* [90] uses Relational Symbolic Execution (RelSE) and invariants to verify constant-time cryptography; [130] formally verifies secret independence.

Lurking Eyes [50] detects execution features linked to timing and speculative attacks, while *PC-detector* [43] identifies processor-port contention. Diversification tools such as *WASM-MUTATE* [131] and e-graph approaches generate semantically equivalent but structurally different binaries to reduce predictable microarchitectural behavior.

F. Malware, Cryptojacking, and Anomaly Detection

Malware detectors combine signatures, runtime behavior, and Machine Learning (ML). *MINOS* [84] converts Wasm binaries to grayscale images for Convolutional Neural Network (CNN)-based cryptojacking detection; *MineThrottle* [58] instruments JIT execution to detect and throttle mining threads. *MineSweeper* [132], *MinerRay* [133], *SEISMIC* [7], *RAPID* [134], *Outguard* [8], and *CMTracker* [135] use instruction patterns, signatures, resource monitoring, or learning-based classifiers.

WasmGuard [136] uses adversarial and contrastive learning; *WasmEye* [105] analyzes function-call and memory-access traces; other approaches classify WASI-call sequences; and *RøBguard* [137] detects browser ransomware through File System Access (FSA) calls and file entropy/size changes. Detector robustness is evaluated using *SWAMPED* [106] and tools such as *MADVEX* [138], *wasm-mutate* [131], and *WASMixer* [35], which study evasion, diversification, and obfuscation rather than directly detecting vulnerabilities [139].

G. Security Analysis of Wasm Smart Contracts

Smart-contract frameworks combine path-sensitive and runtime analysis. *EOSAFE* [77] uses symbolic execution and heuristic detectors; *WASMOD* [79] combines instrumentation, runtime validation, grey-box fuzzing, and test oracles; *WASIF* [65] combines concolic execution with invocation sequences and taint analysis; *WASDAM* [80] models contract data access and fuzzes sensitive paths; and *WaLi* [78] combines static critical-path detection with runtime monitoring. These combinations are important because contract vulnerabilities often depend on specific states and input sequences.

H. Formal Verification and Security Assurance

Formal frameworks aim to prove security properties rather than only detect violations. *Iris-Wasm* [88] uses higher-order separation logic and logical relations; *Iris-MSWasm* and *MSWasmCert* formalise memory-safety and capability invariants [140]; and *WasmRef-Isabelle* and *WasmCert-Isabelle* provide verified Wasm semantics and interpreters that can also serve as differential-fuzzing oracles [141].

Arrival [142] verifies instruction selection against Instruction Set Architecture semantics, while *ZKWASM* [143] proves correct Wasm execution using Zero-Knowledge Succinct Non-interactive Arguments of Knowledge. Unlike conventional scanners, these systems establish execution, memory-safety, or semantic properties through formal proof or construction.

It is important to distinguish Wasm security frameworks from supporting infrastructure. Tools such as Z3, AFL/AFL++, libFuzzer, ASan, Intel Pin, VirusTotal, Binaryen, WABT, and Emscripten support security techniques but are not dedicated Wasm security frameworks [32]. This distinction clarifies the Wasm security ecosystem identified under RQ4. Overall, RQ4 reveals two main directions: detection through analysis, fuzzing, monitoring, and machine learning [67], and mitigation through isolation, memory protection, TEEs, constant-time techniques, and formal verification [86], [107], [124].

IX. Research Gaps and Future Directions

Based on the findings of RQ5, recent Wasm security research increasingly focuses on the *scalability, precision, portability, and deployability* of defenses as Wasm expands beyond browsers [121], [144].

Memory safety and isolation remain major gaps. Existing work highlights limitations in temporal memory safety, use-after-free protection, stack isolation, unmanaged heaps, memory64, and recovery of memory bounds from optimized binaries, while some hardware-assisted solutions remain architecture-dependent [6], [13], [44], [82], [98]. Lightweight privilege separation, resource isolation, secure host interaction, and least-privilege system-call control are also needed without undermining Wasm's performance advantages.

Program analysis remains challenging due to indirect calls, linear-memory aliasing, imports, multi-module execution, and control-flow reconstruction, while dynamic instrumentation may incur substantial runtime overhead [93], [112]. Fuzzing

and symbolic execution face input-generation and path-explosion challenges, respectively. Formal verification must cover Wasm stacks, including JIT/AOT compilation, dynamic loading, system interfaces, and features such as threads, SIMD, GC, Reference Types, memory64, and exception handling [82], [124]. Compilation and optimization further obscure source-level information, necessitating improved recovery of types, control flow, memory semantics, and security properties across languages, compilers, and runtimes [60]. The Component Model adds concerns regarding binding correctness, inter-component isolation, and secure data exchange [6].

Malware and anomaly detection remain vulnerable to obfuscation, binary diversification, adversarial transformations, and JavaScript–Wasm interactions. Progress is also limited by the lack of representative datasets for cryptojacking, ransomware, keylogging, smart-contract malware, and obfuscated samples [58]. More robust cross-language detection, standardized datasets, lightweight monitoring, and real-time prevention mechanisms are therefore required [57], [67], [144], [145]. Side-channel risks, including timing, cache, and speculative-execution leakage, also require practical detection and mitigation that remain effective after JIT/AOT compilation.

Wasm's broader deployment introduces further isolation and platform-specific challenges. Cloud and serverless systems require stronger CPU, memory, file-system, kernel, and tenant isolation beyond the basic sandbox [81], [146]. Embedded, IoT, and industrial systems require standardized low-level hardware interfaces, deterministic execution, attestation, and lightweight verified runtimes. Smart-contract analysis continues to suffer from path explosion, incomplete transaction modeling, and limited vulnerability datasets [33], [107]. Browser applications require better analysis of JavaScript–Wasm interactions and safer access to browser and local resources [54], [56]. For machine-learning workloads, model extraction, reverse engineering, and resource-constrained inference remain important concerns [102], [138], [139].

a) Future Directions: Promising directions include *sound memory-safety enforcement* for memory64, GC, and the Component Model across JIT/AOT runtimes; *cross-language analysis* of Wasm, JavaScript, WASI, and host functions; and *multi-tenant isolation* with fine-grained CPU, memory, and system-call control. Other priorities include adversarially robust malware detection using realistic datasets, scalable hybrid analysis and verification, and lightweight verified runtimes and hardware interfaces for embedded and IoT systems. These areas offer measurable opportunities to improve security precision and portability while reducing runtime overhead.

X. Conclusion

We analyzed 135 studies covering Wasm vulnerabilities, attacks, adversarial models, application domains, detection and mitigation techniques, and security tools. To the best of our knowledge, this is the first SoK to integrate these dimensions of Wasm security. Our findings show that sandboxing alone is insufficient: memory vulnerabilities remain significant, particularly in memory-unsafe applications, while runtimes, compilers, host interfaces, and shared resources introduce additional threats. Thus, Wasm security requires defense in depth.

Research has progressed from runtime correctness and cryptojacking to memory safety, runtimes, compilers, host interfaces, and cross-language interactions, with defenses increasingly combining analysis, testing, runtime protection, isolation, and formal methods. Future work should improve memory safety and isolation, address emerging features, evaluate defenses across runtimes and domains, and establish standardized benchmarks for reproducibility.

References

[1] A. Haas *et al.*, "Bringing the web up to speed with webassembly," in *Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation*, 2017, pp. 185–200.

[2] D. Lehmann, J. Kinder, and M. Pradel, "Everything old is new again: Binary security of {WebAssembly}," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 217–234.

[3] B. McFadden, T. Lukasiewicz, J. Dileo, and J. Engler, "Security chasms of wasm," *NCC Group Whitepaper*, 2018.

[4] W. Xin, Z. Kai, and Q. Bin, "Review of webassembly application research for edge serverless computing." *Journal of Computer Engineering & Applications*, vol. 59, no. 11, 2023.

[5] M. Waseem, T. Das, A. Ahmad, P. Liang, and T. Mikkonen, "Issues and their causes in webassembly applications: an empirical study," in *Proceedings of the 28th international conference on evaluation and assessment in software engineering*, 2024, pp. 170–180.

[6] W.-C. Wu *et al.*, "Is this the same code? a comprehensive study of decompilation techniques for webassembly binaries," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2024, pp. 108–130.

[7] M. Kim, H. Jang, and Y. Shin, "Avengers, assemble! survey of webassembly security solutions," in *15th International Conference on Cloud Computing (CLOUD)*. IEEE, 2022, pp. 543–553.

[8] H. Harnes and D. Morrison, "Sok: analysis techniques for webassembly," *Future Internet*, vol. 16, no. 3, p. 84, 2024.

[9] T. Orlando, L. D'Agati, F. Longo, G. Merlino *et al.*, "A survey of webassembly usage for embedded applications: Safety and portability considerations." in *ITASEC/SERICS*, 2025.

[10] G. Perrone and S. P. Romano, "Webassembly and security: A review," *Computer Science Review*, vol. 56, p. 100728, 2025.

[11] B. Kitchenham, S. Charters *et al.*, "Guidelines for performing systematic literature reviews in software engineering," 2007.

[12] D. Moher, A. Liberati, J. Tetzlaff, and D. G. Altman, "Preferred reporting items for systematic reviews and meta-analyses: the prisma statement," *Bmj*, vol. 339, 2009.

[13] C. Disselkoen, J. Renner, C. Watt, T. Garfinkel, A. Levy, and D. Stefan, "Position paper: Progressive memory safety for webassembly," in *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, 2019, pp. 1–8.

[14] H. Lei *et al.*, "Put your memory in order: Efficient domain-based memory isolation for wasm applications," in *ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 904–918.

[15] A. Tolmach, C. Chhak, and S. Anderson, "Defining and preserving more c behaviors: Verified compilation using a concrete memory model," in *15th International Conference on Interactive Theorem Proving (ITP 2024)*, 2024, pp. 36–1.

[16] Q. Stiévenart, D. W. Binkley, and C. De Roover, "Static stack-preserving intra-procedural slicing of webassembly binaries," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2031–2042.

[17] Y. Zhang, M. Liu, H. Wang, Y. Ma, G. Huang, and X. Liu, "Research on webassembly runtimes: A survey," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 8, pp. 1–47, 2025.

[18] K. Brandl, S. Erdweg, S. Keidel, and N. Hansen, "Modular abstract definitional interpreters for webassembly," in *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, 2023, pp. 5–1.

[19] E. Massidda, L. Pisu, D. Maiorca, G. Giacinto *et al.*, "Bringing binary exploitation at port 80: Understanding c vulnerabilities in webassembly," in *International Conference on Security and Cryptography, SECRYPT*, 2024, pp. 552–559.

[20] X. Lu *et al.*, "Detecting webassembly runtime bugs with grammar-guided program mutation," *IEEE Transactions on Reliability*, 2025.

[21] Y. Zhang *et al.*, "Characterizing and detecting webassembly runtime bugs," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 2, pp. 1–29, 2023.

[22] Q. Stiévenart, C. De Roover, and M. Ghafari, "The security risk of lacking compiler protection in webassembly," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2021, pp. 132–139.

[23] L. Silva, J. Metrôlho, and F. Ribeiro, "Efficient data exchange between webassembly modules," *Future Internet*, vol. 16, no. 9, p. 341, 2024.

[24] S. Song, C. Shin, and D. Kwon, "Wasdom: An efficient write protection for wasm jited code with arm domain," *IEEE Access*, vol. 13, pp. 26 260–26 272, 2025.

[25] X. Zhang, J. Wang, X. Du, and S. Liu, "Wasmcfuzz: Structure-aware fuzzing for wasm compilers," in *Proceedings of the 2024 ACM/IEEE 4th International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS)*, 2024, pp. 1–5.

[26] B. Spies and M. Mock, "An evaluation of webassembly in non-web environments," in *2021 XLVII Latin American Computing Conference (CLEI)*. IEEE, 2021, pp. 1–10.

[27] E. Stanley and E. Eide, "Finding bugs in webassembly interface type binding generators," in *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2025, pp. 204–216.

[28] E. Johnson *et al.*, "Wave: a verifiably secure webassembly sandboxing runtime," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2940–2955.

[29] B. Wen, J. Gu, H. Han, P. Yu, and Y. Liu, "Instruction vulnerability prediction for webassembly with semantic enhanced code property graph," in *ACM on Web Conference 2025*, 2025, pp. 4134–4145.

[30] J. Park and E.-S. Cho, "Webmo: Webassembly memory bitwise operation obfuscation," in *2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2024, pp. 1436–1439.

[31] X. Wu, A. Romano, L. Huang, Q. Yan, C. Fu, and W. Wang, "Broken promise: Differential analysis of functional discrepancies between webassembly and native binaries," in *Proceedings of the ACM Web Conference 2026*, 2026, pp. 5144–5154.

[32] A. Hilbig, D. Lehmann, and M. Pradel, "An empirical study of real-world webassembly binaries: Security, languages, use cases," in *Proceedings of the web conference 2021*, 2021, pp. 2696–2708.

[33] N. He *et al.*, "Seewasm: An efficient and fully-functional symbolic execution engine for webassembly binaries," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1816–1820.

[34] Ningyu *et al.*, "Eunomia: Enabling user-specified fine-grained search in symbolically executing webassembly binaries," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 385–397.

[35] S. Cao, N. He, Y. Guo, and H. Wang, "Wasmixer: Binary obfuscation for webassembly," in *European Symposium on Research in Computer Security*. Springer, 2024, pp. 88–109.

[36] S. Bhansali, A. Aris, A. Acar, H. Oz, and A. S. Uluagac, "A first look at code obfuscation for webassembly," in *Proceedings of the 15th ACM conference on security and privacy in wireless and mobile networks*, 2022, pp. 140–145.

[37] S. Cao, N. He, Y. Guo, and H. Wang, "Brewasm: A general static binary rewriting framework for webassembly," in *Static Analysis*, M. V. Hermenegildo and J. F. Morales, Eds. Cham: Springer Nature Switzerland, 2023, pp. 139–163.

[38] G. Choi and S. Jeon, "Stack-based static webassembly binary slicing and mutation for generating valid sub-binaries," *Scientific Reports*, 2026.

[39] S. Narayan *et al.*, "Segue & colorguard: Optimizing sfi performance and scalability on modern architectures," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2025, pp. 987–1002.

[40] M. Fink, D. Stavrakakis, D. Sprokholt, S. Chakraborty, J.-E. Ekberg, and P. Bhatotia, "Cage: Hardware-accelerated safe webassembly," in *Proceedings of the 23rd ACM/IEEE international symposium on code generation and optimization*, 2025, pp. 538–552.

[41] S. Narayan *et al.*, "Hardware-assisted fault isolation: Going beyond the limits of software-based sandboxing," *IEEE Micro*, vol. 44, no. 4, pp. 70–79, 2024.

[42] D. G. Battina, "Resilient web code execution: E-graph driven obfuscation for microarchitectural attack mitigation," in *2026 5th International Conference on Sentiment Analysis and Deep Learning (ICSADL)*. IEEE, 2026, pp. 1238–1244.

[43] T. Rokicki, C. Maurice, M. Botvinnik, and Y. Oren, "Port contention goes portable: Port contention side channels in web browsers," in *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, 2022, pp. 1182–1194.

[44] Q. Michaud, Y. Pipereau, O. Levillain, and D. Ayed, "Robust stack smashing protection for webassembly," in *2024 IEEE Future Networks World Forum (FNWF)*. IEEE, 2024, pp. 861–866.

[45] I. Bastys, M. Algehed, A. Sjösten, and A. Sabelfeld, "Secwasm: Information flow control for webassembly," in *International Static Analysis Symposium*. Springer, 2022, pp. 74–103.

[46] D. Genkin, L. Pachmanov, E. Tromer, and Y. Yarom, "Drive-by key-extraction cache attacks from portable code," in *International Conference on Applied Cryptography and Network Security*. Springer, 2018, pp. 83–102.

[47] M. E. Mazaheri, S. S. Bayat, and A. F. Taheri, "A study of timing side-channel attacks and countermeasures on javascript and webassembly," *The ISC International Journal of Information Security*, 2022.

[48] S. Narayan *et al.*, "Swivel: Hardening webassembly against spectre," in *30th USENIX Security Symposium*, 2021, pp. 1433–1450.

[49] I. Buljić, E. Kadušić, N. Živić, and E. Marevac, "Evaluating legacy and modern cryptography on the web: Rsa, hybrid aes and ed25519 in wasm and javascript," *environment*, vol. 5, p. 6, 2025.

[50] M. E. Mazaheri, F. Taheri, and S. B. Sarmadi, "Lurking eyes: A method to detect side-channel attacks on javascript and webassembly," in *2020 17th international isc conference on information security and cryptology (iscisc)*. IEEE, 2020, pp. 1–6.

[51] M. Musch, C. Wressnegger, M. Johns, and K. Rieck, "New kid on the web: A study on the prevalence of webassembly in the wild," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2019, pp. 23–42.

[52] W. Bian, W. Meng, and Y. Wang, "Poster: Detecting webassembly-based cryptocurrency mining," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2685–2687.

[53] Z. Yu, D. Zhan, L. Ye, H. Yu, H. Zhang, and Z. Tian, "Exploring and exploiting the resource isolation attack surface of webassembly containers," in *34th USENIX Security Symposium*, 2025, pp. 1111–1128.

[54] H. Oz, A. Aris, A. Acar, G. S. Tuncay, L. Babun, and S. Uluagac, "Røb: ransomware over modern web browsers," in *Proceedings of the 32nd USENIX Conference on Security Symposium*, 2023.

[55] J. Cabrera-Arteaga, M. Monperrus, T. Toady, and B. Baudry, "Webassembly diversification for malware evasion," *Computers & Security*, vol. 131, p. 103296, 2023.

[56] S. Wang *et al.*, "Leveraging webassembly for numerical javascript code virtualization," *IEEE Access*, vol. 7, pp. 182 711–182 724, 2019.

[57] M. H. Khan Abbasi, S. Ullah, T. Ahmad, and A. Buriro, "A real-time hybrid approach to combat in-browser cryptojacking malware," *Applied Sciences*, vol. 13, no. 4, 2023.

[58] W. Bian, W. Meng, and M. Zhang, "Minethrottle: Defending against wasm in-browser cryptojacking," in *Proceedings of The Web Conference 2020*, 2020, pp. 3112–3118.

[59] L. Huang, J. He, C. Wang, and W. Wang, "Wascr: A webassembly instruction-timing side channel repairer," in *Proceedings of the ACM on Web Conference 2025*, 2025, pp. 4562–4571.

[60] Q. Stiévenart, C. De Roover, and M. Ghafari, "Security risks of porting c programs to webassembly," in *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, 2022, pp. 1713–1722.

[61] N. He, S. Cao, H. Wang, Y. Guo, and X. Luo, "The promise and pitfalls of webassembly: Perspectives from the industry," in *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, 2025, pp. 480–490.

[62] A. Ramesh *et al.*, "Silverline: Lightweight virtualization and orchestration of distributed systems," in *2025 IEEE 31st Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2025, pp. 375–388.

[63] J. Ménétrey, M. Pasin, P. Felber, and V. Schiavoni, "Twine: An embedded trusted runtime for webassembly," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 205–216.

[64] H. Lei, D. Li, and Y. Guo, "Furina: A light-weight webassembly runtime for ics," in *Proceedings of the 2025 Workshop on Re-design Industrial Control Systems with Security*, 2025, pp. 1–6.

[65] C. Chen *et al.*, "Wasif: In-depth detection of vulnerabilities in wasm smart contracts via information flows and function invocation sequences," *Blockchain: Research and Applications*, p. 100287, 2025.

[66] Y. Wang, Z. Zhou, Z. Ren, D. Liu, and H. Jiang, "A comprehensive study of webassembly runtime bugs," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2023, pp. 355–366.

[67] Y. Xia, P. He, X. Zhang, P. Liu, S. Ji, and W. Wang, "Static semantics reconstruction for enhancing javascript-webassembly multilingual malware detection," in *European Symposium on Research in Computer Security*. Springer, 2023, pp. 255–276.

[68] M. Nieke, L. Almstedt, and R. Kapitza, "Edgedancer: Secure mobile webassembly services on the edge," in *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*, 2021, pp. 13–18.

[69] Y. Kang, J. Zeng, Q. Xia, Z. Wu, and Y. Li, "The impact of cryptographic algorithm performance on smart contracts in webassembly," in *International Conference on Blockchain and Trustworthy Systems*. Springer, 2025, pp. 93–104.

[70] L. Coppolino, S. D'Antonio, G. Mazzeo, R. Nardone, L. Romano, and M. Schmitt, "Wasmbox: A lightweight wasm-based runtime for trustworthy multi-tenant embedded systems," *IEEE Transactions on Emerging Topics in Computing*, 2024.

[71] R. Liu, L. Garcia, and M. Srivastava, "Aerogel: Lightweight access control framework for webassembly-based bare-metal iot devices," in *2021 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2021, pp. 94–105.

[72] B. Li, H. Fan, Y. Gao, and W. Dong, "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*, 2022, p. 261–272.

[73] M. Seidler, A. Krause, and P. Ulbrich, "Wasm-io: Enabling low-level device interaction in webassembly for industry automation," *ACM Transactions on Embedded Computing Systems*, vol. 24, no. 5s, pp. 1–26, 2025.

[74] T. Orlando, M. Arena, L. D'Agati, G. Merlino, and F. Longo, "Secure and flexible webassembly deployment infrastructure for automotive cyber-physical systems," in *2025 IEEE 31st International Symposium on On-Line Testing and Robust System Design (IOLTS)*. IEEE, 2025, pp. 1–4.

[75] S. Wallentowitz, B. Kersting, and D. M. Dumitriu, "Potential of webassembly for embedded systems," in *2022 11th Mediterranean Conference on Embedded Computing (MECO)*. IEEE, 2022, pp. 1–4.

[76] C. Shin, G. Han, S. Song, J. Park, Y.-H. Choi, and D. Kwon, "Ward: Efficient memory protection for webassembly on tiny embedded systems," *IEEE Access*, vol. 14, pp. 5925–5939, 2026.

[77] N. He *et al.*, "Eosafe: security analysis of eosio smart contracts," in *30th USENIX security symposium*, 2021, pp. 1271–1288.

[78] S. Yang, H. Li, and Z. Zheng, "Wali: Control-flow-based analysis of wasm smart contracts," in *International Conference on Blockchain and Trustworthy Systems*. Springer, 2022, pp. 322–335.

[79] J. Zhou and T. Chen, "Wasmod: Detecting vulnerabilities in wasm smart contracts," *IET Blockchain*, vol. 3, no. 4, pp. 172–181, 2023.

[80] C. Chen, X. Wang, P. Ren, B. Yu, and J. Liu, "Wasdam: Effectively detecting vulnerabilities in wasm smart contracts based on the data access model," in *2025 25th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2025, pp. 1–6.

[81] A. Ramesh, T. Huang, B. L. Titzer, and A. Rowe, "Empowering webassembly with thin kernel interfaces," in *Proceedings of the Twentieth European Conference on Computer Systems*, 2025, p. 1–20.

[82] L. Döllerer and A. Engelke, "Performant bounds checking for 64-bit webassembly," in *Proceedings of the 16th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, 2024, pp. 23–31.

[83] W. Wang and J. Li, "A runtime design of webassembly container for embedded real-time operating systems," in *International Conference on Applied Computational Intelligence, Informatics and Big Data*. Springer, 2024, pp. 156–164.

[84] F. N. Naseem, A. Aris, L. Babun, E. Tekiner, and A. S. Uluagac, "Minos: A lightweight real-time cryptojacking detection system." in *NDSS*, 2021, pp. 1–15.

[85] D. Peng, C. Liu, T. Palit, P. Fonseca, A. Vahldiek-Oberwagner, and M. Vij, "μswitch: Fast kernel context isolation with implicit context switches," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2956–2973.

[86] J. Ménétrey, M. Pasin, P. Felber, and V. Schiavoni, "Watz: A trusted webassembly runtime environment with remote attestation for trustzone," in *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2022, pp. 1177–1189.

[87] O. Draissi and L. Davi, "Bento: Fine-grained memory isolation for cots webassembly binaries," in *Proceedings of the ACM Web Conference 2026*, 2026, pp. 3090–3101.

[88] X. Rao *et al.*, "Iris-wasm: Robust and modular verification of webassembly programs," *Proceedings of the ACM on Programming Languages*, vol. 7, pp. 1096–1120, 2023.

[89] D. Youn, W. Shin, and S. Ryu, "West: Specification-based test generation for webassembly," in *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2025, pp. 1403–1414.

[90] R. M. Tsoupidi, M. Balliu, and B. Baudry, "Vivienne: relational verification of cryptographic implementations in webassembly," in *Secure Development Conference (SecDev)*. IEEE, 2021, pp. 94–102.

[91] B. L. Titzer, E. Gilbert, B. W. J. Teo, Y. Anand, K. Takayama, and H. Miller, "Flexible non-intrusive dynamic instrumentation for webassembly," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2024, pp. 398–415.

[92] A. Stuker, A. Munsters, A. L. Scull Pupo, L. Christophe, and E. Gonzalez Boix, "Jasmaint: Portable multi-language taint analysis for the web," in *Proceedings of the ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, 2025, pp. 79–92.

[93] F. Breitfelder, T. Roth, L. Baumgärtner, and M. Mezini, "Wasma: A static webassembly analysis framework for everyone," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2023, pp. 753–757.

[94] E. Massey and P. Olivier, "Wasp: Stack protection for webassembly," *Journal of Systems Architecture*, vol. 172, p. 103666, 2026.

[95] M. Scherer, J. F. Blaabjerg, A. Sjösten, M. Solitro, and M. Maffei, "Wappler: Sound reachability analysis for webassembly," in *37th Computer Security Foundations Symposium (CSF)*, 2024, pp. 249–264.

[96] D. Lehmann and M. Pradel, "Wasabi: A framework for dynamically analyzing webassembly," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 1045–1058.

[97] O. Blaak and T. Van Cutsem, "Towards least-privilege webassembly applications: Transparent interposition for webassembly components," in *Companion Proceedings of the 10th International Conference on the Art, Science, and Engineering of Programming*, 2026, pp. 14–17.

[98] A. E. Michael *et al.*, "Mswasm: Soundly enforcing memory-safe execution of unsafe code," *Proceedings of the ACM on Programming Languages*, vol. 7, no. POPL, pp. 425–454, 2023.

[99] Z. Zhang, W. Zheng, B. Hua, Q. Fan, and Z. Pan, "Vmcanary: Effective memory protection for webassembly via virtual machine-assisted approach," in *23rd International Conference on Software Quality, Reliability, and Security (QRS)*. IEEE, 2023, pp. 662–671.

[100] Z. Jiang and B. Hua, "Washadow: Effectively protecting webassembly memory through virtual machine-aware shadow memory," in *23rd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 2024, pp. 222–233.

[101] S. Wang and A. Prakash, "Webassembly memory tagging," in *Proceedings of the Cloud Computing Security Workshop*, 2025, pp. 63–79.

[102] H. Huang and J. Zhao, "Multi-modal learning for webassembly reverse engineering," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 453–465.

[103] L. Zhang *et al.*, "Waltzz: Webassembly runtime fuzzing with stack-invariant transformation," in *34th USENIX Security Symposium*, 2025, pp. 6159–6178.

[104] T. Heinrich, N. C. Will, R. R. Obelheiro, and C. A. Maziero, "A categorical data approach for anomaly detection in webassembly applications." in *ICISSP*, 2024, pp. 275–284.

[105] A. Vogel, T. Glörfeld, A. Szekely-Schenker, T. Trenner, R. Ermler, and R. Kapitza, "Wasmeye: Language-and platform-independent anomaly detection for webassembly," in *Proceedings of the 26th International Middleware Conference*, 2025, pp. 314–325.

[106] T. Kim *et al.*, "When does wasm malware detection fail? a systematic analysis of their robustness to evasion," in *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2025, pp. 2957–2969.

[107] Q. Stiévenart and C. De Roover, "Compositional information flow analysis for webassembly programs," in *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2020, pp. 13–24.

[108] J. Zhuang, H. Jiang, and B. Hua, "Wasmchecker: Effectively detecting webassembly bugs via static program analysis," in *2024 8th International Conference on Electrical, Mechanical and Computer Engineering (ICEMCE)*. IEEE, 2024, pp. 1669–1678.

[109] T. Brito, P. Lopes, N. Santos, and J. F. Santos, "Wasmati: An efficient static vulnerability scanner for webassembly," *Computers & Security*, vol. 118, p. 102745, 2022.

[110] Z. Jiang and B. Hua, "Powerpoly: Analyzing multilingual programs with the aid of webassembly," in *International Conference on Information and Communications Security*. Springer, 2025, pp. 77–96.

[111] M. Scherer, J. F. Blaabjerg, A. Sjösten, and M. Maffei, "Wanilla: Sound noninterference analysis for webassembly," in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, 2025, pp. 126–140.

[112] W. Zheng and B. Hua, "Wasmdypa: Effectively detecting webassembly bugs via dynamic program analysis," in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2024, pp. 296–307.

[113] A. Munsters, A. L. S. Pupo, J. Bauwens, and E. G. Boix, "Oron: Towards a dynamic analysis instrumentation platform for assemblyscript," in *Companion Proceedings of the 5th International Conference on the Art, Science, and Engineering of Programming*, 2021, pp. 6–13.

[114] X. Wu, J. He, L. Huang, C. Fu, and W. Wang, "Wbsan: Webassembly bug detection for sanitization and binary-only fuzzing," in *Proceedings of the ACM on Web Conference 2025*, 2025, pp. 3311–3322.

[115] S. Song, S. Park, and D. Kwon, "Metasafer: A technique to detect heap metadata corruption in webassembly," *IEEE Access*, vol. 11, pp. 124 887–124 898, 2023.

[116] B. Jiang, Z. Li, Y. Huang, Z. Zhang, and W. Chan, "Wasmfuzzer: A fuzzer for webassembly virtual machines," in *34th International Conference on Software Engineering and Knowledge Engineering, SEKE 2022*, 2022, pp. 537–542.

[117] J. Wang, Z. Guo, X. Ying, P. Qian, and Y. Chen, "S w fuzz: Structure-sensitive webassembly fuzzing," in *2024 31st Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2024, pp. 343–352.

[118] J. Han, Z. Zhang, Y. Du, W. Wang, and X. Chen, "Esfuzzer: An efficient way to fuzz webassembly interpreter," *Electronics*, vol. 13, no. 8, p. 1498, 2024.

[119] K. Haßler and D. Maier, "Wafl: Binary-only webassembly fuzzing with fast snapshots," in *Reversing and Offensive-oriented Trends Symposium*, 2021, pp. 23–30.

[120] W. Zhao, R. Zeng, and Y. Zhou, "Wapplique: Testing webassembly runtime via execution context-aware bytecode mutation," in *Proceedings of the 33rd ACM SIGSOFT international symposium on software testing and analysis*, 2024, pp. 1035–1047.

[121] S. Zhou, J. Wang, H. Ye, H. Zhou, C. Le Goues, and X. Luo, "Lwdiff: an llm-assisted differential testing framework for webassembly runtimes," in *2025 IEEE/ACM 47th International Conference on Software Engineering, ICSE*, 2025, pp. 153–164.

[122] L. Andrès, F. Marques, A. Carcano, P. Chambart, J. F. Santos, and J.-C. Filliâtre, "Owi: Performant parallel symbolic execution made easy, an application to webassembly," *The Art, Science, and Engineering of Programming, Vol. 9, Issue 1, Article 3*, 2025.

[123] B. Jiang, Y. Chen, D. Wang, I. Ashraf, and W. K. Chan, "Wana: Symbolic execution of wasm bytecode for extensible smart contract vulnerability detection," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, 2021, pp. 926–937.

[124] L. Deng *et al.*, "Formal verification platform as a service: Webassembly vulnerability detection application." *Computer Systems Science & Engineering*, vol. 45, no. 2, 2023.

[125] F. Marques, J. Fragoso Santos, N. Santos, and P. Adão, "Concolic execution for webassembly," in *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, 2022, pp. 11–1.

[126] J. Yang and Z. Fan, "Dynamic-aware pruning-quantization compression for browser-based anomaly detection via webassembly optimization," *International Journal of Pattern Recognition and Artificial Intelligence*, 2026.

[127] Z. Jiang and B. Hua, "Wasmsepa: Effectively protecting webassembly through privilege separation," in *25th International Conference on Software Quality, Reliability and Security*. IEEE, 2025, pp. 703–714.

[128] P. M. André, Q. Stiévenart, and M. Ghafari, "Developers struggle with authentication in blazor webassembly," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2022, pp. 389–393.

[129] X. Yu *et al.*, "Wasc: Hardening webassembly sandboxes via system interface decoupling," *ACM Transactions on Architecture and Code Optimization*, vol. 23, no. 1, pp. 1–23, 2026.

[130] J. Protzenko, B. Beurdouche, D. Merigoux, and K. Bhargavan, "Formally verified cryptographic web applications in webassembly," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1256–1274.

[131] J. Cabrera-Arteaga, N. Fitzgerald, M. Monperrus, and B. Baudry, "Wasm-mutate: Fast and effective binary diversification for webassembly," *Computers & Security*, vol. 139, p. 103731, 2024.

[132] R. K. Konoth *et al.*, "Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1714–1730.

[133] A. Romano, Y. Zheng, and W. Wang, "Minerray: Semantics-aware analysis for ever-evolving cryptojacking detection," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 1129–1140.

[134] J. D. P. Rodriguez and J. Posegga, "Rapid: Resource and api-based detection against in-browser miners," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 313–326.

[135] G. Hong *et al.*, "How you get shot in the back: A systematical study about cryptojacking in the real world," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1701–1713.

[136] Y. Sun, H. Chen, Z. Fu, W. Lv, Z. Liu, and H. Liu, "Wasmguard: Enhancing web security through robust raw-binary detection of webassembly malware," in *Proceedings of the ACM on Web Conference 2025*, 2025, pp. 1942–1950.

[137] H. Oz, G. S. Tuncay, A. Aris, A. Acar, L. Babun, and S. Uluagac, "Ransomware over modern web browsers: A novel strain and a new defense mechanism," *ACM Transactions on the Web*, vol. 19, no. 1, pp. 1–28, 2025.

[138] N. Loose, F. Mächtle, C. Pott, V. Bezsmertnyi, and T. Eisenbarth, "Madvex: instrumentation-based adversarial attacks on machine learning malware detection," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2023, pp. 69–88.

[139] Y. Wang, P. Yu, H. Han, J. Gu, F. Xu, and X. Huang, "Wamo: Toward secure browser inference via web model obfuscation in webassembly," in *Proceedings of the ACM Web Conference*, 2026, pp. 2812–2823.

[140] M. Legoupil, J. Rousseau, A. L. Georges, J. Pichon-Pharabod, and L. Birkedal, "Iris-mswasm: Elucidating and mechanising the security invariants of memory-safe webassembly," *Proceedings of the ACM on Programming Languages*, vol. 8, pp. 304–332, 2024.

[141] C. Watt, M. Trela, P. Lammich, and F. Märkl, "Wasmref-isabelle: A verified monadic interpreter and industrial fuzzing oracle for webassembly," *Proceedings of the ACM on Programming Languages*, vol. 7, pp. 100–123, 2023.

[142] M. McLoughlin, A. Sheng, C. Fallin, B. Parno, F. Brown, and A. Van-Hattum, "Scaling instruction-selection verification against authoritative isa semantics," *Proceedings of the ACM on Programming Languages*, vol. 9, pp. 4064–4090, 2025.

[143] S. Gao, G. Li, and H. Fu, "Zkwasm: A zksnark wasm emulator," *IEEE Transactions on Services Computing*, vol. 17, pp. 4508–4521, 2024.

[144] C. Komiya, N. Yanai, K. Yamashita, and S. Okamura, "Jabberwock: A tool for webassembly dataset generation and its application to malicious website detection," *Journal of Information Processing*, vol. 32, pp. 298–307, 2024.

[145] T. Heinrich, N. C. Will, R. Obelheiro, and C. Maziero, "A method for detecting anomalies in webassembly using categorical data," *SN Computer Science*, vol. 7, no. 1, p. 68, 2026.

[146] W. Zheng and B. Hua, "A comprehensive study of bugs in embedded webassembly virtual machines," in *2023 3rd International Conference on Computer Science, Electronic Information Engineering and Intelligent Control Technology (CEI)*. IEEE, 2023, pp. 901–907.