ACTA TECHNICA CORVINIENSIS – Bulletin of Engineering | e–ISSN: 2067 – 3809
Tome XVII [2024] | Fascicule 4 [October – December]

[1,2.]Mircea ȚĂLU

A REVIEW OF VULNERABILITY DISCOVERY IN WEBASSEMBLY BINARIES: INSIGHTS FROM STATIC, DYNAMIC, AND HYBRID ANALYSIS

[1.]The Technical University of Cluj–Napoca, Faculty of Automation and Computer Science, Cluj–Napoca, ROMANIA
[2.]SC ACCESA IT SYSTEMS SRL, Cluj–Napoca, ROMANIA

Abstract: WebAssembly (Wasm) has rapidly gained adoption as a powerful, low–level assembly language designed to enable near–native performance in web browsers, alongside secure execution in other environments such as IoT and edge computing. Despite its secure–by–design nature, WebAssembly is vulnerable to several types of attacks, including memory safety issues, side–channel attacks, and code injection. These vulnerabilities pose significant threats to applications relying on WebAssembly, particularly in performance–sensitive and security–critical environments. This study examines existing research on static, dynamic, and hybrid analysis approaches for discovering vulnerabilities in WebAssembly binaries. By categorizing state–of–the–art methods, are highlighted both their contributions and limitations. This study aims to foster a unified framework for vulnerability discovery that aligns with the needs of both WebAssembly applications and the broader cybersecurity landscape.

Keywords: WebAssembly binaries, Vulnerability discovery, Static analysis, Dynamic analysis, Hybrid analysis

INTRODUCTION

A coalition of companies, including Mozilla, Microsoft, Apple, and Google, created WebAssembly (or "Wasm" for short) in 2015 to address the shortcomings of traditional web technologies. WebAssembly has emerged as a revolutionary technology, designed to serve as a portable compilation target for high–level languages, primarily used to improve performance in web applications and expand into other domains like the Internet of Things (IoT), blockchain, and edge computing [1–4]. It has also been recognized as an official standard by the World Wide Web Consortium (W3C).

Since its inception, Wasm has offered a portable, platform–independent, and highly efficient format that translates high–level languages into a binary instruction set that can be executed within a sandboxed environment. Currently, several programming languages, including C, C++, Rust, AssemblyScript, C#, F#, Dart, Go, Kotlin, Swift, D, Pascal, Zig, and Grain, are supported by WebAssembly, enabling a wide range of applications [5–7].

A WebAssembly binary is structured as a module, which encompasses a set of Wasm functions, the declarations of their shared global variables, and details about the linear memory where these functions and variables are stored. The execution model follows a stack–based machine, with Wasm instructions manipulating the stack by pushing and popping values. A Wasm module is run by an embedder, such as the host JavaScript engine, which manages the loading of modules, resolves imports and exports between them, and coordinates I/O operations, timers, and error handling.

WebAssembly functions as a technology that complements JavaScript, although it does not compile from JavaScript itself. Instead, it allows seamless integration, where JavaScript facilitates interaction between the Document Object Model (DOM) and WebAssembly modules within browsers [5].

The main advantage of WebAssembly is its capacity to enhance application performance while maintaining security. Despite its design for secure execution, WebAssembly is not immune to various security vulnerabilities that can be exploited by malicious actors. These vulnerabilities arise from several inherent characteristics of the WebAssembly architecture, its interaction with host environments, and its operational context within web browsers [8–10].

The key security challenges that have been identified include memory safety vulnerabilities, side–channel attacks, susceptibility to speculative execution vulnerabilities, and the complexities associated with its interaction with JavaScript [11,12]. These vulnerabilities pose serious challenges, underscoring the importance of developing robust vulnerability discovery methods for securing WebAssembly binaries. It is known that the security flaws present in memory–unsafe languages such as C and C++ can propagate into WebAssembly binaries, posing significant vulnerabilities [6].

This research explores current studies on methods for identifying vulnerabilities in WebAssembly binaries through static, dynamic, and hybrid analysis techniques.

RESEARCH METHODOLOGY

A survey was carried out to evaluate the increasing research focus on Vulnerability Discovery in WebAssembly Binaries, with insights drawn from static, dynamic, and hybrid analysis techniques. Due to the practical importance of this area, the review covers the period from 2018 to 2024. This research reviews various journal articles discussing key concepts and real–world applications.

Research on static analysis methods

Static analysis refers to the examination of WebAssembly code without executing it, relying on code inspection to identify potential vulnerabilities. This method typically involves analyzing the program's structure, control flow, and data flow to detect patterns that could lead to security issues. For example, static analysis tools can focus on identifying risks such as memory access violations, type errors, or control flow integrity (CFI) problems. By analyzing code at compile–time, static analysis can provide early detection of vulnerabilities, although it may struggle with issues that only emerge during runtime. This section discusses three prominent static analysis tools – Wassail, Wasmati, and WASP1 – focusing on their methodologies, strengths, and limitations. The development of these tools offer a unique approach to vulnerability detection, from Wassail's focus on information flow to Wasmati's use of code property graphs and WASP1's execution model. While these tools have their respective limitations, they represent critical advancements in securing WebAssembly applications.

- Wassail stands out as the pioneering static analysis tool developed exclusively to identify vulnerabilities in WebAssembly binaries [13,14]. It utilizes a summary–based, compositional analysis technique that focuses on tracking information flow throughout the program.

This approach generates a summary for each WebAssembly function, detailing how information is transmitted within the function. These individual summaries are integrated when analyzing function calls, offering a comprehensive view of the information flow across the whole binary. The analysis itself is performed on a Control Flow Graph (CFG), where data flow analysis identifies potential vulnerabilities related to the movement of sensitive or insecure information within the program.

Stiévenart et al. [15] applied a program slicing (a broad technique that, based on specific criteria, trims a program down to its smallest form while still fulfilling those requirements) and their experiments demonstrated that program slicing reduced binary sizes to 52% of their initial size. They concluded that this technique could also be beneficially applied to loop analysis. On the other hand, Wassail's summary–based approach is particularly noteworthy for its scalability [16]. Compositional analysis has been shown to scale effectively in other domains, as it reduces the complexity of analyzing large binaries by focusing on individual functions before aggregating their effects at a higher level. However, the scalability of Wassail itself has not been thoroughly evaluated, leaving questions about its effectiveness in analyzing larger, more complex WebAssembly binaries [16]. Nonetheless, Wassail's ability to detect information flow vulnerabilities makes it a valuable tool in identifying a wide range of security issues, including unauthorized access to memory or improper handling of sensitive data. While Wassail's focus on information flow is crucial for preventing data leakage and breaches, its reliance on function summaries may lead to limitations in detecting vulnerabilities that arise from complex interactions between functions or external inputs. Therefore, Wassail might be more suitable for smaller binaries or binaries with well–defined and isolated function boundaries. Future work could explore expanding Wassail's capabilities to cover inter–function vulnerabilities and improve its evaluation in larger–scale applications.

- Wasmati builds upon the foundation laid by Wassail by introducing a more sophisticated approach to vulnerability detection in WebAssembly binaries through the construction of a Code Property Graph (CPG) [17].

The CPG is a multi–layered data structure that combines information about the program's execution order, control flow, data dependencies, and other relevant properties. By searching for specific patterns in CPG sub–graphs, Wasmati can detect vulnerabilities such as buffer overflows, use–after–free errors, and other memory corruption issues that are common in low–level code like WebAssembly.

One of the primary challenges Wasmati addresses is the rapid growth of the CPG due to the inability to statically determine the targets of indirect calls. Indirect calls, which allow a function to call another function based on runtime information, complicate static analysis because their targets are not known until the program is executed. Wasmati mitigates this issue by introducing optimizations during the CPG generation process. These optimizations include adding annotations, caching intermediate results, and using more efficient graph traversal algorithms, reducing the complexity of analyzing indirect calls. The scientists reported that this tool could construct a CPG for a WebAssembly binary in an average of 58 seconds, demonstrating its efficiency [17]. Brito et al. [18] by leveraging optimized techniques for CPG generation and four distinct query back–ends, it efficiently identifies various vulnerability types in WebAssembly binaries code. Extensive testing across diverse datasets demonstrated its scalability and reliability, making it a valuable solution for analyzing complex, real–world WebAssembly applications [18]. Additionally, Wasmati was found to have a low false positive rate, meaning it could accurately identify vulnerabilities without flagging benign code as malicious. This is a significant improvement over some other static analysis tools, which often suffer from high false positive rates, leading to wasted time and effort in manual verification. However, like Wassail, Wasmati has its limitations. The reliance on CPG construction can result in scalability issues as the size of the binary increases. Moreover, while Wasmati can efficiently detect certain types of vulnerabilities, it may struggle with more complex attack patterns that require a deeper understanding of runtime behaviors, such as race conditions or time–of–check to time–of–use (TOCTOU) vulnerabilities. Nevertheless, Wasmati's ability to detect structural vulnerabilities in WebAssembly binaries makes it an important tool for static analysis in the WebAssembly ecosystem.

- WASP1 introduces a hybrid approach to static analysis by combining symbolic execution with concolic execution [19]. Concolic execution is a technique that merges concrete and symbolic execution to explore all feasible execution paths within a program, maximizing code coverage and increasing the likelihood of discovering hidden vulnerabilities. Symbolic execution allows the tool to generate concrete inputs for different execution paths, ensuring that all paths are explored and tested for potential issues. WASP1 is specifically designed to uncover vulnerabilities such as integer overflows, buffer overflows, and memory access violations in WebAssembly modules. Unlike purely static analysis techniques, concolic execution enables WASP1 to examine the effects of inputs on the execution of the binary, allowing for a more comprehensive analysis. To demonstrate its effectiveness, the authors of WASP1 developed a framework called WASP–C, which allows for the testing of C programs by converting them into WebAssembly binaries and then analyzing them with WASP1. The results of the WASP–C framework showed that WASP1 could effectively uncover a range of bugs and vulnerabilities in the tested programs. However, one limitation of WASP1 is that it requires access to the high–level source code of the program to perform its analysis. This constraint means that WASP1 is primarily useful for open–source programs or situations where the source code is available, limiting its applicability in cases where only the compiled WebAssembly binary is available. WASP–C showed competitive results against other tools in symbolic execution tasks, focusing on byte–level granularity for Wasm code. Despite this limitation, WASP1's use of symbolic execution provides a powerful tool for vulnerability discovery. Its ability to maximize code coverage ensures that even deeply hidden vulnerabilities can be identified, making it particularly useful for identifying security issues that arise from edge cases or rare execution paths.

Static analysis techniques have the following advantages: early detection, comprehensive coverage, and scalability. Their limitations include false positives/negatives, lack of contextual information, and complexity in analysis.

- **Research on dynamic analysis methods**

In contrast, dynamic analysis involves examining WebAssembly binaries while they are being executed, providing real–time insights into how the program behaves in different scenarios. This approach is particularly useful for identifying vulnerabilities that arise from the interaction of the code with the execution environment, such as buffer overflows, race conditions, or unauthorized memory access. While dynamic analysis offers the advantage of catching runtime–specific issues, it typically requires more

computational resources and may not cover all execution paths of the program.

- Szanto et al. [20] proposed an innovative approach to detecting vulnerabilities in WebAssembly binaries through the implementation of a taint–tracking technique. Taint tracking is a widely–used method in software security to monitor the flow of sensitive data, such as personal information or cryptographic keys, through a program's execution, helping to identify potential vulnerabilities like information leakage or code injection attacks.

What distinguishes Szanto et al.'s [20] approach is its ability to apply taint tracking without requiring modifications to the underlying structure of the WebAssembly binary, thereby preserving the integrity of the original code. To achieve this, they developed a custom virtual machine (VM) that runs natively within JavaScript, which is particularly significant because of the prevalent use of WebAssembly in web applications. By integrating the taint–tracking system into a JavaScript–based VM, Szanto et al. [20] ensured compatibility with existing web technologies, enabling seamless deployment in real–world web environments. The method works by assigning a tainted label to each allocable byte in WebAssembly's memory section and each variable on the stack. This granularity in labeling ensures that every potentially sensitive data point is monitored throughout the execution, thereby improving the precision of taint propagation analysis. A key advantage of this technique is its non–intrusiveness.

Unlike traditional taint–tracking methods that often require binary rewriting or instrumentation, Szanto et al.'s [20] method avoids these invasive alterations, reducing the risk of introducing new vulnerabilities or bugs into the system. This aspect is critical for security–sensitive applications, where even minor modifications to the code can lead to unintended side effects. Additionally, by preserving the original binary structure, their approach ensures better compatibility with existing WebAssembly compilers and execution environments. However, this precision and non–intrusiveness come at a cost in terms of runtime performance.

Szanto et al. [20] reported that the overhead introduced by their taint–tracking technique scales mostly linearly with the execution time of the WebAssembly binary. In practice, this means that the system can incur a performance penalty of up to 100% compared to uninstrumented execution. While such an overhead might be prohibitive for performance–critical applications, it is worth noting that in many security contexts, this trade–off is acceptable. Ensuring the safety and security of sensitive data flows often justifies increased computational costs, especially in web applications that handle personal user information or financial data. The technique proposed by Szanto et al. [20] contributed to the growing body of research aimed at enhancing WebAssembly binaries security, a field that has gained considerable attention due to the increasing adoption of WebAssembly in both client–side and server–side applications.

- TaintAssembly. TaintAssembly [21] introduced a sophisticated taint–tracking mechanism tailored for detecting vulnerabilities in WebAssembly binaries that distinguishes itself through its integration with the V8 JavaScript engine, which is the backbone of widely used platforms such as Google Chrome and Node.js [22].

This strategic modification leverages the existing infrastructure of V8, enabling more efficient and seamless taint tracking within web environments where Wasm is increasingly prevalent. TaintAssembly opted for a more integrated approach by modifying the V8 engine itself to embed taint–tracking functionalities directly into the execution pipeline of WebAssembly binaries. By doing so, it minimized the need for external interventions or the creation of auxiliary VMs, thereby streamlining the taint–tracking process and enhancing performance efficiency. TaintAssembly supports taint tracking for a comprehensive set of data types, including integers (i32, i64) and floating–point numbers (f32, f64), which are fundamental to Wasm's type system. Additionally, it extends taint tracking capabilities to linear memory, which is crucial for monitoring data flows in applications that heavily utilize memory operations. Furthermore, TaintAssembly introduces a probabilistic variant of taint tracking, which employs statistical methods to infer potential taint propagation paths. This probabilistic approach enhances the detection of complex vulnerabilities that may not be easily identifiable through deterministic methods alone.

A notable difference between TaintAssembly and Szanto et al.'s approach lies in the handling of WebAssembly module structures.

TaintAssembly requires modifications to the Wasm module before taint labels can be assigned to variables. This preprocessing step involves analyzing and restructuring the Wasm binary to insert taint labels appropriately, ensuring that all relevant variables are accurately tracked during execution. While this introduces an additional preprocessing phase, it enables more precise taint propagation and reduces the likelihood of false positives in vulnerability detection.

One of the most significant advantages of TaintAssembly is its superior runtime performance. Through its integration with the V8 engine and optimized taint–tracking algorithms, TaintAssembly achieves a runtime overhead of merely 5–12%. This is a stark contrast to the approach proposed by Szanto et al., which incurs an overhead of up to 100%. The reduced overhead makes TaintAssembly more viable for real–world applications, where performance constraints are a critical consideration. The efficiency gains are primarily attributed to the tight integration with V8, which allows for more direct and less resource–intensive taint tracking compared to running a separate VM.

Moreover, by leveraging the V8 engine's optimizations and just–in–time (JIT) compilation capabilities, TaintAssembly can maintain high execution speeds while performing taint analysis. This integration ensures that taint tracking does not become a bottleneck, thereby supporting the deployment of security mechanisms in performance–sensitive environments such as web browsers and server–side applications. However, the requirement to modify the V8 engine introduces certain limitations. Maintaining compatibility with future updates of V8 can be challenging, as engine modifications may need to be reapplied or adjusted with each new release. Additionally, the preprocessing step to modify Wasm modules adds complexity to the deployment pipeline, potentially increasing the effort required to integrate TaintAssembly into existing workflows.

Despite these challenges, TaintAssembly represents a significant advancement in the realm of WebAssembly security. Its ability to provide efficient and accurate taint tracking within widely adopted platforms like Google Chrome and Node.js underscores its practical applicability. By enabling developers to detect and mitigate vulnerabilities such as buffer overflows, code injection, and information leakage in Wasm binaries, TaintAssembly contributes to the robustness and reliability of web applications.

- Wasabi. Wasabi [23] represents a versatile and robust framework designed for the dynamic analysis of WebAssembly binaries. Wasabi addresses this need by employing binary instrumentation techniques that enable comprehensive runtime analysis without necessitating modifications to the original Wasm binaries.

At its core, Wasabi performs binary instrumentation by injecting calls to analysis functions written in JavaScript directly into the WebAssembly binary. This approach leverages the seamless interoperability between WebAssembly and JavaScript within web environments, facilitating the integration of sophisticated analysis capabilities. The instrumentation process involves identifying relevant points within the Wasm binary where analysis functions should be inserted, thereby enabling real–time monitoring and data collection during the execution of the binary.

By embedding these analysis hooks, Wasabi can perform a variety of analyses, including instruction counting, call graph extraction, memory access tracing, and taint analysis. Instruction counting in Wasabi allows developers and security analysts to monitor the execution frequency of specific instructions, providing insights into performance bottlenecks and potential optimization opportunities. Call graph extraction is another critical feature, enabling the construction of a detailed map of function calls within the Wasm binary. This facilitates the identification of complex interdependencies and the detection of anomalous or unauthorized function invocations that may indicate malicious behavior or software defects. Memory access tracing is particularly important for uncovering vulnerabilities related to memory safety, such as buffer overflows and use–after–free errors. By tracking how memory is accessed and manipulated during execution, Wasabi can identify patterns that may lead to security breaches or data corruption.

Taint analysis, one of Wasabi's most powerful features, involves tracking the flow of sensitive or untrusted data through the program. This enables the detection of potential information leaks, injection attacks, and other forms of data misuse by ensuring that tainted data does not reach critical execution points without proper validation. A significant innovation of Wasabi is its

support for selective instruction instrumentation. Instead of indiscriminately instrumenting every instruction in the Wasm binary, Wasabi allows users to specify which instructions are relevant for a particular analysis.

This targeted approach reduces the overhead associated with instrumentation, as only the necessary parts of the binary are instrumented. By focusing on specific instructions, Wasabi can provide more precise and efficient analysis, tailored to the unique requirements of different applications and security contexts. Despite its advantages, the implementation of Wasabi introduces runtime overhead, which varies significantly depending on the application and the specific instructions being analyzed. The authors of Wasabi reported a runtime overhead ranging from 2% to 163% [23].

This variability is attributed to several factors, including the complexity of the instrumentation, the frequency of instrumented instructions, and the nature of the analyzed workload. While a 2% overhead is relatively minimal and acceptable for many applications, a 163% overhead can be prohibitive for performance–critical environments.

Consequently, the selective instrumentation feature of Wasabi plays a crucial role in balancing the trade–off between analysis comprehensiveness and runtime efficiency. Wasabi's ability to perform multiple types of analysis within a single framework enhances its utility for developers and security professionals. By providing a unified platform for instruction counting, call graph extraction, memory access tracing, and taint analysis, Wasabi simplifies the process of conducting thorough security assessments and performance evaluations of WebAssembly binaries. This multi–faceted analysis capability is particularly valuable in complex applications where understanding the interplay between different components is essential for ensuring both performance and security.

- Fuzzm. Fuzzm [24] represents a specialized fuzzer for WebAssembly (Wasm) binaries, leveraging the widely adopted American Fuzzy Lop (AFL) framework to perform fuzz testing on binary–only applications.

Fuzz testing is a critical technique in software security, where random or semi–random inputs are supplied to a program to uncover vulnerabilities such as crashes, memory corruption, or unexpected behavior. Fuzzm extends this principle to WebAssembly, enabling the security analysis of Wasm binaries even in the absence of source code, which is often a challenge in the context of proprietary or third–party applications. The AFL framework, which forms the foundation of Fuzzm, is traditionally used to fuzz applications by compiling them from source code and inserting instrumentation at compile time. This instrumentation tracks path coverage, a critical metric in fuzzing that indicates how thoroughly the fuzzer explores different execution paths within the application.

Path coverage helps guide the fuzzer toward unexplored code paths, improving the chances of discovering bugs or security flaws. However, Fuzzm departs from this source–code–centric approach by operating directly on Wasm binaries. Since Fuzzm does not have access to the source code, it cannot leverage AFL's compile–time instrumentation techniques. Instead, it employs static binary instrumentation to achieve similar functionality. Specifically, Fuzzm inserts code at all branches within the Wasm binary, ensuring that coverage information is collected in a way that is compatible with AFL's feedback–driven fuzzing mechanism. This static binary instrumentation is crucial for enabling effective fuzzing in the absence of source code, allowing Fuzzm to monitor and report the execution paths taken by the WebAssembly binary during fuzz testing.

One of the significant advantages of Fuzzm is its ability to provide detailed coverage information while maintaining a low runtime overhead. Binary instrumentation can often introduce performance penalties due to the insertion of additional code at various points in the program's control flow. However, the authors of Fuzzm demonstrated that the overhead imposed by their static binary instrumentation is minimal, making the tool practical for extensive fuzz testing campaigns without excessively slowing down execution. This low runtime overhead is essential for achieving high throughput in fuzzing, as it allows more inputs to be tested in a shorter period, thereby increasing the likelihood of discovering vulnerabilities.

Furthermore, Fuzzm's design is not tied to a specific WebAssembly runtime environment, enhancing its versatility. This flexibility means that Fuzzm can be used across different Wasm execution environments, including both web–based runtimes (such as those in browsers) and standalone Wasm runtimes (such as Wasmtime or Wasmer). By decoupling the fuzzer from any particular runtime, Fuzzm allows security

researchers to apply it to a wide range of Wasm applications and execution contexts, broadening its applicability. In addition to its core fuzzing capabilities, Fuzzm also incorporates a canary–based protection mechanism to guard against memory corruption vulnerabilities. Memory corruption is a common and serious security issue in low–level programming languages that WebAssembly often interacts with, such as C and C++. Canary–based protection works by placing special "canary" values in memory regions that are vulnerable to overflow, such as function stack frames. If a buffer overflow or similar vulnerability attempts to overwrite the memory, the canary value is altered, signaling the presence of a vulnerability before it can be exploited. This proactive detection mechanism adds an additional layer of defense, enabling Fuzzm not only to identify crashes but also to flag potential memory safety issues before they lead to severe security breaches. Fuzzm's approach to fuzz testing has several implications for the future of WebAssembly security. Given the increasing adoption of WebAssembly in areas such as web development, blockchain smart contracts, and serverless computing, the ability to fuzz test Wasm binaries without access to source code is of paramount importance. Proprietary Wasm binaries deployed in production environments often do not come with readily available source code, making tools like Fuzzm indispensable for ensuring the security of such applications. Moreover, Fuzzm's low overhead and broad runtime compatibility make it an ideal candidate for integration into continuous integration (CI) pipelines, where frequent security testing of binaries is crucial.

- WAFL. WAFL [25] is a cutting–edge binary–only fuzzer designed specifically for WebAssembly (Wasm) binaries, leveraging the robust capabilities of the AFL++ [26]] framework, a community–driven extension of the original American Fuzzy Lop (AFL) fuzzer.

Unlike traditional fuzzers that require access to source code for instrumentation, WAFL operates directly on Wasm binaries, making it particularly valuable for testing proprietary or third–party applications where source code may be unavailable. A key component of WAFL's architecture is its integration with the WAVM [27] runtime, an Ahead–of–Time (AOT) compiler for WebAssembly. WAVM translates Wasm binaries into optimized native machine code prior to execution, which significantly enhances performance.

WAFL extends WAVM by applying a series of patches that enable the generation of coverage information essential for AFL++'s feedback–driven fuzzing approach. These patches modify the WAVM runtime to insert instrumentation hooks at all branch points within the Wasm binary, facilitating precise tracking of execution paths without altering the original binary structure. WAFL employs AOT compilation to minimize runtime overhead, as Wasm binaries are precompiled into native code, reducing the need for Just–In–Time (JIT) compilation during fuzzing. Additionally, WAFL introduces lightweight Virtual Machine (VM) snapshots, allowing the fuzzer to quickly save and restore the VM state between fuzzing iterations. This optimization significantly accelerates the fuzzing process by minimizing setup and teardown times, thereby increasing the overall throughput of fuzzing campaigns.

Empirical evaluations have demonstrated that WAFL can achieve impressive performance, sometimes even outperforming native AFL x86–64 harnesses compiled from source code. This superior performance is attributed to the efficient integration with WAVM and the effective use of VM snapshots, which collectively reduce the overhead typically associated with binary instrumentation. To generate AFL++–compatible coverage information, WAFL implements static binary instrumentation within the WAVM runtime. This process involves inserting instrumentation code at all branch instructions within the Wasm binary, ensuring comprehensive coverage data collection. By capturing detailed execution paths, WAFL enhances AFL++'s ability to guide the fuzzing process towards unexplored and potentially vulnerable areas of the binary. This meticulous coverage generation is crucial for maximizing the effectiveness of the fuzzing efforts, enabling WAFL to uncover a wide range of vulnerabilities, including buffer overflows, memory corruption, and code injection flaws.

In addition to its core fuzzing functionalities, WAFL incorporates a canary–based protection mechanism to detect and prevent memory corruption vulnerabilities. Canary values are strategically placed in memory regions susceptible to overflow attacks. During execution, any attempt to overwrite these canaries triggers an immediate detection of the anomaly, allowing WAFL to flag potential security breaches before they can be exploited. This

proactive defense mechanism not only aids in identifying crashes but also enhances the overall security assurance provided by the fuzzer. Despite its strengths, WAFL is inherently tied to the WAVM runtime, which limits its applicability to environments that utilize WAVM for executing Wasm binaries [28]. This dependency restricts WAFL's use in scenarios where alternative Wasm runtimes, such as Wasmtime or Wasmer, are preferred.

Dynamic analysis techniques have the following advantages: runtime context, reduced false positives, and detection of complex vulnerabilities. Their limitations include limited coverage, performance overhead, and environmental dependencies.

Research on hybrid analysis methods

Recent studies have demonstrated the complementary nature of these approaches, with static analysis offering early detection of code vulnerabilities, and dynamic analysis capturing issues that manifest during execution. Researchers have explored combining both techniques into hybrid models to enhance vulnerability detection in WebAssembly binaries. These combined approaches provide a more comprehensive security assessment by leveraging the strengths of both static and dynamic methods.

WASP2 [29] presents a sophisticated framework for detecting vulnerabilities in WebAssembly binaries by leveraging both static and dynamic analysis techniques informed by known vulnerability patterns. A core component of WASP2 is its deep learning–based vulnerability classification model. The model is trained to identify vulnerabilities by mapping static features from known vulnerable binaries in architectures such as x86 and ARM to their corresponding WebAssembly binary representations. This mapping process involves several steps:

- Feature Extraction – WASP2 extracts a comprehensive set of static features from both the source architectures (x86/ARM) and the WebAssembly binaries. These features include opcode sequences, control flow patterns, data flow characteristics, and other relevant code attributes that are indicative of vulnerabilities.
- Model Training – The extracted features from known vulnerable and benign binaries are used to train a deep neural network. The architecture of the model typically includes multiple layers, such as convolutional layers for feature extraction and fully connected layers for classification. Techniques such as dropout, batch normalization, and regularization are employed to prevent overfitting and enhance the model's generalization capabilities.
- Cross–Architecture Mapping – By mapping features across different architectures, WASP2 ensures that the model can generalize vulnerability patterns from traditional binary formats (x86/ARM) to WebAssembly binaries. This cross–architecture approach is crucial for leveraging the extensive body of knowledge and datasets available for x86 and ARM vulnerabilities, thereby enhancing the model's robustness and accuracy. The authors of WASP2 conducted extensive evaluations to assess the framework's effectiveness in detecting known vulnerabilities within WebAssembly binaries.

The evaluation process involved the following steps:

- Dataset Construction: A diverse dataset comprising WebAssembly binaries with known vulnerabilities, derived from real–world applications and benchmark suites, was assembled. This dataset included various types of vulnerabilities such as buffer overflows, use–after–free errors, integer overflows, and code injection flaws.
- Model Performance – The deep learning model achieved high accuracy in identifying vulnerable subroutines, demonstrating its capability to generalize vulnerability patterns from x86 and ARM architectures to WebAssembly. Precision, recall, and F1–score metrics were used to quantify the model's performance, with WASP2 attaining precision and recall rates exceeding 90% in most categories.
- Runtime Overhead – The integration of static and dynamic analysis introduced minimal runtime overhead, primarily attributable to the efficient feature extraction and model inference processes. The dynamic analysis phase with Wasabi added an additional layer of verification without significantly impacting overall performance. WASP2 offers several key advantages such as:
- High Accuracy: By leveraging deep learning and cross–architecture feature mapping, WASP2 achieves high accuracy in detecting known vulnerabilities, reducing the rate of false positives and negatives.

— Comprehensive Analysis – The combination of static and dynamic analysis provides a holistic view of potential vulnerabilities, enabling the detection of both structural weaknesses and runtime behaviors that could lead to security breaches.
— Automation and Scalability – WASP2 automates the vulnerability detection process, making it scalable for large codebases and suitable for integration into continuous integration/continuous deployment (CI/CD) pipelines.
— Adaptability: The framework can be adapted to incorporate new vulnerability patterns and support additional architectures, enhancing its long–term viability and relevance. Despite its strengths, WASP2 exhibits certain limitations: dependence on known vulnerabilities, runtime environment constraints, and resource intensive.

Hybrid analysis techniques have the following advantages: enhanced coverage, improved accuracy, and contextual awareness. Their limitations include increased complexity, higher overhead, and coordination challenges.

Research on comparative analysis of the static, dynamic, and hybrid detection techniques

Vulnerability detection in Wasm binaries can be approached through various methodologies, primarily categorized into static analysis, dynamic analysis, and hybrid techniques. Each of these approaches offers distinct advantages and faces unique challenges, making a comparative understanding essential for selecting the appropriate strategy in various security contexts (Table 1).

In the context of WebAssembly, the choice between static, dynamic, and hybrid analysis techniques depends on several factors, including the availability of source code, the performance requirements of the application, and the nature of potential vulnerabilities. Static analysis is particularly useful for preliminary security assessments and ensuring code integrity before deployment. Dynamic analysis excels in environments where real–time monitoring and runtime behavior are critical, such as in web browsers and server–side applications running Wasm modules. Hybrid approaches are ideal for scenarios demanding thorough security evaluations, where both code structure and runtime behavior must be scrutinized to uncover a wide array of vulnerabilities. Recent work by M. Țălu [30] discusses advanced data protection techniques in WebAssembly, contributing to the security landscape.

Table 1. Comparative analysis of the static, dynamic, and hybrid detection techniques

Aspect	Static Analysis	Dynamic Analysis	Hybrid Analysis
Coverage	Comprehensive across all possible paths	Limited to executed paths	Comprehensive by combining both static and dynamic coverage
False Positives/Negatives	Higher potential for false positives and negatives	Lower false positives, but may miss some vulnerabilities	Reduced false positives and negatives through cross–validation
Performance Overhead	Generally low as no execution is required	Higher due to instrumentation and monitoring	Higher, combining overheads of both approaches
Contextual Information	Limited, no runtime context	Rich, with runtime context	Enhanced by leveraging both static and dynamic contexts
Complexity	Moderate, depends on the analysis depth	High, due to the need for controlled execution	High, due to the integration of multiple methodologies
Scalability	High, suitable for large codebases	Limited, especially for performance–critical applications	Moderate, balancing thoroughness with resource demands
Detection Capability	Good for structural and syntax–based vulnerabilities	Excellent for runtime and context–dependent vulnerabilities	Superior, covering a wide range of vulnerabilities

CONCLUSIONS

In conclusion, the security landscape of WebAssembly binaries is evolving rapidly as this technology gains prominence in modern web applications and server–side environments. This research has highlighted the critical need for effective vulnerability discovery techniques tailored to the unique characteristics of Wasm. By examining static, dynamic, and hybrid analysis methods, we have provided insights into the strengths and limitations of each approach. Static analysis techniques are valuable for their ability to perform comprehensive code evaluations and early detection of vulnerabilities without execution, yet they often struggle with false positives and a lack of contextual understanding. In contrast, dynamic analysis offers deeper insights into runtime behavior, enabling the identification of context–dependent vulnerabilities. However, it is constrained by the execution paths that are actually traversed during testing, which can limit its overall coverage. Hybrid analysis methods emerge as a promising solution, combining the thoroughness of static analysis with the contextual richness of dynamic analysis. By integrating these approaches, hybrid techniques

can enhance vulnerability detection while mitigating the weaknesses inherent in each individual method. The effective deployment of these techniques in the WebAssembly ecosystem is crucial to ensure the security and reliability of applications utilizing Wasm.

References

[1] WebAssembly official documentation. Available at: https://webassembly.org/ (accessed September 30, 2024).
[2] A. Rossberg, B.L. Titzer, A. Haas, D.L. Schuff, D. Gohman, L. Wagner, A. Zakai, J.F. Bastien, M. Holman. Bringing the Web Up to Speed with WebAssembly. Communications of the ACM, 61(12): 107–115, 2018
[3] P. Mendki. Evaluating Webassembly Enabled Serverless Approach for Edge Computing, 2020 IEEE Cloud Summit, Harrisburg, PA, USA, 2020, pp. 161–166
[4] M.N. Hoque, K.A. Harras. WebAssembly for Edge Computing: Potential and Challenges. IEEE Commun. Stand. Mag. 2022, 6, 68–73.
[5] P.P. Ray. An Overview of WebAssembly for IoT: Background, Tools, State–of–the–Art, Challenges, and Future Directions. Future Internet, 15(8): 275, 2023
[6] D. Lehmann, J. Kinder, M. Pradel. Everything old is new again: Binary security of WebAssembly. In Proceedings of the 29th USENIX Security Symposium (USENIX Security 20), Boston, MA, USA, 2020; pp. 217–234.
[7] Y. Yan, T. Tu, L. Zhao, Y. Zhou, W. Wang. Understanding the performance of webassembly applications. In Proceedings of the 21st ACM Internet Measurement Conference, Virtual Event, 2021; pp. 533–549.
[8] J. Dejaeghere, B. Gbadamosi, T. Pulls, F. Rochet. Comparing Security in eBPF and WebAssembly. In Proceedings of the ACM SIGCOMM 1st Workshop on eBPF and Kernel Extensions, New York City, NY, USA, 10 September 2023.
[9] M. Kim, H. Jang Y. Shin. Avengers, Assemble! Survey of WebAssembly Security Solutions, 2022 IEEE 15th International Conference on Cloud Computing (CLOUD), Barcelona, Spain, 2022, pp. 543–553
[10] J. Sun, D.Y. Cao, X.M. Liu, Z. Zhao, W.W. Wang, X.L. Gong. SELWasm: A Code Protection Mechanism for WebAssembly, 2019 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom), Xiamen, China, 2019, pp. 1099–1106
[11] H. Lei, Z. Zhang, S. Zhang, P. Jiang, Z. Zhong, N. He, D. Li, Y. Guo, X. Chen. Put Your Memory in Order: Efficient Domain–based Memory Isolation for WASM Applications. CCS '23: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, pp. 904 – 918
[12] H. Harnes, D. Morrison. SoK: Analysis Techniques for WebAssembly. Future Internet., 16(3): 84, 2024
[13] J. Shortt, Master thesis of Computer Science: A System for Bounding the Execution Cost of WebAssembly Functions. Carleton University, Ottawa, Ontario, 2023
[14] Q. Stiévenart, D.W. Binkley, C. De Roover. W: a WebAssembly Static Analysis Library (Extended Presentation Abstract). https://soft.vub.ac.be/Publications/2021/vub–tr–soft–21–04.pdf.
[15] Q. Stiévenart, D.W. Binkley, C. De Roover. Static stackpreserving intra–procedural slicing of Webassembly binaries. In Proceedings of the 44th International Conference on Software Engineering, pp. 2031–2042, 2022.
[16] Q. Stiévenart, C. De Roover. Compositional information flow analysis for Webassembly programs. In Proceedings of the 2020 IEEE 20th InternationalWorking Conference on Source Code Analysis and Manipulation (SCAM), Adelaide, SA, Australia, 28 September–2 October 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 13–24. WASSAIL.
[17] P.D.R. Lopes. Master's Thesis: Discovering Vulnerabilities in Webassembly with Code Property Graphs. Instituto Superior Técnico, Lisbon, Portugal. 2021.
[18] T. Brito, P. Lopes, N. Santos, J.F. Santos. Wasmati: An efficient static vulnerability scanner for WebAssembly, Computers & Security, 118, 102745, 2022
[19] F. Marques, J. Fragoso Santos, N. Santos, P. Adão. Concolic Execution for WebAssembly. In Proceedings of the 36th European Conference on Object–Oriented Programming (ECOOP 2022), Berlin, Germany, 2022; Ali, K., Vitek, J. (eds.); Leibniz International Proceedings in Informatics (LIPIcs); vol. 222, pp. 11:1–11:29.
[20] A. Szanto, T. Tamm, A. Pagnoni. Taint tracking for WebAssembly. arXiv 2018. arXiv:1807.08349.
[21] W. Fu, R. Lin, D. Inge, TaintAssembly: Taint–Based Information Flow Control Tracking for WebAssembly, arXiv 2018. arXiv:1802.01050.
[22] Node.js. Node.js. 2022. Available online: https://nodejs.org/en (accessed on 5 October 2024).
[23] D. Lehmann, M. Pradel. Wasabi: A framework for dynamically analyzing webassembly. In Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems, Providence, RI, USA, 13–17 April 2019; pp. 1045–1058.
[24] D. Lehmann, M.T. Torp, M. Pradel. Fuzzm: Finding Memory Bugs through Binary–Only Instrumentation and Fuzzing of WebAssembly. arXiv 2021, arXiv: 2110.15433.
[25] K. Haßler, D. Maier. WAFL: Binary–Only WebAssembly Fuzzing with Fast Snapshots. In Proceedings of the Reversing and Offensive–oriented Trends Symposium, Vienna, Austria, 18–19 November 2021; pp. 23–30.
[26] A. Fioraldi, D. Maier, H. Eißfeldt, M. Heuse. AFL++: Combining Incremental Steps of Fuzzing Research. In Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT 20), Boston, MA, USA, 2020.
[27] WAVM. 2021. Available online: https://wavm.github.io (accessed on 5 October 2024).
[28] K. Haßler, D. Maier. WAFL: Binary–Only WebAssembly Fuzzing with Fast Snapshots. ROOTS'21: Reversing and Offensive–oriented Trends Symposium. 2022, pp. 23–30
[29] P. Sun, L. Garcia, Y. Han, S. Zonouz, Y. Zhao. Poster: Known vulnerability detection for WebAssembly binaries. 2021
[30] M. Țălu, A Review of Advanced Techniques for Data Protection in WebAssembly, Acta Technica Corviniensis – Bulletin of Engineering, Hunedoara, Romania, 2024, In press.

ISSN: 2067-3809