2022 IEEE 15th International Conference on Cloud Computing (CLOUD) | 978-1-6654-8137-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/CLOUD55607.2022.00077

Avengers, Assemble! Survey of WebAssembly Security Solutions

Minseo Kim
School of Cybersecurity
Korea University
Seoul, Republic of Korea
ichbinminseo@korea.ac.kr

Hyerean Jang
School of Cybersecurity
Korea University
Seoul, Republic of Korea
hr_jang@korea.ac.kr

Youngjoo Shin
School of Cybersecurity
Korea University
Seoul, Republic of Korea
syoungjoo@korea.ac.kr

***Abstract*—WebAssembly, abbreviated as Wasm, has emerged as a new paradigm in cloud-native developments owing to its promising properties. Native execution speed and fast startup time make Wasm an alternative for container-based cloud applications. Despite its security-by-design strategy, however, WebAssembly suffers from a variety of vulnerabilities and weaknesses, which hinder its rapid adoption in cloud computing. For instance, the native execution performance attracted cybercriminals to abuse Wasm binaries for the purpose of resource stealing such as cryptojacking. Without proper defense mechanisms, Wasm-based malware would proliferate, causing huge financial loss of cloud users. Moreover, the design principle that allows type-unsafe languages such as C/C++ inherently induces various memory bugs in an Wasm binary. Efficient and robust vulnerability analysis techniques are necessary to protect benign cloud-native Wasm applications from being exploited by attackers. Due to the young age of WebAssembly, however, there are few works in the literature that provide developers guidance to such security techniques. This makes developers to hesitate considering Wasm as their cloud-native platform. In this paper, we surveyed various techniques and methods for Wasm binary security proposed in the literature and systematically classified them according to certain criteria. As a result, we propose future research directions regarding the current lack of WebAssembly binary security research.**

***Index Terms*—Cloud computing security, WebAssembly binary security, WebAssembly security solutions**

I. Introduction

WebAssembly is a World Wide Web Consortium (W3C) open standard for a portable, efficient, and secure runtime of web applications. Abbreviated as Wasm, WebAssembly is essentially a sandboxed binary format that can be executed with different OSes on various hardware platforms with native execution speed [1]. Owing to many beneficial properties, the usage of Wasm is not limited to web applications, but is being extended to various applications such as desktop applications [2], mobile devices [3], IoT [4], [5], and embedded systems [6]. In particular, the versatility of WebAssembly brings huge benefits to cloud-native development platforms. Native execution speed and fast startup time of Wasm applications attracted many developers to consider Wasm as an alternative for container-based cloud applications [7]–[9].

However, despite the isolation-based security by its design, Wasm has many security flaws and vulnerabilities that have prevented its rapid adoption in cloud computing. For instance, the native execution performance of Wasm applications tends to attract many cybercriminals and has thus resulted in the advent of Wasm-based resource-stealing malware (e.g., cryptomining malware [10]). The absence of proper defense techniques may result in massive financial losses for cloud users. Moreover, the design principle of Wasm, which supports type-unsafe languages such as C/C++, makes the Wasm binaries susceptible to a wide variety of memory bugs [11].

Thus, to enable the practical and secure usage of Wasm, it is necessary to conduct studies on the prevention of Wasm-based malware and the identification of vulnerabilities in Wasm binaries. Many researchers have previously attempted to address the various security concerns related to the use of Wasm, by applying memory and code protection mechanisms, and sandboxed environments. Consequently, various analysis techniques have been proposed to identify vulnerabilities in Wasm binaries and to protect Wasm applications from attacks. However, because Wasm is still a relatively new technology, the efforts to improve security remain inadequate. Moreover, to date, there is no reported survey of existing Wasm security techniques, which is necessary for developers and security researchers.

In this paper, we provide an in-depth and extensive survey of the state-of-the-art security techniques for Wasm binaries. In particular, we classify previous works based on their methodology and the target security concern. According to our criteria, previous studies on the security of Wasm can be categorized into two types, as security techniques that protect: (1) from malicious Wasm binaries, and (2) Wasm binaries from attacks. The first category mainly includes techniques to detect malicious Wasm binaries such as cryptojacking malware. The second category includes techniques to protect benign Wasm applications, as well as vulnerability analysis techniques. We analyze the strengths and weaknesses of each technique in comparison with other similar methods. Based on our analysis, we suggest a research direction for the security of Wasm binary as a cloud-native application. We expect our survey to contribute to the WebAssembly security research community applied to cloud environments. The contributions of our paper are as follows.

- We present, to the best of our knowledge, the first in-depth and extensive survey of the state-of-the-art security

2159-6190/22/$31.00 ©2022 IEEE
DOI 10.1109/CLOUD55607.2022.00077

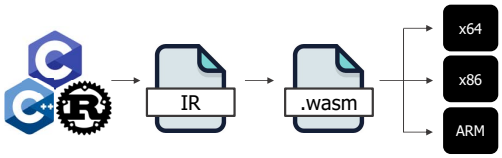

Figure 1. Compilation process of WebAssembly binary

techniques proposed in the literature for Wasm binaries.
- We propose a novel taxonomy of Wasm binary protection mechanisms.
- We present the results of quantitative and qualitative analyses of the strengths and weaknesses of each Wasm security technique compared with other similar techniques.
- We present a discussion of the open problems in current Wasm binary security enhancement research and provide some suggestions to improve security.

The paper is structured as follows. Section II provides an overview of the Wasm binary and describes its structure and basic security mechanisms. Section III describes the usage of Wasm binaries and the associated vulnerabilities in Wasm compilers. Section IV introduces Wasm binary protection methods. Section V describes the weaknesses of existing Wasm binary protection methods and discusses future research directions. Lastly, Section VI presents the conclusions.

II. Background on Wasm

In this section, we provide some essential background information on Wasm. Specifically, we preset an overview of Wasm by describing the structure of Wasm and its built-in security mechanism.

A. Brief overview

The evolution of web browsers has been obstructed by computationally inefficient JavaScript language. Furthermore, as JavaScript can directly access the Document Object Model (DOM), it has a vulnerability that can threaten normal users if an attacker identifies an accessible path to the DOM. To overcome these obstacles, in 2013, Mozilla introduced asm.js [12] as an extension of JavaScript designed to boost execution performance. As an alternative to asm.js, Wasm has been proposed as a new standard by the W3C. Wasm is a portable, low-level bytecode and compilation target designed for safe and fast execution on various target platforms such as x86 and ARM (See Fig.1). Wasm supports high-level languages such as C/C++ [13] and Rust [14] and currently, efforts are being undertaken to extend the support to Python [15] and C# [16]. Because a Wasm compilation produces a compact binary, the process of downloading and executing the Wasm binary on the web is less computationally expensive than that sourced from JavaScript code.

B. Structure of Wasm module

Wasm modules are created as a result of a Wasm compilation process. As shown in Fig.2, the Wasm module generally

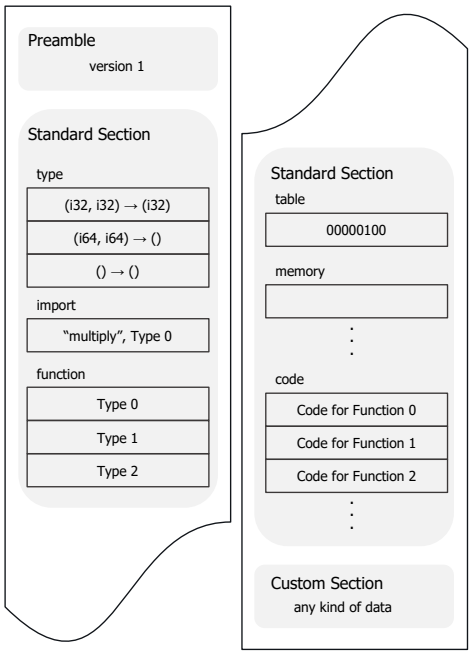

Figure 2. Structure of WebAssembly module

consists of *Preamble*, *Standard*, and *Custom* sections, the details of which are presented below.

1) Preamble: This is a section that indicates a starting point for the Wasm bytecode. More specifically, it informs that the binary file is a Wasm module and also includes additional information such as the Wasm version.

2) Standard section: This section contains all the necessary information for executing a Wasm module, such as the code, function, and memory. Every Wasm module has at least one Standard section, and they each validate their Standard sections before initiating the execution. Each value in the Standard section appears only once in the order shown in Fig.2.

Type. The Type section declares a list of all the unique function signatures used by the Wasm module, including the imported functions. Additionally, multiple functions may share the same function signature. Fig.2 shows an example of the Type section in which three function signatures are declared.

Import. The Import section declares all the functions, tables, memory, and global variables imported by the Wasm module. This section is necessary to enable code and data sharing among multiple modules.

Function. The Function section contains a complete list of the functions defined in the Wasm module. The location of a function in the module is referenced with a function body index in the Code section; the value is applied as an index of the function signature in the Type section.

Table. The Table section provides an indirection to refer to the actual function. That is, the function call occurs through this table mapping. This sections enhance the security of Wasm module by preventing code from directly accessing function pointers.

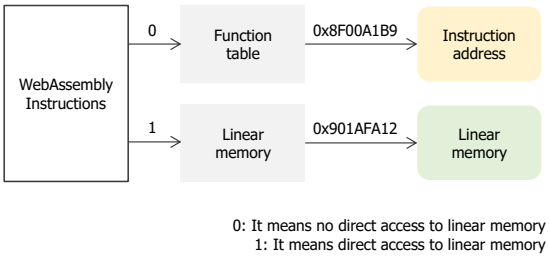

Figure 3. An example of linear memory in a Wasm module

Memory. The Memory section contains the linear memories that are used by a module instance. As with the Table section, it is a key element of Wasm security, as the Memory section prohibits Wasm binaries from directly accessing memory.

Code. The Code section contains the body of functions defined in the Wasm module. The order of functions in the Code section is the same as the order in which the function signatures are declared.

3) Custom section: The Custom section typically contains arbitrary Wasm module data, such as the symbols for function and variable names, which are necessary for debugging. Unlike other user-defined sections, the Custom section only appears after the Standard section. However, the Custom section can appear more than once in a Wasm module and those sections can share the same name. Unlike Standard sections, this section does not introduce error, even if it is not correctly placed.

C. Security features of Wasm

Security is one of the primary design goals for Wasm. Thus, Wasm is designed to have the following security features and properties.

1) Sandboxed environment: Wasm binaries run in a sandboxed environment that is isolated from the outside. Thus, by design, a Wasm module cannot operate outside of the confines of the sandbox without authorized permission and proper APIs. For instance, Wasm modules in a web browser have no direct access to DOM objects, as they are only allowed access via JavaScript APIs. Wasm modules are also subject to restrictions on information flow through the same-origin policy (SOP) enforced by web browsers. Additionally, a standalone Wasm that runs on a sandboxed runtime with OS support, is also requested to use propose APIs to access system resources such as files and sockets.

2) Linear memory: A Wasm module is instantiated, as it uses a specific API to create necessary memory objects for the execution. Then, a JavaScript engine or system runtime internally creates a managed buffer (e.g., an ArrayBuffer for JavaScript). Thus, the Wasm module operates by accessing the physical memory address indirectly. This means that the linear memory of Wasm is implemented in managed buffers to ensure that the Wasm module only reads and writes data in a limited area of the memory.

As shown in Fig.3, the Wasm module does not have direct access to the instruction address. For example, if the Wasm module invokes a function located at `0x8F00A1B9`, it can only access the function through the function table. However, the data located at `0x901AFA12` in the linear memory can be directly accessed by the Wasm module.

3) Control-flow integrity: In Wasm, the control-flow integrity for direct/indirect function calls and returns are guaranteed at runtime. To enable this, the compiler creates a control-flow graph (CFG) for the Wasm program during a compilation process. Thus, any executions that do not follow the CFG would result in failure.

When calling a function directly from the Wasm module, Wasm explicitly uses the function's index to protect the binary so that the function is called normally. Specifically, the functions are indexed in a function table with type information and the type is validated whenever direct function calls are made. Alternatively, returns are protected through a protected call stack.

III. Related Work

In this section, we discuss some previous related work.

A. Surveys on the usage of Wasm

Owing to its portability, Wasm is currently being extended from web applications to desktop, mobile, IoT, and even cloud applications. However, the knowledge on the usage of programming languages in the Wasm ecosystem is minimal. In consideration of this, Hilbig et al. [10] surveyed the usage of Wasm languages in the wild; they discovered that approximately 64.2% of Wasm binaries were compiled from C/C++ source code. The difference between our survey and that conducted by Hilbig et al. [10] is that they mainly focused on investigating the practical usage of Wasm binaries, whereas we attempted to survey the state-of-the-art security enhancement techniques for Wasm binaries.

B. Studies on Wasm vulnerabilities

Although the developers of Wasm did consider security in the design, it is known that Wasm has many security flaws, including traditional buffer overflow vulnerabilities [11].

Romano et al. [17] discovered that the vulnerabilities in Wasm binaries are primarily sourced from the lack of any protections or security mechanisms in most Wasm compilers. They analyzed some Wasm compilers such as Emscripten [13], AssemblyScript [18], and Rustc/Wasm-Bindgen [19], to identify their vulnerabilities and flaws. They consequently found bugs in Emscripten that may cause significant security problems [17]. They also investigated the compatibility-related problems of programming languages, because Wasm tends to have more limited data types (e.g., `i32, f32, i64,` and `f64`) than other languages; this may lead to type-confusion vulnerabilities.

To summarize, the work of Romano et al. [17] focused on investigating various problems with Wasm compilers, including several vulnerabilities that may originate from compiler

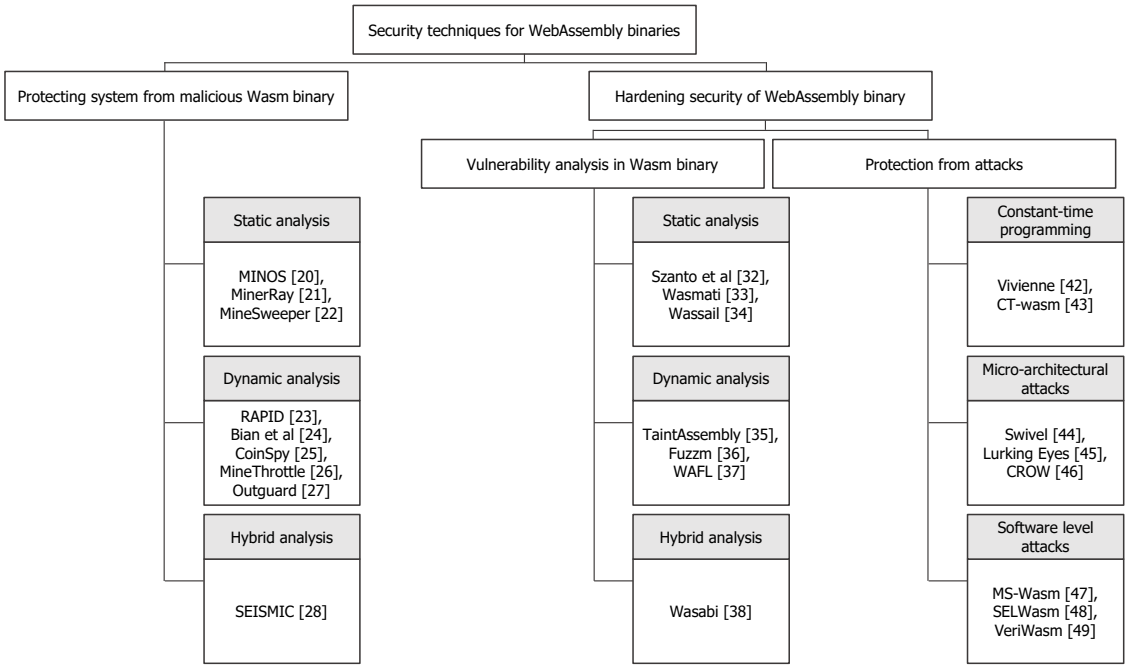

Figure 4. Classification of security techniques for WebAssembly binaries

errors. Our work is different because we focus on security concerns in Wasm binaries, not in compilers. Furthermore, we also surveyed various security enhancement techniques for Wasm binaries.

IV. Security techniques for Wasm binaries

In this section, we present the results of our detailed survey of the state-of-the-art security techniques for Wasm binaries. Generally, the security techniques proposed in the literature can be classified according to their security goals and methodologies. Thus, we have classified previous works into two categories according to the criteria, as security techniques to: (1) provide protection from malicious Wasm binaries (details presented in Section IV-A), and (2) protect Wasm binaries from various attacks (details presented in Section IV-B). Fig.4 shows our taxonomy of the Wasm security solutions.

The most recent work that we surveyed tended to address security problems by constructing attack detection or vulnerability analysis methods. These methods share several general approaches, which have been classified as either static-, dynamic-, or hybrid-based.

For our survey, the surveyed methods were placed in the *static-based approach* category if their primary strategy was based on the analysis of binary files or source code of Wasm modules. Such methods were developed to identify vulnerabilities or detect malicious code inside Wasm applications by conducting static analysis techniques, such as formal verification and reverse engineering.

The surveyed methods were placed in the *dynamic-based approach* category if their strategy was primarily based on monitoring the dynamic behavior of Wasm binaries. The dynamic behavior may include changes to the input/output, environment variables, and/or running states of the program. Any changes in behavior during the execution of a Wasm binary can be traced by using debuggers or dynamic binary instrumentation tools.

Lastly, the surveyed methods were placed in the *hybrid-based approach* category if they used both static- and dynamic-based approaches to achieve their goals.

A. System protection from malicious Wasm binaries

Among the various security techniques that have been proposed to date for detecting malicious Wasm binaries, most focus on the detection or identification of a type of resource-stealing malware (e.g., cryptojacking malware). In this section, we present the results of our survey of such protection schemes against malicious Wasm binaries. Specifically, we have classified them as static-, dynamic-, or hybrid-based approaches. We also present the analysis of the detection performance of these schemes.

Static analysis-based approach. MINOS [20] is a malware detection method that has been specifically designed for the detection of Wasm-based cryptojacking malware. In general, the MINOS malware detection scheme converts a binary file of a Wasm module into a grayscale image. MINOS applies the converted image as an input to a convolutional neural network (CNN) to decide whether the Wasm binary performs a cryptojacking attack. Because the analysis is performed on a static basis, MINOS does not subject the production systems to high computational overhead.

Table I
Classification of scheme to protect systems from malicious WebAssembly binary

Category	Scheme	Open-sourced	Detection feature	Performance				Cryptomining algorithms detection			
				Data collection	Detection time (s)	Accuracy	F1 score	Number of detectable cryptominers	Coin Hive	Crypto Night	Crypto Loot
Static	MINOS [20] (2021)	✗	Grayscale image	Alexa top 1M websites, Tranco top 100K websites of Tranco 1M websites	0.0259	98.97%	0.95	-	✓	✓	✗
	MinerRay [21] (2020)	✓	CFG	Alexa top 1.2M websites of Alexa top 10M websites	1.9	99.99%	0.99	-	✓	✓	✓
	MineSweeper [22] (2018)	✓	JavaScript code, Network requests/response	Alexa top 1M websites	-	99.46%	0.94	13 (CoinHive, CoinImp, Mineralt, JSECoin, CryptoLoot, CryptoNoter, Coinhave, Minr, Webmine, DeepMiner, Cpufun, Monerise, NF WebMiner)	✓	✓	✓
Dynamic	RAPID [23] (2018)	✗	JavaScript API, System resource	Alexa top 330,500 websites	-	99.99%	0.94	-	✓	✗	✗
	Bian et al [24] (2019)	✗	Subsequences of WebAssembly instruction	Alexa top 100K websites	-	100%	1	-	✗	✗	✓
	CoinSpy [25] (2020)	✗	Cryptojacking PoW features	CoinHive pages, Public WWW, Alexa top 5K websites, Alexa 100K websites of Alexa top 1M websites	-	96%	-	4 (CoinImp, CryptoLoot, WebMine, CryptoNight)	✓	✓	✓
	MineThrottle [26] (2020)	✗	Frequency distribution of instructions	Alexa top 1M websites	-	99.69%	0.98	-	✓	✗	✓
	Outguard [27] (2019)	✓	Web workers, Parallel tasks, WebAssembly, WebSockets, Hash algorithms, PostMessage event load, MessageLoop event load	Alexa top 1M websites, Alexa top 600K websites	-	99.99%	0.98	13 (CoinHive, CoinImp, Mineralt, JSECoin, CryptoLoot, CryptoNoter, Coinhave, Minr, Webmine, DeepMiner, Cpufun, Monerise, NF WebMiner)	✓	✓	✓
Hybrid	SEISMIC [28] (2018)	✗	WAT	A-star, Asteroids, Bullet(10), Bullet(1000), CreaturePack, FunkyKarts, Basic4GL, Tanks, CoinHive v0 & v1, NFWebMiner, YAZEC Miner	1.7	98%	0.99	4 (CoinHive v0/v1, NF WebMiner, YAZEC Miner)	✓	✓	✗

MinerRay [21] attempted to detect the cryptojacking behavior of a Wasm module by analyzing a CFG. More specifically, MinerRay constructs a CFG from the Wasm binary and then searches for control flows to certain encryption algorithms that are commonly used for cryptojacking. In the case of Wasm-based cryptojacking malware, the cryptographic algorithms are typically implemented in JavaScript, Wasm, and/or non-standard Wasm libraries such as asm.js. First, MinerRay converts JavaScript and non-standard libraries into standard Wasm binaries. Then, it translates the standard Wasm binary into an intermediate representation from which an interprocedural CFG is constructed. In an experimental environment, MinerRay achieved high detection rates for most cryptojacking Wasm binaries that run on web browsers; it has also been demonstrated to have the lowest false positive rate among all relevant detection schemes in existence at the time.

MineSweeper [22] is another detection method for cryptojacking Wasm binaries. Essentially, the detection strategy of MineSweeper is dependent on the characteristics of CryptoNight [29], which is a specialized algorithm designed for cryptomining. Specifically, MineSweeper attempts to identify cryptojacking attacks by identifying the signatures of the CryptoNight algorithm and its variants in Wasm binaries.

All the abovementioned static analysis-based approaches have the advantage of a low system overhead. This is because Wasm module execution is not required for the detection techniques to be applied to the binary files. However, static approaches have a critical drawback in that they can be easily bypassed by attackers utilizing code obfuscation techniques.

Dynamic analysis-based approach. RAPID [23] is a dynamic analysis-based scheme that identifies cryptojacking attacks. RAPID uses JavaScript API calls and system resources to monitor the behavior of suspected applications during runtime. To this end, information on JavaScript API usage is collected from web browsers by using Chromium2 and Chrome debugging features. The runtime usage of memory, CPU, and network transmission of the web browser is also collected by using Docker containers. Then, a support vector machine is employed to allow this raw data to be applied as the basis of a classification model. Because the classification-enabling features are obtained with the aid of resource-monitoring tools, RAPID is likely to be bypassed by attackers who can disturb system resource monitoring.

To overcome the limitation of RAPID, Bian et al. [24] proposed an approach that applies the instruction stream of a Wasm application as a feature. The effectiveness of their scheme was predicated upon the fact that the instruction sequence of cryptojacking malware is distinguishable from benign Wasm binaries. Specifically, cryptojacking malware performs a considerably large number of hash computations that make the instruction sequences uniform. The results of evaluation revealed that the scheme proposed by Bian et al. afforded higher cryptojacking attack detection accuracy than previously developed schemes.

CoinSpy [25] attempts to identify cryptojacking attacks by exploiting the fact that proof-of-work (PoW) algorithms are common in cryptojacking malware. In particular, the CoinSpy approach was established based on the results of previous observations that indicated that the number of parallel threads and memory usage substantially increase while the PoW algorithm is running. The CoinSpy detection method was also predicated upon previous observations that have indicated that communication from the malware only occurs when the CPU is in an idle state. With these features as its basis, a CNN-based classification model was built to detect cryptojacking attacks.

MineThrottle [26] is predicated upon previous observations that indicate that the frequency distribution of cryptojacking malware instructions is unique and distinguishable from that of normal applications. Accordingly, an attack is judged to occur if the frequency of specific instructions is abnormal or far from the threshold. As attackers may bypass the detection by using some dummy instructions, MineThrottle was also designed to include a feature that prevented detection circumvention through the implementation of dynamic thresholds that were specific to each attack case. The results of evaluation revealed that MineThrottle had a high attack detection accuracy and low false-positive rate.

Outguard [27] attempts to detect cryptojacking attacks based on the characteristics of Wasm binary execution. In particular, this method analyzes all the functions in a JavaScript runtime and rendering engine to accurately extract the behavioral characteristics of cryptomining. The results of analysis revealed that functions related to the web worker, parallel operation, Wasm, WebSockets, the hash algorithm, PostMessage event loading, and message loop event loading, were important as signatures for attack detection.

To summarize, all the abovementioned dynamic analysis-based methods apply the behavioral characteristics of Wasm binaries as the basis of their attack detection schemes. Thus, this type of approach is more difficult for attackers to bypass than the static-based approach. However, this type of approach has the following limitation: the detection can only be performed when the suspected binary is being executed; therefore, this type of approach may have a higher computational overhead.

Hybrid analysis-based approach. As an attempt to overcome the limitations of static and dynamic analysis-based approaches, certain methods have been designed to incorporate both types of approaches to achieve high-accuracy and efficient attack detection. SEISMIC [28] is a detection method that adopts such a hybrid analysis-based approach. SEISMIC first converts a Wasm binary into Wasm text format (WAT) that is subsequently parsed into an abstract syntax tree form. Then, it inserts a new global 64-bit integer variable into global sections of the Wasm binary to profile the Wasm instructions. This allows SEISMIC to monitor all the Wasm binary instructions that are executed during runtime. The results of the evaluation of SEISMIC revealed that the hybrid analysis-based approach has the following advantage: higher accuracy can be obtained even under the conditions of a small set of features.

Comparative analysis. We now present our comparative analysis of the static-, dynamic-, and hybrid-based methods presented in this section. The results of analysis are summarized in Table I.

- *Dataset.* We first investigated the types of datasets used in the surveyed works to evaluate the performance of their approaches. As shown in Table I, the investigation revealed that the dataset from the Alexa site [30] was employed for most studies; this means that the real Wasm binaries applied for the evaluation were mostly collected from websites in the wild. Because Wasm is currently being extended for applications beyond the web, including the cloud, we were able to determine that the datasets used in the surveyed works did not cover all the Wasm use cases.
- *Performance.* We compared the performances of the surveyed methods in terms of the attack detection time, accuracy, and F1 score. The comparison was based on the results of evaluation for each scheme. The results of this performance comparison are also presented in Table I. The table shows that the scheme developed by Bian et al. [24] has a relatively high detection accuracy and F1 score compared with those of other schemes. We attribute this to their evaluation methodology; specifically, the number of datasets that were applied for the evaluation was relatively small.
- *Cryptomining algorithms.* We also investigated the detection coverage of the surveyed methods. Because the primary goal of all the schemes was to identify cryptojacking attacks in Wasm binaries, we compared the coverage with respect to a set of commonly used cryptomining tools such as CoinHive, CryptoNight, and CryptoLoot [31]. As can be seen in Table I, we found that almost all the schemes can detect malicious Wasm binaries that employ those cryptomining tools. In addition, certain techniques can detect a wider variety of cryptomining algorithms.

B. Hardening the security of Wasm binaries

In addition to enlightening a research direction for Wasm-based malware detection, there have been efforts dedicated to hardening the security of Wasm binaries from the perspectives of (1) vulnerability analysis and (2) protection from attacks. In this subsection, we present the in-depth results of a survey of such schemes, as well as a comparative analysis.

1) Vulnerability analysis of Wasm binaries: Because Wasm was designed to be a polyglot compilation target, it supports multiple programming languages, including type-unsafe languages such as C/C++. Given this fact, Wasm binaries compiled from unsafe languages may contain memory corruption vulnerabilities that make them susceptible to classical buffer overflow attacks [11]. Such vulnerable Wasm binaries become attractive targets as they are easily exploited by attackers. Thus, vulnerability analysis techniques for Wasm binaries are necessary to reinforce Wasm security. In the literature, there

Table II
Classification of WebAssembly binary vulnerability analysis scheme

Category	Scheme	Open-sourced	Analysis technique	Implementation method	Runtime overhead
Static-based	Szanto et al [32] (2018)	✗	Taint analysis	Own tool	-
	Wasmati [33] (2021)	✗	CPG	Own tool	-
	Wassail [34] (2020)	✓	Compositional analysis	Own tool	-
Dynamic-based	TaintAssembly [35] (2018)	✓	Taint tracking	JavaScript engine modified	$5\% \sim 12\%$
	Fuzzm [36] (2021)	✓	Fuzz testing	Own tool	$5\% \sim 6\%$
	WAFL [37] (2021)	✓	Fuzz testing	Own tool	-
Hybrid-based	Wasabi [38] (2019)	✓	Static instrumentation, Analysis at runtime	Own tool	$< 1\%$

are a number of papers that describe a method of analysis to identify vulnerabilities in Wasm binaries. Table II lists the binary analysis techniques that we have classified according to their methodology, i.e., static analysis-, dynamic analysis-, and hybrid analysis-based approaches.

Static analysis-based approach. Szanto et al. [32] proposed a static analysis-based vulnerability detection method that inserts a taint into the program's memory, then tracks this data and observes the impact of that value on the program. In this method, the input parameters of the functions in the Wasm binary are applied as taint labels. With this method, Wasm binaries can be analyzed without modifying the structure.

Alternatively, Wasmati [33] identifies vulnerabilities in a Wasm binary by constructing a code property graph (CPG) from the binary file. In this method, the execution order, execution path, dependency of the data, and control flows of the Wasm binary, are all expressed in the CPG. The experimental results reported in the paper revealed that Wasmati was able to detect all vulnerabilities included in binaries written in C/C++ language. Moreover, Wasmati was also demonstrated to be able to find vulnerabilities in the interface between a Wasm binary and JavaScript.

Wassail [34] is the first static analysis-based method to utilize a compositional analysis technique [39] to find vulnerabilities in a Wasm binary. Compositional analysis techniques are techniques that infer knowledge about an entire system based on an analysis of partial components of the system. In the case of Wasm, all functions within a Wasm binary are isolated and only accessible via JavaScript. This characteristic hinders vulnerability analysis because it is difficult to determine the entire analysis scope affected by interesting functions. Wassail is able to overcome this type of challenging problem by integrating a compositional analysis technique into a method of summary-based configuration analysis.

Static analysis-based approaches have an advantage as vulnerabilities can be identified in the software development stage. However, its drawback is that the dynamic behavior of a program is invisible; thus, it cannot detect vulnerabilities that only occur in certain control flows.

Dynamic analysis-based approach. TaintAssembly [35] is a dynamic analysis-based method that utilizes a taint analysis technique to detect vulnerabilities in Wasm binaries. In TaintAssembly, all Wasm variables of type i32, i64, f32, and f64 are labeled as tainted inputs. The results of their evaluation revealed that all the vulnerabilities in a Wasm binary can be detected with an acceptable computational overhead. Nonetheless, a limitation of TaintAssembly is that the structure of the Wasm binary needs to be reconstructed before the taint labels can be set for all the variables in the program.

Fuzzm [36] constitutes the first method to utilize greybox fuzzing to detect vulnerabilities in Wasm binaries. In addition, Fuzzm applies a canary-based protection mechanism that isolates the Wasm binary from memory corruption vulnerabilities such as stack and heap overflow.

WAFL [37] is a Wasm binary-only fuzzing technique that utilizes an AFL++ fuzzer [40]. More specifically, WAFL applies coverage-based fuzzing to the Wasm binary with better performance; this is possible because running instances can be quickly reset by using VM snapshots. WAFL is not only applicable to Wasm binaries running on a web browser but it is also applicable to standalone binaries with a Wasm system interface [41].

As previously mentioned, dynamic analysis-based methods can only be implemented under the conditions of program execution. This implies that vulnerability analysis can only be performed within the context of a running process; this may limit the applicability of dynamic-based approaches.

Hybrid analysis-based approach. Wasabi [38] is a hybrid analysis-based framework for vulnerability analysis in Wasm. The first step of this method involves modifying a bytecode of the Wasm binary to perform binary instrumentation. After the binary has been instrumented, Wasabi monitors the dynamic behavior of the Wasm module on an instruction-by-instruction basis.

Comparative analysis. We conducted comparative analysis on the vulnerability analysis techniques presented in this

subsection based on their implementation method and runtime overhead. The results of comparison are summarized in Table II.

- *Implementation.* Most works discussed in this subsection also introduced tools that implement the proposed vulnerability analysis techniques. Most of these tools were implemented by the corresponding authors and have been made public under open-source licenses (See Table II). An exception is that, in the case of TaintAssembly [35], the proposed technique is implemented by modifying a JavaScript engine.
- *Runtime overhead.* We evaluated the performance of the surveyed techniques in terms of their runtime overhead. Note that the results of our analysis are based on the results of the evaluations that are available in the original papers. Table II lists the runtime overhead of each vulnerability analysis technique.

2) Protection from attacks: The types of attacks targeting Wasm binaries continue to increase in number because Wasm applications are widely applied in various environments other than web browsers. Thus, in this section, we introduce several techniques to protect Wasm from such attacks.

Ensuring constant-time programming. Wasm applications have become attractive targets for attackers who possess the ability to mount side-channel attacks [50]. To mitigate such attacks, several techniques that employ a constant-time programming approach have been proposed to reinforce the security of Wasm binaries. Constant-time programming is an approach that establishes a software-based defense mechanism to protect against side-channel attacks. It restricts secret-dependent control flow and memory operations in a program. There are several reports on the application of constant-time programming to Wasm binaries.

For example, Vivienne [42] is a verification scheme that checks whether a Wasm binary adheres to the constant-time principle. It applies relational symbolic execution for verification. More specifically, it performs verification by first passing the input values that include the same public values and different secret values, to two instances of programs; then, it evaluates the relational symbol execution result. If the two instances output the same result, then it verifies that the program adheres to the constant-time principle.

CT-Wasm [43] is a method that enhances the security of Wasm language such that it can be safely applied to cryptographic implementations. Specifically, Wasm has been reinforced by strictly restricting the flow of security information [51] under the constant-time programming principle [52]. This allows Wasm binaries to be protected from various timing attacks.

Protection from microarchitectural attacks. Recently, several studies on Wasm protection against microarchitectural attacks have been conducted.

Swivel [44] is a compiler-based protection method that provides protection against microarchitectural attacks that exploit conditional branch predictors, branch target buffers, and return stack buffers. Swivel comprises two implementations, i.e., Swivel software-based fault isolation (Swivel-SFI) and Swivel control-flow enforcement technology (Swivel-CET). Swivel-SFI lays out the basic blocks as linear blocks and each linear block secures against speculative execution. Then, it protects the backward edge by replacing the return instructions and using a separate return stack. Alternatively, Swivel-CET employs a hardware shadow stack to protect Wasm binaries.

Lurking Eyes [45] attempts to protect Wasm modules based on information about known cache side-channel vulnerabilities. In particular, it extracts JavaScript code from HTML data and determines whether the code snippet contains the same characteristics of the known vulnerabilities.

CROW [46] employs a code diversification approach to protect Wasm binaries from cache side-channel attacks. Specifically, it creates various binaries that have different execution flows and variables for the same application. The diversity of the Wasm binary hinders the ability of attackers to deliver cache side-channel attacks.

Protection from software attacks. Because Wasm applications are generally susceptible to a variety of software attacks, defense methods have also been devised to protect Wasm binaries from software-level attacks.

MS-Wasm [47] is a defense method that entails the use of dedicated hardware to ensure Wasm memory security. Specifically, this technique protects the memory in a Wasm module from attacks by restricting the access between the Wasm memory and other segments.

SELWasm [48] is a defense method that protects the execution code in a Wasm module by encrypting the code with cryptographic algorithms.

VeriWasm [49] is a verification method that validates the isolation of the memory in Wasm binary. It utilizes a static software-based fault-isolation verification technique that verifies the security of Wasm modules during compile time. In particular, VeriWasm creates a CFG based on the compiled Wasm binary and then analyzes the safety properties of each function to determine if the module is safe.

Comparative analysis. Here, we discuss the results of our comparative analysis of Wasm binary protection schemes. Note that we focused on comparing the primary protection approach and implementation method of the surveyed schemes. The results of our comparative analysis are summarized in Table III.

- *Protection approach.* Regarding the protection approaches, there are two general approaches to ensuring the security of Wasm. The first approach involves hardening the Wasm binaries to eliminate internal vulnerabilities. The second approach involves the real time and accurate detection of active attacks on Wasm modules. We have categorized various protection schemes based on these criteria, as can be seen in Table III.
- *Implementation method.* The results of our analysis revealed that most of the works presented in this subsection entail the application of custom-made tools to implement the proposed techniques; these tools have been made public under open-source licenses. Thus, these protection

Table III
Classification of WebAssembly binary protection schemes from attacks

Category	Scheme	Protection approach	Description	Implementation method	Open-sourced
Ensuring constant-time programming	Vivienne [42] (2021)	Hardening	Relational symbolic execution	Own tool	✓
	CT-Wasm [43] (2019)	Hardening	Use the flow of security information	Own tool	✓
Protection from micro-architectural attacks	Swivel [44] (2021)	Hardening	Software-based isolation, Control-flow enforcement technology	Own tool	✓
	Lurking Eyes [45] (2020)	Detection	Use of known vulnerability information	Own tool	✗
	CROW [46] (2021)	Hardening	Code diversification	Own tool	✓
Protection from software-level attacks	MS-Wasm [47] (2019)	Hardening	Use of known vulnerability information	Compiler modification	✗
	SELWasm [48] (2019)	Hardening	Code encryption/decryption	Own tool	✗
	VeriWasm [49] (2021)	Hardening	Software-based Fault Isolation	Own tool	✓

techniques can be easily applied to Wasm applications without any restrictions.

V. Open problems

In this section, we describe the limitations of existing Wasm binary protection methods and suggest future research directions.

A. Protection from malicious Wasm binaries

The surveyed schemes that are designed to protect against malicious Wasm applications has some limitations. In particular, the data set for evaluating the proposed approach is limited to specific algorithms used in cryptojacking attacks. This may cause difficulty in detecting attacks implemented with other algorithms. Thus, more extensive data sets should be used for the performance evaluation.

Furthermore, these schemes may be easily bypassed by code obfuscation techniques. Given that there are existing code obfuscation techniques for Wasm, such as Wobfuscator [53], it is necessary to develop techniques to detect malicious Wasm even if a code obfuscation technique is applied.

B. Hardening the security of Wasm binaries

1) Wasm binary vulnerability analysis: It is important to identify internal Wasm binary vulnerabilities before it is deployed to users. In particular, the most frequently compiled languages for Wasm binaries are C/C++, so it is necessary to check for vulnerabilities in advance. However, to date, there have been no investigations into internal Wasm binary vulnerability detection methods.

Static analysis-based methods utilize a taint analysis technique to identify internal Wasm binary vulnerabilities when the program is not being executed. However, its drawback is that its range of detection is limited to the range affected by the input value.

Alternatively, dynamic analysis-based methods use fuzzing to analyze Wasm binaries. However, current fuzzing techniques are only available for Wasm binaries compiled from C/C++. Thus, its limitation is that it is not possible to analyze the vulnerabilities in Wasm binaries compiled from other programming languages such as Rust or Python.

In addition, current research on Wasm binary vulnerability analysis tends to be biased toward web applications. Given that Wasm is currently being extended to various applications, it is necessary to correspondingly extend the research on Wasm vulnerability analysis.

2) Protecting Wasm binaries from attacks: As attacks against Wasm binaries continue to increase, some researchers have opted to focus on hardening the security of Wasm binaries.

Most of the existing techniques focus on Wasm binaries used in web environments. However, Wasm is currently being extended for application as an alternative for containers; this means that it is increasingly being employed as a standalone platform. Thus, research on attacks and countermeasures against Wasm applications used in various environments should be preceded.

VI. Conclusion

Wasm is beginning to be widely adopted in cloud computing, as it offers the promise of a high execution performance with better security. However, research to extend the applicability of Wasm to cloud computing is still in its early stage. Thus, security concerns related to the application of Wasm in a cloud environment have not been extensively investigated.

In this paper, we surveyed existing state-of-the-art techniques and solutions related to the security of Wasm binaries. We constructed a new taxonomy of Wasm security solutions and applied it to classify all the techniques proposed in the literature. We also discussed the results of qualitative and quantitative comparative analyses of these techniques. Moreover, we discussed the limitations of the current protection techniques and suggested future research directions. In particular, we discussed future research directions to enhance the security of Wasm binaries in cloud-native applications. We

believe that our survey constitutes a significant contribution to research on strengthening the security of Wasm binaries.

Acknowledgement

This research was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No.2020R1F1A1065539)

References

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

[2] A. Møller, "Technical perspective: Webassembly: A quiet revolution of the web," *Communications of the ACM*, vol. 61, no. 12, pp. 106–106, 2018.

[3] V. A. B. Pop, S. Virtanen, P. Sainio, and A. Niemi, "Secure migration of webassembly-based mobile agents between secure enclaves." University of Turku, 2021.

[4] 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.

[5] N. Mäkitalo, T. Mikkonen, C. Pautasso, V. Bankowski, P. Daubaris, R. Mikkola, and O. Beletski, "Webassembly modules as lightweight containers for liquid iot applications," in *International Conference on Web Engineering*. Springer, 2021, pp. 328–336.

[6] F. Scheidl, "Valent-blocks: Scalable high-performance compilation of webassembly bytecode for embedded systems," in *2020 International Conference on Computing, Electronics & Communications Engineering (iCCECE)*. IEEE, 2020, pp. 119–124.

[7] (2021) Cloud native webassembly applications are already here. [Online]. Available: https://thenewstack.io/cloud-native-webassembly-applications-are-already-here/

[8] (2019) Announcing lucet: Fastly's native webassembly compiler and runtime. [Online]. Available: https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime

[9] (2018) Webassembly on cloudflare workers. [Online]. Available: https://blog.cloudflare.com/webassembly-on-cloudflare-workers/

[10] 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.

[11] 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.

[12] Mozilla. (2014) asm.js. [Online]. Available: http://asmjs.org/

[13] A. Zakai, "Emscripten: an llvm-to-javascript compiler," in *Proceedings of the ACM International Conference companion on Object oriented programming systems languages and applications companion*, 2011, pp. 301–312.

[14] S. M. Jain, "Webassembly with rust and javascript: An introduction to wasm-bindgen," in *WebAssembly for Cloud*. Springer, 2022, pp. 57–85.

[15] Mozilla. (2019) Pyodide. [Online]. Available: https://pyodide.org/en/stable/

[16] B. Joshi, "Blazor," in *Beginning Database Programming Using ASP. NET Core 3*. Springer, 2019, pp. 337–380.

[17] A. Romano, X. Liu, Y. Kwon, and W. Wang, "An empirical study of bugs in webassembly compilers," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 42–54.

[18] (2017) Assemblyscript. [Online]. Available: https://www.assemblyscript.org/

[19] (2018) wasm-bindgen. [Online]. Available: https://github.com/rustwasm/wasm-bindgen

[20] F. Naseem, A. Aris, L. Babun, E. Tekiner, and S. Uluagac, "Minos: A lightweight real-time cryptojacking detection system," in *Annual Network and Distributed System Security Symposium(NDSS)*, 2021.

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

[22] R. K. Konoth, E. Vineti, V. Moonsamy, M. Lindorfer, C. Kruegel, H. Bos, and G. Vigna, "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*. ACM, 2018.

[23] 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*. ACM, 2018.

[24] 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*. ACM, 2019.

[25] C. Kelton, A. Balasubramanian, R. Raghavendra, and M. Srivatsa, "Browser-based deep behavioral detection of web cryptomining with coinspy," in *Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb)*. NDSS, 2020.

[26] W. Bian, W. Meng, and M. Zhang, "Minethrottle: Defending against wasm in-browser cryptojacking," in *Proceedings of The Web Conference 2020*. ACM, 2020.

[27] A. Kharraz, Z. Ma, P. Murley, C. Lever, J. Mason, A. Miller, N. Borisov, M. Antonakakis, and M. Bailey, "Outguard: Detecting in-browser covert cryptocurrency mining in the wild," in *The World Wide Web Conference*, 2019, pp. 840–852.

[28] W. Wang, B. Ferrell, X. Xu, K. W. Hamlen, and S. Hao, "Seismic: Secure in-lined script monitors for interrupting cryptojacks," vol. 11099 LNCS. Springer Verlag, 2018, pp. 122–142.

[29] Monero. (2017) Cryptonight. [Online]. Available: https://monerodocs.org/proof-of-work/cryptonight/

[30] Amazon. (2022) Alexa. [Online]. Available: https://www.alexa.com/topsites/

[31] (2017) Cryptoloot. [Online]. Available: https://crypto-loot.org/

[32] A. Szanto, T. Tamm, and A. Pagnoni, "Taint tracking for webassembly," *arXiv preprint arXiv:1807.08349*, 2018.

[33] P. D. R. Lopes, "Discovering vulnerabilities in webassembly with code property graphs." Técnico Lisboa, 2021.

[34] 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.

[35] W. Fu, R. Lin, and D. Inge, "Taintassembly: Taint-based information flow control tracking for webassembly," *arXiv preprint arXiv:1802.01050*, 2018.

[36] D. Lehmann, M. T. Torp, and M. Pradel, "Fuzzm: Finding memory bugs through binary-only instrumentation and fuzzing of webassembly," *arXiv preprint arXiv:2110.15433*, 2021.

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

[38] 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.

[39] P. Cousot and R. Cousot, "Modular static program analysis," in *International Conference on Compiler Construction*. Springer, 2002, pp. 159–179.

[40] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "Afl++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[41] (2019) Wasi. [Online]. Available: https://wasi.dev/

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

[43] C. Watt, J. Renner, N. Popescu, S. Cauligi, and D. Stefan, "Ct-wasm: type-driven secure cryptography for the web ecosystem," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[44] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen *et al.*, "Swivel: Hardening webassembly against spectre," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1433–1450.

[45] 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.

[46] J. Cabrera Arteaga, O. Floros, O. Vera Perez, B. Baudry, and M. Monperrus, "Crow: code diversification for webassembly," in *Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb)*. NDSS, 2021.
[47] M. Vassena and M. Patrignani, "Memory safety preservation for webassembly," *arXiv preprint arXiv:1910.09586*, 2019.
[48] J. Sun, D. Cao, X. Liu, Z. Zhao, W. Wang, X. Gong, and J. Zhang, "Selwasm: A code protection mechanism for webassembly," in *2019 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*. IEEE, 2019, pp. 1099–1106.
[49] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan, "Доверяй, но проверяй: Sfi safety for native-compiled wasm," in *Annual Network and Distributed System Security Symposium(NDSS)*, 2021.
[50] (2021) Memory safety in wasm. [Online]. Available: https://www.adservio.fr/post/memory-safety-in-webassembly
[51] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on selected areas in communications*, vol. 21, no. 1, pp. 5–19, 2003.
[52] G. Barthe, G. Betarte, J. Campo, C. Luna, and D. Pichardie, "System-level non-interference for constant-time cryptography," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 1267–1279.
[53] A. Romano, D. Lehmann, M. Pradel, and W. Wang, "Wobfuscator: Obfuscating javascript malware via opportunistic translation to webassembly," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022.