Received 27 September 2023, accepted 18 October 2023, date of publication 26 October 2023,
date of current version 10 November 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3327817

RESEARCH ARTICLE

metaSafer: A Technique to Detect Heap Metadata Corruption in WebAssembly

SUHYEON SONG[1], **SEONGHWAN PARK**[1], **AND DONGHYUN KWON**[2]
[1]Department of Information Convergence Engineering, Pusan National University, Geumjeong-gu, Busan 46241, Republic of Korea
[2]School of Computer Science and Engineering, Pusan National University, Geumjeong-gu, Busan 46241, Republic of Korea

Corresponding author: Donghyun Kwon (kwondh@pusan.ac.kr)

This work was supported in part by the National Research Foundation of Korea (NRF) grant funded by the Korea government [Ministry of Science and ICT (MSIT)] under Grant RS-2023-00217689, and in part by the MSIT, Korea, through the Information Technology Research Center (ITRC) support program supervised by the Institute for Information Communications Technology Planning Evaluation (IITP) under Grant IITP-2023-RS-2023-00259967.

ABSTRACT WebAssembly (Wasm), a technology enabling efficient native code execution in web browsers, has seen a significant rise in adoption as a popular compilation target. This has led to the emergence of lightweight web services powered by Wasm, characterized by their small binary size and reduced data transfer overhead, thanks to the inherent efficiency of Wasm. Despite their lightweight nature, these services can deliver powerful features like image/video processing, AI and graphical application that surpass the capabilities of JavaScript. To ensure lightweight web services and enhance the overall web experience, Wasm has been extensively optimized. However, these optimizations have raised concerns about memory safety, leading to memory-related vulnerabilities. Wasm's characteristic memory structure, linear memory, has vulnerabilities that provide various attack vectors to attackers. In particular, it presents various attack possibilities through metadata modification containing memory structure information. Attackers can exploit heap memory overflow in Wasm applications, allowing them to target arbitrary memory addresses, modify data, or execute arbitrary code. Such overflows can corrupt memory metadata, resulting in incorrect memory behavior. While research has mitigate memory-related weaknesses in languages such as C and C++ and architectures like X86 in recent decades, the direct application of security solutions designed for different domains to Wasm is not a practical approach. Consequently, allocators in Wasm remain vulnerable to issues like heap overflow and metadata corruption. Thus, there is a pressing need for tailored memory safety techniques and solutions that accommodate Wasm's architecture-agnostic and linear memory structures. In this paper, we propose metaSafer as a solution. By shadowing metadata from Wasm linear memory to JavaScript virtual machine memory and conducting metadata verification, metaSafer effectively blocks attack attempts and vectors. Notably, our solution achieves fast memory shadowing and validation while maintaining a small code size. Through various verification processes, we measured the performance and code size of metaSafer, revealing that it is a software-only security solution with no additional hardware requirements. metaSafer demonstrates robust metadata protection for Wasm applications with an acceptable performance overhead of up to 8% in SQLite speed tests and Polybench benchmarks.

INDEX TERMS WebAssemlby, javaScript, security, spatial memory safety, metadata corruption.

I. INTRODUCTION

As the web environment matures, recent web services are increasingly equipped with web applications that provide quality content, such as user interaction or visual graphics in the browser.

Accordingly, the WebAssembly(Wasm) [1] appeared in 2015. Rather than a web application driven by JavaScript, a high-performance and functional web application developed in a language that provides more diverse features and

The associate editor coordinating the review of this manuscript and approving it for publication was Yassine Maleh.

extensibility, such as C, C++, and RUST, is executed in the web environment leveraging a Wasm. It runs native code in the browser, and the execution speed is similar to native code. Therefore, users can enjoy advanced applications that provide various interactions and visual graphics through Wasm in a web browser without downloading or installing them on the local machine. In addition, there is an advantage of providing broad compatibility that allows Wasm applications to be used through a browser without depending on the architecture of the host machine or a specific runtime.

As a result of these advantages, Wasm is being used to create a wide variety of high-performance web applications, including games, video editing, and machine learning [2]. The significance of Wasm in the evolving web environment is that it allows developers to create high-performance applications that can run on the web without having to sacrifice security or portability. However, due to Wasm's optimized components, it lacks many security features. Therefore, various old security problems solved when performing in the existing legacy environment are reappearing in the Wasm environment [3]. By this problem, despite the short history of Wasm, many security and defense techniques studies are being conducted [4], [5], [6], [7], [8].

Since Wasm applications are distributed and executed through the web, the application size and performance are essential considerations [9]. To satisfy these requirements, Wasm applications often use simplified allocators such as emmalloc, which does not provide memory protection mechanisms or is used in limited coverage. In addition, Wasm's memory area consists of one large array, which is managed by a region using metadata. Therefore, due to the characteristics of these allocators and memory structures, linear memory is vulnerable to heap overflow attacks and allows arbitrary memory writes through attack vectors such as metadata corruption attacks. These vulnerabilities can lead to attack threats such as indirect control flow divert [10].

In this paper, we propose metaSafer, which saves metadata stored in the existing Wasm linear memory structure into JavaScript virtual machine area to prevent various attacks through metadata corruption. It shadows metadata to provide metadata access control and protection from a malfunction of the Wasm application.

metaSafer protects Wasm applications against attacks using metadata corruption through conventional Wasm linear memory overflow and, at the same time, guarantees execution time similar to that of existing Wasm applications through metadata handling optimization techniques. In addition, since the metaSafer prototype was implemented with a small code size of 7%, the existing advantages of Wasm can be maintained.

Contribution: The contributions of metaSafer can be summarized in three ways. First, metaSafer provides a robust defense against attacks targeting the metadata contained within Wasm linear memory. Second, metaSafer provides high-level metadata security while minimizing the increase in code size and providing optimized performance. Lastly, since the security features are provided through JavaScript, metaSafer has high compatibility that does not depend on a specific architecture and runtime environment.

- **Metadata protection** metaSafer is a novel security solution specifically designed to address the increasing threat of attacks targeting Wasm linear memory. In particular, metaSafer builds the defense from attacks that exploit metadata containing memory structure information. These types of attacks can be hazardous as they can result in the execution of arbitrary code and even enable an attacker to take control of a system. metaSafer mitigates these attacks by isolating and shadowing the metadata, making it much more difficult for attackers to exploit.
- **Optimization** metaSafer is designed to optimize the performance and minimize the code size increase when porting from specific allocator functions written in C to JavaScript that applied for metaSafer protection. Regarding code size, metaSafer efficiently and flexibly patches existing instructions through an API provided in the form of a library. More specifically, all metaSafer code is designed in statement units, maintaining a small code size even when all protection functions are applied throughout the application operation. So it detects metadata corruption and defends against various attacks with only an increase in file size of about 7%. This optimization minimizes the overhead incurred when distributing application packages. Regarding performance optimization, We introduce optimized metadata access between Wasm and JavaScript via the address encoder. Using an address encoder allows access to random addresses rather than a linear search method for metadata stored in VM memory. It enables faster metadata access in the JavaScript Virtual Machine and leads to overall performance improvement of Wasm application.
- **Compatibility** metaSafer is a software-only security solution, delivering platform-agnostic security features without the need for additional hardware extensions or any alterations to the Wasm runtime. And it does not require specific hardware extension and hardware based security features. This inherent adaptability enables "metaSafer" to seamlessly function across diverse environments, all without being tethered to any particular system configuration. Furthermore, the inclusion of all security functions directly within the compiled code guarantees a consistently high level of stability and reliability in its security features. Therefore, metaSafer ensures usability and applicability in various environments without compromising Wasm's scalability and compatibility features.

In conclusion, metaSafer represents a novel approach/ solution in addressing the metadata corruption within the WASM linear memory structure while effectively thwarting

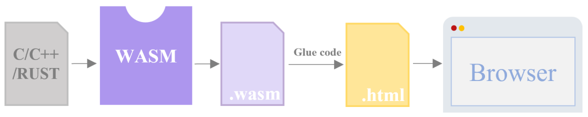

FIGURE 1. Overview of WebAssembly.

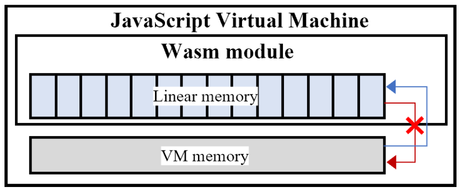

FIGURE 2. The Wasm module inside of JavaScript virtual machine and Wasm linear memory in Wasm module.

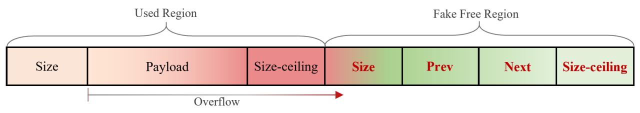

FIGURE 3. Overview of unlink exploit attack.

potential attack vectors stemming from malicious metadata manipulation. This software-only protective technique offers a significant advantage in its independence from hardware requirements. Delivered to developers as a library, metaSafer seamlessly integrates into existing systems without requiring runtime modifications. It achieves these deterministic security defenses on Wasm linear memory while incurring minimal performance overhead and maintaining a compact code footprint.

metaSafer is a formidable and adaptable solution tailored to combat a critical security vulnerability, memory-targeted attacks on Wasm-powered web applications. The significance of metaSafer lies in its potential to elevate the security standards for web-based systems, ensuring the integrity and reliability of these platforms in the face of evolving cybersecurity threats.

II. BACKGROUND

A. WEBASSEMBLY

The incorporation of JavaScript in the creation of web services has occasionally hindered the evolution of web-based services and applications, primarily due to the inherent constraints of JavaScript with regards to its functionality and performance [11]. It's important to note that many web services rely heavily on JavaScript for their implementation. However, this approach may not always be practical when compared to technologies like Wasm, which allows the development of web services using native source code languages such as C, C++, or Rust. This distinction highlights that Wasm provides a more versatile and efficient alternative for certain applications, addressing some of the limitations associated with JavaScript in web service development. As a result, the capabilities and speed of JavaScript have posed significant challenges and bottlenecks, slowing down the progress of creating web-based services and applications in certain contexts. Wasm enables codes written in C, C++ and Rust to be executed quickly and safely regardless of specific runtime environments such as hardware and platforms (Figure 1). Unlike JavaScript, Wasm is provided in a low-level bytecode format. It is executed in an isolated execution space for a safe execution environment. Nevertheless, it still provides a fast runtime execution speed similar to the native code execution speed. Since all contents and data for Wasm applications are transmitted through the network, minimizing the size of transmitted data is a vital optimization target factor that affects overall application performance and web experience [9]. By reducing the size of this code, the data transmission time and the time consumed for jit compile is reduced, which can affect actual application execution performance. In addition, components of Wasm are managed as modules, which include functions, globals, indirect call tables, and memories. The embedder, such as a JavaScript virtual machine or an operating system, provides the instantiation operation for modules. Due to this characteristic of Wasm, the embedder can access the module, but Wasm cannot access the embedder area according to the isolation policy. Figure 2 shows the state in which the Wasm memory module is included in JavaScript. Within the web application context, two types of memory are used; JavaScript virtual machine memory and Wasm linear memory. JavaScript can access Wasm memory through specific APIs (e.g., WebAssembly.Memory), but Wasm itself cannot directly access JavaScript memory due to security and isolation measures. In addition, Wasm memory is managed in the form of a linear array, which is a memory area of a single linear byte array specially designed to be used with Wasm applications. Although this linear memory enables efficient and fast memory access, it does not provide security functions such as memory access control and paging for the heap area within the linear memory, making it vulnerable to attacks such as buffer overflow (BoF) [12] and memory leak.

B. WASM MEMORY ALLOCATOR AND VULNERABILITIES

Wasm uses allocators to manage memory, and one commonly used allocator is "emmalloc," known for its lightweight design. Emmalloc is favored for Wasm applications due to its smaller source code, occupying only 1/3 the size of native allocators, resulting in more compact Wasm binary files. However, it is important to note that emmalloc lacks certain security features present in native allocators.

In emmalloc, memory is organized into regions, each containing metadata and payload. The metadata in a region includes information such as bottom and ceiling size, which helps determine whether the region is free or in use. When a region is allocated (in use), the payload is located between metadata. On the other hand, if a region is freed, its metadata

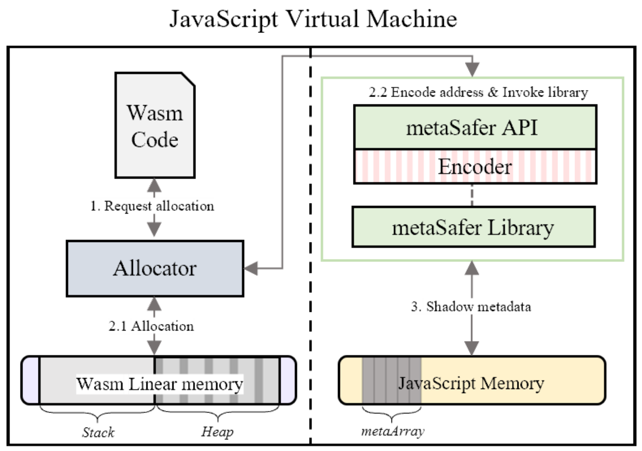

FIGURE 4. Overview of metaSafer API and library. The green box is the parts of metaSafer. We shadow the metadata from Wasm Linear memory to JavaScript memory which is trusted area.

contains pointers to the next and previous free regions, enabling efficient management of the free list. Figure 3 shows the data structure of these regions. This linked-list-based free list management poses a vulnerability, as Wasm linear memory is susceptible to metadata corruption attacks through heap overflow. Due to this specific characteristic of Wasm linear memory management, attackers can potentially exploit heap overflow vulnerabilities to tamper with metadata and the free list, leading to arbitrary memory writes and security breaches. Despite its advantages in terms of code size and compact Wasm binary files, the lack of robust security features in emmalloc raises concerns about memory safety. As Wasm applications are transmitted over networks and executed in various runtime environments, ensuring memory safety is crucial to protect against potential attacks.

In emmalloc allocator, vulnerabilities named unlink exploit [3] can be rise in wasm linear memory. And it shown in Figure 3. When memory chunks are deallocated using the free function, allocators attempt to consolidate adjacent free chunks into a single larger one to prevent fragmentation. In the event of data overflow in previous region, which might occur due to an incorrect length parameter in a memcpy operation, an attacker gains the ability to write to the metadata adjacent to following region. This manipulation involves clearing the used bit to free bit and creating a fabricated metadata. Subsequently, when previous region is freed, the allocator checks whether it can merge the newly freed chunk with an adjacent free chunk. As the tampered metadata(free bit) indicates that the following chunk is free, the allocator invokes removeFromFreeList to unlink it in preparation for merging. This enables the attacker to write an arbitrary value to an arbitrary address.

III. THREAT MODEL AND ASSUMPTION

In this paper, we make the assumption that Wasm binaries are distributed by a trusted entity, ensuring the integrity and authenticity of the files. We assume that these binaries are designed and implemented without intentional malicious code, adhering to standard security practices. However, the Wasm application executed on the client machine browser is vulnerable to memory safety issues, specifically targeted by metadata corruption attacks through heap overflow vulnerabilities. These vulnerabilities can potentially lead to control flow diversion [10] and arbitrary memory writes within the Wasm binary.

Furthermore, we assume that the client-side machines operate flawlessly and are regularly updated with the latest operating system and hardware driver patches. We rely on the correct functioning of the browser runtime, JavaScript virtual machine, and compiler, which collectively provide a secure execution environment for Wasm binaries

IV. DESIGN

metaSafer is a solution detects metadata corruption and prevents further attacks. It shadows each region's metadata located in Wasm linear memory to JavaScript virtual machine memory. And that shadowed metadata used as a criteria while verification for metadata integrity. This section describes the challenges of achieving the design of metaSafer.

A. DESIGN PRINCIPLE

To design and implement metaSafer, we follow three design principle. Solutions to each principle are described in the following sections.

- P1. Wasm provides access control to linear memory from external sources, based on the Wasm sandbox design. However, the current Wasm security policy does not provide access control between memory regions within the linear memory. Therefore interference between memory region should be prevented.
- P2. Security features should not have dependency on a specific runtime or environment. In other words, security features operating within Wasm should not affect Wasm's compatibility and versatility.To ensure environment-agnostic features, metaSafer security features should be provided irrespective of the host architecture and browser environment.
- P3. Security features should not significantly impact the ease of deployment and the performance of Wasm applications. To achieve this, the size of the Wasm deployment package should be kept to a minimum, and runtime performance should be optimized.

B. DESIGN OVERVIEW

Metadata can be tampered with through overflow in Wasm linear memory, allowing attackers to perform arbitrary memory accesses. To address this critical security factor, we designed metaSafer to protect the heap memory metadata, which serves as a key attack surface. Each region's metadata is shadowed into the JavaScript VM, acting as our trust anchor. This ensures that even if overflow occurs between adjacent memory areas in linear memory, any metadata

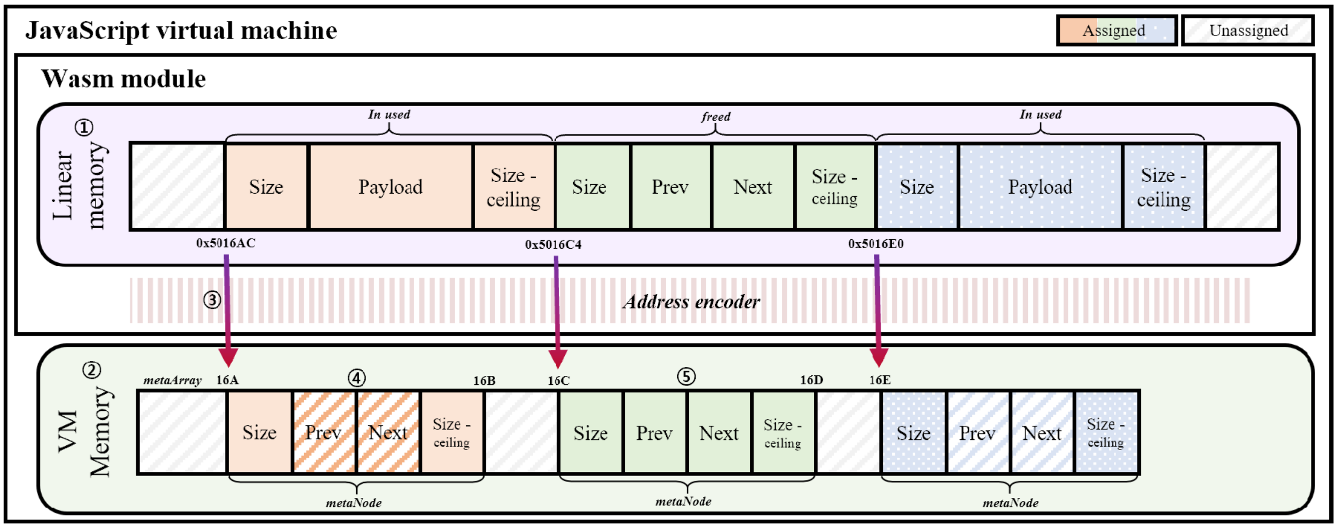

FIGURE 5. **Example of Wasm linear memory layout and JavaScript memory layout that applied metadata management mechanism of *metaSafer*. Upper side is the Wasm linear memory and the lower side is the JavaScript virtual machine memory(VM memory). Metadata shadowed in *metaNode* and the *metaNode* used as element of *metaArray*. Address encoder located between Wasm linear memory and VM memory and encode the memory address into *metaArray* array index value.**

corruption can be detected during verification using the shadowed metadata in the JavaScript VM (P1).

All security features and components of metaSafer are embedded in the Wasm binary during the compile stage. As a result, metaSafer is not bound to a specific runtime or host architecture, ensuring ease of distribution and compatibility across various environments. Moreover, metaSafer's protection features do not rely on hardware-specific features, such as Intel MPK or ARM Domain, which are only available in specific architectures (P2).

metaSafer facilitates communication between Wasm bytecode and the metaSafer library through APIs. These APIs include a setter and a validator, responsible for shadowing and verifying metadata, respectively. The shadowing and verification processes are conducted in statement units to minimize code size and avoid significant performance overhead (P3). Further details on this process are provided in Section VI-A.

Figure 4 shows the overview of metaSafer in JavaScript virtual machine. In compile time, the metaSafer API is applied on emmalloc allocator automatically. By doing this, metaSafer API located in Wasm module and the metaSafer library written in JavaScript operates on the JavaScript virtual machine area. ① In every requests related with memory allocation (malloc, free and etc), the Wasm application uses memory allocator, and ② every time the allocator performs any instruction that has metadata reference and writes(2.1), metaSafer API works with it(2.2). To shadow and validate the metadata, API calls the library and proceeds to address encoding and switch the world from Wasm to JavaScript. ③ metaSafer library shadow the metadata in the JavaScript memory, and in case of validation, the metaSafer library takes the metadata from JavaScript memory and proceeds validation process. All this metaSafer operation is based on the fact that the memory operation of the Wasm application is limited within its linear memory and cannot access JavaScript virtual machine memory directly. In other words, the Wasm application cannot modify the shadowed metadata.

C. EFFICIENT METADATA ACCESS

In the emmalloc allocator, accessing the metadata of the preceding region can be achieved through a pointer. To achieve a similar approach in metaSafer, this method is not applicable in JavaScript due to the lack of pointer support. To address this issue, we attempted to design a method using a linked list named freelist to manage a series of freed regions in JavaScript. However, this approach resulted in substantial search overhead due to linear search operations. To mitigate this issue and improve efficiency, metaSafer employs a different approach by handling metadata as an array in JavaScript, providing random access via index values.

Metadata is managed as an object of an array, with the address of each memory region transformed into an array index. This enables direct and random access to metadata without the need for linear search. To generate an array index from a memory region address, we designed an address encoder, the operation of which will be described in the following section.

For efficient metadata management and fast access to JavaScript shadowed metadata, we designed a *metaArray* data structure containing each region's metadata as element and we called each element as *metaNode*. *metaNode* has metadata as object. *metaArray* is stored and managed in the JavaScript virtual machine as an array. It's mechanism describes in next paragraph.

```
Encode address to index
    ENCODE_ADD(address) {
        index = (address - STACK_SIZE) >> 4
        return index
    }
```

FIGURE 6. Algorithm of address encoder. it subtract the stack size from the provided address and performing a right bit shift of 4.

```
static void unlink_from_free_list(Region *region)
    {
    % original code(1)
    region->prev->next = region->next;
    % original code(2)
    region->next->prev = region->prev;
    }
```

(a)

```
1       static void unlink_from_free_list(Region *region)
2           {
3           % check the integrity of the wasm memory metadata
4(+)        metaSafer_checkMeta(ENCODE_ADD(region), metaNext, \
            region->next);
5           % original code(1)
6           region->prev->next = region->next;
7           % update the metadata from WASM to JS
8(+)        metaSafer_setMeta(ENCODE_ADD(region->prev),
            metaNext, region->next);
9
10          % check the integrity of the wasm memory metadata
11(+)       metaSafer_checkMeta(ENCODE_ADD(region), metaPrev, \
            region->prev);
12          % original code(2)
13          region->next->prev = region->prev;
14          % update the metadata from WASM to JS
15(+)       metaSafer_setMeta(ENCODE_ADD(region->next), \
            metaPrev, region->prev);
16          }
```

(b)

FIGURE 7. (a) is original code snippet of emmalloc memory allocator and (b) is the patched emmalloc code with ***metaSafer***. The red code is the API of ***metaSafer*** and it encode the region memory address into ***metaArray***'s index and handover the arguments to ***metaSafer*** library.

Data Structure: The heap data area in Wasm linear memory is represented as an array buffer, allowing for random access to specific memory areas using address values. One approach to shadowing the corresponding array buffers into JavaScript involves using a specific memory address as a metadata array index, resulting in fast search times. However, this approach sacrifices memory efficiency, as it requires storing the entire array buffer in memory. On the other hand, storing only the metadata in the metadata array of JavaScript reduces wasted memory space. However, this approach comes with slower search times, as it requires traversing the entire metadata array to locate a specific memory region. Consequently, the overall performance of the Wasm application is significantly impacted.

To address these various challenges comprehensively, we have developed a solution that involves the design of a *metaNode* data structure, mirroring the existing region struct, and a corresponding *metaArray* to manage the shadowed metadata. Additionally, we have implemented an address encoder that converts specific addresses in Wasm linear memory into indices of the *metaArray*, enabling efficient random access to specific memory areas while minimizing array search time (P3). The *metaArray* resides in the JavaScript VM memory and its size is variable, dynamically increasing as the Wasm application allocates more memory regions. This scalable approach ensures optimal performance and adaptability to varying memory requirements.

Figure 5 shows the metadata management mechanism. Part ① is the series of memory regions allocated in Wasm linear memory, and the status of each region is divided into two; used and free. Part ② is the *metaArray* that has *metaNode* as an element. Each *metaNode* has metadata of each region and metadata is different depending on region status. Part ④ shows that the used region has two metadata; Size and size-ceiling. The free region (⑤) has four metadata; Size, Prev, Next, Size - ceiling. Part ③ shows how the address encoder works. the region address $0 \times 5016AC$ encoded into *16A*.

The **Encoder**, an essential component of the metaSafer API, plays a crucial role in facilitating efficient address encoding for metadata management. When processing address encoding, the metadata of each region in the Wasm linear memory is shadowed onto the *metaArray*, with the encoded region address value serving as the index to reference the corresponding *metaNode*. The address encoding process involves subtracting the stack size from the provided address value and performing a right bit shift of 4 (Figure 6). This adjustment accounts for the linear memory structure, ensuring proper alignment with 16-byte units for heap memory regions. The resulting value is then utilized as the index in the *metaArray* for the given region address. By employing this approach, metadata is stored efficiently, and faster access speeds are achieved compared to linear search. The optimized metadata access speed between Wasm and JavaScript significantly enhances the overall runtime performance of the system.

V. IMPLEMENTATION

A. METASAFER LIBRARY

This section describes the functions of the metaSafer API and library. APIs are consisted of *setter* and *validator* which have a role in calling the metaSafer libraries, and encoding the region address to index for *metaNode* using encoder. The libraries consist of *setter* and *validator* that write and validate metadata to *metaArray*.

To minimize changes to the existing behavior of emmalloc and minimize JavaScript code size overhead, instead of porting all allocator functions to JavaScript, a statement-level code modification that accesses JavaScript only when metadata is created and referenced was applied. Through this, the protection technique provided by metaSafer can be applied to all operations of the existing emmalloc with minimal code addition.

```
1  Write Metadata in JS VM Memory
2      metaSafer_setMeta(index, type, value)
3      {
4      % if there is no array for 'index', make new one
5          if(index = NULL){
6              metaArray[index] = new metaNode
7           }
8      % write metadata value to corresponding type
9      metaArray[index].type = value
10     }
11
12 Validate Metadata with shadowed metadata
13     metaSafer_checkMeta(index, type, value)
14     {
15     % retrieve metadata to corresponding type
16         JS meta = metaArray[index].type
17     % check whether the shadowed metadata(JS meta) is same
18       with WASM's metadata(value)
19         JS meta != value {
20             abort()
21         }
22     % return true if the metadata is safe and not forged
23     return true
24     }
```

FIGURE 8. Pseudo code of metaSafer library (setter and validator).

1) METADATA VALIDATION (ADDRESS, TYPE, VALUE)

Before the metadata reference operation(Figure 7 (b), line 6 and 13) for the current region, integrity is ensured by checking that metadata corruption or forgery(Figure 8, from line 15 to 21) has not occurred through verification(Figure 7 (b), line 4 and 11) of the metadata of the WASM linear memory and *metaArray*. In this process, the metadata of *metaArray* is accessed through the encoder in the same way as the setter. When accessing the metadata of the previous memory region(Figure 7 (b), line 6), metaSafer trust the metadata in the previous *metaNode* and validate the metadata of Wasm linear memory instead of backward search the whole *metaArray* with encoded previous region address(previous index). This approach, it guarantees a faster metadata validation time than a backward search of whole *metaArray* with the previous index.

2) METADATA WRITE (ADDRESS, TYPE, VALUE)

If all metadata is confirmed that there is no corruption on it, the original operation (Figure 7 (b), line 6 and 13) works normally and updated metadata will saved on VM memory (Figure 7 (b), line 8 and 15). The setter encodes the region address value and passes the metadata value and type to the metaSafer library located in JavaScript (Figure 7 (b), line 4 and 11). In the library (Figure 8), if *setter* does not have a *metaNode* with certain index, create a new *metaNode* (Figure 8, line 5 and 6) and save the received metadata (Figure 8, line 9).

B. DETECTION OF METADATA CORRUPTION

The Figure 9 illustrates a particular situation wherein the metadata associated with a subsequent unallocated memory region has been falsified as a result of a buffer overflow event that transpired within the payload of the preceding memory region. This incident, typified by a metadata corruption attack, such as an unlink exploit, has led to the tampering of metadata within Wasm linear memory space. It is important to note that under this attack attempt, the metadata resident in the VM Memory area remains unaffected by the overflow-induced metadata modification. Subsequently, when memory access or operations involving the compromised unallocated region (denoted as "2" in the figure) are executed, the metaSafer system employs its metadata validation mechanism to verify the integrity of the metadata. In the event of any tampering being detected, the executing process is promptly terminated.

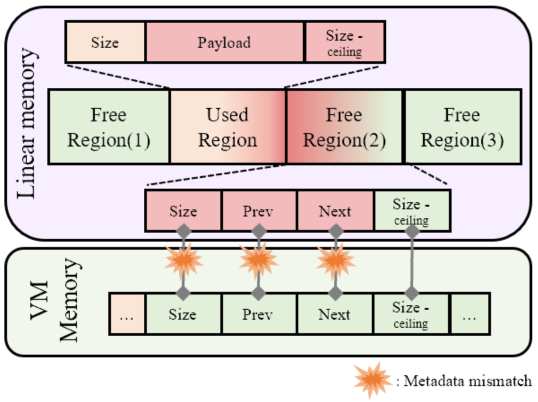

FIGURE 9. Illustration of metadata mismatch situation. Metadata saved in Linear memory are corrupted by overflow that occurred from the previous used region's payload. And it detected when metaSafer try to validate metadata using metadata saved in VM memory.

TABLE 1. Micro benchmarks of each metaSafer API. 'First execution' refers the runtime with compilation time. 'Single execution' shows the single execution of API function without compilation. Compilation is required only in first time of execution.

Type	First execution	Single execution
Setter	51.87us	0.41us
Validator	36.3us	0.41us
Encoder	4.22us	0.28us
Switch	10.36us	0.28us

VI. EVALUATION

In this section, we discuss the runtime performance and code size overhead of metaSafer compared with the emmalloc allocator.

A. ENVIRONMENT SETUP

We implement the benchmark on a system with a 10 core 20 thread Comet lake Intel i9-10900K (3.70Ghz and 5.30Ghz with turbo boost max technology) and DDR4-3200 64GB. The system runs with Ubuntu 18.04.6 LTS and Node JS v14.18.2. Our metaSafer applied on emmalloc lightweight memory allocator, and the benchmarks scenario is as follows; Micro Bench (subsection VI-B), which measures the performance of each API applied to metaSafer, the stress test (subsection VI-C) that measuring the execution time of single malloc and free functions and malloc benchmark provided by

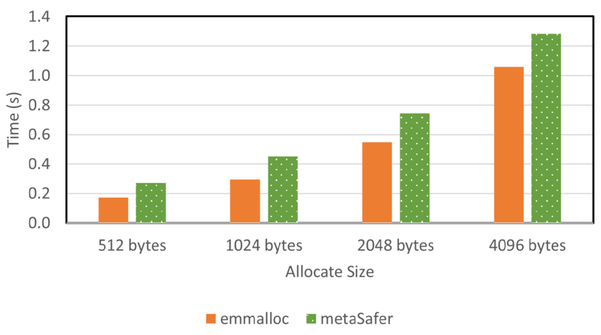

FIGURE 10. **Stress test of emmalloc and metaSafer. The orange solid bar graph is test execution time of emmalloc and dotted green bar is for metaSafer. The graph represents how much overhead occur under heavy loads of memory allocation and free.**

TABLE 2. **This table represents the execution time of single memory allcocation(malloc) and free instruction in each allocator. The 'First' includes instruction compilation time and operation and 'Single' is the execution only time of each instruction.**

	malloc		free	
	First	Single	First	Single
emmalloc	6.41us	0.49us	4.34us	0.30us
metaSafer	89.37us	4.73us	35.32us	1.03us

emsdk [13]. For the Macro benchmarks (subsection VI-D), PolyBench/C 4.2.1 [14] are used and SQLite [15], [16], [17] is used for real world application test case (subsection VI-E).

B. MICRO EXECUTION OVERHEAD

In this section, we measure the individual performance of the APIs provided by metaSafer and the time spent on world switching between Wasm and JavaScript. Measured APIs are setters, validators, and encoders. Wasm Binary takes the JIT (Just In Time) compile method, translating bytecode into machine code at execution. Therefore, the code executed for the first time includes compilation time on their execution time. From the second round onward, compile time is not required.

In Table 1, 'First execution' is an execution time that includes both compile time and single API execution time. 'Single execution' shows the single execution time of each API. It calculated on average value after executing a single API 10,000 times consecutively. Single execution time represents the execution speed in an environment where the same API is repeatedly executed. This table proves that each API and world switching does not make a significant performance overhead during application runtime. The case of *validator* consists of a read and comparison(validation) operation. And in the case of the *setter*, only the memory write operation is performed. In the first execution, the *validator* is faster than *setter*. The setter takes time to make *mateNode* in JavaScript memory in the first time of saving the metadata.

C. STRESS TEST

This section compares metaSafer and emmalloc allocator through stress tests. The stress test measures the overall stability and performance of the allocator in a heavy-load environment through repeated malloc and free instruction execution. The stress test demonstrates the allocation efficiency of metaSafer through a runtime performance comparison with emmalloc. Stress conditions are as follows; We malloc and free the four maximum memory allocation sizes (512, 1024, 2048, and 4096 bytes) 100,000 times each. The minimum allocation size set is 16 bytes. It also shows the time for a single run of malloc and free through each allocator. We additionally used the following compilation options to cope with the significantly increasing memory usage. '-sALLOW_GROWTH_MEMORY'

In the Figure 10, metaSafer tends to be slower than emmalloc. metaSafer shows a minimum of 20% and a maximum 57% runtime overhead compared to emmalloc. This shows that runtime overhead can occur under the extreme number of malloc and free operation. As an additional insight, the performance overhead incurred decreases as the MAX size increases. Over 50% overhead was presented in 512bytes allocation, but as the MAX size was increased to 4096bytes, the overhead was reduced to 20%.

In the Table 2, metaSafer has about nine times of performance overhead compared to emmalloc in the malloc, and the free shows about three times of runtime overhead. Like the APIs, malloc and free must also be compiled in the first execution, so additional runtime is consumed for the first malloc and free. We note that single malloc and free execution with metaSafer have huge overhead compared with emmalloc allocators, but as it is performed repeatedly, the runtime overhead gradually decreases.

D. MACRO BENCHMARKS

This section evaluates performance and memory allocation efficiency through macro benchmarks using Polybench/C. The polybench/C (v4.2.1 beta) written in C compiled using emscripten (v3.1.22) [13] to use emmalloc and metaSafer allocator and perform each test using Wasm.

The graph Figure 11 shows the 30 test results of two allocators performed PolyBench/C. Emmalloc shows a slightly faster execution time than metaSafer. metaSafer shows a maximum 2% execution time overhead compared with emmalloc. More specifically, more than 1% overhead has occurred in bicg, burbin, jacobi-1d, and trisolv, but metaSafer shows an average 0.43% of overhead in all test cases. These test results suggest that the execution time overhead of metaSafer is insignificant in arithmetic-intensive test environments.

E. REAL WORLD APPLICATION

To perform a realistic scenario, we performed a benchmark through SQLite. SQLite is an embedded relational database management system (RDBMS) designed to be lightweight, and it compiles to WASM and can be run in the browser. We used Firefox (v110.0.1) to run the benchmark. This measures the runtime overhead in real applications caused by metaSafer. The database format used

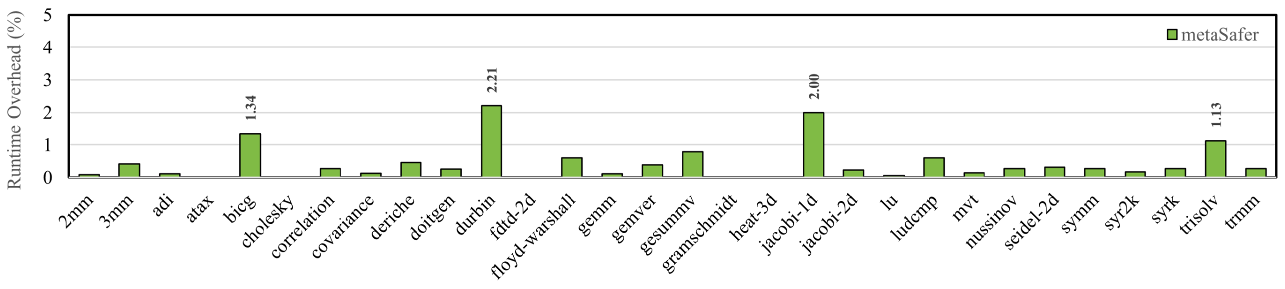

FIGURE 11. This graph shows the result of Polybench benchmarks. The green solid bar represents the metaSafer execution runtime overhead based on emmalloc(baseline) execution time. In durbin test case, metaSafer required 2.21 percent more time to run than emmalloc.

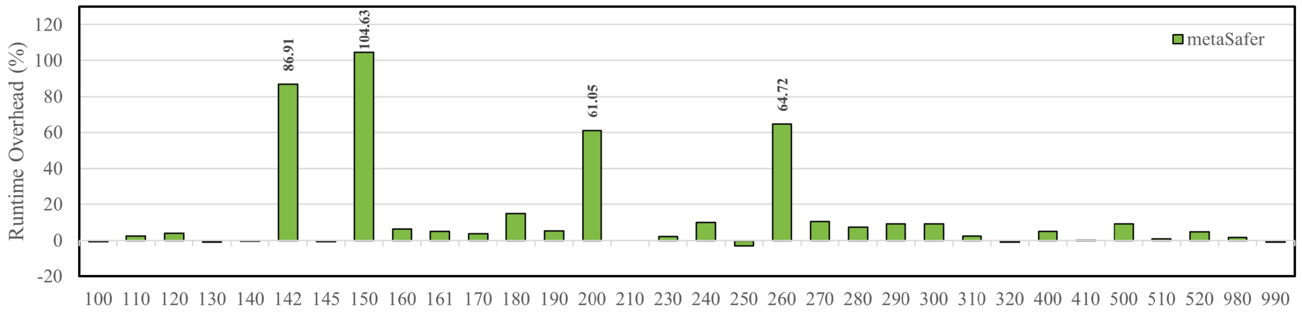

FIGURE 12. This graph is about runtime overhead in SQLite which required over thousand of memory operations. The green solid bar is runtime overhead of metaSafer and shows 8% of overhead in total.

by SQLite benchmark is memfs. The Memfs is a virtual file system that tests and simulates file system operations in Wasm applications. Through this, consistent and reliable benchmark results can be obtained without being affected by the I/O delay of physical storage and hardware performance instability. The compilation options used for Wasm are '-sALLOW_GROWTH_MEMORY' and '-O3'. With these options, Wasm linear memory can be additionally increased in size when required.

The tests are mainly database updates (table creation, insert, update, delete), and its result shows in Figure 12. In experiment 260 (Query added column after filling), metaSafer is about four times slower than emmalloc. This means that when a new column is added, SQLite needs to dynamically allocate additional memory while adding the new column's data. Experiment 200 runs the VACUUM command, which compresses and optimizes files. It reclaims unused space, rearranges data, and frees up memory space. While running this task, metaSafer incurs additional performance overhead. The remaining 142, 150 experiments generate additional indexes and do data collation. The number of mallocs and frees used in the total SQLite benchmark is 6499 and 6865, respectively. Also, realloc, which changes the size of a memory region, was used 587 times. Emmalloc took 151.33 seconds for the total execution, and metaSafer took 163.58 seconds, showing 8% slower performance. This indicates the lower impact of runtime overhead in more general applications than in brutal situations like the stress test in the previous section.

F. CODE SIZE

As mentioned in the previous section, it is crucial to minimize the size of the application package delivered over the network due to the characteristics of Wasm applications used in the web environment. This section compares the size of the compiled Wasm and JavaScript that applied various allocator.

The graph (Figure 13) shows the difference in file size created by compiling metaSafer and emmalloc, respectively. Optimization levels were set to -O2 and -O3. In a previous study, there was a difference in the final Wasm file size according to compiler flags (ref), which also showed a difference in metaSafer's file size measurement. -O2 option is a moderate level of optimization which is less aggressive optimization with faster code execution than -O3. metaSafer's JavaScript has a size of 947 bytes at -O2 and 915 bytes at -O3. This means that the compiled file size of the Wasm application with metaSafer applied is 947 bytes larger than the existing compiled file size without metaSafer. Then metaSafer's Wasm code has a size of 1259 bytes in -O2 and 1600 bytes in -O3. Finally, metaSafer brings a total file size addition of 2206 bytes in the -O2 optimization scenario and 2515 bytes in the -O3 optimization scenario. In other words, the effect of metaSafer's code size is variable depending on the source size and scale of the application. 7% code size increase due to metaSafer in compiled Polybench/C code. But in SQLite, metaSafer takes only 0.15% extra size. The absolute size of this metaSafer may vary depending on the version of the compiler used, the compilation options selected, and other user environments. We used emscripten

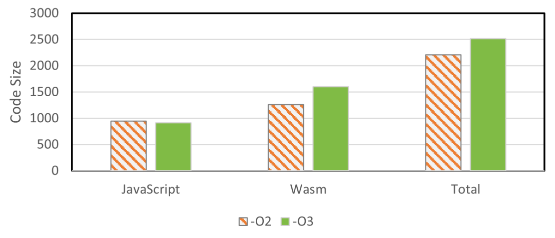

FIGURE 13. Compiled code size of metaSafer. Added code size are fixed under same optimization level. -O2 and -O3 refers the optimization level 2 and 3 respectively.

(v3.1.22) with '-sALLOW_GROWTH_MEMORY' and -O2, -O3 respectively to compile metaSafer.

Summary: metaSafer tends to be slower than emmalloc, but has an acceptable runtime overhead of at least 1% and at most 10% in macro benches and applications, excluding stress tests. It is reasonable overheads considering that it provides security features not offered by existing lightweight allocators. We analyzed the effect of each API on the execution time through micro-benchmarks and proved the metaSafer shows a small level of runtime overhead even the API is repeatedly used. Despite providing these security features and low overhead, metaSafer still proves to be a practical allocator with a small size.

VII. DISCUSSION

In this section, we discuss the future work and the expandability of metaSafer strategy. metaSafer provides security features with JavaScript in the browser. It has the potential to provide extended usability and enhanced performance with the following extra work.

A. SUPPORT STANDALONE RUNTIME

metaSafer relies on the memory space within the JavaScript VM running within the browser as a trust anchor. However, it is important to note that JavaScript is not supported in the Wasm standalone runtime, which is commonly used in embedded devices or environments running Wasm applications without a browser. As a result, using the metaSafer library based on JavaScript is not feasible in such scenarios. To apply metaSafer in a standalone runtime environment, significant modifications to the Wasm standalone runtime at a specific low level are necessary. Currently, there are at least 10 different WASM standalone runtimes [18], [19], [20], [21], [22], each with its unique implementation. Ensuring full compatibility and providing the same level of security as metaSafer would require substantial engineering efforts and time. While porting metaSafer to a standalone runtime environment could reduce the code size of Wasm applications and minimize performance overhead, it may come at the cost of compatibility due to the constraints of the limited runtime environment.

B. METADATA MIGRATION

metaSafer detects metadata corruption in linear memory through metadata validation, relying on the shadowed metadata in JavaScript VM memory for the validation process. However, we can extend metaSafer with an additional feature to defend against metadata corruption attempts by fully migrating the metadata stored in linear memory to JavaScript VM memory and isolating it from the Wasm application. This would involve comprehensive changes to the memory structure in Wasm and the data structure of the allocator. By adopting this approach, metadata would be isolated from Wasm linear memory, thereby increasing the availability of Wasm heap memory. As it would eliminate the need for metadata validation, this approach has the potential to provide slightly faster runtime speeds with the same level of security as the current metaSafer. However, one drawback of this method is that linear memory integrity verification through metadata becomes impossible, which could potentially expose linear memory to data-only attacks.

VIII. RELATED WORK

A. WASM SECURITY

In the past five years since WASM was released, many previous studies have been conducted in terms of memory safety and Control Flow Integrity (CFI) of WASM [4], [5], [6], [7], [8]. CFI is a security mechanism crucial for protecting computer programs from control flow attacks that can compromise software integrity and security. It enforces strict rules on program execution paths, reducing vulnerabilities and preventing unauthorized code execution. In order to guarantee the integrity of WASM applications, there are studies that run WASM in an environment where a trusted platform is supported. To ensure the utmost integrity of Wasm applications, research endeavors have focused on executing Wasm within a shielded environment empowered by a trusted platform, explicitly referring to a Trusted Execution Environment (TEE) like Intel Software Guard Extensions (SGX) or ARM TrustZone. TEE provides secure enclaves within hardware architecture, such as SGX's isolated compartments, safeguarding critical code and data from unauthorized access, software vulnerabilities, and hardware threats. AccTEE [23] executes WASM binary compiled for a specific target system in Intel's SGX enclaves to protect critical operations in unsafe OS or host systems. TWINE [24] runs a lightweight Wasm virtual machine in a trusted execution environment and supports WASI (WebAssembly system interface) so that existing WASM applications can be executed without recompilation. WaTZ [25] guarantees the integrity of the application code through remote attestation of the Wasm application binary in the ARM-based trusted execution environment. However, the above studies partially limit the versatility of WASM because there are requirements for additional hardware (x86-based and limited CPU range) that support a trusted execution environment. In addition, it does not provide the detection or prevention of attack threats and CFI integrity that may occur during Wasm application execution. These previous studies have greatly contributed to improving the security of WASM, but the purpose of each study [23], [24], [25] is different from the security of linear memory protected by metaSafer.

B. MEMORY HARDENING

There are previous studies to enhance memory safety. VIP [26] protects heap metadata through a virtual address with Intel MPK and builds a defense against an attack that tampers the CFI or executes arbitrary code that leverages metadata corruption vulnerabilities and heap overflow. This protection methodology relied on the x86 architecture. The AddressSanitizer(ASan) [27] instruments the program's memory accesses to check for violations of the heap red zones; the area outside of the dedicated memory region. The memory metadata is located in the heap red zone. Through this, it implements heap memory protection. This is the software-only memory protection approach that provides higher compatibility than using hardware extension. However, extra memory-bound check instructions must be added to every memory access attempt. And it leads to an increase in runtime performance overhead. These previous methodologies relied heavily on specific hardware or made high runtime performance overhead while implementing memory safety techniques. Therefore it is hard to apply this methodology to Wasm directly. The ASLR(Address Space Layout Randomization) [28] is a critical security measure that protects against memory-based attacks by randomizing the location of code and data in memory. However, Wasm currently lacks support for ASLR, a critical security feature employed by traditional systems. Even if ASLR were to be introduced to Wasm in the future, the limited memory size of Wasm (32-bit) would still provide insufficient entropy for effective randomization, potentially leaving vulnerabilities exposed. In contrast, metaSafer offers deterministic protection, a marked improvement over ASLR's probabilistic protection. Deterministic metadata protection in a software-only implementation is crucial as it provides a reliable and predictable defense against metadata attacks, reducing the risk of both false positives and false negatives compared to probabilistic protection methods like ASLR. With metaSafer, the predictability and reliability of metadata protection are significantly enhanced, as it ensures consistent and reliable safeguarding of sensitive information.

metaSafer adopts a software-only approach which does not rely on Intel SGX and MPK. It relies on the JavaScript virtual machine's memory as a trust anchor, thereby circumventing the need for hardware-based protection methodologies. This approach successfully mitigates issues related to context or world switching, thanks to Wasm sharing memory with JavaScript. This symbiotic relationship facilitates rapid data transmission and ensures that our software-only implementation maintains exceptional compatibility while keeping performance overhead to a minimum. The metaSafer methodology represents a novel approach aimed at identifying instances of metadata corruption within the Wasm environment. Its primary objective differs from methodologies [26], [27], [28] utilized in other domains and environments(x86) or those reliant on hardware-based protection mechanisms. Since Wasm takes a memory access method through metadata in a unique structure called linear memory, this paper suggests the need for a new protection technique suitable for Wasm linear memory, rather than protection that reuses existing protection techniques.

IX. CONCLUSION

In conclusion, this study proposes metaSafer, a robust security solution that effectively defends against attacks targeting Wasm linear memory by shadowing metadata into JavaScript VM memory, thereby ensuring metadata integrity and detecting corruption. The performance overhead of metaSafer has been rigorously verified through various tests, including micro benchmarks, stress tests, macro benchmarks, and real-world application testbenches. The results demonstrate that metaSafer introduces only a small file size overhead and shows performance impact ranging from 1% to 8%, making it a highly efficient and practical security solution for Wasm applications.

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 *Proc. 38th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2017, pp. 185–200.

[2] S. Bano and S. Khalid, "BERT-based extractive text summarization of scholarly articles: A novel architecture," in *Proc. Int. Conf. Artif. Intell. Things (ICAIoT)*, Dec. 2022, pp. 1–5.

[3] D. Lehmann, J. Kinder, and M. Pradel, "Everything old is new again: Binary security of WebAssembly," in *Proc. 29th USENIX Conf. Secur. Symp.*, 2020, pp. 217–234.

[4] B. McFadden, T. Lukasiewicz, J. Dileo, and J. Engler, "Security chasms of WASM," Austin, TX, USA, NCC Group White Paper, 2018, vol. 1.0.

[5] A. Hilbig, D. Lehmann, and M. Pradel, "An empirical study of real-world WebAssembly binaries: Security, languages, use cases," in *Proc. Web Conf.*, Apr. 2021, pp. 2696–2708.

[6] Q. Stiévenart and C. D. Roover, "Compositional information flow analysis for WebAssembly programs," in *Proc. IEEE 20th Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, Sep. 2020, pp. 13–24.

[7] Q. Stiévenart, C. De Roover, and M. Ghafari, "Security risks of porting C programs to WebAssembly," in *Proc. 37th ACM/SIGAPP Symp. Appl. Comput.*, Apr. 2022, pp. 1713–1722.

[8] Q. Stiévenart, C. De Roover, and M. Ghafari, "The security risk of lacking compiler protection in WebAssembly," in *Proc. IEEE 21st Int. Conf. Softw. Quality, Rel. Secur. (QRS)*, Dec. 2021, pp. 132–139.

[9] J. C. Arteaga, S. Donde, J. Gu, O. Floros, L. Satabin, B. Baudry, and M. Monperrus, "Superoptimization of WebAssembly bytecode," in *Proc. Conf. Companion 4th Int. Conf. Art, Sci., Eng. Program.*, Mar. 2020, pp. 36–40.

[10] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM Comput. Surv.*, vol. 50, no. 1, pp. 1–33, 2017.

[11] S. Khalid, S. Khusro, and I. Ullah, "Crawling AJAX-based web applications: Evolution and state-of-the-art," *Malaysian J. Comput. Sci.*, vol. 31, no. 1, pp. 35–47, Jan. 2018.

[12] A. One, "Smashing the stack for fun and profit," *Phrack*, vol. 7, no. 49, pp. 14–16, Nov. 1996.

[13] (Mar. 12, 2023). *Emscripten*. [Online]. Available: https://github.com/emscripten-core/emsdk

[14] (Mar. 8, 2023). *Polybench/C Official Site Address*. [Online]. Available: http://www.ohio-state.edu

[15] (Mar. 12, 2023). *Sqlite*. [Online]. Available: https://www.sqlite.org/index.html

[16] S. Bhosale, T. Patil, and P. Patil, "Sqlite: Light database system," *Int. J. Comput. Sci. Mob. Comput*, vol. 44, no. 4, pp. 882–885, 2015.
[17] S. Ashraf, T. Ahmed, Z. Aslam, D. Muhammad, A. Yahya, and M. Shuaeeb, "Depuration? Based efficient coverage mechanism for? Wireless sensor network," *J. Electr. Comput. Eng. Innov.*, vol. 8, no. 2, pp. 145–160, 2020.
[18] *Wasmer*. Accessed: Mar. 14, 2023. [Online]. Available: https://wasmer.io/
[19] *Wasmtime*. Accessed: Mar. 14, 2023. [Online]. Available: https://wasmtime.dev/
[20] *Wavm*. Accessed: Mar. 14, 2023. [Online]. Available: https://github.com/WAVM/WAVM
[21] *Life*. Accessed: Mar. 14, 2023. [Online]. Available: https://github.com/perlin-network/life
[22] *Lucet*. Accessed: Mar. 14, 2023. [Online]. Available: https://www.fastly.com/products/edge-compute/runtime/
[23] D. Goltzsche, M. Nieke, T. Knauth, and R. Kapitza, "AccTEE: A WebAssembly-based two-way sandbox for trusted resource accounting," in *Proc. 20th Int. Middleware Conf.*, Dec. 2019, pp. 123–135.
[24] J. Ménétrey, M. Pasin, P. Felber, and V. Schiavoni, "Twine: An embedded trusted runtime for WebAssembly," in *Proc. IEEE 37th Int. Conf. Data Eng. (ICDE)*, Apr. 2021, pp. 205–216.
[25] J. Ménétrey, M. Pasin, P. Felber, and V. Schiavoni, "WaTZ: A trusted WebAssembly runtime environment with remote attestation for TrustZone," in *Proc. IEEE 42nd Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2022, pp. 1177–1189.
[26] M. Ismail, J. Yom, C. Jelesnianski, Y. Jang, and C. Min, "VIP: Safeguard value invariant property for thwarting critical memory corruption attacks," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2021, pp. 1612–1626.
[27] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *Proc. USENIX Annu. Tech. Conf.*, 2012, pp. 309–318.
[28] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proc. 11th ACM Conf. Comput. Commun. Secur.*, Oct. 2004, pp. 298–307.

SUHYEON SONG received the B.S. degree in computing systems from the Unitec Institute of Technology, New Zealand, in 2021. He is currently pursuing the master's degree in computer engineering with Pusan National University, Republic of Korea. His research interests include computer system security and ARM TrustZone.

SEONGHWAN PARK received the B.S. degree in computer engineering from Dongseo University, South Korea, in 2021. He is currently pursuing the Ph.D. degree with Pusan National University, Busan, Republic of Korea. His research interests include system security and H/W architecture.

DONGHYUN KWON received the B.S. and Ph.D. degrees in electrical and computer engineering from Seoul National University, South Korea, in 2012 and 2019, respectively. He is currently a Professor with the School of Computer Science and Engineering, Pusan National University, South Korea. His research interest includes system security against various types of threats.