OmniWasm: Efficient, Granular Fault Isolation and Control-Flow Integrity for Arm Microcontrollers

Maorui Bai
The George Washington University
mbai@gwu.edu

Runyu Pan
Shandong University
rypan@sdu.edu.cn

Gabriel Parmer
The George Washington University
gparmer@gwu.edu

***Abstract*—Traditional embedded systems often ignore security, instead focusing on simplicity. Unfortunately, increasingly pervasive network connectivity exposes these systems to malicious input, and the consolidation of code of various qualities and sources onto single systems increases the chance of errant behavior. Microcontrollers do not often have advanced security facilities to help prevent malicious threats – even the use of memory protection to constrain the ill-effects of a compromise is not pervasive. Software techniques to provide strong security properties for application execution often focus on limiting accessible memory through dynamic checks on memory accesses through software fault isolation (SFI), and on ensuring that software cannot suffer from control-flow hijack attacks through control-flow integrity (CFI). This paper introduces OmniWasm, which provides sandboxes in which applications execute that provide both SFI and CFI. OmniWasm focuses on two core contributions. First, it reduces the overhead of these techniques by using a novel application of common, but obscure memory operations to both limit sandboxed memory accesses (SFI) using hardware, and allow memory accesses outside the sandbox to enable CFI. Second, it focuses on enabling embedded software to be decomposed into multiple sandboxes by providing optimized communication facilities between them that avoid thread context switching overheads and providing single-copy message passing. We show that the overheads of OmniWasm are better than existing software address validation techniques while also providing better memory utilization. We also show that OmniWasm enables significant performance gains for inter-sandbox communication across varying message sizes.**

I. Introduction

Security has often been an after-thought in embedded systems. In embedded systems that are *closed* – those that are not exposed to network traffic, and running only approved and integrated code, security techniques are often seen as only increasing overheads and complexity. However, modern systems are increasingly directly attached to, and interact with networks. Vehicles are connected to the cloud, infrastructure, and each other (in V2X), industrial plants are integrated into analytics-driven control loops (Industry 4.0), and embedded systems that are part of the Internet of Things (IoT) interact with each other and users through the Internet. Even the smallest embedded systems must assume that they are exposed to potentially malicious communication over the Internet. At the same time, the code that executes on these systems is increasingly open-source, complex, and generally more difficult to certify for the absence of bugs that are potentially exploitable.

This material is based upon work supported by the National Science Foundation under Grants No. CPS 1837382, CNS 1815690, and by the Office of Naval Research under N000142212084. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF nor ONR.

Microcontrollers provide the low-power, inexpensive computation in many of these embedded domains. Arm Cortex-M microcontrollers are popular and have between low tens to hundreds of MHz, with tens to low hundreds of KiB of SRAM. Unfortunately, these systems do not have many of the features that promote security such as $W \oplus X$, MMUs, or Pointer Authentication (PA) [1] to provide protection against shell-code, address virtualization and protection, or control-flow integrity, respectively. The combination of an increasing exposure to potentially malicious network input, and lack of hardware security features motivates new approaches to provide strong security on these pervasive systems.

Software-based approaches to adding security to legacy code can bridge this gap. Software Fault Isolation (SFI) [2], [3], [4] adds dynamic checks into existing C code, that cause access exceptions to memory outside of an application's *sandbox*. This prevents potentially compromised or errant code from accessing memory outside of the application. Control-Flow Integrity [5], [6] techniques ensure that the control flow of an application follows only the prescribed paths intended by the compiler and developer. This prevents control-flow hijacking attacks like stack smashing and Return-oriented Programming (RoP) [7], [8].

WebAssembly [9] (Wasm) is an abstract assembly language that is low-level enough that C code can compile to it – for example, it does not require garbage collection, enables pointers, pointer arithmetic, stack allocation of memory. A Wasm runtime provides strong SFI and CFI properties, thus sandboxing the code within. Wasm is an output target for the LLVM and clang C/C++ compiler. It has been used in embedded systems [10], [11], [12], [13] to sandbox existing C code.

A key mechanism used by Wasm runtimes to provide SFI is *bounds checks* on memory accesses. To ensure that an application sandbox accesses only the set of memory it has been allocated, dynamic checks on pointer dereferences trigger software exceptions if outside of the sandbox's memory. These checks are often optimized to remove conditional branches, instead masking addresses to keep them within a power-of-two amount of sandbox memory. eWasm [10] investigates various implementations and found that they present an unfortunate trade-off: waste memory with wrapping-based pointer updates, or suffer more overhead for explicit condition-based bounds checking.

Microcontrollers often provide Memory Protection Units (MPUs) that explicitly specify a set of memory ranges that user-level execution can access without exception. While

MPUs seem ideal for preventing memory accesses outside of a sandbox, this is complicated as most CFI runtimes require interleaved memory instructions for accessing the runtime's data-structures with application memory accesses within the sandbox memory. Without gratuitous mode switching, the MPU cannot both prevent access beyond the sandbox's memory, *and* allow the runtime's data-structures.

OmniWasm provides a Wasm runtime that both avoids the overheads of bounds checking and is memory efficient. As such, it avoids the trade-offs between performance and memory of previous implementations. To do so we leverage little known, but commonly deployed, load and store instructions that can be executed in kernel mode, but that perform MPU checks as if the load/store were made from user mode. Importantly, there are no algorithmic changes made when compiling C into Wasm sandbox, executable code, so predictable C code maintains bounded execution times.

In sandboxing code, OmniWasm also enables the fine-grained isolation between different software applications. It enables system software to be decomposed into multiple, smaller sandboxes, and to efficiently communicate between them. This enables controlling the impact of faulty execution to subsets of the system, that can then be independently rebooted. To maintain strong isolation, we copy messages between sandboxes, but optimize the system to require a single copy compared to conventional RTOS communication APIs. Similarly, the Wasm model enables multiple sandboxes to execute sequentially in the same thread, which also optimizes IPC overhead, and saves stack memory.

A key property of OmniWasm's Wasm implementation is that it maintains key real-time and latency properties of embedded code, while enabling stronger security and reliability. Specifically, C code that provides bounded execution times, maintains bounded execution times in the output assembly – though overheads might result in latencies increasing (see §V-A). The relatively low-level mechanisms of Wasm avoid the complicated lookup structures or garbage collection common in language virtual machines that can introduce unpredictability. By maintaining the bounded execution times of C code down to the generated assembly, OmniWasm establishes a foundation for secure real-time execution.

Given the increasing software complexity, and exposure to networks of modern embedded systems, techniques to increase software security are necessary. OmniWasm represents a significant innovation in Wasm compilers and runtimes for microcontrollers by providing efficient execution and efficient use of memory, while providing optimized IPC to enable decomposing system software into fault domains.

Contributions. OmniWasm's contributions include:

- **HW/SW co-design for efficient SFI/CFI.** We design and implement a new mechanism for performing bounds-checks for sandboxes. It uses instructions that are little-used, but pervasively deployed on Arm Cortex-M microcontrollers to enable the MPU to validate that pointer dereferences are within a sandbox.
- **Efficient inter-sandbox communication.** To aid in decomposing the system into various sandboxes, each of which can fail independently, we introduce a novel IPC infrastructure that is optimized to enable single-copy message passing, and stack sharing.
- **Wasm sandbox evaluation.** We also provide a benchmark and application-driven evaluation of our techniques compared to native C code, and eWasm. We demonstrate that though there is overhead for strong SFI and CFI, we improve over existing techniques.

II. Background

In this paper, we propose a hardware bounds-checking mechanism that provides efficient memory sandboxing for Wasm running on microcontrollers. This section focuses on background for two technologies integral to this mechanism: Wasm and MPUs.

A. WebAssembly

Wasm is an abstract assembly language [14] designed to run across various platforms at speeds comparable to native execution that is not tied to any specific language. Unlike native code execution, a key property of Wasm code is that it is executed in a *sandbox*. Wasm employs both software fault isolation (SFI) and control flow integrity (CFI), thus a Wasm runtime provides a secure environment for running untrusted code. Importantly, both SFI and CFI are provided in a Wasm runtime using simple mechanisms that add only constant overheads to low-level assembly operations (*e.g.* loads and stores, or function pointer invocation). Thus, code compiled to Wasm that has bounded execution times, will maintain a bounded execution time, though latency might increase due to overheads. Figure 1 displays the Wasm sandbox and runtime layout.

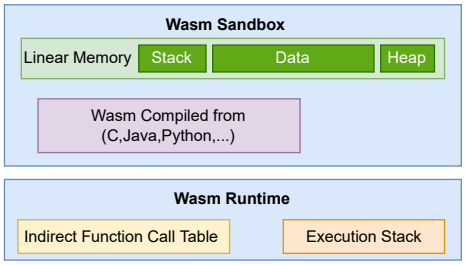

Fig. 1: Wasm Sandbox and Runtime Layout

Memory Safety via Software Fault Isolation. Each Wasm sandbox accesses a range of memory, represented as a contiguous array of bytes known as the "linear memory". This linear memory is separate from the memory of other Wasm sandboxes and from the system's memory. Through SFI, load and store memory operations performed by a sandbox are restricted to be only within its linear memory. A Wasm compiler and runtime, which implements the semantics of the Wasm specification, ensures the safety of a sandbox's memory accesses by inserting bounds checks. These prevent out-of-bounds memory accesses, such as buffer overflows, accessing memory outside of linear memory.

Control Flow Integrity. CFI is a security property that prevents attackers from altering program control flow beyond what was intended by the programmer and compiler. The

Sandbox module requires CFI to ensure that no bugs can be leveraged to modify the program's control flow. To guarantee CFI, Wasm uses a data stack and an indirect function call table.

- *Separate data and execution Stack.* When the execution stack in C holds data (e.g. arrays), there is a vulnerability risk: stack smashing [7] can index past stack-allocated array bounds, and overwrite instruction-pointer return values. Thus, upon return, an attacker can hijack control flow. To counteract this, Wasm separates the execution stack that tracks function invocations and return addresses, from a data stacks that hold addressable data that is placed inside linear memory. The normal execution stack includes only values to which code does not have an address, including function call return addresses. As a result, sandboxed code cannot programmatically modify the execution stack, thus preventing stack smashing attacks.
- *Indirect function call table.* Another way that attackers hijack a program's control flow is by overwriting function pointers or vtable entries, redirecting them to addresses of malicious code or to sequences of Return-Oriented Programming (ROP) [8] gadgets. Wasm prevents function pointers from taking arbitrary values using a typed, indirect function call table. The table is *outside* of linear memory, thus not modifiable by sandboxed code. Each entry in this table points to a function, initialized at compile time (from a valid function address). Function pointers in the Wasm code are *offsets* into this table, instead of direct addresses. When a call through a function pointer is made, the function to call is looked up in the call table using the index. The Wasm runtime performs two checks. First, the runtime checks that the provided index is within the valid range of the function table. Second, if the index is valid, the runtime checks that the function at that index in the function table matches the expected type signatures. This function type signature check can prevent the attacker from calling a function with incorrect arguments or exploiting type conversion vulnerabilities. This support enables even low-level code (*e.g.* C) to safely avoid function pointer control-flow hijacking attacks.

B. eWasm: an Efficient, Embedded Wasm Implementation

Memory bounds-checking is a crucial aspect of memory sandboxing, aiming to prevent out-of-bounds memory accesses. Software bounds-checking needs to add additional code instructions to validate every memory access operation. Figure 2 illustrates how eWasm distributes the Wasm sandbox data.

Software Bounds Checking. eWasm [10] provides three separate bounds-checking mechanisms:

- *Condition-based bounds-checking.* This method inserts naive condition-based bounds checks on every linear memory load and store. For example, conditional checking for an address `a`, with a given linear memory bound (`lm_bound`):
  `if (a >= lm_bound) exception();`

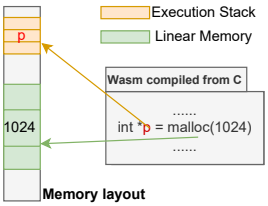

Fig. 2: eWasm's implementation of Wasm sandboxes splits sandbox data across the stack and linear memory. It ensures that data that can be indexed into (thus requires checks to ensure that the index is within valid ranges) is allocated into linear memory. Software bounds checks are added to each load and store to linear memory to prevent memory accesses outside the sandbox. Stack-allocated variables that are not indexed, along with control flow meta-data (e.g. function return addresses) are tracked in the normal execution stack, identified with the stack pointer.

- *Masking-based sandboxing.* This approach applies a bitmask to offsets during each linear memory load and store to prevent accesses beyond power-of-two sized linear memory. Assuming that `lm_bound` is a power-of-two:
  `a = a & (lm_bound - 1);`
- *Software page-tables.* This technique enables non-contiguous linear memory and necessitates a page lookup before every memory load and store.

Note that the examples above are simplified and do not consider the size of the data retrieved at the address `a`.

While masking-based sandboxing demonstrates the best performance – being only 1.5x slower than native execution – it carries a significant drawback. Specifically, it can lead to up to 100% linear memory wastage, averaging 50%. In contrast, condition-based bounds-checking and software page-tables methods, which do not result in linear memory wastage, lag in performance, operating at 2x and 4x the native speed, respectively. This presents a clear trade-off in software bounds-checking: to achieve enhanced performance, one may need to compromise on memory efficiency and vice versa. Beyond this trade-off, software bounds-checking inherently introduces performance overhead by necessitating additional code for each load and store operation.

C. MPUs for Hardware Memory Protection

MPUs are hardware units that are commonly featured on microcontrollers. Unlike Memory Management Units (MMUs) that provide address virtualization (*i.e.* translation from virtual to physical addresses), MPUs focus only on subsetting the memory accessible to executing code.

A load or store to an address outside of the MPU-defined accessible set, results in a processor exception, thus enabling the RTOS to handle erroneous accesses. This has been used in RTOSes [15], [16] to provide minimal process abstractions that provide memory protection by limiting the memory accessible to different tasks.

To restrict memory access, the MPU divides physical memory into several regions, each defined by its start address, size, access permissions, and attributes. While the access permissions and attributes of a region dictate a region's

accessibility status, its size and start address delineate the region's range.

MPU-supported region quantities and sizes depend on specific processor designs. In Arm architectures, Armv7-m accommodates up to 8 regions. The size of each region in these architectures must be a power of two, and its start address needs to align with the region's size. In contrast, the Armv8-m architecture can support 16 or more regions, and it allows regions of varied sizes without the strict power-of-two constraint. Therefore, the configuration of region numbers, size, and start address of Armv7-m's MPU varies from that of Armv8-m's MPU. We will discuss this further in the implementation section.

While newer Arm architectures may introduce more advanced attributes for the MPU, a number of fundamental attributes remain consistent across Arm architectures. A key attribute of the MPU region is the access level that tells whether the region is accessible in user mode or privileged modes. The access permissions of the MPU region can be read, write, or execute.

III. Design

The core of our contribution is the HW/SW co-design for processor and memory efficient sandbox bounds checking, and IPC that enables the system to be decomposed into multiple sandboxes. This section will discuss OmniWasm's design.

A. Using the MPU for Bounds-Checking

In the preceding section, we examined three software bounds-checking techniques employed in eWasm. These software mechanisms impose a trade-off between performance and memory utilization, offering no avenue to optimize for both CPU and memory resources. Furthermore, these methods introduce performance overhead due to the insertion of additional, explicit bounds-checking instructions into the code. In the case of address wrapping approaches, sandbox memory must be rounded up to a power of two, decreasing utilization.

Consider a strawman sandbox runtime that avoids this trade-off by using an MPU. It includes a monitor that is responsible for maintaining the integrity of sandboxed Wasm code by enforcing bounds checks and maintaining data structures to ensure CFI. Using a hardware MPU potentially offers strict enforcement of linear memory boundaries, ensuring that the sandboxed Wasm code cannot overstep its allocated memory. However, the MPU limits *all* load and store instructions, including those of the monitor's data structures. The monitor requires data-structures *outside* of the sandbox to maintain the sandbox's integrity (including, for example, the execution stack and indirect function pointer table). As a result, the sandbox monitor faces a dilemma: if it uses the MPU for isolation, it must temporarily alter MPU settings to access these data structures. Practically, this means effectively making mode transitions for each function call, which is a slower solution than a simple conditional bounds check.

In contrast, OmniWasm's goal is to leverage MPU-based bounds check so that software can elide bounds-checking instructions, while still enabling the sandboxing logic to directly access its runtime state. To achieve this, we extend the isolation mechanisms of the eWasm compiler and runtime to use "unprivileged load and store" instructions on popular Arm chips.

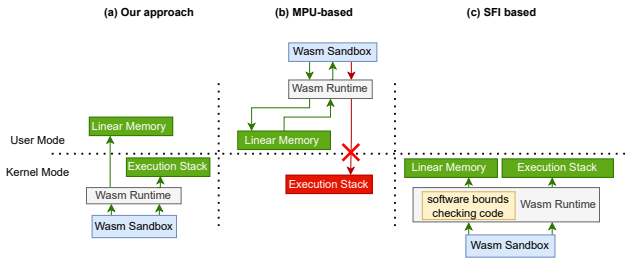

Fig. 3: Comparison between OmniWasm, MPU-based, and SFI bounds-checking approaches in Wasm.

The Arm architecture manages memory access through standard load and store instructions. Using normal instructions, once the MPU is configured to limit the linear memory access to user mode execution, a sandbox monitor that is operating in user mode can only perform load and store operations on this linear memory. Direct load and store operations on the monitor's data structures that are outside of linear memory are disallowed. To side-step this limitation, OmniWasm uses *unprivileged load and store* instructions.

Arm architecture versions v5, v6, v7, and v8, as well as the microcontroller profiles v7m and v8m, support *unprivileged load and store* instructions – `ldrt` and `strt` (or `ldtr` and `sttr` in AArch64 v8a). These instructions ensure that memory access operations are performed as if from user mode, regardless whether the processor is currently in user mode or kernel mode. We leverage these instructions to execute *MPU-restricted* linear memory accesses while enabling standard load and store instructions to directly access privileged data structures. In this setup, the unprivileged load and store instructions allow the sandbox monitor to operate in kernel mode, ensuring fast access to privileged data structures by avoiding mode switches, while still enforcing that Wasm logic can only interact with the linear memory, thereby providing SFI. Figure 3 compares this approach with two other bounds-checking approaches.

SFI memory isolation properties in OmniWasm translate program loads and stores directly into unprivileged loads and stores. By utilizing these unprivileged load and store instructions, OmniWasm uses direct access to memory without the need for frequent mode switches, lookups, or additional bounds-checking code. Since this introduces no additional, unpredictable overheads, C code with a bounded execution time should maintain bounded execution times in OmniWasm thus integrating cleanly into existing real-time systems.

B. RTOS Support for Multi-Sandbox Systems

Unlike traditional operating systems with features like virtual machines and containers for sandboxing, RTOS environments often lack these sandboxing options due to hardware resource constraints. Most RTOSs employ IPC mechanisms to support task communication, and temporal isolation by

decoupling task's execution. However, these IPC mechanisms imply significant overheads. These often include messages being copied twice through intermediate buffers, and thread communication necessitating context switching. In this paper, we introduce a solution that aims to reduce both context-switching and memory copy overheads without compromising task memory isolation and security features within an RTOS environment.

OmniWasm sandboxes execute preemptively. Thus, when executed in separate RTOS threads, sandboxes execute independently. While Wasm has atomic instruction extensions for multi-threaded sandbox execution *within* a sandbox, eWasm and OmniWasm do not yet support them. Thus, OmniWasm's current sandboxes are single-threaded.

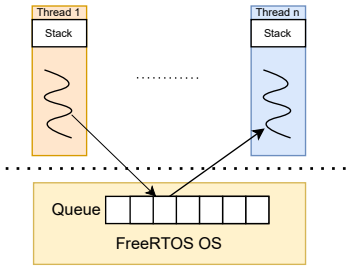

Fig. 4: FreeRTOS IPC communication between threads

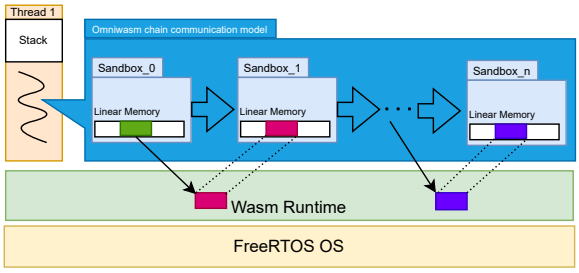

Fig. 5: Wasm chain model for IPC.

Figure 4 shows the traditional IPC communication between threads using traditional RTOS mechanisms (*e.g.* in FreeRTOS). Given the concurrent execution of communicating threads, an intermediate buffer is often used to maintain independent execution of each thread. In contrast, Figure 5 shows OmniWasm's IPC mechanism for communication between Wasm sandboxes. Our design is based on the assumption that sandboxes in a chain of communication execute sequentially based on message dependencies. We focus on the isolation of memory and logic between each sandbox, and not on concurrency for sandboxes in a chain. This assumption is natural given the communication dependencies intrinsic in chains of processing, but it does limit concurrency. Should temporal isolation between sandboxes be required, the Wasm runtime can leverage timer interrupts to detect overruns.

To limit a sandbox's communication to only with allowed, down- and up-stream sandboxes, we provide a capability-based API [17]. With this, a sandbox specifies the capability it wishes to send or receive messages from, and the runtime maps that capability to the target sandbox.

Sandbox Context Switching. In Figure 4, switching between threads requires RTOS overheads such as scheduling and saving and restoring registers. In contrast, in Figure 5, Wasm modules are self-isolating and they all execute sequentially (in chain order) in a *single* host thread. The runtime is able to switch between sandboxes effectively with function calls within a single thread, thereby reducing the overhead associated with thread context switching and stack management. This has an additional benefit in memory-restricted embedded systems of requiring allocations only for a single thread with a stack requirement of the maximum stack usage of any its chained sandboxes.

Inter-sandbox Communication for Chain-based IPC. Most RTOSs support communication methods including mailboxes and queues [18]. While mailboxes are designed for single messages, queues facilitate the transmission of multiple messages. Both mechanisms generally involve two memory copy operations through an intermediate buffer.

Inter-sandbox communication in our system uses chain-based IPC primitives explicitly provided by OmniWasm's runtime. In this chain model, data flows sequentially from one sandbox into the next, a common pattern in embedded systems used for subsequent stages of sensor processing. Each stage processes the incoming data and passes its own output to the subsequent stage. The first and last stages act as the source and sink, respectively, while the intermediate stages both send and receive messages. When a sandbox wants to transmit a message, the OmniWasm runtime directly allocates space within the receiving sandbox's own memory for that message, and copies it directly from transmitting buffer to the freshly allocated memory. This approach eliminates the need for a second memory copy through the RTOS buffers while maintaining strong inter-sandbox isolation.

OmniWasm focuses on low and bounded latency communication between sandboxes, while also bounding memory consumption for messages. Latency is optimized by processing various sandbox stages in a chain sequentially in a single (RTOS) thread, while message buffers are directly pre-allocated in sandbox memory, enabling direct transfer between sandboxes. The capabilities of each sandbox limit their communication only to the next, intended sandbox in the chain. This sequential, chain-driven execution and direct message transfer, combined with bounded execution of OmniWasm sandbox code, enable end-to-end, bounded execution of chains of isolated computations.

C. Threat Model and Security Analysis

OmniWasm's goal is to provide strong SFI and CFI to prevent sandboxed code from impacting code or data outside the sandbox using mechanisms aside from defined communication channels. Our threat model is inherited from WebAssembly. Unlike managed runtimes that require strong type-safety, Wasm aims to support even low-level C code, thus enables the creation of arbitrary pointers. As a consequence of this, conventional attempts to threaten data integrity or to hijack control flow of the sandbox can be attempted. These include attempting to jump to malicious dynamic function pointers and to utilizing buffer overflow attacks to manipulate values held in the stack.

We assume that the code provided by the application is WebAssembly (*e.g.* compiled from C into Wasm). At the Wasm level, not all operations available in C are expressible. This makes it impossible (given Wasm limitations) to express other mechanisms for control flow hijacking including manipulating computed gotos and switch statements based on jump tables. Further, executable logic specified in Wasm *cannot* include native ISA assembly instructions as it is limited to the Wasm instruction-set. To ensure these properties, all Wasm code is explicitly validated.

Security Analysis. While arbitrary addresses can be synthesized in a sandbox, the WebAssembly execution model prevents attempts to load or store memory outside of linear memory, instead raising a runtime exception. Stack-allocated data (in C) that can be accessed through a pointer, is allocated in the data stack *within linear memory*. This is separate from the execution stack, and ensures that any buffer overflows in stack-allocated data are subject to the normal linear memory access checks. To prevent (potentially malicious) function pointers from hijacking control, runtime code validates the function before jumping to it. Fundamentally, while sandboxed code is allowed to directly modify linear memory, the Wasm execution model ensures that addresses used in control-flow (function return addresses, and function pointers) are *outside of linear memory*, thus not corruptible by sandboxed code. In eWasm, memory accesses are limited to linear-memory via bounds checks, the data stack is allocated in linear memory, and the function pointer table is allocated statically by the compiler.

The key to understanding OmniWasm's security properties is that we aim to maintain the Wasm execution model. First, OmniWasm must limit sandboxed loads and stores to the allowed linear memory range. While existing approaches use software bounds checks or wrapping logic, OmniWasm uses unprivileged loads and stores confined by the MPU to the linear memory range. Sandboxed code is prevented from accessing memory using normal (kernel mode) instructions that would not be MPU-protected as the OmniWasm compiler generates only unprivileged memory instructions corresponding to Wasm load and store instructions. The monitor ensures that the unprivileged load and stores are properly MPU-constrained by loading the MPU before executing the sandbox and relying on the RTOS to save/restore MPU context on thread switches.

We additionally consider complexities due to the execution of sandboxed code in *kernel mode*. The sandboxed logic is prevented from executing privileged instructions – or accessing CPU state that could threaten sandbox integrity (*e.g.* MPU registers) – due to (1) the code's inability to express assembly code in validated Wasm – preventing direct execution of instructions not generated by the OmniWasm compiler, (2) the inability to access memory-mapped CPU structures that are outside of linear memory – prevented by the MPU-limited unprivileged memory instructions, and (3) the CFI properties that prevent the sandbox from generating new executable instructions (*e.g.* shell code). OmniWasm inherits the indirect function pointer table logic, and execution and data stack separation from eWasm, thus maintaining their key CFI properties.

OmniWasm is designed to maintain the strong SFI and CFI guarantees of the Wasm execution model, while limiting resource consumption, and integrating into an existing RTOS environment. WebAssembly's focus is on sandboxing code, not on minimizing the system's Trusted Computing Base (TCB). The code generated by the eWasm compiler, the OmniWasm extensions to it, the monitor's runtime, the FreeRTOS-based OS, and the hardware are all required to be correct to maintain SFI and CFI properties. As such, these all constitute the TCB necessary to sandbox applications.

IV. Implementation

Our implementation of MPU-based bounds checks and single copy IPC in OmniWasm is a series of compiler and runtime extensions of the publicly accessible eWasm [10] project [19]. eWasm provides a compiler named aWsm [19] that converts Wasm code to LLVM binary code and a wasm runtime that maintains sandboxed isolation. We use the WASI-SDK [20] to generate Wasm abstract assembly code from the source application. aWsm compiles that Wasm code into LLVM-IR, and we then use LLVM to generate the Arm target binary. By default, Wasm linear memory is comprised of pages with a default size of 64KB [9]. This means that all linear memory sizes are rounded up to the nearest page size. In a resource constraints microcontroller system, this can lead to a prohibitive amount of memory waste (due to internal fragmentation). Correspondingly, we modified the default page size in WASI-SDK to 1KB, thus ensuring that the WASI-SDK compiler is now capable of generating Wasm code that understands the significantly smaller page size, thus decreasing memory internal fragmentation.

Similar to eWasm, we assume that each sandbox has a known maximum linear memory size. Thus we statically allocate the linear-memory for each, an assumption that is reasonable and often necessary in many resource-constrained, embedded systems.

A. MPU-based Bounds Checking

The MPU-based bounds-checking implementation consists of two parts: MPU configuration/programming and the use of unprivileged memory load and store instructions. Programming the MPU hardware includes setting a specific region number to have a specific region base address, region size, and region permission. The MPU memory region in Armv6-m and Armv7-m architectures must be aligned to a base address that is a multiple of the region size, and must be a power of two. Thus, we must calculate the base address and size of all regions that are necessary to cover the linear memory which is not guaranteed to have a power-of-two length, nor be aligned on a power-of-two base address.

Figure 6 shows an example of how the MPU is configured to perform bounds-checking for linear memory. We have a Wasm application with a 67KiB linear memory that has a start address at `0x20000000`. This Wasm application, as an example, tries to load the value both inside and outside of linear memory. When the OmniWasm runtime executes the

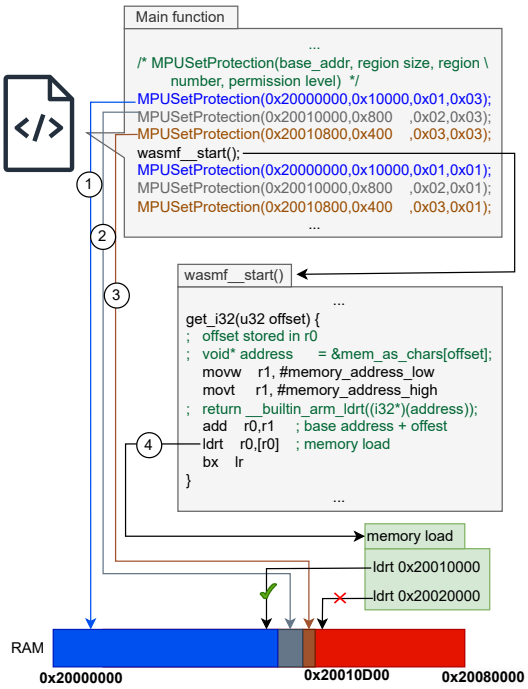

Fig. 6: The setup and execution of MPU bounds-checking method for a sandbox with linear memory that spans between `0x20000000` and is of size `0x10D00`.

sandbox, it must first load the MPU. First, (1), we configure the first MPU region with a base address that is equal to the start address and a region size of 64KB. Second, (2), we configure the second MPU region with a base address that starts from the end of the first MPU region and a region size of 2KB. Third, (3), we configure the MPU region number three with a base address that starts from the end of the second MPU region and a region size of 1KB. All three MPU regions convey both read and write permissions (denoted with permission level `0x3`). As the OmniWasm runtime is executing in kernel-mode, normal loads and stores are not limited by the MPU.

After the configuration phase, we start to run the Wasm program's main function wasmf__start(). The OmniWasm compiler ensures that all memory loads are performed using the logic summarized in `get_i32(...)`, which uses the `ldrt` instruction. This is the unprivileged load instruction which proceeds as if made from user mode execution. The memory accesses of the program are shown in (4), which includes example loads from memory addresses `0x20010000` and `0x20020000`. Since the first memory address is inside the memory range, the hardware MPU check implicit in the `ldrt` instruction allows the access. However, the second memory address is outside of the allowed ranges, thus the MPU's checks during the `ldrt` will trigger a memory fault.

Computing MPU Configurations. The linear memory can be mapped into N MPU regions, which must be equal to or less than the maximum number of MPU regions that are supported by the hardware. For each MPU region, we assign the MPU region size and MPU region base address.

The configuration is determined by iteratively assigning regions that each cover the largest power-of-two amount of linear memory, leaving a smaller uncovered portion which is considered in the next iteration. In this paper, R is a sequence of MPU region sizes ($\forall_{r_i \in R} r_i \times 2 = r_{i+1}$), M is linear memory size that is not yet covered by MPU regions, $\{A_x, 1 < x < N\}$ represents the size of MPU region x, and $\{B_x, 1 < x < N\}$ is the base address of MPU region x. The formula below shows how to find elements of R to cover M

$$\exists r_j \in R : r_j \leq M < r_{j+1}, \quad A_x = r_j \tag{1}$$

Now uncovered linear memory size is $M = M - r_j$. Here we iterate until either $x > N$ or $M = r_j$. $x > N$ implies that the linear memory cannot be fully protected by MPU regions. $M = r_j$ indicates that the linear memory is accurately protected by the assigned MPU regions. With this logic, and with 8 regions and a minimum page size of $1KiB$, we should be able to cover any sandbox size less than $2^N KiB$ or $256KiB$.

Assuming that the MPU region base address is aligned to the MPU region size, and a sandbox's linear memory base is `BASE`, which must be aligned on an A_1 address, the formula used to calculate each region's base is simple, contiguously assign each region after the previous:

$$B_x = \begin{cases} \texttt{BASE} & \text{if } x = 1 \\ B_{x-1} + A_x & \text{if } 1 < x \leq N \end{cases} \tag{2}$$

Unprivileged Load and Store Instruction Utilization. Neither GCC nor LLVM support unprivileged memory access instructions (*e.g.* `ldrt`) by default, leading to an absence of optimizer support. Given this challenge, we have incorporated our optimization techniques in the implementation. However, unlike the regular load and store instructions that support multiple addressing modes, the unprivileged load and store instructions are constrained to a single addressing mode, resulting in suboptimal performance. As a result, this single addressing mode restriction can lead to performance that is less optimal compared to the regular load and store instructions that support multiple addressing modes.

OmniWasm uses an extended version of LLVM. We investigate two options to enable LLVM to support these instructions. We can use either simple inlined assembly expressions in the runtime, or attempt to add intrinsic functions in LLVM. The linear memory load and store operations in eWasm are defined in C code, so using inlined assembly is a trivial extension, but inline assembly blocks are not amenable to as many optimization passes as native LLVM IR. In contrast, intrinsic functions can be well-integrated with the optimization passes of LLVM, but they do not already exist for these instructions.

The use of inline assembly is straightforward:

Listing 1: Inline assembly code for unprivileged memory load instruction. Comparable code provides access to unprivileged stores.

```
asm ("ldrt %0,%1 "
    :"=r"(src)
    :"m"(*(const char(*)[4])(dst)));
```

However, despite the convenience of inline assembly in generating assembly code for unprivileged memory access

instructions, the performance of the resulting programs is almost never faster than conditional bounds checking. This deficiency is attributable to the absence of LLVM optimization. LLVM treats inline assembly code as an unknown (black-box) code for LLVM (modulo input, output, and clobber lists) until it is converted into machine instruction at the LLVM backend stage. But most of the LLVM optimization passes will be performed in the preceding LLVM IR stage. Consequently, LLVM has limited capabilities for optimizing inline assembly code.

However, directly modifying LLVM to add new load and store instructions is also not straightforward. Normal memory operations must be discriminated from those we wish to make to linear memory. This motivates the use of explicitly utilized intrinsic functions to perform the unprivileged memory operations. Intrinsic functions are explicitly considered in all LLVM optimization passes.

In the LLVM backend, OmniWasm leverages the TableGen language to define the custom intrinsic functions, specify the properties of intrinsic functions, and add patterns that match both the custom intrinsic functions and corresponding unprivileged memory access instructions. Furthermore, we have also incorporated our custom offset range check function into the immediate offset addressing mode pattern for unprivileged memory access instructions because this part is missing in the LLVM backend. Consequently, LLVM is now capable of recognizing the custom intrinsic function and generating accurate machine code.

Given that intrinsic functions are visible to each LLVM optimization pass, OmniWasm's implementation has another option: either modify the existing optimization passes or write new optimization passes for the custom intrinsic function. Both approaches require a deep understanding of the inner workings of the compiler and are typically undertaken by experienced compiler developers. This is due to the fact that memory operations and their semantics are at the core of many optimization passes. Therefore, OmniWasm uses a technique that (1) enables explicit use of the instruction through C programmatic intrinsics, (2) leverages the *existing optimization passes* for normal load and store instructions, and (3) results in linear-memory accesses using `ldrt` and `sdrt`.

The compiler front end (`clang`) translates the intrinsics used in the OmniWasm runtime and generates the LLVM IR (Intermediate Representation), as shown in the first step (green arrow) in Figure 7. Passing these instructions to the optimization stages requires manual optimization which would be redundant with most of the normal memory optimizations. Instead, OmniWasm substitutes our custom intrinsic functions with regular load and store instructions augmented with OmniWasm-specific *metadata* that is tracked through optimizations. LLVM metadata represents information appended to the IR code, which has no effect on the semantics of the code. This is shown in transformation (1). Note the `!custom.intrinsic.ldrt` meta-data now bound to a normal `load` instruction. The specially defined metadata serves to differentiate between the load and store instructions generated

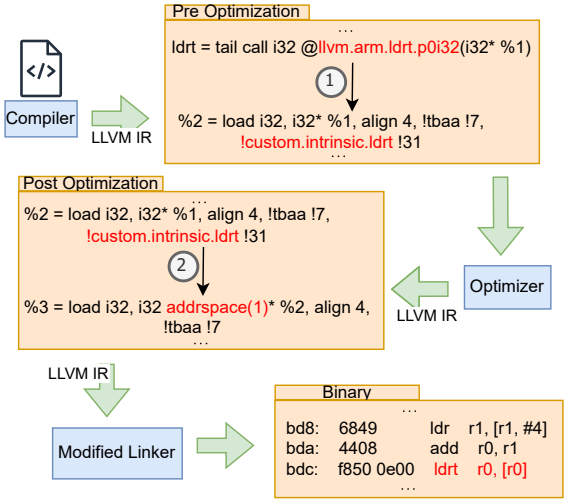

Fig. 7: The Compilation Pipeline.

from custom intrinsic functions and those from standard instructions. Where everything left unchanged from this point on, the compiler would generate normal loads/stores for linear memory access. Once LLVM completes the optimization process in the IR stage (see the "Optimizer" box), we swap instructions annotated with OmniWasm's metadata with a custom "address space" in the optimized IR file. This is seen in OmniWasm's transformation in (2). The LLVM backend that runs as part of the linker ("Modified Linker"). In the LLVM backend, at the outset of the instruction selection phase, we revert the load and store instructions that are supplemented with the custom address space, back into their originally intended form of `ldrt` and `sdrt`. This process enables our custom intrinsic functions to be used explicitly only for linear memory accesses, yet also be optimized similarly to conventional load and store instructions.

Optimized Inter-Sandbox Communication. The initial implementation of the OmniWasm runtime was inspired by the UVWasi [21] project and satisfied the minimal requirement for executing a single WASI application on a microcontroller. The UVWasi project implements the WASI system call API [22] and has been used both by NodeJS and Wasm3. A WASI application is a Wasm sandbox that contains its own linear memory, a memory protection mechanism, and a WASI environment, similar to the file-descriptor set for a UNIX process.

Unfortunately, this WASI environment results in redundant sandbox variables and symbols across various sandboxes when linked together into a conventional single-image microcontroller system. To ensure isolation between sandboxes, we process the ELF objects for each sandbox, renaming all symbols to be sandbox-specific. We provide a "allow-list" of symbols that the global OmniWasm runtime exposes that are not renamed, including software exception handlers, and IPC functions.

To enable sandboxes to interact with sensors, actuators, and communicate with other sandboxes through IPC, we provide a runtime for the WASI API that effectively exposes a set of capabilities (that are used similar to file-descriptors in UNIX) to use for streams of input and output. The normal,

polymorphic `read` and `write` API equivalents are used to send and retrieve data.

Listing 2: Host functions to read and write data from sensors/actuators/IPC.

```
__attribute__((import_name("omniwasm_msg_send")))
void omniwasm_msg_send(const void* pTxData, unsigned
     int u32Size);

__attribute__((import_name("omniwasm_msg_recv")))
void omniwasm_msg_recv(void** pMsgBuffer, unsigned
     int** u32Size);
```

When building the microcontroller, the OmniWasm runtime builds the pipelines of sandboxes statically, thus creates a compile-time table mapping sandbox output to input. Each WASI descriptor is associated with an entry in the table. When such a sandbox attempts to either transmit or receive data, the host system refers to this table to verify the existence of the Wasm module associated with the upstream or downstream sandbox. If a `send`ing sandbox references a descriptor, the runtime identifies the downstream sandbox, allocates memory in it, and copies the memory. If a `recv`ing sandbox references a descriptor, identifies the upstream sandbox, validates that it has sent data, and returns the allocated memory.

V. Evaluation

This section summarizes the evaluation of MPU-based bounds checking and the evaluation of OmniWasm IPC.

All evaluations are based on the hardware `STM32F767IGT6`. The hardware contains a `Cortex M7` series microcontroller with a 216 MHz, dynamic branch predictor, a single-level I/D cache of 16KB/16KB, 64-bit ITCM interface, 2x32-bit DTCM interfaces, SRAM of 512KB, and Flash of 1MB. ITCM and DTCM are specialized memory areas closely coupled to the processor, designed to provide fast and deterministic access times for instructions and data, respectively. Programs allocated in ITCM and DTCM generally exhibit significantly better performance compared to when they are allocated in general-purpose RAM. If programs fit entirely into ITCM and DTCM, we utilize them; when they do not, we ensure that applications use only normal memory.

Our base system for evaluation contains `FreeRTOS` operating system version 10.3.1, `arm-none-eabi-gcc` compiler version 10.3.1, `LLVM` version 13.0 [23], `WASI-SDK` version 13.0. The Wasm code version of the benchmarks and applications is built with the `WASI-SDK`-provided LLVM C compiler. The native C code version of the benchmarks and applications, along with eWasm runtime are built with `LLVM` C compiler. The `FreeRTOS` is built with `arm-none-eabi-gcc`.

Benchmarks. While `Polybench` [24] was employed in the original Wasm paper, its emphasis on floating-point arithmetic makes it less suited for our embedded system evaluation. In our evaluation, we use `CoreMark` [25] and `MIBench` [26]. `CoreMark` is a small, reproducible benchmark focusing on basic read/writes, integer operations, and control operations. Due to its simplicity and reproducibility, `CoreMark` provides a clear and stable assessment of the performance overhead of the processor's core functionalities caused by our solution. On the other hand, `MIBench` is a collection of applications

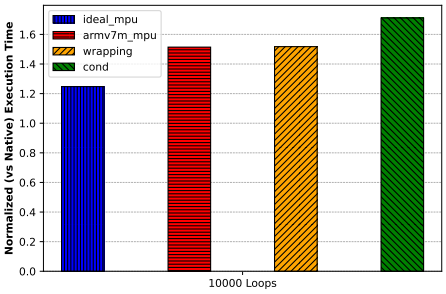

Fig. 8: `CoreMark` benchmark execution time for different bounds-checking methods. The vertical axis is the execution time, normalized to C.

in a variety of embedded domains. Due to its diverse set of applications, `MIBench` offers a broader perspective on how our solution performs under varied and complex scenarios. While the test cases are based on `MIBench`, we specifically use versions from `TACLeBench` [27] in our experiment due to their fit for resource-constrained environments.

Applications. In our evaluation, we have selected three real-world applications: TinyEKF Kalman filter, Arduino PID (Proportional-Integral-Derivative) library, and CMSIS-NN. TinyEKF is an algorithm used for sensor fusion and state estimation. PID controller is a control loop mechanism used for embedded control systems. Both TinyEKF and PID controllers are ubiquitous from automotive to industrial applications. CMSIS-NN is a set of neural network functions used in microcontrollers for image classification. Benchmarking with CMSIS-NN will give us insights into how our solution performs in the domain of AI-centric embedded applications.

Within the TinyEKF project, an established benchmark is provided specifically for sensor data fusion applications. We harness this benchmark directly for our evaluation. Similarly, for the Arduino PID library, we deploy the "adaptive tuning" example as our test case. For CMSIS-NN, we use its CIFAR-10 example to assess performance.

A. MPU-based bounds-checking

To assess our MPU-based bounds-checking method, we compared it against three other methods using both execution efficiency and memory consumption as metrics. We have named our method `armv7m_mpu` since it's tested on a hardware platform based on Armv7m. The other methods we considered are `ideal_mpu`, `wrapping`, and `cond`. The `ideal_mpu` method utilizes regular load and store instructions. The performance of `ideal_mpu` can represent the potential performance of our `armv7m_mpu` if the unprivileged load and store instructions support multiple address modes and receive official compiler support. The `wrapping` method refers to the wrapping bounds checking approach in eWasm, while `cond` denotes the conditional bounds checking method in eWasm. We conducted these comparisons using the benchmarks and applications mentioned above.

Execution Efficiency. We initiated our evaluation using the `CoreMark` benchmarks to assess the execution efficiency across different bounds-checking methods. For each of the

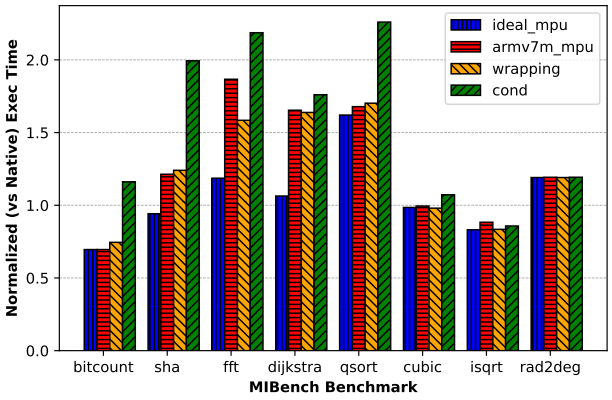

Fig. 9: `MIBench` benchmark execution time for different bounds-checking methods. The horizontal axis is the different benchmarks, while the vertical axis is the execution time, normalized to C.

four bounds-checking methods, we ran the corresponding Wasm code version of the `CoreMark` benchmark and recorded the total running time. Since CoreMark requires a minimum execution time of 10 seconds, we ran it 10,000 times to ensure accurate measurement. Subsequently, we repeated the benchmarking process for the native C code version of the `CoreMark`. Figure 8 plots the result, with all Wasm execution times normalized to the native code execution time that is 2534498743 CPU cycles.

In our subsequent evaluation, we focused on the `MIBench` benchmarks. `MIBench` encompasses six embedded application categories: automotive, office, consumer, network, security, and telecom. We excluded the office and consumer categories due to their primary design focus on resource-abundant embedded systems, such as those running Linux, which contrasts with our target of a constrained microcontroller environment. From the remaining four categories, we further refined our selection by eliminating benchmarks exceeding a size of 512 KB, the maximum capacity of our hardware's SRAM. After filtering, we assessed the execution efficiency of the remaining benchmarks using different bounds-checking methods, following the same process as with the `CoreMark` benchmark. Figure 9 plots the `MIBench` results. We also track the maximum measured execution time for MIBench benchmarks. We do not plot these values as, in most cases, the difference ratio between the maximum and average measured execution times is less than 1%.

While `MIBench` includes some applications, we augmented these with more complicated applications to assess the performance of the bounds-checking methods in real-world applications. We utilized three representative applications: TinyEKF [28], a lightweight Extended Kalman Filter implementation; PID Controller [29], a fundamental control loop mechanism; and CMSIS-NN [30], which is used in image recognition tasks. Figure 10 plots the runtime normalized to native C code.

Discussion. The performance overhead of using the `armv7m_mpu` is observed to be approximately 1.2 times slower compared to `ideal_mpu`. This deviation was expected. In Armv7-m, since unprivileged load and store in-

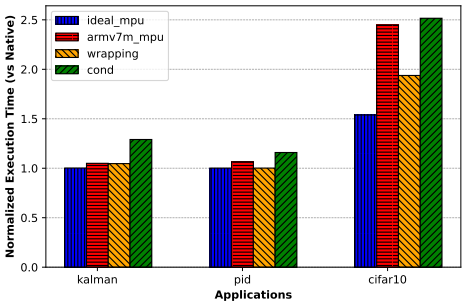

Fig. 10: Application execution time for different bounds-checking methods. The horizontal axis is the different applications, while the vertical axis is the execution time.

structions only support an immediate addressing mode with a limited range of 256, the LLVM backend can offer more optimization opportunities for the regular load and store instructions that support more addressing modes. Furthermore, in `armv7m_mpu`, enabling and disabling MPU memory regions during Wasm module transitions introduces additional overhead. As a result, when unprivileged load and store instructions receive hardware support comparable to the regular ARM load and store operations, we anticipate that the average performance overhead of our approach will still be slightly slower than that of the `ideal_mpu`. Importantly, `armv7m_mpu` has results generally similar to `wrapping`, and faster than `cond`. Even on this hardware with restrictive unprivileged load and stores, OmniWasm represents competitive performance. We also observed that for several benchmarks, the Wasm code executed faster than the native code. It's possible that in certain scenarios, the Wasm code, once compiled, benefited from certain optimizations that the native code did not, leading to a counterintuitive outcome.

Memory Consumption. We evaluate the memory footprint of `CoreMark`, `MIBench`, and applications. Since the benchmarks and applications do not use dynamic allocation, both RAM and flash usage are calculated through static analysis of the generated binaries. The fundamental flash and RAM consumption used in our experiments are shown in the table below:

Memory	`Driver`	`OS`	`WASI SDK`
RAM	1.55 KB	19.35 KB	1.77 KB
Flash	3.7 KB	8.5 KB	14.33 KB

Figures 11, 12 and 13 show the RAM consumption of `MIBench`, `CoreMark`, and applications, respectively. Figures 14, 15 and 16 show the flash consumption of `MIBench`, `CoreMark`, and applications, respectively. The test results presented exclude the fundamental consumption listed in the previous table. Furthermore, the reported RAM consumption figures include the linear memory configured with a 1KB page size, not the default 64KB, as mentioned in the implementation section. The size of data stack of linear memory is adjusted based on the requirements of the benchmarks and applications. Specifically, we determine the data stack size by iteratively reducing its size and running the applications until they fail, ensuring that we allocate just enough memory without compromising functionality. We also determine the

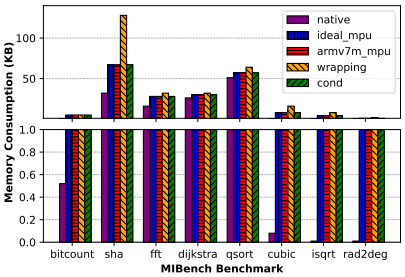

Fig. 11: MIBench RAM footprints for different bounds-checking methods.

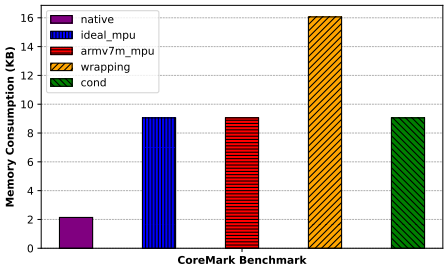

Fig. 12: CoreMark RAM footprints for different bounds-checking methods.

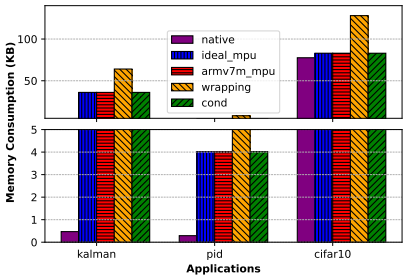

Fig. 13: Application RAM footprints for different bounds-checking methods.

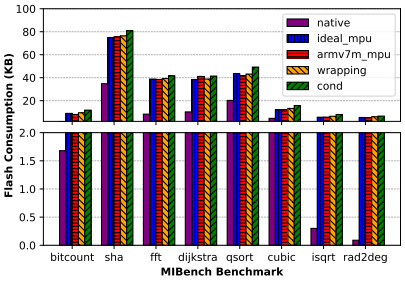

Fig. 14: MIBench flash footprints for different bounds-checking methods.

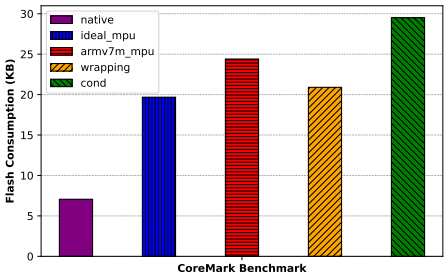

Fig. 15: CoreMark flash footprints for different bounds-checking methods.

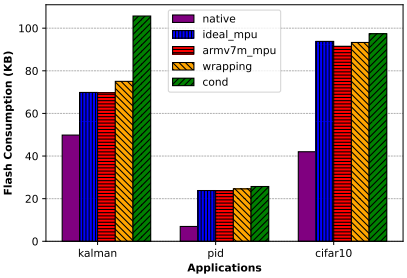

Fig. 16: Application flash footprints for different bounds-checking methods.

size of the indirect function call table statically from the table information in the WebAssembly Text format file. In our memory comparisons, we favor C by not accounting for the execution stack size, even though it's larger for C due to stack allocations.

Discussion. In our evaluation, the flash footprint figures indicate that the flash consumption for most Wasm versions is within four times that of the native code version. However, the Wasm versions of rad2deg and isqrt consume more than four times the flash compared to their native code counterparts. Our measurements of Wasm code, containing only an empty main function, revealed a flash consumption of around 5 KB. This result can illuminate the unexpected outcomes observed in rad2deg and isqrt. Given that the flash consumption of rad2deg and isqrt ranges between 5KB and 6KB, the default Wasm code size of 5KB significantly impacts the results. The RAM footprint figures show that there is a substantial memory overhead with the wrapping bounds-checking approach, in comparison to alternative bounds-checking methods. This stems from its need for memory sizes in powers of two, often doubling memory usage. Conversely, our MPU bounds-checking achieves comparable performance without the extra memory demands seen with wrapping bounds-checking. This positions our MPU bounds-checking approach as the most favorable among the three memory protection methods we examined.

We believe these results validate the fundamental contributions of OmniWasm: *even modern Arm microcontrollers with only unprivileged load and store instructions with restricted memory addressing, performance is competitive with the fastest bounds-checking technique* – `wrapping`. At the same time, *the memory consumption dominates* `wrapping`, *validating that OmniWasm provides the better of performance and memory usage*.

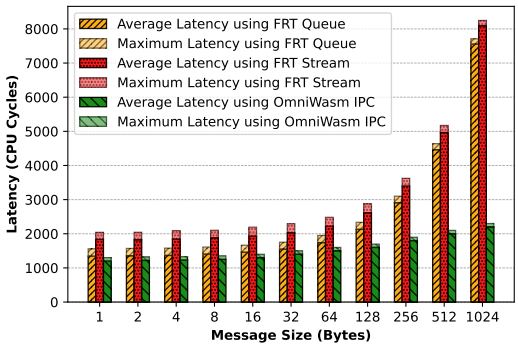

Fig. 17: IPC baseline test.

B. OmniWasm IPC evaluation

We first focus on latency in our assessment of IPC mechanisms. We benchmark our single-copy IPC approach against the two IPC methodologies offered by FreeRTOS: Queue, which is designed for passing items between tasks or between interrupts and tasks, and Stream Buffer, tailored for sending a stream of bytes between entities. In the baseline test, data of varying sizes, starting from 1 byte and incrementing in powers of 2 up to 1024 bytes, is transferred using each IPC method. We record the latency associated with each data transmission size. Figure 17 displays the result of the baseline test.

We then focus on stack usage in our assessment of IPC mechanisms under real-world communication scenarios between the TinyEKF and PID controller applications. In the first scenario, the two applications were executed in separate

threads, utilizing message queues for communication within FreeRTOS. Subsequently, in the second scenario, the Wasm versions of the applications were executed within a single thread in FreeRTOS, employing our custom IPC methods for communication. The stack usage was measured utilizing the uxTaskGetStackHighWaterMark API, provided by FreeRTOS. The results revealed that the stack usage was 0.87 KB in the first scenario, while it was reduced to 0.58 KB in the second scenario, saving approximately 33% of the memory.

Discussion. The baseline test results show that as data size grows, our single memory copy IPC's latency approaches half of what FreeRTOS IPC produce. This aligns with our initial expectations, given that both Queues and Stream Buffers in FreeRTOS involve two memory copy operations, while ours requires just one. Notably, for smaller data transmissions—such as a few bytes—our IPC significantly outperforms FreeRTOS. This advantage arises because our method operates within a single thread, avoiding the context switch overhead inherent to FreeRTOS IPC.

VI. Related Work

SFI. SFI research has a long history, with various strategies emerging over the years to ensure code safety within sandboxed environments. Among these, binary verification [3], [4], [31] and compiler verification [32], [33], [34] are two common approaches to verify the safety of sandboxed code. Binary verification uses static analysis on the binary to ensure that the required safety checks have been correctly implemented. Compiler verification proves that any binary produced by the verified SFI compiler is sandboxed. Since we simply use special load and store instructions, that happen to be safety-checked by hardware, our work is complementary to existing binary verification and compiler verification processes. We'd expect they could be straight-forwardly extended to use OmniWasm's techniques for linear-memory protection.

CFI. RECFISH [5] instruments the program binary by inserting safety checks to maintain control-flow integrity and leverage the MPU to enforce memory protections. To avoid the problems of needing to both use the MPU to isolate memory, and update the shadow stack (tracking function invocations), they make a system call on each function prologue and epilogue. The overhead on their Arm Cortex-R processor increases function overheads from 19 to 275 cycles. We believe that OmniWasm's optimizations are complementary, and might enable RECFISH to avoid this overhead, thus speed up function call operations.

PAC-PL [35] provides CFI support on FPGA-based platforms by utilizing a hardware accelerator for real-time pointer authentication and leveraging ARM TrustZone and hypervisor technologies for enhanced security. μRAI [36] ensures CFI by relocating return addresses to non-writable memory and utilizing a dedicated State Register to resolve the correct return locations through direct jump instructions derived from static call graph analysis. While prior approaches have adopted diverse strategies for ensuring control-flow integrity, our study extended this domain by implementing a hardware-based memory protection mechanism for Wasm programs in embedded systems, further reinforcing CFI.

Techniques centered around isolating shadow stacks [37], [38] have aimed to provide CFI using the same unprivileged load and store instructions used in OmniWasm. Unlike these works, OmniWasm focuses on providing both SFI and CFI, thus a strong sandboxing model.

WebAssembly. The open-source community has implemented many different types of Wasm runtime. Among them, WebAssembly Micro Runtime (WAMR) [39] and Wasmer [40] are practically designed for resource-constrained embedded systems. While the Wasm3 [41] interpreter is designed for a broad range of hardware platforms, it can also be used in resource-constrained embedded systems. These Wasm engines, similar to eWasm [10], rely on explicit bounds checking to provide memory safety. We've extended eWasm with efficient bounds checking and efficient IPC based on OmniWasm's use of hardware instructions, and we expect that the same mechanisms could be applied to other runtimes.

Microcontroller Isolation. Many hardware approaches have been taken to increasing embedded system security. These include the use of μ-kernels that can run isolated VMs [42], Rust kernels that run conventional C code in user-level using MPU protection [15], and minimal OSes that extend Sloth [43] that interweave mode transitions into application code [16]. Generally, such techniques provide strong SFI execution of application code, but do *not* provide strong CFI guarantees for that code. OmniWasm demonstrates that Wasm sandboxes overhead can be reasonable while providing both SFI and CFI.

VII. Conclusions

This research has approached the core challenge of how to provide strong security protections for code executing on microcontrollers. To do so, we leverage a WebAssembly compiler and runtime to provide strong SFI and CFI properties for legacy C/C++ code. We've introduced OmniWasm which provides both a novel mechanism to perform bounds checking that leverages MPU hardware, while still maintaining access for the runtime to sandbox CFI and SFI meta-data. We've also introduced a multi-sandbox communciation which enables finer-grained sandbox coordination to decrease the scope errant or malicious behaviors. OmniWasm's evaluation demonstrates that these techniques are able to both have strong performance while also avoiding signification memory fragmentation, and efficiently communicate between chains of sandboxes. We believe that OmniWasm is a useful tool for strongly sandboxing code in future microcontroller systems that have strong security requirements.

References

[1] S. Yoo, J. Park, S. Kim, Y. Kim, and T. Kim, "In-kernel control-flow integrity on commodity oses using arm pointer authentication," in *31st USENIX Security Symposium (USENIX Security)*, 2022.

[2] T. A. R. Wahbe, S. Lucco and S. Graham, "Software-based fault isolation," in *Proceedings of the 14th SOSP*, Asheville, NC, USA, December 1993.

[3] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Orm, S. Okasaka, N. Narula, N. Fullagar, and G. Inc, "Native client: A sandbox for portable, untrusted x86 native code," in *Proceedings of the 2009 IEEE Symposium on Security and Privacy*, 2009.

[4] L. Zhao, G. Li, B. De Sutter, and J. Regehr, "Armor: Fully verified software fault isolation," in *Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*, 2011.

[5] R. J. Walls, N. F. Brown, T. L. Baron, C. A. Shue, H. Okhravi, and B. C. Ward, "Control-Flow Integrity for Real-Time Embedded Systems," in *31st Euromicro Conference on Real-Time Systems (ECRTS)*, 2019.

[6] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, 2005.

[7] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *IEEE Symposium on Security and Privacy*, 2013.

[8] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 1, 2012.

[9] 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*, ser. PLDI '17, 2017.

[10] G. Peach, R. Pan, Z. Wu, G. Parmer, C. Haster, and L. Cherkasova, "eWASM: Practical software fault isolation for reliable embedded devices," in *Proceedings of the International Conference on Embedded Software (EMSOFT)*, 2020.

[11] B. Li, W. Dong, and Y. Gao, "Wiprog: A webassembly-based approach to integrated iot programming," in *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, 2021.

[12] R. Gurdeep Singh and C. Scholliers, "Warduino: A dynamic webassembly virtual machine for programming microcontrollers," in *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR)*, 2019.

[13] A. Hall and U. Ramachandran, "An execution model for serverless functions at the edge," in *Proceedings of the International Conference on Internet of Things Design and Implementation*, ser. IoTDI '19, 2019.

[14] "Webassembly Specification." Online, 2019, https://webassembly.github.io/spec/core/, Release 1.0.

[15] L. Amit, C. Bradford, G. Branden, G. Daniel, P. Pat, D. Prabal, and L. Philip, "Multiprogramming a 64 kB computer safely and efficiently," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.

[16] D. Danner, R. Muller, W. Schröder-Preikschat, W. Hofer, and D. Lohmann, "SAFER SLOTH: efficient, hardware-tailored memory protection," in *20th IEEE Real-Time and Embedded Technology and Applications Symposium, (RTAS)*, 2014.

[17] J. B. Dennis and E. C. V. Horn, "Programming semantics for multiprogrammed computations," *Commun. ACM*, vol. 26, no. 1, pp. 29–35, 1983.

[18] "FreeRTOS: http://www.freertos.org, retrieved 5/1/13."

[19] "aWsm. An Awesome Wasm Compiler and Runtime, https://github.com/gwsystems/aWsm/awsm—an-awesome-wasm-compiler-and-runtime," 2020.

[20] "WASI C/C++ SDK repository, https://github.com/WebAssembly/wasi-sdk/tree/wasi-sdk-13," 2021.

[21] "UVWasi, https://github.com/nodejs/uvwasi," 2021.

[22] "WASI. WebAssembly System Interface, https://github.com/WebAssembly/WASI/blob/main/legacy/preview1/docs.md," 2021.

[23] "The LLVM Compiler Infrastructure, https://llvm.org/," 2021.

[24] "PolyBench/C: the Polyhedral Benchmark suite, https://web.cse.ohio-state.edu/ pouchet.2/software/polybench/," 2019.

[25] "EEMBC's CoreMark, https://github.com/eembc/coremark," 2022.

[26] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, 2001.

[27] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener, "TACLeBench: A benchmark collection to support worst-case execution time research," in *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, 2016.

[28] "TinyEKF: Lightweight C/C++ Extended Kalman Filter with Python for prototyping, https://github.com/simondlevy/TinyEKF.git," 2019.

[29] "Arduino PID Library, https://github.com/br3ttb/Arduino-PID-Library," 2019.

[30] "CMSIS NN Software Library, https://arm-software.github.io/CMSIS_5/NN/html/index.html," 2019.

[31] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan, "Rocksalt: better, faster, stronger sfi for the x86," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2012.

[32] F. Besson, S. Blazy, A. Dang, T. P. Jensen, and P. Wilke, "Compiling sandboxes: Formally verified software fault isolation," in *ESOP*. Springer, 2019.

[33] J. Bosamiya, W. S. Lim, and B. Parno, "Provably-safe multilingual software sandboxing using webassembly," in *31st USENIX Security Symposium*, 2022.

[34] J. A. Kroll, G. Stewart, and A. W. Appel, "Portable software fault isolation," in *CSF*, 2014.

[35] G. Serra, P. Fara, G. Cicero, F. Restuccia, and A. Biondi, "PAC-PL: enabling control-flow integrity with pointer authentication in FPGA soc platforms," in *RTAS*, 2022.

[36] N. S. Almakhdhub, A. A. Clements, S. Bagchi, and M. Payer, "μrai: Securing embedded systems with return address integrity," in *NDSS*, 2020.

[37] J. Zhou, Y. Du, Z. Shen, L. Ma, J. Criswell, and R. J. Walls, "Silhouette: Efficient protected shadow stacks for embedded systems," in *29th USENIX Security Symposium*, 2020.

[38] Y. Du, Z. Shen, K. Dharsee, J. Zhou, R. J. Walls, and J. Criswell, "Holistic {Control-Flow} protection on {Real-Time} embedded systems with kage," in *31st USENIX Security Symposium*, 2022.

[39] "Wamr. a lightweight standalone WebAssembly (Wasm) runtime, https://github.com/bytecodealliance/wasm-micro-runtime," 2022.

[40] "Wasmer. The universal WebAssembly runtime supporting WASI and Emscripten, https://github.com/wasmerio/wasmer," 2018.

[41] "Wasm3. A high performance WebAssembly interpreter written in C, https://github.com/wasm3/wasm3," 2019.

[42] R. Pan, G. Peach, Y. Ren, and G. Parmer, "Predictable virtualization on memory protection unit-based microcontrollers," in *24th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2018.

[43] W. Hofer, D. Lohmann, F. Scheler, and W. Schröder-Preikschat, "Sloth: Threads as interrupts," in *2009 30th IEEE Real-Time Systems Symposium*. IEEE, 2009, pp. 204–213.