eWASM: Practical Software Fault Isolation for Reliable Embedded Devices

Gregor Peach, Runyu Pan, Zhuoyi Wu, Gabriel Parmer, *Member, IEEE,*
Christopher Haster, Ludmila Cherkasova, *Member, IEEE*

***Abstract*—As we connect more microcontrollers to the Internet and employ them to control the physical world around us, their reliability and security is increasingly important. Many microcontrollers provide limited facilities for hardware isolation, and real-time OSes offer custom APIs, that require coupling applications into the ecosystem and abstractions of that specific OS to leverage isolation.**

This paper investigates the use of *software sandboxing of applications* to support isolation for resource constrained devices. Toward this, we detail the design of eWASM, a processes abstraction that adapts a popular sandbox, Wasm, for microcontrollers. eWASM provides a runtime to constrain memory accesses and control flow, enabled by our aWsm Wasm compiler. We discuss and evaluate its multiple implementations that effectively trade time and space, optimizing for the constraints of embedded systems. This enables popular languages (e.g., C) to be effectively sandboxed by software. We demonstrate performance within 40% of native C on Polybench. We believe this is a practical and compelling result for many IoT domains, and it represents the first compiled sandboxing environment for microcontrollers. We show that restrictions of the current Wasm specification lead to significant memory consumption and provide suggestions for the creation of an embedded-specific Wasm variant.

***Index Terms*—Web Assembly, Software Fault Isolation, Control-Flow Integrity, Embedded Systems, Sandboxing**

I. Introduction

Industry 4.0 and the Internet of Things are driving the integration of microcontrollers into IoT networks that effectively interconnect sensors and actuators to provide useful context about the environment. Arm believes that a *trillion* new IoT devices will be produced by 2035 [1]. At the same time as embedded systems are increasingly exposed to a network connection, their complexity is significantly increasing. Network connections open the embedded systems to malicious intents, meaning that even a *single* buggy line of code can lead to compromise, threatening the controlled physical assets, and potentially human safety.

Unfortunately, current microcontroller software stacks are complicated to use in trustworthy embedded systems. These stacks generally lack strong facilities for isolation. Lacking isolation, compromises or faults can allow malicious code to alter parameters (*e.g.*, replacing `a` and `b` in `memset(a, 0, b)` with `SRAM_BASE`, and `SRAM_SZ`), or hijack control flow. For example, a buffer overflow in telnet can corrupt UAV flight software [2]. Isolation mechanisms decompose code and data into separate protection domains, each of which can only access the resources within that domain. Short of full system verification, defense-in-depth practices motivate constraining the effects of any errant or compromised behavior, so they impact a controlled subset of the system.

Hardware support is often used to isolate applications from each other and the OS from applications. Dual-mode protection enables the kernel code to be inaccessible from applications, at the cost of requiring mode transitions. In addition, Memory Protection Units (MPUs), which are common in microcontrollers, enable the partitioning of memory into subsets, each accessible by separate applications. MPUs are limited in that they can only protect a finite (small) number of memory regions, and some implementations have significant alignment constraints on those regions. Despite this, research has used them to provide an isolation infrastructure based on VMs [3], to protect user-level applications [4], and, when interwoven into code, to inline the protection switches [5]. MPUs require OS, hardware, and build-system support, thus tend to require integration between OS and application (*i.e.*, the application to use custom OS abstractions). Unfortunately, the uptake of hardware memory isolation has been limited in microcontrollers.

As one alternative, software techniques for *type safety* and for *memory safety* are promising avenues to provide isolation without relying on hardware features nor OS support. Languages, that provide type safety, ensure that all memory accesses adhere to the proper type of the data being accessed. Thus, memory accesses are constrained only to memory provided by the language runtime, and are prevented from corrupting other applications and OS data. The popularity of scripting languages including javascript, python, and lua (through `duktape`, `micropython`, and the `lua` interpreter, respectively) demonstrates the appeal of a safe programming environment.

As a second alternative, Software Fault Isolation (SFI [6]) techniques provide memory safety, in which loads and stores are constrained to a *sandbox*. When this data sandboxing is paired with control flow integrity (CFI) [7] – which ensures that execution can only follow paths intentionally generated by the compiler – the application's execution and memory accesses are constrained to the sandbox. Unlike in systems that use type safety, sandboxed environments allow the application to freely access any memory *within* the sandbox. This property enables legacy, widely deployed, and unsafe code written in

Gregor Peach, Runyu Pan, Zhuoyi Wu, and Gabriel Parmer are with the Dept. of Computer Science, George Washington University, Washington, DC e-mails: peachg@gwmail.gwu.edu, {panrunyu, zhuoyiw1,gparmer}@gwu.edu.

Christopher Haster and Ludmila Cherkasova are with Arm Research, USA e-mails: {Christopher.Haster,Ludmila.Cherkasova}@arm.com

languages such as C to execute with the benefits of isolation.

There is a long history of research into SFI sandboxes [6], [8], [9], [10], [11], [12], [13], and they have been broadly deployed in web browsers [11]. There has been less research into sandboxes on microcontrollers [13], [14], despite their potential to increase isolation in an environment, where limited resources encourage legacy and unsafe code. SFI approaches ensure memory safety by inserting software checks on load and store instructions to bounds-check their target addresses. To ensure CFI, indirect function call invocations (through function call pointers) have their target address and function types validated when invoked. Additionally, the return address saved in the invocation stack must be protected from buffer overflow attacks, often by separating out a control stack to save these addresses, and a data-stack to pass variables [13]. Unfortunately, the overheads of performing the necessary SFI checks are a concern, given the limited resources of microcontrollers [15]: "The [...] pure software mechanism based on Software Fault Isolation [6] would be too expensive for our embedded applications because it requires that all memory operations in a program are masked". Memory consumption is an additional concern for SFI sandboxes: all checks require additional instructions, increasing ROM size. Thus, in this paper, we design various sandboxing implementations and analyze their trade-offs in computation and memory. We believe this shows the practicality of SFI and its ability to provide strong isolation in resource constrained systems.

SFI standardization: WebAssembly (Wasm). There has been a popular effort to standardization of SFI in *WebAssembly* [16]. Despite the name, Wasm is a *general* sandboxing standard applicable beyond web browsers. It defines an *intermediate assembly language* (Wasm) and a semantics for that language. Different implementations exist for Wasm, and interpreters are gaining popularity on microcontrollers (see Wasm3 [17]). The Wasm standard has the significant benefits that it has created a strong community, an expansive set of tools for managing and debugging Wasm, and broad language support for compiling to Wasm (many LLVM [18]-supported languages).

Motivated by the strengths of standardization, this paper introduces eWASM– a Wasm compiler[1] and runtime for embedded devices. In addition to providing a Wasm runtime, eWASM enables an evaluation of various *generic* mechanisms for memory bounds checking and exploration of the trade-offs between the processing efficiency and memory consumption. The *research* questions we aim to answer include the following ones:

- How should memory bounds checks within a sandbox be designed to provide effective resource usage and strong isolation?
- What performance can be achieved relative to native code and a Wasm interpreter?
- What adaptations to the Wasm standard are needed to support and optimize for the embedded systems?

Contributions. The paper offers the following contributions:

- We detail the design and implementation of eWASM: an efficient sandbox for strong SFI targeting Arm Cortex-M microcontrollers.
- We investigate various mechanisms for sandbox memory isolation and demonstrate their trade-offs in CPU and memory efficiency. This analysis leads us to a solution that supports the fastest sandbox for microcontrollers (to the best of our knowledge).
- Finally, we investigate and analyze these mechanisms to identify the aspects of the Wasm specification that are ill-suited for using the limited resources of microcontrollers and provide the suggestions for creating an embedded-specific Wasm variant.

II. Related Work

Type safe languages. There is long history of using safe languages in embedded systems. TinyOS' nesC [19] used a type-safe environment, paired with a component-based system composition style to program deeply embedded systems. Tock [4] uses the Rust language to enable type-safe capsules in the kernel to extend system functionality, while using the MPU to isolate user-level applications. Both rely on cooperative scheduling for components/capsules. eWASM provides a process model with preemptive scheduling, and focuses on providing a secure application sandbox more-so than system extensibility.

Type-safe languages such as javascript, lua, and python (through `duktape`, `lua`, and `micropython`, respectively) have proven to be popular programming environments for IoT devices. They enable a safe programming environment in a high-level language that is used to dynamically extend device functionality. In contrast, eWASM leverages the *language-independent* Wasm environment to execute a large array of languages in a safe sandbox with performance that is both better and more predictable than a comparable interpreter.

Memory safe runtimes. Software sandboxing techniques have a long history [6], [8], [9], [10], [11], [12], [13], [14]. Some focus on the verification of the sandbox [12], [13], while others only focus on CFI [14].

Most of these efforts focus on non-resource-constrained systems and microprocessors. In contrast, ARMor [13] provides a verified sandbox for microcontrollers, but does not constrain loads. By not limiting the loads, this approach does not provide *confidentiality* as all memory can be accessed. This is particularly dangerous on microcontrollers as devices are often memory mapped, thus safety is dependent on each device's semantics. ARMor does not focus on, nor extensively evaluate performance (between 10% and 240% slowdowns despite unconstrained loads). Walls [14] find that CFI alone on microcontrollers imposes a 30% slowdown. In contrast, eWASM imposes on average a 40% slowdown (§VI-A) when providing both CFI and strong memory safety (for both loads and stores). eWASM evaluates how various implementations of memory safety for SFI trade efficiency and memory consumption This includes a novel software page-table mechanism that enables non-contiguous, fine-grained memory allocations. Importantly, using a safe intermediate representation (Wasm), enables the

[1]Our Wasm compiler is called aWsm and is open-sourced along with eWASM at www.github.com/gwsystems/awsm/.

elision of some memory safety tests (*e.g.*, by lifting them out of loops), and lays the foundation for language-independent binaries that can be retargeted between microcontrollers and more capable hardware (*e.g.*, in the cloud).

Hardware isolation mechanisms. In contrast to these software techniques, past research has investigated using hardware to provide isolation. Systems have explicitly leveraged hardware to increase isolation: from virtual machine abstractions using MPUs [3], to Cortex-M Trustzone [20], to fine-grained switching between modes inlined into code [5]. Others have used compiler and static analysis techniques to provide CFI checks on sensitive instructions using dual mode execution hardware [15], and to automatically separate system code into MPU-isolated compartments [21], [22]. These approaches do not provide strong isolation *within* a compartment, instead attempting to reduce CFI threats. These works differ from eWASM in the following ways: (1) eWASM provides strong CFI within the sandbox, while also preventing access to memory or control outside of the sandbox. (2) eWASM focuses on sandboxing a subset of code, while these other approaches attempt to provide stronger isolation for all microcontroller code (*e.g.*, including the RTOS). (3) eWASM has no hardware isolation dependencies, thus avoids conflicting with how the RTOS or surrounding software use the MPU. This enables the use of eWASM in existing software systems by linking with the sandbox at compile time.

Wasm runtimes. Wasm interpreters (including [23], [17]) focus on debuggability and interpreter performance. These runtimes are often between 14 and 25 times slower than native C (as we show in §VI-D). Though useful for dynamically updating system code and debugging, interpreted code is not applicable for code that must be predictable and/or efficient. We find that when performance is increased to within a factor of 2x native C, the mechanisms for memory sandboxing are a dominant factor in determining performance and memory consumption. A focus of this paper is to closely examine existing and novel mechanisms for this.

eWASM summary. This paper presents the first SFI sandbox for microcontrollers that targets performance and memory consumption that suffer only a relatively small degradation compared to C. eWASM investigates various memory isolation mechanisms, and applies them to an existing SFI standard (Wasm). We focus on Wasm to pragmatically leverage its extensive existing software ecosystem.

III. WASM BACKGROUND

This paper investigates the use of WebAssembly (Wasm) [16] to provide memory and control sandboxing of embedded code. Wasm is a portable, language-agnostic, low-level bytecode paired with a runtime semantics, which provides a memory-light sandbox for untrusted execution based on software fault isolation [6] and control flow integrity [7]. Despite being driven by web standards bodies, the specification has been designed to work outside the browser as well, and there are numerous implementations for non-Web environments. No efficient implementation exists for deeply embedded system on microcontrollers with performance and predictability on par with native code. This paper investigates mechanisms for efficient sandboxing on microcontrollers, evaluating their trade-offs, and providing suggestions for a version of the Wasm standard customized for embedded systems. This section focuses on the existing Wasm standard, and its properties, and does not discuss this work's contributions.

The key features of Wasm include the following:

- *Efficiency.* WebAssembly targets near native speed while only assuming common hardware capabilities.
- *Safety.* WebAssembly describes a memory-safe, sandboxed execution environment based on restricted memory access and control flow. Wasm sandboxes, despite the name, may be linked into non-Web programs.
- *General execution environment.* Unlike language-specific bytecodes that define typed object layouts and differentiate between memory containing pointers and memory containing integer values, Wasm exposes access only to an untyped (yet bounded) array of bytes. This enables the execution of low-level languages such as C *within* the sandbox.
- *Open and debuggable.* WebAssembly provides a human-readable textual format [24] to simplify learning, debugging, and optimization tasks.

Memory-safety. Code execution within the Wasm sandbox is only able to access memory within a contiguous *linear memory* region. Each Wasm memory access addresses linear memory at an *offset* from the base, L, of the linear memory. Thus, there is some amount of address *virtualization* as an address N in the sandbox is located at $L + N$ in physical memory (assuming linear memory is laid out contiguously). Additionally, accesses are only allowed *within* linear memory, *i.e.*, accessing an address beyond the linear memory's size generates a sandbox violation. Thus, the Wasm runtime is responsible for translating linear memory accesses, and bounds-checking them to prevent accesses outside the sandbox. These checks are a foundation for the Wasm sandbox's *isolation*.

Linear memory is dynamically sized. It is extended in increments of a 64KiB page size, when asked to `expand_memory`. The bounds checking logic is correspondingly updated.

Control-flow integrity. Sandboxing untrusted execution requires control-flow integrity (CFI) [7], which constrains execution to a safe control flow graph, considered by the compiler. Wasm achieves this using two mechanisms: a data stack and a validated function pointer dispatch.

Wasm code's execution stack must be protected from a potentially malicious logic within the Wasm code. This is required to prevent stack smashing attacks that overwrite function return address values. Thus, the execution stack should be *outside* of linear memory. Unfortunately, this complicates passing pointers to stack-allocated variables. To prevent these pointers from accessing the stack arbitrarily, Wasm separates the execution stack into (1) the execution stack tracking function calls and local variables *external* to linear memory (2) a data stack *within* linear memory containing stack-allocated variables.

Wasm dynamically ensures that only valid function pointers are called by sandboxed code. This is ensured by requiring

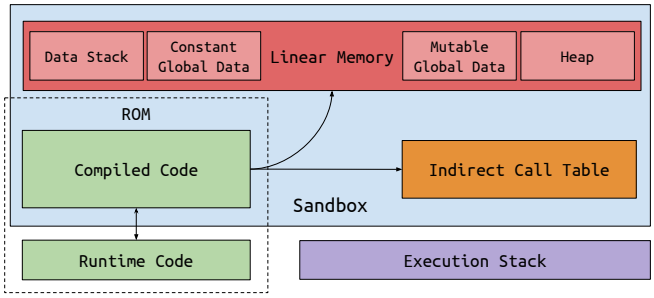

Fig. 1: The layout of the Wasm sandbox.

all function pointers (including C++ vtable dispatch) be not pointers, and rather offsets into a runtime table of valid function entry points. When a function pointer is invoked, the Wasm runtime (1) validates they point to valid function addresses, and (2) validates that the type of the caller matches those of the function. Similarly, function entry-points into the sandbox, and calls out of the sandbox to functions explicitly provided by the runtime, are indexed and type-checked using runtime function pointer tables.

We depict the key mechanisms of a Wasm sandbox in Figure 1. Memory safety depends on constraining loads and stores to linear memory that contains the sandboxed global and heap data. Control-flow integrity requires both the separation of the execution stack (that tracks function call return addresses) from the data stack – in linear memory – used to hold data referenced by pointers, and type and control flow checks on function pointer calls through an indirect function call table. Runtime code for interacting with the RTOS, and requesting memory are the main means to exit the sandbox.

A. Opportunities for Wasm on Embedded Systems

The Wasm standard has a number of features that make it appealing for embedded devices.

- *Broad support.* Wasm is supported in all major browser vendors. There is a large ecosystem of vendors, tools, and languages providing Wasm support. With the ability to leverage Wasm, the embedded system community would benefit from a broader ecosystem.
- *Wasm as a portable IR.* Wasm is a platform-independent Intermediate Representation (IR). It can be generated for different source languages (*e.g.*, a Wasm back-end for LLVM works for C, C++, and Rust), and can run on many platforms (our compiler runs on 32-bit and 64-bit x86, Arm64, and Arm Thumb/Cortex-M architectures). Solving how to effectively run Wasm on microcontrollers opens the possibility to sandbox existing code, regardless if it executes on a microcontroller, or in the cloud.
- *No mandatory garbage collection.* Many broadly used language runtimes such as javascript, lua, or python (through `duktape`, `lua`, and `micropython`, respectively) cannot provide predictable execution and often require more memory, or are slower. The most popular implementations are all interpreters, which emphasizes their utility as extension mechanisms to embedded system. The Wasm sandbox's linear memory is managed by the program. It can use garbage collection, but can also use manual or static memory allocation.
- *Lightweight runtime.* Wasm mandates only a small number of runtime features around maintaining memory sandboxing and CFI (see above). They also provide a simple specification for the runtime to safely invoke functions from the runtime, into the sandbox, and vice-versa. These light requirements show promise in an embedded adaptation.

B. SFI Challenges on Embedded Systems

Currently, only *interpreted* Wasm environments exist for Cortex-M processors. Though Just-In-Time (JIT) compilers exist on larger systems, their size, complexity, and unpredictability make them untenable on microcontrollers. In general, running SFI on resource constrained microcontrollers presents a number of challenges. Despite the long history of SFI approaches [6], [8], [9], [10], [11], [12], [13], multiple specific aspects of microcontrollers complicate their adoption in embedded systems. Zhao et al. [13] focus on formally verifiable SFI, and do not report extensive evaluations nor study memory isolation mechanisms beyond condition-based bounds checking.

- *Memory consumption.* Wasm's 64 KiB pages are too large for microcontrollers that often have between 16-256 KiB SRAM. Additionally, it is not clear, how SFI approaches are impacted by the separation of system memory into read-write SRAM and read-only flash, that supports execute-in-place program execution.
- *Performance.* Though the bounds checking on linear memory and the indirect checking of function pointer invocations are necessary for proper isolation, they add overhead over non-sandboxed code. On out-of-order architectures with intelligent dynamic branch predictors, these checks will likely have less of an impact than on a simple 3-6 stage, in-order pipeline with limited dynamic branch prediction.
- *Assumed ISA features.* Wasm specifically is a 32-bit virtual architecture. A common technique [16] to optimize away linear memory bounds checks on 64-bit MMU-based systems by devoting 2^{32} bytes of contiguous virtual memory to linear memory, thus preventing any access beyond the 32 bits available to Wasm. Unfortunately, without virtual memory support and with Cortex-M's 32 bit architecture, this optimization is not possible. Additionally, Wasm also supports floating point, but not all Cortex-Ms do. In such cases, floating-point emulation is broadly deployed.

Given the specific constraints and limitations of microcontrollers, the extensive past SFI research is not sufficient to offer a practical solution for embedded software isolation.

IV. eWASM Design

We introduce the eWASM system that integrates the aWsm Wasm compiler, a microcontroller Wasm runtime, and a RTOS to provide strong software-provided isolation for reliable embedded systems. We discuss the implementation of the Wasm sandbox, then integration into FreeRTOS to provide a software processes abstraction.

A. Control-Flow Integrity

The Wasm specification ensures strong control flow integrity such that code inside the sandbox can only execute code generated to maintain the sandbox protection. Wasm's emphasis on a structured control flow lets us easily ensure the generated code does not branch to unexpected locations. eWASM implements the Wasm standard's CFI mechanisms in a straightforward manner, but we discuss the design here for completeness. The design of CFI in Wasm considers three main control structures:

Direction function calls. The compilation process is constrained by structured control flow based on conditionals, loops, and function calls. We statically ensure that all direct branches are to valid targets, thus the only way to violate control flow is to jump indirectly. In C environments, control flow is typically compromised with either indirect function calls (*e.g.*, heap smashing) and stack corruption (*e.g.*, stack smashing).

Indirect function calls. Indirect functions calls are caused by function pointer invocations, for example, through virtual function tables or function arguments (*e.g.*, `qsort`'s comparator argument). The Wasm specification strictly controls indirect function calls. It forces all indirect calls to go through a global table of function pointers. This table is statically initialized with the valid addresses of functions and their types. At runtime, the invoked function pointer is replaced with an offset into this table, and the call-site of the function pointer includes the expected type. The runtime looks up which funtion is stored at the offset into the table, and verifies that the type of the function matches that expected by the call-site. Only then does the runtime perform the dynamic branch. If the signature does not match, or there is not a valid address in the table slot, a sandbox exception is generated.

Stack-based execution tracking. The other way the program could violate control-flow integrity is through stack smashing – that is, maliciously overwriting function return addresses on the stack. The data stack, which is used to pass arguments and do function-local allocations, is allocated in linear memory. Corrupting this stack cannot violate control flow integrity – the runtime does not trust it. We use the native execution stack for executing sandboxed code. This stack is inaccessible from within the sandbox, thus preventing the corruption of its return addresses.

B. Memory Safety

To prevent Wasm code from accessing and potentially corrupting memory outside of the sandbox, loads and stores must only be performed in linear memory. The Wasm specification gives no guidance on the mechanisms to be used for constraining or responding to illegal memory accesses. eWASM explores *four* implementations for validating linear memory accesses that make different trade-offs between isolation, performance, memory usage, and standards conformance.

- *No isolation.* To establish a baseline, this does no safety checks on loads or stores. To execute an instruction under this model, the runtime simply adds the offset to the address of the start of linear memory, then dereferences the address. This does *not* provide memory safety, but is useful as a baseline to separate eWASM overheads for linear memory access checking, and the other mechanisms (*e.g.*, those required for CFI).
- *Condition-based bounds-checking.* This approach inserts a naive bounds check on each load and store, based on the size of linear memory. This provides memory safety, but necessitates an extra branch every time data is loaded or stored. This implementation is specification compliant. This is a straightforward implementation of bounds checking and is also used in [13].
- *Masking-based sandboxing.* To increase performance and avoid branches, eWASM includes an implementation, based on masking offsets into linear memory. On each load and store, the runtime applies a bitmask to offsets to ensure that they fall within the bounds of linear memory. This has no effect on in-bounds accesses, but causes out of bounds accesses to wrap around (using modulo arithmetic). This approach generates branch-free code that is faster than condition-based code. Unfortunately, an efficient version of masking-based sandboxing requires power-of-two sized linear memory, which can waste memory due to internal fragmentation. This approach does not detect and trap on accesses beyond linear memory, instead wrapping within it. Due to this, it is not strictly Wasm standard-conforming. Note, that languages such as C define out-of-bounds accesses as *undefined behavior*, similar to this masking approach. Regardless, masking maintains strong isolation as linear memory accesses are still confined to the sandbox.
- *Software page-tables.* The previous implementations have the shortcoming that linear memory must be contiguous. This is appropriate for many embedded systems where maximum application memory consumption should be known statically, but where dynamic allocation is necessary, contiguous memory is restrictive as microcontrollers do not provide address virtualization (*i.e.*, they have a MMU). Inspired by hardware MMUs, we design a software page-table implementation. In this implementation, all offsets are split into a n-bit page number and a 10-bit page offset. The runtime maintains a single-level page-table, tracking how pages are mapped to backing memory. On access, it simply looks up the page corresponding to the linear memory address, and then loads or stores from there. If the access spans two pages, then the eWASM runtime does two loads and reconstructs the final value. This approach tends to be slow, but is extremely flexible. It allows sandboxed programs to be given limited access to arbitrary pieces of memory. As linear memory is expanded, pages can be *separately and non-contiguously allocated* with the RTOS's `malloc`.

C. eWASM-based Process Abstraction

A primary benefit of using a software approach to provide isolation is that it avoids requiring hardware support, and also avoids interfering with the system outside of the sandbox. Many microcontrollers do not have MPUs, thus *must* use software means for isolation. When available, MPUs might

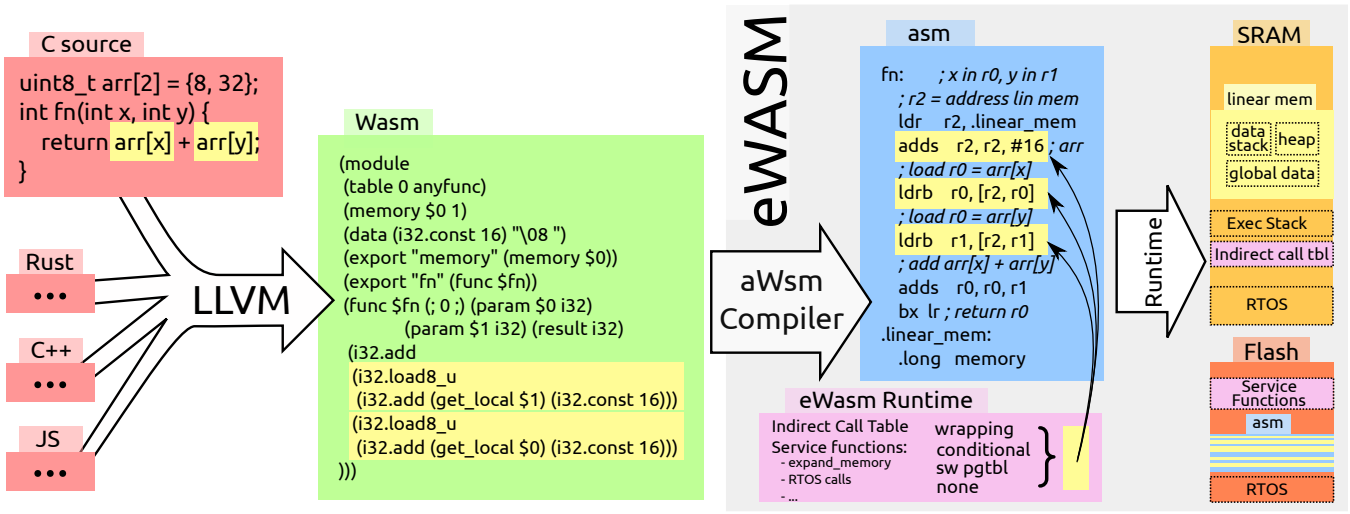

Fig. 2: The compilation pipeline and execution in eWASM. Languages compile to Wasm (LLVM provides strong support for this), and eWASM provides the compiler and the runtime to sandbox the execution for microcontrollers. eWASM consists of the aWsm compiler, and the runtime, which includes bounds-checks to ensure sandboxed memory accesses. The yellow boxes emphasize memory accesses which carry through the C, to the Wasm, and to the assembly. The assembly on the right offsets loads and stores into linear memory (yellow in SRAM), but does not bounds check: it implements the "no isolation" (none) policy.

be used by the RTOS, or the programming environment, preventing their use to specific sandbox and application. In this way, SFI approaches are complementary to, and can co-exist with hardware techniques.

We pair the sandboxing facilities of Wasm, with the temporal isolation properties of an underlying RTOS to provide a process abstraction for the execution of lower-assurance or untrustworthy code.

Even if control-flow integrity and memory safety are assured, the system is of limited use without temporal isolation. An infinite loop within the sandbox could preclude fairness and predictability. Thus, we focus on using the Wasm sandbox as a means for isolating an application, and executing each sandbox in a separate RTOS thread. We use FreeRTOS as our RTOS. As FreeRTOS uses preemptive, priority-driven scheduling, sandboxes are prevented from monopolizing the CPU and causing unpredictability in other, higher-priority tasks.

The Wasm runtime and generated code runs on a separate thread from the non-sandboxed parts of the program. To interact with sandboxed code, one can use FreeRTOS message passing channels to coordinate between tasks. Note that messages passed to the sandbox must be copied into its linear memory. In our design, communication into the sandbox is possible with no more overhead than a simple function call, copying the message. This leaves the door open for different approaches for cross thread coordination, depending on the application. Though we focus on providing temporal isolation via threaded execution, the Wasm code could be integrated into existing threads, or interrupt handlers.

V. eWASM Implementation

eWASM consists of three different components: (1) our ahead-of-time compiler, aWsm, that takes Wasm code and generates sandboxed Arm Cortex-M executable code, (2) the Wasm runtime, that maintains sandboxed isolation and provides a set of functions (*e.g.*, syscall emulation) to be called from the sandbox, and (3) integration into an RTOS to provide a runtime process abstraction. Below, we discuss each one in detail.

A. eWASM Compilation Pipeline

Programs go through a multi-step compilation pipeline before being ready to execute in the sandbox, as depicted in Figure 2. We leverage existing support in the LLVM compiler infrastructure [18] to generate the Wasm code for a source language. An LLVM backend for Wasm enables it to generate Wasm for any of the (many) languages that use LLVM.

During this step, we compile the application with a custom fork of musl libc that directs all system calls to the eWASM syscall handler. Our current implementation supports a small number of system calls, including writes to standard out and error, and memory allocation requests (via an emulated `mmap`). In this way, we support the legacy code that uses our restricted set of system calls.

The generated Wasm file specifies the logic for a well-typed, stack-based architecture. Figure 2 gives a simple example of a Wasm program which calculates a variable offset into an array (located at offset 16 in linear memory), and dereferences that value. It does this twice and adds both values, returning the result. The Wasm file is fed into our ahead-of-time compiler, aWsm, which translates the Wasm into LLVM IR. aWsm performs a number of operations: (1) It translates from a stack machine (Wasm) to a register machine (LLVM IR). The types of data on the stack are always statically known, which eases the transition to a SSA form. (2) It maps most Wasm instructions (*e.g.*, for arithmetic) to an equivalent sequence of LLVM IR instructions. (3) However, for specific instructions integral to the sandbox isolation properties, aWsm generates an invocation to a pluggable function, which we

Listing 1: No protection

```
i16 get_i16(u32 offset):
; offset stored in r0
; char* address = &memory[offset];
    movw    r1, #memory_addr_low
    movt    r1, #memory_addr_high
; return *(i16 *) address;
    ldrsh   r0, [r1, r0] ; array access
    bx      lr
```

Listing 2: Conditional bounds checking

```
i16 get_i16(u32 offset):
; offset stored in r0
    push    {r4, r5, r7, lr}
    sub     sp, #40
; assert( offset <= memory_size - sizeof( i16 ));
    movw    r1, #memory_size_addr_low
    movt    r1, #memory_size_addr_high
    ldr     r1, [r1]      ; load memory size
    subs    r1, #2
    cmp     r1, r0.       ; do bounds check
    bhs     #40 <get_i16+0x3e>
; [Fault handling logic omitted ...]
    ...
; char* address = &memory[offset];
    movw    r1, #memory_addr_low
    movt    r1, #memory_addr_high
; return *(i16 *) address;
    ldrsh   r0, [r1, r0]          ; array access
    add     sp, #40
    pop     {r4, r5, r7, pc}
```

Listing 3: Wrapping

```
i16 get_i16(u32 offset):
; offset stored in r0
; return *(i16 *) &memory[offset % MEM_SIZE];
    movw    r1, #memory_addr_low
    movt    r1, #memory_addr_high
    bfc     r0, #18, #14.
    ldrsh   r0, [r1, r0]      ; bitmask, array access
    bx      lr
```

define in the C runtime rather than emitting the corresponding LLVM instructions. This enables an efficient interface to the eWASM runtime, and allows us to easily update, validate, and test different bounds checking implementations (see the integration of bounds logic from the runtime into the assembly in Figure 2). A focus of this paper is on the impact of various bounds checking implementations, thus enabling the configurable coupling of the runtime sandboxing logic into the application logic is essential. We heavily rely on LLVM to optimize the resulting code and inline the runtime operations, thus removing any overhead of the separate functions.

Then we use LLVM/clang to combine this bytecode with our runtime, written in C which defines the bounds checking mechanism. We use LLVM's link time optimization (LTO) to ensure that these mechanisms are efficiently inlined into the application. Once this process is done, we are left with a self-contained, sandboxed object file ready for deployment onto the target platform. Our infrastructure makes it easy to create new pluggable functions, allowing custom communication protocols to develop between the sandboxed code and code outside of the sandbox. We use these to interface with the RTOS The aWsm compiler is currently 3500 lines of code written in Rust and is open source.

B. eWASM Sandboxing Runtime

The eWASM runtime consists of a number of functions that are directly invoked from the Wasm sandbox. These include:

- linear memory accesses, that include the bounds-checking logic,
- function pointer invocation indirection tables and type-checking,
- the implementations of select system calls by pulling in a modified musl libc for backwards compatibility, and
- service functions to expose the underlying RTOS's functionality, where appropriate (*e.g.*, message passing based communication, or APIs for accessing I/O devices).

We investigate different implementations of the bounds checking mechanism for linear memory that in many cases trade efficiency for memory consumption. For reference, Listing 1 shows the logic for doing a load of a 16-bit value from linear memory. It offsets the address into the linear memory (`memory`) using `ldrsh` to perform an array load. Note that Arm inlines the address for `memory` into the code by splitting it into its higher- and lower-order bits across instructions. Similarly, the assembly in Figure 2 performs 8-bit loads using comparable instructions.

As eWASM provides isolation of sandboxed execution, the other bounds checking mechanisms ensure access is constrained to within linear memory. The straightforward conditional checking mechanism is detailed in Listing 2. This mechanism (1) compares the address (+ 16) to the maximum linear memory size (`memory_size`), (2) produces a sandbox exception if the access is beyond the bounds, and (3) offsets into memory. In contrast, Listing 3 elides the condition and error handling code by simply wrapping the address within a power-of-two, static sandbox size (*i.e.*, by masking off the higher order bits), and then offsets into memory.

The software page-table implementation is not shown here. It uses a single-level page-table and a page size of 1kB. It must check if a memory access crosses between page boundaries, and splits it up the access across pages (later recombining the memory), if necessary. Thus, this mechanism is relatively complex, and generates a fair amount of code for each memory access.

The eWASM runtime is 1200 lines of C code. The relative simplicity of this runtime is important: it enables us to have heightened confidence that the sandboxing of each application is properly provided.

C. eWASM RTOS Integration

The sandboxed application object (ELF object) is linked into the surrounding RTOS, and executed as a single thread inside of it. This provides the temporal isolation provided by the preemptive, priority-driven scheduling of FreeRTOS. Message queues can link normal FreeRTOS tasks to the sandboxed application, and vice-versa. The runtime is activated directly by function calls, so we avoid all mode switching and hardware protection domain switching overheads (*e.g.*, of writing to the MPU registers). Stronger temporal isolation could be provided if FreeRTOS implemented rate-limiting servers [25]. For this work, we focus on decoupling the

memory and control isolation from the RTOS implementation to leverage the facilities it provides for temporal isolation.

D. Practical Considerations and Limitations

- *Unaligned memory accesses.* The Wasm specification does not allow us to assume that loads and stores are aligned. This means we must account for that, and make memory operations and bounds checking mechanisms work with unaligned addresses. A prototype version of eWASM – that converted unaligned accesses to a set of aligned smaller access – has large slowdowns, so eWASM currently relies on activating unaligned accesses in the microcontroller (supported by all but the smallest of Cortex-M processors).
- *Statically-sized backing memory.* With limited memory and without virtual memory support, growing large contiguous allocations is challenging. Thus, conditional bounds checking and wrapping use a statically-sized allocation. In contrast, software page-tables are exceptional in that they do allow non-contiguous expansion of linear memory by performing address virtualization in software.
- *Read-only memory.* Wasm does not differentiate between read-only and read-write memory. This has a significant impact on eWASM memory usage. Read-only memory can be stored and directly accessed on flash in microcontrollers. As the flash often is larger than SRAM, applications are commonly designed to have large lookup tables in read-only memory. eWASM must then copy that read-only memory from flash into SRAM when constructing a sandbox. This is sub-optimal because (1) read-only memory consumes both SRAM *and* flash resources, and (2) SRAM consumption can significantly increase. This is what we feel is the single Wasm specification nuance that has the most friction with embedded systems. In § VII, we discuss a set of suggested extensions to Wasm for embedded systems.
- *Expanding linear memory.* Dynamic allocation is a common requirement, and we implemented it in our modified musl libc. Wasm specifies that the sandbox should expand memory by 64KiB chunks, which is not granular enough for constrained embedded systems. Thus, the libc `malloc` requests heap expansions by making a system call to the runtime, allowing it to allocate with 1KB granularity.
- *Undefined behavior in C.* Since our compiler pipeline uses optimizations, undefined behavior in the underlying C program can cause unexpected results. Some legacy C programs with undefined behavior make assumptions about the underlying machine that do not hold in Wasm. These include assuming that specific addresses have semantic meaning, or that different structs have the same bitwise layout. In practice, eWASM caught undefined behavior in several programs we tested, although none in the programs we used for our final benchmarks.
- *Stack overruns.* The execution stack is outside of the Wasm sandbox, thus stack overflows are a potential hazard. We require the stack to be guarded with the MPU. The shadow stack is in linear memory, so overruns due to, for example, large stack-allocated data-structures, are protected by the existing bounds checks. For embedded systems, applications must often be profiled to ascertain stack usage. eWASM applications are no exception, but insufficient stack sizes will be detected and properly faulted.
- *Libc replication.* Libc is compiled into the code of each sandbox. This provides strong protection as even the libc code is sandboxed. Unfortunately, it also means that the libc code is replicated in the sandboxes. This increases the code-size for the sandboxed code, but we err on the side of strong isolation.

VI. Evaluation

Hardware and software configuration. We use `STM32F767ZIT6`, a `Cortex-M7` based microcontroller for evaluation of eWASM. It runs at 216 MHz, has a 6-stage pipline that is dual-issue, and has a dynamic branch predictor. It has a 16k/16k I/D cache, 512kB SRAM, and 2MB Flash with the ability to execute code directly from Flash (aided by a prefetcher). We believe that the results from this evaluation should generalize to other processors in the `Cortex-M` family. We would expect that on systems without dynamic branch prediction, the results for bounds checking will be worse.

All evaluations use the FreeRTOS V9.0.0 operating system. The base system is compiled with GCC 8.2 using the `Musl` C library. The interpreter we tested `wasm3` [17] is built with this toolchain as well. All benchmarks and applications are built with `Clang-LLVM`, both WASM and C native. All graphs in this section depict averaged measurements.

Benchmarks. The benchmark suite used in the original Wasm paper is `Polybench` [26]. It has some relevance to embedded systems: it includes common matrix operations and statistical operations.

Applications. We use a number of applications to evaluate eWASM. The applications we analyzed are:

- CMSIS-NN V1.0.0 (`nn`)- A neural network library designed for microcontrollers that is used to perform image categorization.
- Arduino PID library (`pid`) - A typical Proportional, Integral, Derivative controller used for physical control.
- TinyEKF Kalman filter (`kalman`)- Used for sensor fusion and state estimation.
- TinyCrypt (`crypto`) - A small library of crypto primitives.

For CMSIS-NN, we do image recognition using a CIFAR-10 configuration that has three convolution layers, then ReLU activation and max pooling layers, finally followed by a fully-connected layer. The input to the network is a 32x32 pixel color image which is classified to one of ten categories. For the Arduino PID library, we run the official "Adaptive Tunings" example. For TinyEKF Kalman filter, we use the official `Fuser` benchmark. TinyCrypt is a crypto library for resource constrained devices, and it is run with its own standard benchmarking suite, with several tests that use too much memory (for even the C code to run) elided.

A. Benchmark Evaluation

To investigate the overheads of different WASM bounds-checking methods and the architectural effects to these over-

heads, we leverage the Polybench benchmark suite which is commonly used in Cortex-M performance profiling. We evaluate both the execution efficiency and the memory footprint of WASM binaries against their native counterparts.

Execution efficiency. We first evaluate the execution efficiency of the benchmarks. We ran each Polybench benchmark for 10000 times, and registered their total running time for all four bounds-checking methods. We followed the same procedure for the native C code. Figure 3 plots the results. All execution times are normalized to native code. We run these tests and report the results in a manner consistent with the evaluation in [16].

Discussion. We see that the execution time of all WASM binaries are higher than that of the directly compiled native binaries (native). The "no bounds checking" (none) Wasm binaries lack any security properties but are almost as fast as native in some benchmarks. The "wrapping bounds checking" (wrapping) mode offers security and those binaries are only 1.5x slower than native, however this mode is not fully standards compliant. The "conditional bounds checking" (cond) mode is 2x slower than native but uses standard compliant bounds-checking. If the system needs to run WASM on non-contiguous memory blocks, "software page table" (swpgt) is required; this approach is even 2x slower than cond but does not require the linear memory to be contiguous. For most embedded systems, wrapping can be applied to maintain security while close to native performance, shrinking the bounds checking overheads.

Memory consumption. We evaluate the memory footprint of all benchmarks. The (read-write) RAM and ROM usage is obtained by static analysis of the generated binaries before they are loaded into FreeRTOS. The dynamic RAM usage is detected with hooks to malloc and free. Figure 4 shows the ROM usage of the Wasm binaries in flash, compared with native C, while Figure 5 shows the RAM consumption.

Discussion. The ROM consumption for Polybench is fairly consistent across all benchmarks. Most benchmark code is small, and the main contributor to the ROM size is the Polybench library. We expect that ROM consumption will increase in eWASM due to three reasons: (1) The bounds checking and function pointer indirection mechanisms provide sandboxing, but require additional code to be generated for the checks. (2) Additionally, libc is compiled in with each Wasm object including copies of the functions used by the applications. (3) Last, initialization data for the linear memory and function lookup table are stored in ROM, consuming around 32kB. Due to these factors, we notice in Figure 4 that the code ROM size bloats for none, wrapping and cond, and results in around a 4x increase in ROM consumption over C. The difference in memory consumption between no bounds checking, and the other eWASM approach is due entirely to the increased code size due to memory safety. The main outlier here is the software page-tables. The inlined software page-table walking code causes significant increases in the binary's code.

As discussed in §V-D, the Wasm standard does not differentiate read-only and read-write memory. This means that what would be ROM (and placed in flash) in C, must be copied into the initial image of the linear memory in Wasm. Thus, the ROM is represented in both ROM and RAM measurements. This effect is particularly pronounced in benchmarks like jacobi-1d that include very little read-write memory.

eWASM's additional RAM consumption is due to runtime structures including the shadow stack, and the function lookup table. Though the shadow stack could be tweaked to be no larger than a specific application requires, we allocate it a constant 32kB. We do *not* report the execution stack RAM consumption (which is larger for C as it is used for stack allocations).

RAM consumption for wrapping is higher than the other bounds checking approaches as the linear memory must be statically sized to be the next higher power-of-two over the maximum heap usage. This leads to a maximum 100% RAM linear memory wastage (and 50% on average) due to internal fragmentation. It should be noted that industry practice is to often pad application memory for future firmware or application updates.

Note that we *under-estimate* the RAM consumption for native C in two ways: (1) We do not measure memory requirements for the runtime execution stack. eWASM generally needs less memory for this as local allocations that can be referenced are allocated in the shadow stack in linear memory instead of on the runtime stack. In practice, this means that stack allocations in C are not counted in these numbers, and appear in linear memory RAM (and ROM) consumption in eWASM. (2) We measure all memory committed to linear memory in eWASM. In native C, we only record the maximum of all active malloc requests. This means we do not measure RAM consumption for malloc meta-data (e.g. headers), nor internal or external fragmentation. Thus, our comparisons with eWASM favor C.

B. Low-end Evaluation with M4

We re-run Polybench using eWASM on a Nordic nRF52840, which is a Cortex-M4F, with 1MiB Flash and 256KiB SRAM. We ran without floating point enabled (thus using emulation) to emulate an M3 without the floating point support, that Wasm assumes. The code and SRAM consumption are close to the results for the M7, thus we omit them here. We execute each benchmark 100 times, and observe deviation from the average for each run of less than our measurement resolution (± 1 nanosecond).

The average slowdown of the code over native C:

Microcontroller	none	wrapping	cond	swpgt
Cortex-M7 (from Fig.3)	1.238	1.402	1.675	3.964
Cortex-M4	1.227	1.249	1.519	1.628

Discussion. As floating point emulation is used, the relative cost of different memory sandboxing approaches is smaller than on the M7. Regardless, the results show that even for hardware that does not support features of Wasm (*e.g.*, floating point), performance of the various memory protection mechanisms is consistent and reasonable.

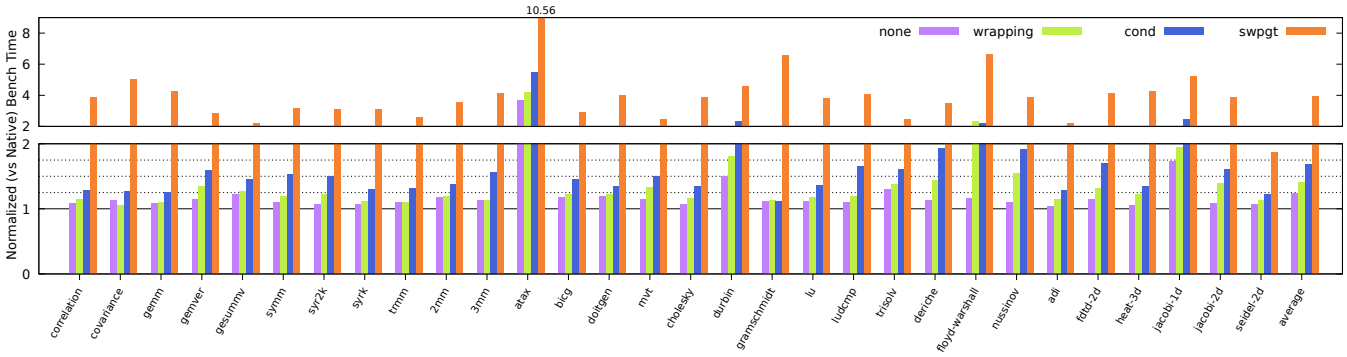

Fig. 3: `Polybench` benchmark execution time for different bounds-checking methods. The horizontal axis is the different benchmarks, while the vertical axis is the execution time, normalized to C.

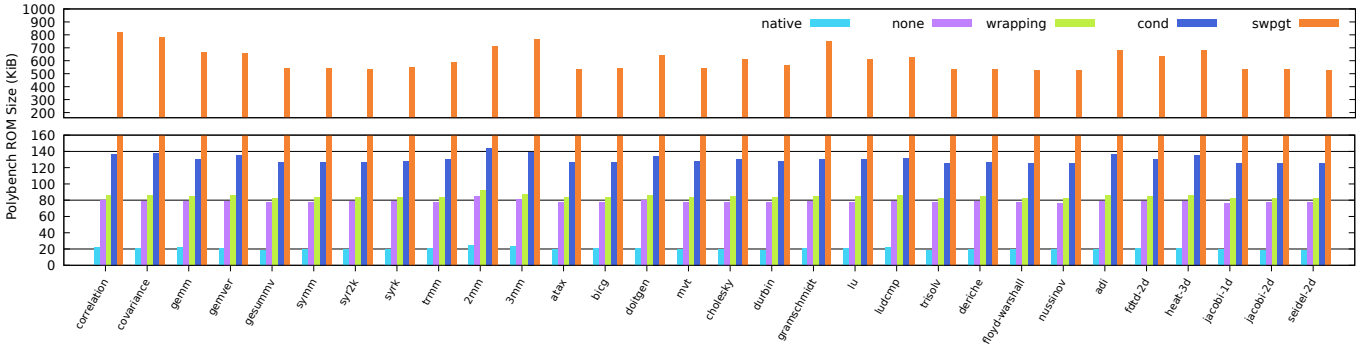

Fig. 4: `Polybench` benchmark ROM footprints for different bounds-checking methods. The horizontal axis is the different benchmarks, while the vertical axis is the memory size.

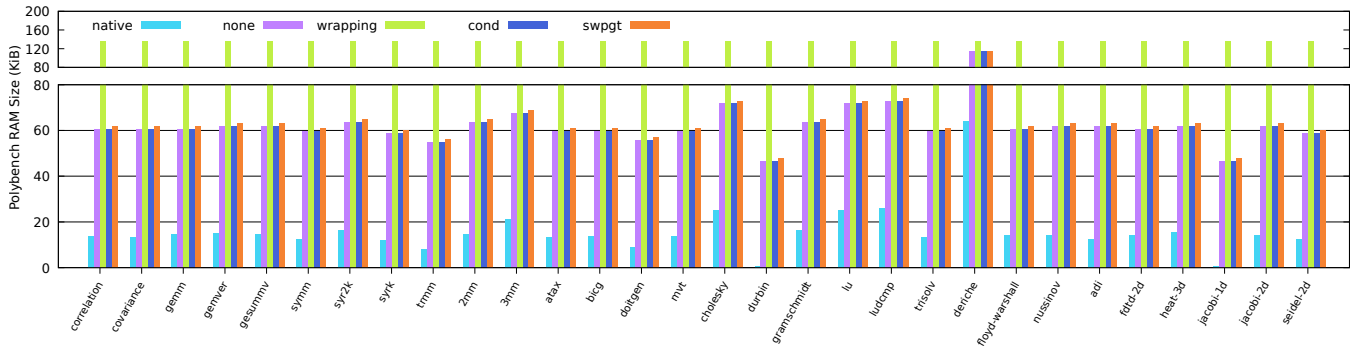

Fig. 5: `Polybench` benchmark RAM footprints for different bounds-checking methods. The horizontal axis is the different benchmarks, while the vertical axis is the memory size.

C. Application Evaluation

In this section, we present real-world application evaluations using the four applications detailed above, `nn`, `pid`, `kalman` and `crypto`. We measure both the *application's running time* as a metric of execution efficiency and memory footprint.

Execution efficiency. We first evaluate the execution efficiency of the applications under all four bounds checking methods and compare this with that of native C binaries directly compiled from C source. We report the execution for a single run of `crypto`, and the average of 1000 runs for the other applications.

Discussion. For all four applications, we can see that `native` binaries are faster than WASM binaries. This is expected due to bounds checking and shadow stack maintenance. Even `none` is slower as it must translate addresses into linear memory.

As with `Polybench` results, `none` is faster than `wrapping`, `wrapping` is faster `cond`, and `swpgt` is still the slowest. For all applications, the running time of `wrapping` and `cond` is within between 1.5x and 3x when compared to native binaries, and the running time of `swpgt` is with in 6x. Even the software page-tables are significantly faster than an interpreter (see §VI-D).

Memory consumption. We then evaluate the memory footprint of all applications. The code ROM size and the RAM size are obtained as in the `Polybench` results and depicted in Figures 7 and 8, respectively.

Discussion. The `none` and `wrapping` mechanisms maintain ROM sizes within a factor of four of C. The same is generally true of `cond`, while `kalman` is an outlier at around 10x. As with `Polybench`, the page-table approach results in very large binaries due to the per-memory access paging logic. The same logic around increased memory consumption for eWASM sandboxes as introduced for `Polybench` is true here, and the same trends present themselves. The software page-table mechanism vastly expands ROM consumption, while

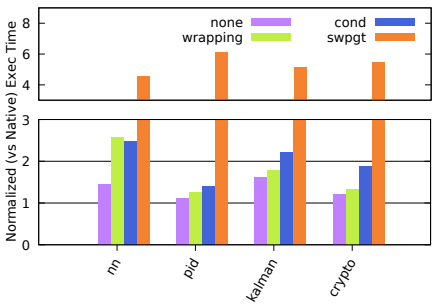

Fig. 6: Application execution time for different bounds-checking methods. The horizontal axis is the different applications, while the vertical axis is the execution time.

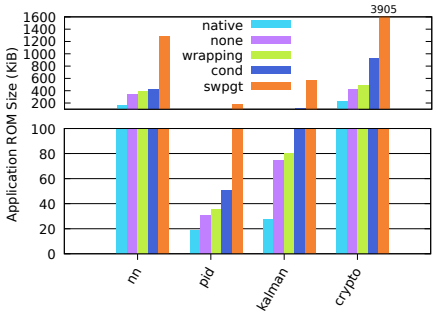

Fig. 7: Application ROM footprints for different bounds-checking methods. The horizontal axis is the different applications, while the vertical axis is the memory size.

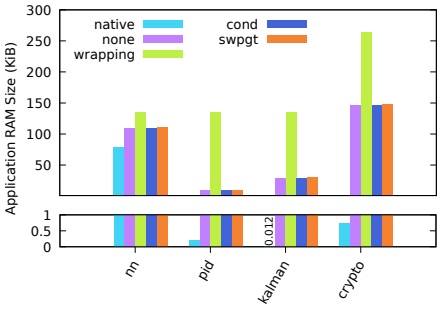

Fig. 8: Application RAM footprints for different bounds-checking methods. The horizontal axis is the different applications, while the vertical axis is the memory size.

`wrapping` remains with a factor of 4x in ROM consumption. The RAM consumption is dominated by pages statically-allocated for use as linear memory.

D. Comparison to Interpreter

Interpreted execution environments are common on IoT devices. We selected `wasm3` as a well established interpreter for Wasm on the `Cortex-M`, with a reputation for strong performance. To assess the properties of the `wasm3` interpreter, we ran our four "real-world" application benchmarks on it.

`wasm3` struggles to run `crypto`, taking more than two hours to complete a single run, thus we judge it to have timed out. `pid` is run for 100 rounds and the other two application benchmarks are run for 10 rounds given their long runtimes.

We present the normalized running time (`exec`) and ROM/RAM usage (`rom` and `ram`) of those applications, for both `wasm3` and eWASM (`wrapping`).

App	wasm3			ewasm		
	exec	rom	ram	exec	rom	ram
nn	67.11	254.4KiB	176.8KiB	2.56	385KiB	136KiB
pid	30.17	94.4KiB	148.8KiB	1.26	35.6KiB	136KiB
kalman	26.43	96.4KiB	167.4KiB	1.78	80.2KiB	136KiB
crypto	timeout	363.4KiB	timeout	1.33	486KiB	264KiB

TABLE I: The `wasm3` interpreter versus eWASM `wrapping`.

Discussion. For these applications, interpreter performance is more than an order of magnitude worse than that of native code. The design of `wasm3` necessitates using RAM to store the intermediate representation it uses to achieve relatively fast interpretation. The interpreter itself is complex and large which leads to significant ROM usage. This culminates in RAM and ROM usage that is larger by an order of magnitude than native. eWASM performs better than `wasm3` for all bounds checking mechanisms. Notably, `wrapping` executes at least an order of magnitude faster that `wasm3` on all the benchmarks we tested. eWASM's memory usage is also superior to `wasm3`, even in the relatively `RAM` hungry `wrapping` mode.

Conclusions. The different memory sandboxing approaches represent different trade-offs in time and space, which we will summarize here. Wrapping-based approaches have the best performance (within 40% of C on `Polybench`), has the lowest ROM usage of any isolation approach, but requires significant RAM consumption due to the power-of-two size requirement. Conditional-based bounds checking increase the ROM size due to the overhead of error paths, and decreases performance compared to wrapping. However, its RAM consumption is relatively tight to that required by the application, increasing in 1kB increments as the heap is expanded. Software page-tables impose significant overheads in terms of performance (though still significantly less overhead than the interpreter), and in ROM usage (due to inlined page-table translation code). However, it shares the RAM efficiency of condition-based checking, and is unique in that it performs address virtualization. Thus linear memory can be non-contiguously allocated. This is useful if an application has unknown maximum memory requirements. We show that all approaches have a significant limitation which is a side effect of the Wasm specification: read-only memory must be allocated in linear memory, thus read-write RAM.

VII. Lessons Learned and Recommendation for Wasm Standards

Configurations of eWASM that perform best on microcontrollers are *not* compliant with the Wasm standard. They maintain sandboxing, but do so by loosening some of the guarantees provided by the standard. This section discusses the specific ways the standard either leads to inefficiencies on microcontrollers, and argues for *an embedded variant of the Wasm standard*. Note, that all suggestions maintain strong sandboxing properties.

- *Page-size, and initial memory allocation.* The Wasm page size is 64KiB. This was chosen as the least common multiple of common hardware page sizes. On embedded systems, especially microcontrollers, a 64KiB page a major limitation. On many M series microcontrollers, there is between around 16 and 128KiB of usable read-write memory. Our implementation departs from the standard by allowing byte-granularity, 1 KiB page-granularity, or static allocation of linear memory for conditional, software page-table, or wrapping bounds checks. The initial image of the linear memory would normally be placed in a single 64KiB page, but we instead size the initial allocation to the actually required memory. To optimize for memory, we recommend that a Wasm embedded variant allow expanding memory by a significantly smaller granularity, both in the initial linear memory and for dynamic allocations.

- *No separate allocation of read-only memory.* As touched on in the previous section, all Wasm linear memory is read-write. On embedded systems, it is common to have large read only data, such as lookup tables, and to store them in the, often larger, flash memory. The *Wasm specification does not provide facilities for separate, read-only memory*. Thus, our implementation copies read-only data into the linear memory, causing significant SRAM consumption.
- *Exceptions on Out of Bounds (OOB) accesses.* The Wasm specification requires that a load to a linear memory address greater than the current size of linear memory should fault. A naive approach to this is bounds checking, but as we have shown, this has overhead, especially on embedded systems with limited branch prediction.

  We have shown the benefit of a wrapping-based approach to sandbox memory accesses. This behavior is not specification conforming, but does give a significant speedup.

  We recommend that a specialized embedded Wasm specification optionally allow undefined behavior of memory accesses outside of the sandbox, so that the runtime can use wraparound to efficient provide sandboxing.

Failed design: MPU-assistance. We attempted a co-design of the linear memory OOB checking using the hardware Memory-Protection Unit (MPU). We could not make this work. In retrospect, it is clear to see why we would inevitably fail. There is a conflict between the goals: (a) generated code needs fast access to the execution stack *and* the linear memory, and (b) code in the sandbox must not be able to access the execution stack. A single static MPU configuration cannot accommodate this, so fine-grained switching, potentially on each access to the execution stack, would be necessary. The overhead of this [3] led to us abandoning attempts to use the MPU.

VIII. Conclusions

This paper introduces eWASM that provides a *software sandbox* with temporal and spatial isolation for legacy code on resource constrained microcontrollers. eWASM consists of an ahead-of-time compiler, aWsm, and a runtime to safely execute even typically unrestricted code such C in what we believe is the first compiled, sandboxed environment for microcontrollers. We compare various mechanisms for ensuring that sandboxed code cannot access memory outside of the sandbox, and find that they all represent interesting time/space trade-offs. Though eWASM shows performance within 40% of C, we also find that the Wasm specification has a few friction points with an effective microcontroller implementation. We make specific recommendations that might further increase the capabilities of strong sandboxing in resource-constrained devices.

The eWASM and aWsm source is publicly available at www.github.com/gwsystems/awsm/.

Acknowledgments

We'd like to thank the reviewers of this paper that provided immensely valuable feedback that has improved the presentation of the research. We'd like to thank the support from the NSF through awards CNS-1815690 and CPS-1837382, and from ARM and SRC through Task 2911.001.

References

[1] Philip Sparks, "The route to a trillion devices; the outlook for IoT investment to 2035," Online, 2017, https://learn.arm.com/route-to-trillion-devices.html.

[2] M. Hooper, Y. Tian, R. Zhou, B. Cao, A. P. Lauf, L. Watkins, W. H. Robinson, and W. Alexis, "Securing commercial wifi-based uavs from common security attacks," in *MILCOM 2016 - 2016 IEEE Military Communications Conference*, 2016.

[3] R. Pan, G. Peach, Y. Ren, and G. Parmer, "Predictable virtualization on memory protection unit-based microcontrollers," in *RTAS*, 2018.

[4] 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.

[5] 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.

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

[7] 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.

[8] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *OSDI'06: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2006, pp. 11–11.

[9] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Software fault isolation with api integrity and multi-principal modules," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[10] B. Ford and R. Cox, "Vx32: lightweight user-level sandboxing on the x86," in *USENIX 2008 Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2008, pp. 293–306.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] A. A. Clements, N. S. Almakhdhub, K. Saab, P. Srivastava, J. Koo, S. Bagchi, and M. Payer, "Protecting bare-metal embedded systems with privilege overlays." in *IEEE Symposium on Security and Privacy (S&P)*, 2017.

[16] 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.

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

[18] "The LLVM Compiler Infrastructure, https://llvm.org/," 2019.

[19] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesc language: A holistic approach to networked embedded systems," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[20] S. Pinto, H. Araujo, D. Oliveira, J. Martins, and A. Tavares, "Virtualization on trustzone-enabled microcontrollers? voil!" in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

[21] A. A. Clements, N. S. Almakhdhub, S. Bagchi, and M. Payer, "Aces: Automatic compartments for embedded systems," in *Proceedings of the 27th USENIX Conference on Security Symposium (CSS)*, 2018.

[22] C. H. Kim, T. Kim, H. Choi, Z. Gu, B. Lee, X. Zhang, and D. Xu, "Securing real-time microcontroller systems through customized memory view switching." in *Network and Distributed System Security Symposium (NDSS)*, 2018.
[23] 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.
[24] "Webassembly Specification." Online, 2019, https://webassembly.github.io/spec/core/, Release 1.0.
[25] B. Sprunt, L. Sha, and J. P. Lehoczky, "Aperiodic task scheduling for hard real-time systems," *Real-time Systems*, vol. 1, pp. 27–60, 1989.
[26] "PolyBench/C: the Polyhedral Benchmark suite, https://web.cse.ohio-state.edu/ pouchet.2/software/polybench/," 2019.

Gregor Peach recently graduated with a B.S. in Computer Science from The George Washington University as part of the class of 2020. Currently he is working at Amazon as a Software Engineer. His research interests include: virtualization, software sandboxing, embedded systems, garbage collection, as well as language design and implementation.

Runyu Pan is a Computer Science PhD Candidate supervised by Prof. Gabriel Parmer at The George Washington University. His research interests mainly incoporates embedded systems, real-time software stacks, and capability-based microkernels. This includes microkernels on microcontrollers (MoM), real-time virtualization and development of modular, secure and robust IoT software infrastructures. His current research are conducted on the Composite operating system deployed on microcontrollers.

Zhuoyi Wu got his undergraduate degree in Computer Science from University of California, Irvine in 2018, and completed his Masters degree at The George Washington University in 2020. His interests span Operating Systems, Networking, Artificial Intelligence, Machine Learning, Security, Algorithms, and Embedded Software. He will start to work for Amazons AWS department from August 2020. His future research directions will investigate the integration of AI and Operating Systems.

Gabriel Parmer is an Associate Professor of Computer Science at The George Washington University. His research interests encompass operating systems, real-time and embedded systems, and scalable parallel systems. This includes the development of a new operating system, Composite, that aims to enable component-based design of low-level system software, while optimizing for end-to-end, non-functional goals such as security, predictability, and scalability. He is a member of the IEEE.

Christopher Haster is a senior software research engineer at Arm Research, Austin, USA. His focus is software development on resource constrained devices, with an interest in non-traditional systems, languages, and execution models. His past work includes LittleFS, a power-resilient RAM bounded filesystem designed to enable external storage on devices with very little internal memory, as well as contributions to Mbed OS around high-level languages and non-RTOS composition strategies.

Lucy Cherkasova is a principal research scientist at Arm Research, San Jose, USA, since 2018. Before that for 20+ years she was a principal scientist at Hewlett Packard Labs and led to success multiple RD projects, with prototypes, algorithms, or features implemented in HP products. Her current research interests are in developing quantitative methods for the analysis, design, and management of distributed systems (such as emerging Smart environments, Edge computing, Big Data processing, and next generation datacenters). She is the ACM Distinguished Scientist and is recognized by multiple Certificates of Appreciation from Usenix and IEEE Computer Society.