Practical Considerations for Sandboxing On-Target-Compiled WebAssembly Modules on Embedded Systems

Fabian Scheidl
BMW Group AG
Munich, Germany
Technische Universität Wien
Vienna, Austria
fabian.scheidl@bmw.de

***Abstract*—Across software architectures and platforms, latest developments have evolved towards flexible and sandboxed software modules that allow safe execution of untrusted code in an inherently side-effect free manner. Recent trends have further led towards portable and statically validatable representations of software in a bytecode format like WebAssembly. In order to facilitate the transfer into the domain of embedded systems and the internet of things, this paper explores the practical considerations, strategies and requirements for integrating an on-target sandboxing WebAssembly compiler into bare metal embedded systems. This in turn enables deployment of corresponding runtimes on embedded systems that demonstrate the need for predictable, resource-efficient and fast sandboxed execution. This paper further evaluates benchmarks to compare the arising overhead of different sandboxing strategies against analogously compiled non-sandboxed executables.**

***Index Terms*—internet of things, safety, sandbox, fault isolation, virtual machine, runtime, webassembly, performance, embedded, memory**

I. Introduction

With the steadily increasing complexity of embedded devices and the growing demands placed upon them, especially as internet of things devices, modularization (and specifically flexibilization) of embedded software vastly gains in importance [1], [2]. The goal of this is enabling safe execution of flexibly hot-loaded software components on embedded systems while at the same time ensuring that, potentially malicious or erroneous, software modules are perfectly isolated from the rest of the system and are thus side-effect free.

With that objective in mind, lightweight virtualization options for embedded systems have been explored [3], [1]. Lightweight virtualization is similarly relevant in the field of desktop browser systems which allow websites to execute untrusted code on the client computer. Departing from JavaScript as the only natively available execution mechanism in browsers, WebAssembly has been introduced. WebAssembly represents a compact, lightweight and portable precompiled bytecode format of a virtual instruction set that can be more efficiently compiled and executed than JavaScript. WebAssembly is inherently sandboxed and statically validatable in an efficient manner [4]. The small instruction set and strict semantics of WebAssembly allow for both a compact implementation of the runtime and humble resource requirements.

Embedded systems often do not offer any complex memory management units, virtualized memory, might not run an operating system and often do not have a process scheduler managing execution times of different processes. It is due to this reason that lightweight virtualization of software modules is particularly relevant for bare metal systems.

Previously, interpreting virtual machines have been utilized for virtualization and deployment of portable hot-loaded sandboxed software to embedded systems [5]. Unfortunately, interpreting techniques are associated with a significant overhead during execution. Even with modern threading techniques for optimizing the dispatch of an interpreter, performance is still orders of magnitudes slower than execution of natively compiled software [6].

In recent times, lightweight on-target compilers for embedded devices has been researched [7], [8]. Now, with WebAssembly, and with it the possibility of bringing lightweight high-performance compiling runtimes to embedded systems and at the same time utilizing an open, standardized and portable representation of software, this approach could form the basis for novel lightweight virtualization options for embedded systems.

We have previously demonstrated that on-target compilation of WebAssembly bytecode via the valent-block compilation technique is a feasible approach for implementation of a WebAssembly runtime on embedded systems with minimal resource requirements and only minimal logic and offers scalable high-performance [9].

Software attacks, specifically those using hot-loaded software modules as an attack vector, often exploit vulnerabilities such as buffer overflows, stack-smashing techniques or overload the system by inducing a locking state by entering an infinite loop. Especially on safety-critical systems, which embedded systems are often part of, this behavior is completely unacceptable. Thus, methods of preemptively avoiding behavior of this kind have to be developed.

II. Contribution and Related Work

In the following sections, we will outline various techniques for practical memory sandboxing of on-target-compiled WebAssembly bytecode and enforcing execution time limits for embedded systems and thus provide the foundation for safe (and largely platform-agnostic) multitasking of local WebAssembly-based nanoprocesses. Due to the high variability and feature-diverseness of embedded systems we constrain this analysis to the most portable techniques for sandboxing, i.e. those which demand the least features of the underlying software- and hardware architecture. We further present benchmarks illustrating the resulting overhead by implementing the presented sandboxing strategies, building on valent-block compilation technique [9].

So far, most pursuits of bringing WebAssembly to embedded systems have focused on interpreting techniques. [5], [10], [11]. Interpreters implicate a significant overhead with raw compute-performance generally being orders of magnitude slower than for compiling implementations [6], [12]. Interpreters further implement sandboxing in a way that is vastly different from compiling runtimes. Based on these aspects and CPU time being especially precious and limited on (often very resource-constrained) embedded systems, we will, in this paper, solely focus on on-target compiled sandboxing of WebAssembly bytecode and will leverage the syntactic restrictions WebAssembly's specification imposes upon valid WebAssembly bytecode.

Most compiling WebAssembly runtimes like Google's V8 rely on the underlying operating system, virtual memory and memory management unit to allow for so-called guard pages, which deliberately forbid access to a specified region of memory and raise a signal once that region is accessed.

Generalized SFI (Software Fault Isolation) and memory-safe masking have previously been shown to be feasible to provide as part of a formally verified compiler (CompCert) that implements *a priori* sandboxing security [13]. Due to data integrity being of paramount importance on safety critical embedded systems and memory masking potentially leading to undetected invalid (but still non-crashing) memory accesses, this sandboxing strategy is not apt for this use case.

III. Managerial Impact

WebAssembly paves the way towards a novel deployment and programming mechanism for embedded devices. Internet of things networks and systems often comprise of a number of embedded systems and could thus, especially in the case of remote resource-constrained internet of things devices, benefit from highly efficiently sandboxed hot-loadable software. This paper demonstrates practical approaches for implementation of guaranteed sandboxing for WebAssembly modules in a compiling runtime for embedded systems. This not only increases flexibility and resource efficiency (and thus, depending on the use-case battery life) of those devices, but does also have the potential to lead to vastly improved device security if logic is flexibly deployed to the platform.

```
(module
  (func (param i32 i32) (result i32)
    local.get 0
    local.get 1
    i32.add
  )
)
```

Fig. 1: A sample WebAssembly module comprising a function which returns the sum of two 32-bit integers.

IV. WebAssembly

WebAssembly (Wasm, an open W3C standard) defines a stack-based minimal virtual instruction set architecture for executable programs [4]. WebAssembly was originally conceived to provide fast execution speeds for computationally intensive tasks in Webbrowsers and builds upon asm.js with contributions from Google's Portable Native Client (PNaCl, [14]). The structure of WebAssembly bytecode enables almost trivial static validation, is strictly typed and offers inherent memory safety and well-defined interfacing to the runtime. Fig. 1 shows a sample WebAssembly module in its textual assembly-like representation.

A. Memory Concept

The memory structure of a WebAssembly module is as follows: WebAssembly employs an opaque stack that is used to perform numeric and logical operations, conversions and evaluate conditions. WebAssembly modules further have access to a separate region of contiguous memory, so-called linear memory that is logically independent from the stack. As of now, the linear memory can only be grown dynamically by multiples of 64kB. There is no method to dynamically shrink linear memory once grown. Interaction with data in linear memory is restricted to loading data onto and storing data from the stack.

B. Control-Flow Integrity

WebAssembly does not offer branches and jumps to arbitrary target addresses. Instead, WebAssembly can only call functions that are either explicitly defined in the respective WebAssembly module or those which are provided and successfully imported from the runtime. WebAssembly functions can be called directly or indirectly via a dynamically provided index onto a function table comprising functions with a strictly and homogeneously defined signature. The WebAssembly instruction **block** enters a block which allows branch-instructions within the respective block to jump to the end of the block which is denoted by the instruction **end**. Similarly, **loop** allows branch instructions to jump to the beginning of the loop. With these restrictions, a correctly executed and successfully validated (and thus spec-conforming) WebAssembly module therefore guarantees control-flow integrity during execution.

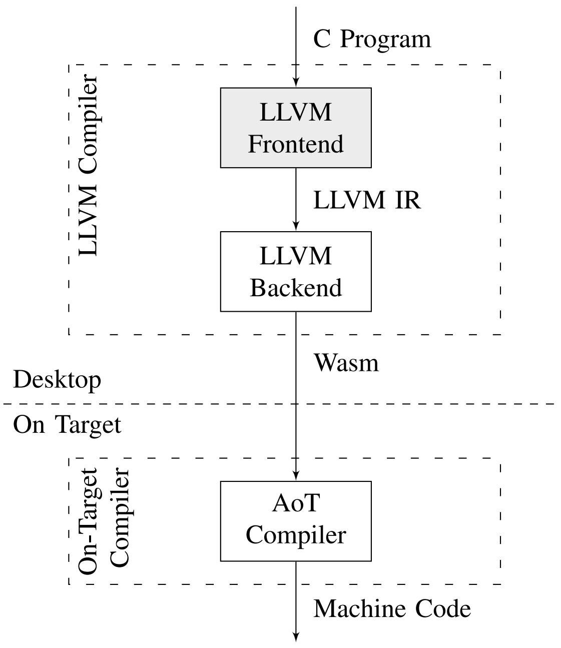

Fig. 2: Compiler toolchain flowchart. Programming language-specific compiler frontend is marked in grey.

V. On-Target-Compilation

Based on the idea of bringing on-target compilation to resource-constrained bare-metal systems and Google's V8 baseline WebAssembly compiler Liftoff [15] we implemented a proof of concept of an extended streaming quasi-singlepass on-target-compiler (Valent-block compilation) for WebAssembly that forgoes complex optimizations, keeps compile-time memory consumption to a minimum and is thus well suited for embedded systems [9].

V8 Liftoff employs a compile-time stack as an intermediate representation (IR) [15]. Valent-block compilation modifies that idea by pushing actual pure expression trees (including arithmetics, as opposed to only values in the case of V8 Liftoff) from the instruction stream to the compile-time stack and then generating code based on the stack contents.

This strategy allows the compiler to be designed in a very compact manner while exhibiting exceptional performance and scalability.

Fig. 2 shows the overarching simplified compiler toolchain for executing logic via an on-target compiler. A program (e.g. C-Code) is compiled to WebAssembly via LLVM (or an equivalent compiler producing valid WebAssembly bytecode), which performs aggressive compiler optimization passes. The WebAssembly module is then transferred to the target (e.g. an embedded vehicular ECU), where it is passed to the on-target ahead-of-time (AoT, or alternatively just-in-time) WebAssembly compiler, in this case the valent-block compiler. The compiler produces executable machine code, that can be, depending on the underlying system and the concrete use-case, either executed from RAM or saved to and executed from flash memory.

```
1 (module
2   (func $f
3     loop $L0
4     br $L0
5     end
6   )
7 )
```

Fig. 3: Valid WebAssembly module leading to an unconditional infinite loop.

VI. Sandboxing

The following section will provide a quick overview over WebAssembly's inherent sandboxing and capabilities in regard to ensuring memory safety. For the purpose of this paper, we extend the meaning of sandboxing to the runtime having control over execution time of the WebAssembly module's functions. Take for example Fig. 3: Line 3 defines a loop label (a branch label that can be targeted as a backwards branch) with the name `$L0`. Once the branch instruction on line 4 is encountered, a branch targeting that loop is executed leading to an unconditional branch to `$L0` and thus an infinite loop is initiated. Note that the WebAssembly module presented in Fig. 3 still represents a specification-conform, validation-passing WebAssembly module. This behavior is therefore something that WebAssembly's restrictions and sandboxing do not cover. If this WebAssembly module is naively translated to machine code without any consideration for execution time sandboxing, while no process scheduling (as it is effectively the case on bare-metal systems) is available, the system will be stuck executing this infinite loop until the system is reset. This behavior is unacceptable for hot-loaded software on embedded systems and represents a major vulnerability, particularly on safety-critical systems.

A. WebAssembly's Sandboxing Features

WebAssembly, given correct implementation of the runtime, enables inherent sandboxing of executable modules. Sandboxing in the sense of WebAssembly's definition is restricted to only allowing safe memory accesses and predefined control flow branching.

Based on the memory concept of WebAssembly (See Section IV-A), it is easy to see that in order to guarantee memory safety, the stack has to be guaranteed not to over- and underflow. Additionally, it has to be ensured that no memory accesses outside of the bounds of allocated linear memory are executed. To achieve this in an efficient manner, a WebAssembly runtime must employ a combination of static (i.e. validation-related) and dynamic checks. The structure, deliberate restrictions and static validation of WebAssembly do shift a significant portion of the responsibility for proper sandboxing from dynamic runtime checks to those which can be statically carried out and thus do not impact runtime performance.

```
1 (module
2   (func (param i32 i32) (result i32)
3     local.get 0
4     i32.add
5   )
6 )
```

Fig. 4: An invalid WebAssembly module, if erroneously executed as-is leading to an underflowing stack.

1) Static Checks: WebAssembly brings along formal specifications and strict typing that enable easy validation of a module's structure. Due to WebAssembly being in essence a strongly typed stack-machine, module validation can be relatively trivially mechanized, can be implemented in a resource-efficient manner and incorporates only minimal logic.

Static validation averts dangers arising due to malformed WebAssembly modules. This includes the guarantee of general in-module code-flow integrity (See also Section IV-B), correct typing and the prevention of stack underflows.

Fig. 4 shows a malformed WebAssembly module that would not pass a correctly implemented validation procedure. Line 3 (**local.get**) pushes the first function parameter (index 0, representing a 32-bit integer) to the stack. Line 4 (**i32.add**) in turn pops two 32-bit integers from the stack and returns their sum. Due to only one integer being present on the stack at that time, namely the function parameter pushed by line 3, this would lead to underflowing of the stack and thus unsafe memory accesses. WebAssembly's validation rules prevent such a module from being executed.

2) Dynamic Runtime Checks: Dynamic runtime checks have to prevent stack overflows. Due to recursive function calls representing valid control flow, the stack can technically grow indefinitely and the runtime must thus ensure that maximum stack height is not exceeded.

With the address of linear memory addresses often being dynamically calculated during runtime, all linear memory accesses have to be furthermore dynamically (i.e. during execution) guaranteed to lie within the allocated address space.

B. Sandboxing WebAssembly on x86-64 Systems

Memory sandboxing WebAssembly modules on x86-64 systems is decidedly easy. Due to the fact that x86-64 desktop systems offer virtualized memory via a memory management unit (MMU), the stack and linear memory can be virtually arranged in such a way that the top of the stack (i.e. the root of a downward growing stack) is aligned with the origin (index 0) of linear memory. Guard pages, (allocated pages of memory with no access rights for the application, e.g. realized with mmap's `PROT_NONE`) can then be positioned before the (downwards growing) stack region and after the linear memory. Due to deliberately restricted access rights, accesses onto these guard pages then raise a signal that can be handled once a fault is encountered. (The WebAssembly module can then either be killed or the reserved memory can be extended in accordance with the WebAssembly specification) Due to being fully handled via the native system providing virtualized memory, this approach has an overhead that is essentially nonexistent.

With WebAssembly using a non-saturating, non overflowing sum of two 32-bit integers (base and offset) to index onto the linear memory [4], the compiler correspondingly has to take care to only allow constants in a 32-bit range or 32-bit registers to index onto linear memory. The guard page is then sized accordingly, in such a way that the sum of the size of the guard page and the actual acessible region of linear memory match the maximally addressable index and thus no memory access can skip past the guard page and access memory beyond the guard page. An analogous guard page can be utilized for limiting the maximum stack size. With memory being fully virtualized, this concept can be extended to multiple concurrently allocated WebAssembly modules within a single process.

As for implementing sub-process scheduling and multitasking (i.e. running multiple WebAssembly modules concurrently within a single process), threads can be utilized in order to hand off scheduling tasks to the underlying system. If a certain module not yielding control (returning) is observed, the respective thread can then be simply aborted from the main thread.

C. Sandboxing WebAssembly on Embedded Systems

Due to the varying availability of memory protection units (MPUs) and memory management units (MMUs) and their vastly differing capabilities, we will discuss some largely platform-agnostic approaches and strategies suited for embedded systems to safely sandbox WebAssembly in ways previously outlined.

In order to enforce memory sandboxing for the WebAssembly stack and linear memory, the compiler has to employ bounds checks for every single memory access (in the case of linear memory) and every time the stack is grown. (e.g. on function entry) By employing complex static analyses, certain bounds checks can be dispensed with. Singlepass compilers (as valent-block compilers are), however, do not have the capacity to perform those analyses. Memory fault isolation via masking is not feasible if out-of-bounds faults shall actually be *detected* and not only redirected to be guaranteed to lie within a valid region. This will therefore often simply lead to corrupted data, which is less than optimal.

As for limiting execution time, there are generally two traditionally used approaches to do so and effectively implement the basis for multitasking: cooperative and preemptive multitasking. The latter leverages interrupts and forced context switches to regain control from an executing program, whereas the former relies on the executing program to (voluntarily and cooperatively) yield its control over CPU resources. In order to preemptively (and enforceably) avoid system locks due to infinite loops, we introduce the concept of *forced cooperative multitasking*. In this sense, we instruct the on-target compiler to emit callbacks or tentative yields at all points that could possibly lead to an infinite loop. This way, the runtime can guarantee that it can regain control over the CPU within a certain maximum execution time. Because it does not rely on specific intricacies and features of a specific platform, this logic is (apart from the concrete machine code) portable across various embedded systems.

VII. Compiler-Guaranteed Sandboxing

We will now discuss how an on-target compiler (e.g. a valent-block WebAssembly compiler like in [9]) can in practice enforce the previously outlined conditions to guarantee sandboxing. For the sake of simplicity and ease of demonstration, we will focus on a single embedded platform, namely one with a 32-bit ARM CPU based on the ARM Thumb-2 instruction set.

Section VII shows exemplary C-code, an equivalent WebAssembly module and the resulting ARM Thumb-2 assembly code (compiled via valent-block compilation). We will use this sample program to illustrate concrete sandboxing concepts and compiler-emitted checks and to demonstrate the effect of the various sandboxing measures on actual emitted machine code (For human readability shown as ARM Thumb-2 assembly).

The assembly code shown is not yet sandboxed and could lead to invalid memory accesses (The address for linear memory that is written to is calculated during execution based on the parameters of the function) or potential locking due to entering an infinite loop (i.e. if $index \bmod 4 \neq 0$);

In this example, the ARM general purpose register **r11** contains the base address of our linear memory region. (Initial setup of this pointer not depicted here) Indexing onto that region via positive offsets will access the linear memory, while negative offsets will access the bookkeeping section containing values like maximum stack height and accessible linear memory size.

In order to maintain brevity, the following descriptions will focus on a single WebAssembly module. The outlined concepts can however be easily extended to multiple parallelly executed WebAssembly modules.

A. Sandboxing of Linear Memory Accesses and Stack Overflow Prevention

In order to sandbox linear memory accesses (stores and loads) without utilizing guard pages or other features of memory management or management protection units and thus be as portable as possible, we have to instruct the compiler to emit bounds checks in order to guarantee that accessed addresses lie in a valid region of memory that is allowed to be accessed.

Section VII-A (left) shows in blue how those bounds checks can be implemented in practice. The compiler encountered a **i32.store** WebAssembly instruction and prepares to emit an equivalent **str** ARM Thumb-2 instruction with the register **r0** indexing from **r11**. Before doing so, the compiler emits additional instructions comparing the accessed memory address to the maximally allowed address stored in the bookkeeping portion in memory (negatively indexed from the linear memory base pointer, in this case **r11**). If the index lies out of bounds, a branch to a **trap** label is emitted. From there on, the runtime can safely abort execution of the executed WebAssembly module.

Section VII-A (right) shows an analogous approach for ensuring that no stack overflow occurs. This is especially important when functions are recursively called. Every time a function is called, a function call frame is created on the stack, preserving the values of registers at call time and recording **lr**, the link register, to allow returning to the caller. Contrary to linear memory, the stack grows downwards. The stack pointer **sp** is, equivalently to linear memory accesses, compared to a reference value in the bookkeeping section of memory. Similarly, a **trap** is executed when a stack overflow is encountered.

B. Limiting Execution Time

In order to limit the runtime of WebAssembly modules and allow to safely abort WebAssembly programs that are stuck in an infinite loop or computations that take too much time, a mechanism for limiting the runtime or transferring control over the CPU back from the WebAssembly module back to the runtime itself is needed. We propose two simple approaches for this, one fully portable (as long as the produced machine code is supported, naturally) and another one depending on statically defined timer interrupts.

The general approach is as follows: In order to be able to limit maximum execution time, the WebAssembly module needs to return control to the runtime either in specified intervals ensuring a maximum execution time interval or regularly check a flag that has been asynchronously set via an interrupt handler and conditionally yield control. This implies that checks need to be emitted at least at any branch target allowing backwards branches and, additionally in the case of very long branchless instruction sequence every N instructions. This implies that, in the case of WebAssembly, those checks have to be inserted at the beginning of functions and at loop labels.

Section VII-A illustrates the two approaches. On the left, at the beginning of the function definition **main**, the compiler emits an unconditional branch to the label **checkyield**, which is under the authority of the runtime. On the right, an asynchronous timer interrupt handler triggered at regular intervals sets a flag in memory (in the module's bookkeeping information adjacent to the linear memory, thus indexing from the same base register) that is analogously checked. If the flag is set, a conditional branch to the, again, runtime-controlled label **yield** is performed.

VIII. Sandboxing Overhead

In order to be able to estimate the approximate performance implications of the previously outlined sandboxing approaches and their respective impacts on execution time, we performed benchmarks comparing execution times off different

```
void main(int base, int index) {
  do {
    *((int *)(base + index)) = index;
    index -= 4;
  }while(index != 0);
}
```

(a) C function

```
(func $main
  (param i32 i32)
  loop $L1
  local.get 0
  local.get 1
  i32.add
  local.get 1
  i32.store
  local.get 1
  i32.const 4
  i32.sub
  local.tee 1
  br_if $L1
  end
)
```

(b) Wasm function

```
.main:
push {r4, r5, lr}
subs sp, #128
.L1:
add r0, r4, r5
str r5, [r11, r0]
subs r5, #4
cmp r5, #0
bne .L1

adds sp, #128
pop {r4, r5, pc}
```

(c) ARM Thumb-2 assembly

Fig. 5: Sample demonstration program in three different, but semantically equivalent, representations

```
.main:
push {r4, r5, lr}
subs sp, #128
.L1:
add r0, r4, r5
ldr r1, [r11, #-4]
cmp r0, r1
IT GE
bge trap
str r5, [r11, r0]
subs r5, #4
cmp r5, #0
bne .L1

adds sp, #128
pop {r4, r5, pc}
```

```
.main:
push {r4, r5, lr}
ldr r0, [r11, #-8]
cmp sp, r0
IT LT
blt trap
subs sp, #128
.L1:
add r0, r4, r5
str r5, [r11, r0]
subs r5, #4
cmp r5, #0
bne .L1

adds sp, #128
pop {r4, r5, pc}
```

Fig. 6: Left: Compiled function with sandboxed linear memory access, right: Compiled function with stack overflow prevention

sandboxing variants, baseline (non-sandboxed) WebAssembly compilation and native compilation.

A. Test Setup

WebAssembly modules were created by compiling C program code via Clang/LLVM (v8.0.1) to its 32-bit WebAssembly target. The same code was also compiled using ARM GCC (all optimizations enabled) for a comparison to natively compiled software. Benchmarks were executed on an ARM (Embedded development board, ARM Cortex M3, 84MHz, ARM Thumb-2 ISA) platform. Due to flash alignment of machine code having an effect on performance on that platform,

```
.main:
push {r4, r5, lr}
bl checkyield
subs sp, #128
.L1:
bl checkyield
add r0, r4, r5
str r5, [r11, r0]
subs r5, #1
cmp r5, #0
bne .L1

adds sp, #128
pop {r4, r5, pc}
```

```
.main:
push {r4, r5, lr}
ldr r0, [r11, #-12]
cmp r0, #0
IT NE
blne yield
subs sp, #128
.L1:
ldr r0, [r11, #-12]
cmp r0, #0
IT NE
blne yield
add r0, r4, r5
str r5, [r11, r0]
subs r5, #1
cmp r5, #0
bne .L1

adds sp, #128
pop {r4, r5, pc}
```

Fig. 7: Left: Compiled function with callbacks potentially yielding control over CPU and thus allowing to limit runtime, right: Compiled function with branchless checking of a flag in memory and yielding if it is set

benchmarks were performed with all possible different flash alignments and the fastest result of each was chosen.

B. Benchmarks

The following benchmarks were used:

- **fib** Calculates the Fibonacci sequence by recursion
- **correlation** An arbitrarily selected benchmark from the PolyBenchC benchmark suite [16]

C. Variants

As outlined in Section VII, the following sandboxing variants were benchmarked:

- **Wasm linmem** Compiled WebAssembly module with only linear memory bounds checks enabled.
- **Wasm stack** Compiled WebAssembly module with only stack overflow protection enabled.
- **Wasm rtcb** Compiled WebAssembly module with runtime limit via callback.
- **Wasm rtflag** Compiled WebAssembly module with runtime limit via setting a flag in a timer interrupt handler.
- **Wasm full** Compiled WebAssembly module linear memory bounds checks, stack overflow protection and runtime limit via flag checking enabled. (Effectively a combination of **Wasm linmem**, **Wasm stack** and **Wasm rtflag**)
- **Wasm base** Baseline WebAssembly performance, none of the described sandboxing features enabled.
- **gcc -O3** Same logic compiled to WebAssembly natively compiled from C to the ARM Thumb-2 target via GCC with the highest optimization level enabled.

D. Results

Benchmark results are illustrated in Fig. 8. Execution times were normalized and are given relative to the result of GCC. The results show, as expected, that runtime limitation via simple flag checking and incorporation of a simple interrupt handler is faster than an unconditional branch and full callback to a function that decides whether execution should be pause or aborted. Furthermore, the results show that even on bare metal embedded systems without memory management units (and thus virtualized memory) memory safety for WebAssembly can be very efficiently enforced by having the compiler emitting bounds checks. The worst results of our benchmarks still lie in the ballpark of native execution with worst-case overheads for full sandboxing in these limited benchmarks of only 29% over native implementation. Interestingly, for fib, the full sandboxing including memory sandboxing and enforced runtime limitation is faster than runtime limitation on its own. We suspect this might be due to lucky alignments of branch targets in flash.

IX. Conclusion

We conclude that WebAssembly, even in a fully software-sandboxed embodiment, can be very efficiently and safely executed on bare-metal embedded systems. Sandbox enforcing on-target WebAssembly compilers represent a novel way of hot-loading and dynamically installing and executing logic on very resource-constrained embedded systems while enforcing memory safety and full control over execution and providing near-native performance. This approach can represent a future solution for remote deployment of executable logic to embedded internet of things devices.

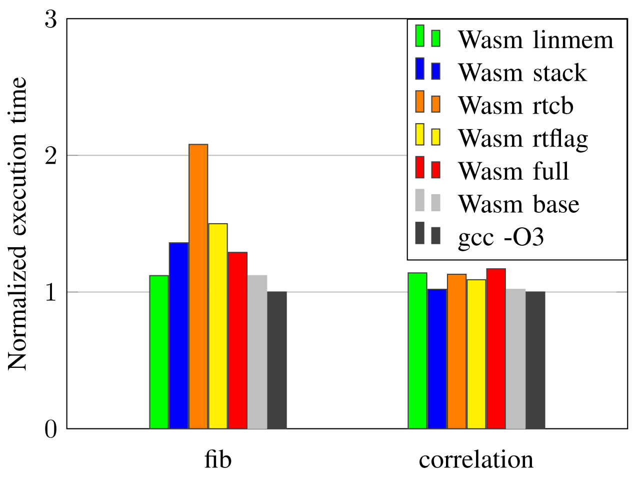

Fig. 8: Benchmark results for various sandboxing mechanisms on an ARM Cortex M3. Results relative to gcc.

References

[1] R. Morabito, V. Cozzolino, A. Y. Ding, N. Beijar, and J. Ott, "Consolidate IoT edge computing with lightweight virtualization," *IEEE Network*, vol. 32, pp. 102–111, Jan 2018.

[2] F. Marchiò, B. Vittorelli, and R. Colombo, "Automotive electronics: Application technology megatrends," in *Proc. 44th Europ. Solid State Device Research Conf. (ESSDERC)*, pp. 23–29, 2014.

[3] R. Morabito, R. Petrolo, V. Loscrì, N. Mitton, G. Ruggeri, and A. Molina, "Lightweight virtualization as enabling technology for future smart cars," in *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pp. 1238–1245, May 2017.

[4] 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," *Proc. 38th ACM SIGPLAN Conf. on Prog. Lang. Design and Impl. (PLDI)*, pp. 185–200, 2017.

[5] M. Jacobsson and J. Wåhslén, "Virtual machine execution for wearables based on WebAssembly," in *Proc. 13th EAI Int. Conf. on Body Area Networks (BodyNets)*, pp. 381–389, 2018.

[6] N. Brouwers, K. Langendoen, and P. Corke, "Darjeeling, a feature-rich VM for the resource poor," *Proc. of the 7th ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, pp. 169–182, 2009.

[7] N. Reijers and C.-S. Shih, "Improved ahead-of-time compilation of stack-based JVM bytecode on resource-constrained devices," *ACM Transactions on Sensor Networks*, vol. 15, no. 3, 2019.

[8] N. Shaylor, "A just-in-time compiler for memory-constrained low-power devices," in *Proc. of the 2nd Java Virtual Machine Research and Technology Symp.*, pp. 119–126, USENIX Association, 2002.

[9] F. Scheidl, "Valent-Blocks: Scalable high-performance compilation of WebAssembly bytecode for embedded systems," *Proc. AEIR Int. Conf. on Computing, Electronics & Communications Engineering (iCCECE)*, 2020. (Forthcoming).

[10] R. P. Putra, "Implementation and evaluation of WebAssembly modules on embedded system-based basic biomedical sensors," Master's thesis, KTH, School of Engineering Sciences in Chemistry, Biotechnology and Health (CBH), Biomedical Engineering and Health Systems, Health Informatics and Logistics, 2019.

[11] R. Gurdeep Singh and C. Scholliers, "WARDuino: a dynamic WebAssembly virtual machine for programming microcontrollers," *Proc. of the 16th ACM SIGPLAN Int. Conf. on Managed Programming Languages and Runtimes (MPLR)*, pp. 27–36, 2019.

[12] K. Hong, J. Park, S. Kim, T. Kim, H. Kim, B. Burgstaller, and B. Scholz, "TinyVM: an energy-efficient execution infrastructure for sensor networks," *Software: Practice and Experience*, vol. 42, pp. 1193–1209, Oct 2011.

[13] F. Besson, S. Blazy, A. Dang, T. Jensen, and P. Wilke, "Compiling sandboxes: Formally verified software fault isolation," *Europ. Symp. on Programming*, 2019.

[14] A. Donovan, R. Muth, B. Chen, and D. Sehr, "PNaCl: Portable Native Client executables," 2010.

[15] C. Hammacher, "Liftoff: a new baseline compiler for WebAssembly in V8," 2018. Accessed Nov 20, 2019.

[16] L.-N. Pouchet, "PolyBench/C the polyhedral benchmark suite," 2010.