Appears in: Proceedings of the 32nd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '26)

WASM-WCET: Worst-Case Execution-Time Analysis of WebAssembly Modules on Updatable Resource-Constrained Embedded Devices

Maximilian Seidler*§, Martin Michelis, Peter Wägemann*, Rüdiger Kapitza*

*Friedrich-Alexander-Universität Erlangen-Nürnberg, {maximilian.seidler,peter.waegemann,ruediger.kapitza}@fau.de

***Abstract*—Updating programmable logic controllers (PLCs) is mostly reserved for their manufacturers (i.e., original equipment manufacturers, OEMs), as modifications to the underlying software may jeopardize system integrity and safety. Consequently, operators of manufacturing systems remain reliant on OEMs for ongoing support, particularly as systems often operate for 25 years or longer. WebAssembly (Wasm), a portable bytecode format, is well-suited to enabling controlled software updates thanks to its spatial isolation, which could give operators greater freedom to adapt their systems. However, temporal isolation remains unaddressed so far. Moreover, using a portable bytecode prevents the application of typical two-stage (i.e., path and hardware) worst-case execution-time (WCET) analysis to ensure timing protection, as the binary is no longer executed directly on the hardware. Instead, the interpreter introduces an additional layer of control flow, including unbounded loops—even for fundamental instructions. Moreover, gaining independence from OEMs requires minimal system interweaving, whereas OEMs must preserve trust in their initial safety mechanisms and their integrity. Finally, PLCs are usually highly resource-constrained.**

With our WASM-WCET framework, we provide the first WCET analysis for Wasm designed to be conducted *on* the resource-constrained device, thereby reducing system interweaving and improving updatability. We integrate a third stage to the WCET analysis and develop the first interpreter cost model for Wasm bytecode. Additionally, we incorporate a specialization prior to analysis to manage unbounded loops in the interpreter's path component, thereby reducing overestimation by up to 98%. We implement a timing-schema–based prototype and present a memory-efficient in-place analysis leveraging Wasm's structured control flow. Our prototype implementation yields results on a microcontroller with 128 KiB of RAM in under 27 ms.

***Index Terms*—WebAssembly (Wasm), worst-case execution time (WCET), static analysis, real-time systems**

I. INTRODUCTION

Updating software components in industrial environments is crucial. Although industrial systems often remain in operation for 25 years or longer [1], [2], [3], the difficulty of future-proofing leads to a demand to modify these systems. The aging of system components negatively affects production by increasing the likelihood of accidents and quality issues [2], with changes to sensors and actuators necessitating adjustments to the operational software. Moreover, technological advancements, including sophisticated data processing, present opportunities for improving manufacturing by lowering production [4] and operational cost, and minimizing downtime [5], [6]. Furthermore, evolving regulatory landscapes may necessitate updates due to extended monitoring of production processes and their environmental conditions [7], [8], [9].

However, operators managing the production system face significant limitations in adapting to evolving requirements due to the design of programmable logic controllers (PLCs), the backbone of modern manufacturing systems. PLC programming languages [10], [11] are primarily designed for operational logic, which deliberately limits their capabilities, safeguarding non-functional system properties against accidental interference, such as timing and data violations. Additionally, producers of PLCs, the original equipment manufacturers (OEMs), hesitate to allow custom firmware modifications by operators to preserve the integrity of the base software's certification and avoid accountability concerns. Consequently, operators and OEMs seek update mechanisms that provide greater autonomy, while maintaining the system's safety, and without violating its certification. These mechanisms should facilitate post-deployment integration of isolated modules with programmatic capabilities beyond standard PLC code.

WebAssembly in Industry: One potential solution to address update mechanisms is WebAssembly (Wasm). Wasm is an emerging bytecode instruction set. Its flexibility and modularity allow for the integration of software components at the firmware level, while its robust memory isolation safeguards against system interference and ensures compliance with certification. Its small execution overhead and memory footprint make it well-suited for utilization on resource-constrained devices in a sandboxed environment. Consequently, interest in the embedded [12], [13], [14], [15], [16], [17] and safety-critical [18], [19] area and its industrial adoption [20], [21], [22], [23] is growing. While Wasm supports multiple execution modes, including ahead of time (AOT) compilation, in safety-critical contexts, compiling to native code places the burden of verification on the compiler, complicating the safety assurance processes. Conversely, interpreters typically have a smaller codebase and well-defined functionality, leading to a smaller trusted code base. Furthermore, the mechanization [24] of Wasm's formal specification allows the derivation of formally correct interpreters [25]. Consequently, current works [18], [26] in safety-critical domains prefer interpreters, particularly when execution speed is not a limiting factor.

§also affiliated with Siemens AG

The Necessity of On-Device Timing Protection: While Wasm ensures memory isolation, post-deployment integrated software modules must also adhere to the initial timing requirements. Operators responsible for creating the updated software modules should adhere to these requirements. However, to ensure the integrity of the PLC and to maintain compliance with liability requirements, OEMs must implement mechanisms to enforce the correct temporal behavior. Therefore, we envision a timing protection mechanism that rejects non-conformant updates. To safeguard against bypassing this protection mechanism, the analysis must occur at a non-skippable point. One such location is the device itself when receiving the updates. However, ensuring timing protection at the device level necessitates the estimation of worst-case execution-times (WCETs) for incoming programs locally.

Research Goal & Challenges: In this paper, we present WASM-WCET, a framework to provide timing protection in updatable, long-life devices for industry automation. To facilitate modular updates, Wasm binaries are utilized. For preventing the execution of timing-violating programs, an acceptance test is performed on the device, based on an on-device WCET analysis. However, the on-device WCET analysis of Wasm programs presents novel, unique challenges.

Traditional WCET analysis [27] models worst-case execution paths and integrates them with hardware models that define execution costs for instructions, resulting in two stages (i.e., hardware & path analysis). However, portable instruction formats like Wasm use a runtime that serves as a virtual execution hardware. Consequently, the instruction cost model for physical hardware cannot be applied. This issue had already been identified by the real-time systems community more than twenty years ago with Java [28], [29], [30], and Wasm's simpler architecture appears more suitable for WCET analysis. However, its different design complicates quantifying runtime effects, an aspect that has not been addressed so far.

For practical long-term on-device analysis, WCET analysis must act independently of OEM dependencies as prolonged device lifetimes increase maintenance, operational infrastructure, and the risk of supply chain attacks. Finally, while traditional WCET analyses typically occur offline with access to substantial computational resources, on-device analysis must comply with strict resource constraints (i.e., $\leq$ 256 KiB RAM). Besides these strict memory constraints, the analysis times have to be low to avoid impractical update durations.

Contribution & Outline: WASM-WCET is an approach to enable on-device timing protection for updatable constrained devices at the example of interpreted Wasm bytecode. Our contributions are three-fold:

1) **Interpreter Cost Model & Load-Time Interpreter-Model Specialization**: We develop the first interpreter cost model that provides execution costs per Wasm instruction. To achieve feasibility and reduce overestimation, we introduce three measures consisting of (1) restrictions on the Wasm bytecode, (2) a software cache model, and (3) *load-time interpreter-model specialization*.
2) **Updatability by Self-Contained Analysis**: We introduce a comprehensive timing-schema–based method [31] for WCET estimation, which provides timing bounds for arbitrary Wasm programs. As we aim for the updatability of long-life devices, dependencies are minimal, rendering the analysis *self-contained*.
3) **Resource Constraints & In-Place Mechanism**: We conduct the first WCET analysis on memory-constrained (128 KiB) devices. To enhance memory efficiency, we leverage features inherent to the Wasm bytecode, allowing us to parcel out WCET calculation and avoid creating the entire control-flow graph. Thereby, we introduce an *in-place* implementation for our timing-schema–based mechanism.

II. Background

This section presents relevant background on Wasm, its use in industry, and current methodologies in WCET analysis.

A. WebAssembly

```
WASM_OP_BR_IF:
    uint32_t label = read_leb128_u32(&p);
    int32_t condition = *(--osp);
    if (condition) {
        Frame *tgt_frame = &cs[csp - label];
        find_end_address(&p, tgt_frame);
    }
goto *op_handles[*p++];
```

Listing 1: An excerpt of a Wasm interpreter: It shows a conditional branch that encompasses multiple unbounded loops, including the label LEB128 decoding and several iterations for finding the block's end address to jump to.

Wasm [32], [33], a low-level intermediate instruction format, was designed as a compile target characterized by its formal specification, strong isolation, and portability. To enhance analyzability, Wasm ensures structured control flow at the binary level by (1) creating control blocks with explicit instructions that mark the start and end and by (2) referencing jump targets relative to these blocks. Its bytecode format is designed for compactness, using little endian base 128 (LEB128) number representation for reduced code size. With its formality and small execution overhead, Wasm is well-suited as a sandboxed alternative for constrained devices, finding numerous applications in industry [20], [21], [22], [23].

Wasm operates on a stack-based virtual machine. Although it offers AOT and just in time (JIT) compilation, safety-critical contexts often prefer interpretation [18], [19]. AOT compilation produces native binaries, preventing the leveraging of Wasm's benefits: Native binaries offer limited portability and lack a formal instruction set, requiring individual certification for each instruction set—or relying on a certified compiler. JIT compilation is impractical due to its resource requirements and volatile execution times. In contrast, interpretation can utilize the formal specification of Wasm, enabling the development of formally correct interpreters [25] and is resource-efficient.

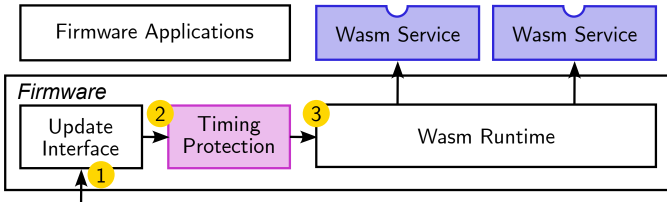


Figure 1: The architectural design has a timing protection mechanism (2) situated between the update interface (1) and the subsequent processing and executing of Wasm programs (3), to facilitate rejecting received programs that could compromise the system's temporal integrity when scheduled.

#	Operation	Occurrences	Number of Unbounded Loops
1	`local.get`	22.58%	1× (LEB 128 decode)
2	`i32.const`	19.92%	1× (LEB 128 decode)
3	`i32.add`	10.92%	0×
4	`local.tee`	6.92%	1× (LEB 128 decode)
5	`i32.load8_u`	5.15%	2× (LEB 128 decode)
	...	...	...
15	`br_if`	1.13%	**16×** (find target, LEB 128 decode, copy values)

Table I: Even the most fundamental and frequent Wasm instructions involve the execution of unbounded loops, with `br_if` exhibiting sixteen of them.

The concept of an interpreter is illustrated in Listing 1. The pointer `p` iterates through the bytecode by jumping to C labels that handle the corresponding Wasm bytecode instruction. The excerpt demonstrates a conditional branch that pops its condition from the current operand stack pointer (`osp`). If the condition holds true, the interpreter accesses the target frame from the control stack (`cs`) using the relative `label` and searches the bytecode to locate the branch target and set `p` accordingly. The `goto` statement fetches the next opcode and finds the handler in the predefined `op_handles` table.

B. WCET Analysis

Traditional WCET analysis follows a two-stage approach [27], as illustrated on the left side of Figure 2: First, the hardware-agnostic path analysis determines the value ranges that influence program behavior and identifies possible control-flow paths. Second, a hardware analysis assigns the execution cost of machine-code instructions to program paths by consulting a hardware model. To address the path component, two main techniques have emerged: (1) the tree-based timing schema [31] and (2) the implicit path enumeration technique (IPET) [34], [35]. The former consolidates possible control flow paths into one worst-case path, while the latter models all possible control flows as an integer linear program (ILP). Most WCET tools, including commercial analyzers (e.g., aiT [36], TimeWeaver [37]), rely on the IPET. Although IPET can yield more precise results, it necessitates the use of ILP solvers, which imposes significant demands on computational performance and memory. As detailed later, for the WASM-WCET approach, we identified that the timing schema is the only feasible way to tackle on-device WCET analysis for resource-constrained embedded devices. For portable bytecodes, an additional stage that attributes the runtime contributions must be integrated, e.g., by using parametric costs [29]. However, the modeling of this contribution, including its origin and complexity, poses significant challenges with Wasm.

III. System Model & Problem Statement

As OEMs aim to ensure long-term reliability, we assume an additional acceptance test directly on the device. The architecture, illustrated in Figure 1, features devices equipped with an update interface (1) that manages communication, integrity checks, storage, etc., to receive new software components. Prior system integration, the timing acceptance test (2) is performed, enabling the device to reject any component with unsatisfactory schedulability, thus ensuring timing protection. Once accepted, the software can be scheduled and executed through the Wasm runtime (3).

We base our work on the 1.0 version of Wasm with the multi-value extension [38], which limits programs to single-threaded execution, primitive data types, and prevents garbage collection. Due to its predictability, this version is widely used in industry, with efforts to lay the foundation for standardized long-term support [39]. Consistent with real-time programming practices and the use of standards like Misra-C [40], we exclude dynamic function calls (i.e., `call_indirect`). We expect the use of an interpreter for its straightforward use in safety-critical environments, and assume the WebAssembly micro runtime (WAMR) due to its broad industry adoption. To address a diverse range of PLCs, we assume memory constraints of less than 256 KiB. The tasks of Wasm modules run without preemptions and, as a consequence thereof, scheduling effects and (cache-/pipeline-related) preemption delays are not relevant in the scope of WASM-WCET.

A. Challenges

To achieve portability across different hardware, industrial systems employ portable intermediate bytecode formats. However, typical static WCET analysis, illustrated in Figure 2 on the left, is closely tied to the hardware. The two-stage WCET analysis assumes that program execution occurs on hardware modeled by the cost model. In contrast, this assumption is no longer valid, especially in the context of interpretation. Rather than directly interfacing with physical hardware, program instructions interact with the runtime environment. However, handling these instructions involves executing sequences of native instructions, as they must translate the more abstract intermediate format into the specific hardware platform format. This translation process is specific to the bytecode and involves additional lookups, decodings, checks, and environment management, thereby introducing an extra layer of control flow, depicted in Figure 2 on the right side. For Wasm, this extra layer is particularly complex because it incorporates additional loops, where the number of iterations cannot be bound off-line in the general case (*unbounded loops* in the following). These unbounded loops at the intermediate layer differ from those in traditional WCET analysis at the program

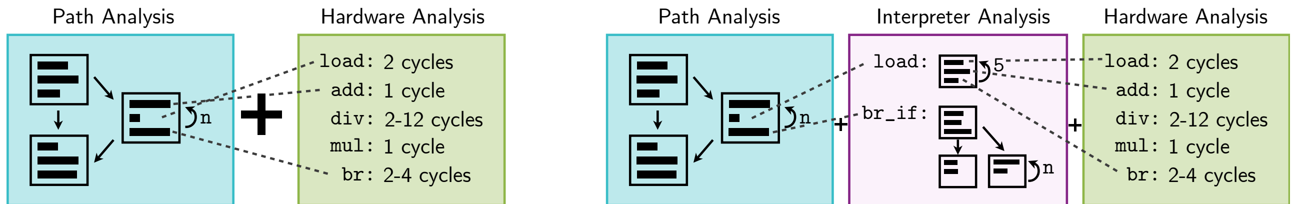


Figure 2: *Left side:* The traditional two-stage WCET estimation integrates path analysis with a cost model based on hardware analyses. *Right side:* In contrast, estimating WCETs for interpreted bytecode requires consideration of the interpreter, which introduces an additional path component for costs containing unbounded loops.

layer, as they prevent the creation of an interpreter cost model. Table I presents the most frequent opcodes from the Wasm benchmark used later in this work's evaluation. Notably, even basic—and most frequent—operations exhibit unbounded loops. Branches introduce even more complexity, with `br_if`, the most frequent one, yielding *16 unbounded loops*. Without considering such unbounded loops, creating an interpreter model for intermediate formats, such as Wasm, is infeasible. Subsequently, we first detail the conceptual challenges that arise, followed by the problems when applying it to the scenario of on-device analysis.

Challenge 1 – Problem of Interpretation: We identify three sources of unbounded loops that must be addressed. First, unbounded loops may arise from the bytecode's *specification*. Each portable bytecode is designed with specific goals that set it apart from native instruction sets. Therefore, execution entails the runtime translating back into the native instruction set while accounting for the intermediate format's design. This includes adhering to the format's memory model, performing integrity checks, managing housekeeping tasks, and decoding values. If this adds a constant offset to the execution time, it is unproblematic, but it can also lead to unbounded loops, depending on the format's design. For instance, in the case of Wasm, integer values are encoded using *LEB128*, which permits arbitrarily large values, but also leads to unbounded decoding and encoding. By revisiting Table I, we can see that this de- and encoding affects even the most fundamental operations, such as loads, stores, and reading constant values.

The second source of unbounded loops stems from additional unavailable information when comparing the analysis of intermediate formats to contemporary analysis: the intermediate *binary* itself. Therefore, upper bounds must also be found for constructs constrained by the binary, even though the binary is static after loading. This scenario is analogous to constant global variables in contemporary analysis and can be found in other portable bytecodes. In the example of Wasm, this is evident with functions, as they permit an arbitrary number of parameters and return values. Consequently, the number of push and pop operations on the operand stack is unknown during interpreter model creation, although the bytecode layout gives an upper bound. This flexibility results in a runtime implementation with unbounded loops.

The third origin of unbounded loops is attributed to the *runtime state*, specifically illustrated by the interpreter's end instruction cache. For Wasm, control structures (i.e., branch targets, conditionals, and loops) consist of a starting operation and an `end` instruction. However, enclosed branch operations require the increment value for the instruction pointer to jump to the correct `end` location. Therefore, the interpreter must perform a forward search in the bytecode each time a branch is encountered. The interpreter excerpt presented in Listing 1 exemplifies this mechanism. To enhance runtime efficiency, runtimes employ software caching mechanisms that significantly impact the costs of branch instructions. Thus, effective cost models must account for the current runtime cache state to determine the appropriate instruction costs.

To address the various influences from different levels (i.e., specification, binary, runtime state), we derive bounds based on the runtime's implementation and introduce artificial upper bounds by non-critical restrictions to the binary. Further, we introduce a mechanism that we term *load-time interpreter-model specialization* to enhance binary-level bounds and effectively model the software cache via software mechanisms.

Challenge 2 – Problem of OEM Dependence: The objective of this work is to empower OEMs to ensure the long-term updatability of PLCs by explicitly enabling post-deployment integration of program code while preventing temporal interference with integrated components. Given that standards acknowledge the potential for errors in data and code, including human-induced issues, OEMs strive to mitigate this risk. Thus, any input data that OEMs cannot independently verify is classified as untrusted. However, in practice, these devices are typically organized in isolated networks, disconnected from the public internet. While this enhances overall device security, it limits the OEM's ability to conduct remote integrity checks post-deployment, leaving the device itself or its local infrastructure to ensure its safety.

Long lifetimes, however, increase the burden that any external dependency imposes, as they require the conservation of this infrastructure for long periods—including software and hardware. Contemporary state-of-the-art WCET analyses typically employ advanced static analysis techniques. While this approach accounts for the complex control flow, it introduces substantial computational complexity, requiring offloading the workload of corresponding (ILP) solvers to operate on commodity or server-grade hardware. Preserving the corresponding soft- and hardware dependencies grows organizational efforts and increases the risk of supply chain attacks.

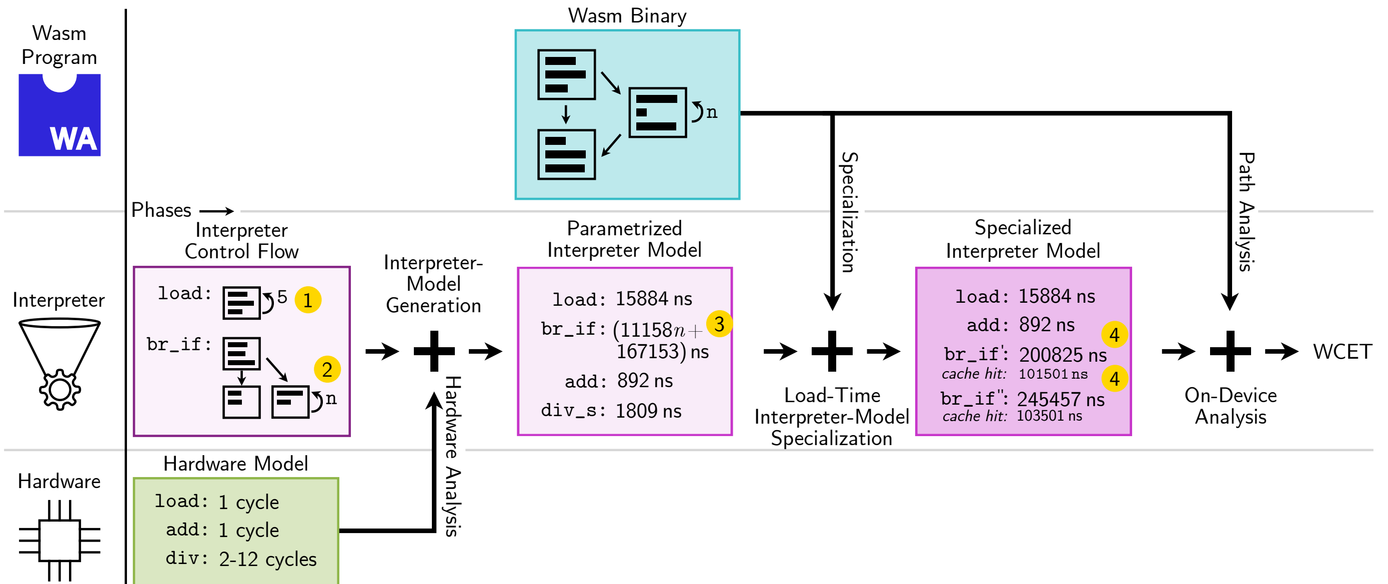


Figure 3: WASM-WCET approach: We address the additional layer added by interpreters in multiple phases. During the analysis of the interpreter's control flow, we bound loops through semantic analysis, combined with adding limitations to the bytecode (1). Unbounded loops (2) result in parametric costs (3). During loading the Wasm bytecode, we record the bytecode's property that corresponds to the loop bound and derive specific costs for each variant of the instruction that occurs (4).

To minimize reliance on OEM dependencies and avoid the need for external computational power, we employ a fully self-contained timing-schema–based [31] approach that performs binary parsing, control-flow graph (CFG) construction, loop reduction, and WCET calculation.

Challenge 3 – Problem of Resource Constraints: PLCs encompass a broad range of functions, resulting in similarly diverse hardware capabilities, sometimes with limited computational resources, such as those found in sensors or programmable gateways. Typical memory requirements limit RAM to below 256 KiB [41]. Given that the analysis must be conducted on the device, we must address these significant resource constraints. Choosing a timing-schema-based approach conceptually demands less memory than IPET-based solutions, which can require gigabytes of RAM. Nevertheless, implementing this analysis with constrained memory necessitates careful memory management. Furthermore, the on-device requirement mandates that the timing schema retains the entire CFG in memory, resulting in a linear increase in memory consumption proportional to the program's size.

To reduce memory consumption, we carefully optimize our analysis and convert our analysis algorithm to an *in-place* approach that leverages Wasm's structured control flow, allowing for the collapsing of basic blocks (BBs) and yielding a solution with minimal memory consumption.

IV. CONCEPT & IMPLEMENTATION

This section decomposes the overall analysis process into distinct phases, as shown in Figure 3. Prior to deployment, OEMs create the parametrized interpreter model followed by a specialization, once the Wasm binary gets available at load time. In the following, we present the interpreter model generation and its specialization, followed by the path analysis of the Wasm binary. Finally, we address resource constraints by introducing our in-place algorithm.

A. Interpreter & Bytecode Cost Model

To derive the interpreter cost model for a specific device, we use aiT [36], an industrial-grade static timing analyzer, in conjunction with the hardware model of the device and the native binary. In addition to the Wasm runtime, this binary encompasses the entire final firmware and operating system (OS), as any calls to firmware, OS, or library functions influence timing behavior and must be accurately integrated into the interpreter cost model. Since OEMs can perform this analysis offline, advanced analysis techniques can be utilized to achieve precise timings. For each Wasm opcode, the corresponding interpreter instruction sequence is analyzed from its entry point until it fetches the next opcode, resulting in an upper bound on execution time per Wasm opcode, i.e., the interpreter cost model. However, the loops presented in Challenge 1 hinder the generation of costs for most instructions entirely using this procedure, while others suffer from significant overestimation. Therefore, we introduce four mechanisms to derive additional loop bounds for the interpreter's unbounded loops, thereby enhancing feasibility and reducing overestimation.

Loop Bounds Through Semantics: For our first measure addressing *specification*-induced loops, we find upper bounds through semantic code analysis. These loops are mainly a direct result of the intermediate format's specification and its implications for the runtime implementation. For Wasm, a significant number of unbounded loops arise from operations performing bitwise processing, such as `popcnt`, `ctz`, and LEB128 integer processing, as they operate on

`while(true)`-loops with a special termination criterion. However, Wasm's numeric values are specified as fixed-length values of either 32-bit or 64-bit, implicitly defining upper bounds. Consequently, we derive and annotate these bounds for the analysis of the runtime, illustrated on the left in Figure 3 (1). The same methodology also applies to upper bounds concerning the normalization of floating-point numbers according to IEEE-754, which are utilized in Wasm. Additionally, we identified one unbounded loop where tracing of the call sites reveals its upper bound.

Artificial Loop Bounds: Our second method involves applying artificial loop bounds, as no loop bounds can be derived from the loop's semantics: We introduce limitations on specific bytecode features, allowing us to assume the corresponding bounds in the runtime. Therefore, this category represents one approach to address *binary*-level influences. For Wasm, the specification allows the number of parameters and return values of functions or blocks to be infinite. In practice, the runtime implementation gives a bound, as it uses an unsigned 32-bit value for this number. However, using the implicit bound of $(2^{32} - 1)$ leads to high pessimism. Additionally, we find an implementation-specific case in the WAMR interpreter where consecutive NOP instructions after a `br_table` opcode introduce another unbounded loop. To ensure feasibility and reduce pessimism, we introduce artificial upper bounds for all three loops, thereby restricting the functions to a maximum number of return values and parameters, and limiting the number of consecutive NOPs after a `br_table`. This is also applied offline for interpreter-model generation (Figure 3, 1).

Load-Time Interpreter-Model Specialization: In some instances, these artificial loop bounds can exhibit significant volatility across different binaries, making fixed artificial restrictions overly limiting for some binaries while resulting in excessive overestimations for others. For these loops (2), we employ parametrized costs, as shown in the center of Figure 3 (3). Rather than imposing fixed artificial loop bounds, we use an abstract parameter that adjusts the cost for each Wasm instruction affected by the specific loop, i.e., the Wasm instruction cost is a linear function of this parameter. Then, we identify how the bytecode defines the parameter's value by analyzing the interpreter's implementation. As this parameter depends solely on the bytecode, we incorporate a specialization prior to analysis by recording the parameter's value during binary parsing. Subsequently, we resolve our parametrized interpreter model to provide specific instruction cost variants (Figure 3, 4). We call this the *load-time interpreter-model specialization*. This specialization applies to both *specification*-level and *binary*-level unbounded interpreter loops. For Wasm, we apply this specialization for branches in control blocks, i.e., `br`, `br_if`, `br_table`, and `if`, and the parameter is the distance between the branch instruction and the control's `end`. Consequently, our cost model provides costs specific to the opcode's position in the bytecode. Notably, even in practical programs, the forward search for the `end` opcode can range from a single instruction to the entire program.

Runtime State & Software Cache Model: The fourth measure addresses *runtime state* influences by modeling the interpreter's state. We demonstrate its impact on the example of the instruction-end cache state as a critical subset of the overall runtime state. Estimating the cache state enables us to account for unbounded loops in specific scenarios. The WAMR used in this work features a set-associative two-way first-in-first-out (FIFO) software cache with 64 sets, optimizing end-of-control-block lookups without iterating over the ensuing bytecode during runtime. While our interpreter model generally assumes cache misses, we enhance the cost model by incorporating loop-level persistence analysis and assuming first-miss behavior within loop bodies.

The set index is derived from the lowest five bits of the control block's instruction address in the bytecode, which is statically known. Consequently, we can integrate the persistence analysis into the load-time specialization step, which comprises a form of abstract interpretation. During binary traversal, we confirm that identical lower five bits appear at most twice per loop, thereby proving first-miss cache lookups [42]. To account for function invocations within loop iterations, we must capture their impact on the global cache. Our simplified confirmation approach can employ the union of all reached set indices in the subsequent call graph, since order is negligible. Notably, cache lookups occur at control block starts, which are less frequent than typical load or store instructions. Additionally, the maximum control stack size limits control block nesting, and OEMs may adjust the (software) cache sizes accordingly.

In our prototype, we leverage this first-miss property and adjust costs based on previous loop iterations when encountering branch instructions within a control block. We incorporate these alternative costs into our specialized interpreter cost model. We derive miss and hit latencies from two offline analyses: one for the worst-case path of a cache miss that includes the unbounded search for the end instruction, and another for a cache hit that avoids the search.

B. Self-Contained On-Device Analysis

As outlined in Challenge 2, the WCET analysis must be self-contained and conducted on the device. This necessitates the execution of the four tasks locally: parsing the received binary, generating its CFG, reducing the cyclic CFG into a directed acyclic graph (DAG), and calculating the WCET.

To parse the incoming Wasm binary, we leverage the WAMR parser, which is optimized for resource constraints. This parsing process validates the binary against the Wasm specification and produces a Wasm module structure that enables identification of the code sections corresponding to each function. Constructing the CFG for the marked entry function involves traversing the function's code to collect instructions into BBs. Upon encountering control opcodes, we link BBs based on the opcode, forming a CFG with the BBs as nodes. In our case of Wasm, we leverage its bytecode design. Wasm employs a control stack in which control instructions push/pop frames. We apply this concept and push/pop BBs.

When we encounter `loop`, `if`, or `block` opcodes, we push a new control frame onto the stack. Upon hitting an `end` opcode, we pop the frame and connect a new BB. Branch instructions, i.e., `br`, `br_if`, and `br_table`, reference their targets relative to this control stack, facilitating straightforward identification of target BBs. If the target BB does not yet exist, we create a temporary BB as a jump target. This architecture ensures proper nesting and prevents arbitrary jumps, thereby eliminating the need for splitting BBs during CFG generation. The opcodes `return` and `unreachable` denote termination points of the CFG, as each function is analyzed individually. Additionally, for each BB, we record the parent loop, if applicable, and the count of instructions within the BB, which we require for later analyses.

To address loop bounds, we propose two approaches. In many programs for real-time embedded systems, data constructs are fixed-size, leading to constant loop bounds. As traversing the bytecode for CFG creation is a form of symbolic execution, we can monitor the constant values pushed to the operand stack and directly leverage them as loop bounds when used in the loop's break condition. For more complex bounds, we introduce a pseudo-function that enables programs to express their loop bounds. Wasm supports function *imports* at the binary level, which are commonly utilized. For instance, in C, such calls can be generated by calls to external functions. Upon encountering this function call, we store its parameters for the subsequent loop annotation. It is essential to note that the loop-bound pseudo-function call is removed during the copying of the Wasm binary before execution. Additionally, during this process, we can mark the loop to instruct the runtime to ensure compliance with the established bounds when executing; however, this incurs considerable overhead that must be accounted for in the analysis.

After CFG creation, we transform the cyclic CFG into a DAG for the summation of BBs' costs. Each loop, represented as a cyclic subgraph in the CFG, is replaced with a substitute BB, as explained in Section IV-C. The incoming and outgoing edges of all BBs within the subgraph are then reconnected to this substitute BB. This procedure is applied recursively to nested loops, leveraging Wasm's structured control flow that prevents irreducible or overlapping loops. In our prototype, we identify all BBs that constitute the loop body by traversing their successors and matching the parent loop recorded during CFG construction with the loop currently under reduction.

Given the unavailability of the high resource demands of IPET-based WCET calculation, we employ a timing-schema-based [31] approach. We accumulate the WCET of sequential statements, utilize the maximum for branches, and multiply the loop body by the upper bound for loops:

$$
\begin{aligned}
t_{\text{WCET}}^{\text{sequential}}(\mathcal{BB}) &= \sum_i t_{\text{WCET}}(BB_i), \\
t_{\text{WCET}}^{\text{branch}}(C, BB_1, BB_2) &= t_{\text{WCET}}(C) + \\
&\qquad \max\left(t_{\text{WCET}}(BB_1), t_{\text{WCET}}(BB_2)\right), \\
t_{\text{WCET}}^{\text{loop}}(n, C, \mathcal{BB}^{\text{body}}) &= n \cdot \left[t_{\text{WCET}}(\mathcal{BB}^{\text{body}} + t_{\text{WCET}}(C))\right]^{1},
\end{aligned}
$$

with BB_i a BB, $\mathcal{BB} = \{BB_1, BB_2, \dots, BB_i\}$ a set of BBs, C the condition and $n \in \mathbb{N}_0$ the upper loop bound.

Addressing the runtime state depends on the specific state type and fraction being modeled. Consequently, modeling the runtime's state can influence CFG creation, loop reduction, the WCET of Wasm instructions, or any combination of these factors. In our context of WAMR, we address software cache modeling during loop reduction by selectively applying an alternative WCET for each branch instruction. During loop substitution, we calculate two WCETs for the loop body, i.e., the substitute BB: one based on the software cache hit WCET and the other on the cache miss WCET. We model a simple cache policy with a cache miss for the initial loop iteration and cache hits for all subsequent iterations. Although not accurately reflecting the actual cache policy—given its assumption that each loop has not been previously executed—this cache model demonstrates a significant reduction in overestimation as shown in our evaluation, especially for nested loops. A notable limitation is when functions are invoked within loops, as this can invalidate the cache during iteration; however, this situation was not encountered in our evaluation.

C. Resource-Efficient In-Place Technique

In our previous section, we presented four conceptual tasks for WCET estimation: parsing the Wasm binary, creating the CFG, reducing the DAG, and calculating the WCET. However, given the stringent memory constraints of our target platforms, we favor a more efficient implementation.

In our implementation, we consolidate these tasks into a single pass, referred to as the *in-place* variant. Moreover, we directly perform WCET estimation on subgraphs, eliminating the need to store the complete CFG. For that, we require an intermediate bytecode that, during CFG construction, denotes when subgraphs are completed, i.e., no further nodes or edges are added, and ensures that no jumps can target the generated subgraph. Otherwise, the WCET calculation of a partial subgraph is required. This property holds for Wasm due to its end markers for control structures.

We jointly conduct CFG construction, DAG reduction, and WCET computation: For each popped control frame, we know that—by design—the BBs corresponding to the frame are no longer addressable through jumps. As a result, we promptly compute the WCETs for these BBs and subsequently apply the timing schema to the frame's subgraph. It is noteworthy that branch instructions may introduce edges that exit these subgraphs, which are excluded from consideration in this process. While exclusion normally does not compromise the WCET calculation, because BBs where those instructions are located record the timing of the branching instruction, it prevents the determination of the opcode distance between the branch instruction and the referenced control structure's `end` instruction. As our load-time interpreter-model specialization requires this distance, we postpone collapsing subgraphs with

[1] This expression differs from [31], as in Wasm, loop conditions are evaluated after the execution of the loop body, and we integrate the exit cost into the condition, providing a sound overestimation.

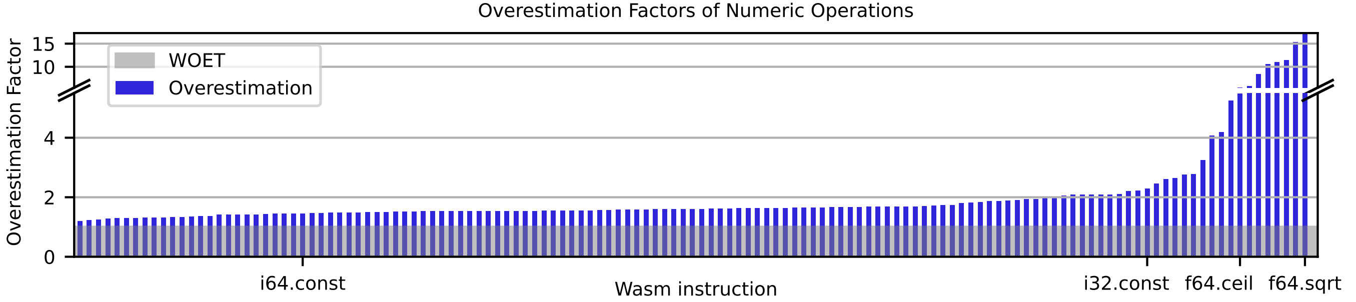


Figure 4: Overestimation for executing numeric single instructions consecutively: The overestimation factors range from 1.21 to 17.29, with 25% being 1.50 or less, 79% below 2.00, and 95% below 5.50, starting at the expected factors but increasing due to unencountered worst-case paths in the interpreter.

outgoing edges until the parent subgraph is closed. Although this presents a limitation, it only occurs in the special case where every branch instruction references the binary's end, resulting in an equivalent to the non-in-place (i.e., out-of-place) variant that requires storage of the full CFG. In all other cases, it reduces memory usage.

V. Evaluation

We subsequently present our evaluation, demonstrate feasibility, and pinpoint overestimations. We conduct our evaluation on an XMC4500 Relax Kit Lite featuring a 120 MHz Cortex-M4F CPU and 64 KiB + 64 KiB SRAM. We select this hardware platform for its accessible processor and its similarity to small PLCs [41]. In practice, PLCs often use proprietary CPUs, and we assume OEMs will provide the necessary processor models. We implement Wasm-WCET based on the Zephyr OS with WAMR. For time measurements, we instrument the WAMR classic interpreter by probing the creation and destruction of Wasm execution frames.

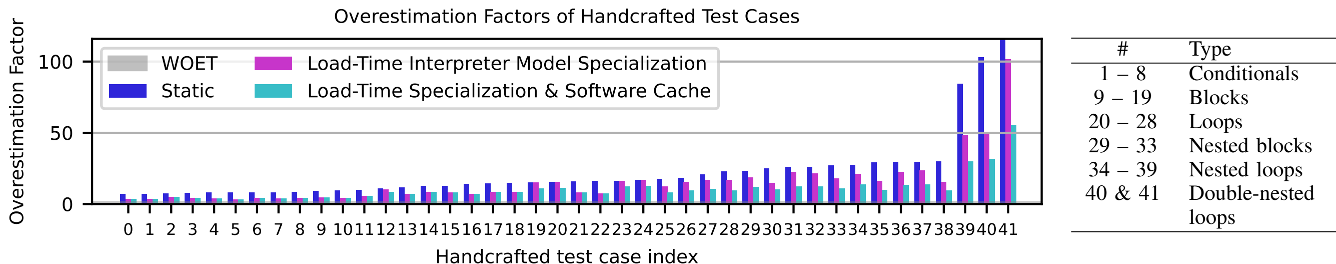
When conducting the on-device WCET, we allocate the full 64 KiB of DSRAM1 for the analysis. For execution, we reuse the same memory for storing the Wasm binary and WAMR housekeeping. Moreover, we reserve the entire 64 KiB of PSRAM1 for Wasm's linear memory. Further, we heuristically adjust the default Wasm memory page size from 64 KiB to 15.75 KiB, resulting in a maximum of 4 pages. In explicitly mentioned cases, we conduct the analysis on a commodity Linux system with an AMD Ryzen CPU and 28 GiB of RAM.

A. Interpreter Cost Model Overestimation

Single Instructions: To assess our interpreter cost model, we quantify the overestimation associated with numeric instructions, including constant usage, comparisons, type conversions, arithmetic operations, mathematical floating-point computations, and bitwise operations. We generate test cases that repetitively execute single instructions. We prepare the operand stack with values; WCET estimation and measurement start after this preparation. As some instructions require dropping values from the stack post-execution, we subtract this part from the WCET and the measured execution time. Due to the lack of a baseline, we calculate overestimations by dividing estimates by worst-observed execution-times (WOETs).

The resulting factors, depicted in Figure 4, range from 1.21 to 17.29, with 25% being 1.50 or less, 79% below 2.00, and 95% below 5.50. For the specified setup, aiT typically yields a geometric mean overestimation of 1.36 [43], closely aligning with our measurements. We observe increased overestimations for Wasm's special mathematical instructions (e.g., `nearest`, `floor`, `ceil`, etc.), with the `sqrt` opcode exhibiting the highest measured overestimation. Moreover, bitwise operators show factors exceeding eight. We attribute the significant overestimations of all these cases primarily to not encountering the worst-case path within the interpreter. For instance, our constant numbers are selected for worst-case LEB128 decoding but may not correspond to the operation's worst-case input. In many arithmetic and comparison operations, the 32-bit variant displays approximately 1.5 times the overestimation of its 64-bit counterpart, particularly evident when comparing `i64.const` (1.46) and `i32.const` (2.30). This discrepancy arises because the interpreter applies uniform routines across varying bit sizes, leading to a higher loop bound assigned to the 32-bit variant during our interpreter model creation. Additionally, our measurement process introduces minor inaccuracies. Overall, the results indicate which specific instructions involve complex control flow in the interpreter, resulting in a pronounced overestimation, including cases where the worst-case input data of different code segments in the interpreter vary. Nevertheless, the general range of our measurements is consistent with aiT's mean overestimations.

Control-Flow–Based Overestimation: To evaluate the influence of the control flow, we handcraft a benchmark set in Wasm's text format that directly translates to the binary format, thereby ensuring exact control over the resulting Wasm binary. This benchmark set consists of binaries with key control-flow operations across diverse nesting configurations mixed with non-control instructions. To ensure the worst-case path is executed, we establish static loop bounds and conditions. To assess the mechanisms presented in our concept, we employ the interpreter model in three configurations by subsequently stacking them. First, both semantic and artificial interpreter bounds are necessary for feasibility, and in combination will



#	Type
1 – 8	Conditionals
9 – 19	Blocks
20 – 28	Loops
29 – 33	Nested blocks
34 – 39	Nested loops
40 & 41	Double-nested loops

Figure 5: Overestimation factors for key programs: The overestimations remain high, while our load-time interpreter-model specialization and cache model demonstrate significant reductions of up to 62% and 45%, respectively.

henceforth be referred to as the *static interpreter model*. Second, we add the load-time interpreter-model specialization. Third, we integrate both load-time interpreter-model specialization and software cache modeling. For the static interpreter model, we identify the maximum distance between a branch and its corresponding block end across all handcrafted benchmarks, thereby ensuring analyzability without prior binary scan. Further, we perform a test run before the measurement run to confirm that our assumptions about cache evictions remain valid for our specific benchmark set. We then estimate the WCET of each program using the three configurations and execute them on the XMC microcontroller.

The resulting overestimation factors are illustrated in Figure 5. Our evaluation shows that these factors range from 7.05 to 115.92, indicating a significant overestimation. We observe that increased control-flow complexity correlates with greater overestimation, particularly in loops, when loop bodies become more complex. This phenomenon illustrates the pessimism inherent to the timing schema, with double-nested loops (*#40* & *#41*) showing the most pronounced pessimism.

Implementing our load-time interpreter-model specialization results in a notable reduction in the overestimation factor, now ranging from 3.45 to 101.61, reflecting a reduction of up to 62%. The most significant decreases are observed in instances with `block` instructions, particularly when there is a considerable distance between the block's end and the preceding branching instruction. No improvements are noted in the four cases with a single loop and a branch, as the jump targets the loop head, which precedes the branch instruction. Consequently, the specialization does not provide additional information. As expected, no overestimation worsens.

In the third configuration, which incorporates software cache modeling, we observe additional enhancements for all loop test cases by 18% to 45% relative to the load-time interpreter-model specialization approach, resulting in final overestimation factors ranging from 3.45 to 55.49, including the double-nested loops. Only improvements are recorded.

In conclusion, our interpreter model demonstrates effective analysis of key control flows without underestimation. Incorporating load-time interpreter-model specialization and cache awareness significantly reduces overestimation. Thereby, we enable WCET estimation for interpreted Wasm for the selected cases, confirming that we resolved Challenge 1.

B. Self-Contained Analysis Benchmark Evaluation

Despite its limited representation of industrial control scenarios, we compile the TACLeBench [44] suite (commit `baefd262`) to Wasm to evaluate our analysis with openly available, less artificial programs. We exclude the *parallel* class because Wasm 1.0 does not support such execution. Moreover, we opted not to implement function calls, as recursion requires annotating additional flow constraints that are not in the scope of this work. Thus, we perform function inlining for the remaining benchmarks.

Table II presents the results, ordered by the Wasm binaries' size. We exclude fourteen of the fifty-seven benchmark cases, as we are unable to execute them *and* cannot analyze them. Specifically, nine cases involve unimplemented call opcodes, and one is not compilable to Wasm. The remaining four lack sufficient memory, as discussed further below. Notably, all programs that lack analysis results—except for statemate—are not executable on the microcontroller due to memory constraints, rendering the missing analysis results unproblematic.

The table lists (1) the WOETs (*Msr*), (2) WCET estimations derived from our analysis (*Est*), executed on the Linux device (*L*) and on the XMC board (*XMC*, discrepancies indicated by a slash), and (3) the overestimation ratio (*O'est*). We categorize our results by the three interpreter models: static, load-time specialization, and load-time specialization with software cache modeling. For subsequent models, we denote the percentage difference in the overestimation factor in parentheses (*Diff*). Errors that prevent the determination of results are noted in the table.

Once more, the range of overestimation factors varies significantly, spanning from 2.43 to 13451.28, in the static interpreter model. The load-time interpreter-model specialization already narrows this range to between 2.29 and 13096.37, while the cache awareness further narrows it to between 1.89 and 4446.41. Although some overestimation factors remain considerably high, our specializations on the interpreter model drastically improve the results. Specifically, load-time specialization demonstrates improvements of 98% (*statemate*) and 82% (*cosf*), while the cache modeling reduces the same overestimations by 79% and 82% once more. In a few cases, none of the models show any improvement; however, these are the test cases with the lower overestimation factors. Consistent

Benchmark		Msr	Static		Interpreter Model Specialization		Specialization & Software Cache	
#		(ms)	Est L/XMC (ms)	O'est	Est L/XMC (ms)	O'est (Diff)	Est L/XMC (ms)	O'est (Diff)
1	cover	0.03	0.24	7.89	0.24	7.89	0.24	7.89
3	deg2rad	4.57	83.66	18.32	83.66	18.32	59.42	13.01 (-29%)
4	rad2deg	4.56	83.43	18.32	83.43	18.32	59.25	13.01 (-29%)
5	binarysearch	0.16	6.18	37.54	4.39	26.69 (-29%)	2.98	18.12 (-32%)
6	fac	0.50	55.22	110.13	42.23	84.22 (-24%)	28.36	56.56 (-33%)
7	prime	0.56	75.27	134.23	26.25	46.82 (-65%)	15.59	27.80 (-41%)
8	duff	0.07	0.40	6.19	0.40	6.19	0.40	6.19
9	isqrt	872.22	14016.53	16.07	14016.53	16.07	11859.31	13.60 (-15%)
10	bsort	136.81	1840330.62	13451.28	1791773.85	13096.37 (-3%)	608333.51	4446.41 (-66%)
12	insertsort	1.73	52.21	30.16	45.74	26.42 (-12%)	35.70	20.63 (-22%)
13	matrix1	12.17	141.97	11.67	141.97	11.67	135.30	11.12 (-5%)
14	lms	41.70	436.28	10.46	436.28	10.46	422.81	10.14 (-3%)
15	bitcount	14.13	12586.22	890.56	1943.85	137.54 (-85%)	997.66	70.59 (-49%)
16	jfdctint	2.93	26.35	8.99	26.35	8.99	25.41	8.66 (-4%)
17	st	61.61	2539.81	41.23	2473.62	40.15 (-3%)	2098.74	34.07 (-15%)
18	iir	0.20	1.71	8.40	1.71	8.40	1.71	8.40
19	cosf	23.24	16428.54	706.90	2913.57	125.37 (-82%)	510.62	21.97 (-82%)
20	filterbank	oob	309483.95 / oob		131861.23 / oob		105234.97 / oob	
21	countnegative	7.39	342.87	46.38	302.70	40.95 (-12%)	144.52 / oob	19.55 (-52%)
22	complex_updates	0.62	4.64	7.52	4.64	7.52	4.64	7.52
23	ludcmp	oob	1319.97 / oob		181.21 / oob		93.74 / oob	
24	minver	3.58	368.88	103.06	67.10	18.75 (-82%)	38.18	10.67 (-43%)
25	cjpeg_wrbmp	102.85	1570.04	15.27	1544.66	15.02 (-2%)	1441.48	14.02 (-7%)
26	huff_dec	oob	325176.70 / oob		13064.75 / oob		6528.95 / oob	
27	petrinet	2.83	246.10	87.00	212.31	75.06 (-14%)	136.77 / oob	48.35 (-36%)
28	adpcm_dec	1.21	34.74	28.74	32.02	26.49 (-8%)	25.59	21.17 (-20%)
29	statemate	230.57	445606.07 / oom	1932.62	10188.47 / oom	44.19 (-98%)	2155.35 / oom	9.35 (-79%)
30	lift	482.14	1172.72	2.43	1103.29	2.29 (-6%)	910.54	1.89 (-17%)
31	adpcm_enc	4.09	277.96	67.93	156.06	38.14 (-44%)	95.99	23.46 (-38%)
32	fir2dim	2.11	16.81	7.98	16.78	7.97 (-0%)	16.58	7.87 (-1%)
33	md5	oob	772348.80		338255.84		224226.80	
34	g723_enc	oob	136818.26 / oob		30056.17 / oob		10092.64 / oob	
37	h264_dec	oob	384.20 / oob		291.01 / oob		225.19 / oob	
38	fmref	oob	1690315.58		75394.73		23240.08	
39	dijkstra	oob	1098.07		1033.13		0.02	
40	cjpeg_transupp	1781.77	280482.42	157.42	90186.28	50.62 (-68%)	59264.40	33.26 (-34%)
41	fft	oob	1912012.34		1908161.73		1727626.40	
46	pm	.data	2856519.58		587310.34		448113.21	
47	audiobeam	.data	9564.99		5540.67		4484.29	
49	sha	.data	79525.58		78574.70		66890.72	
50	susan	oob	66557782.11 / oob		4879382.91 / oob		2164560.08 / oob	
53	epic	oob	191855258.74		11966404.99		8343216.65	
54	cubic	oom	4353241.46 / oom		328968.82 / oom		102854.63 / oom	

Table II: WASM-WCET's overestimations range from 1.89 to 13451.28, which stem from both the pessimism in the interpreter model and the timing schema. The evaluation demonstrates WASM-WCET's feasibility of self-contained on-device analysis.

with our previous experiments, we see no cases where the improved models worsen overestimation.

Besides demonstrating feasibility, our evaluation allows us to identify the primary source of the significant overestimation as the inherent pessimism of the timing schema, which *multiplies* with the pessimism of both the interpreter and the hardware models. This issue is particularly pronounced in the bubblesort (*bsort*) algorithm, where multiple factors converge. The Wasm compiler introduces a triple-nested loop, which poses considerable challenges to the timing schema. Early conditional branches within the innermost loop, which target the outermost loop, result in high overestimations. Consequently, compiler toolchains must avoid such patterns. Moreover, our cache model assumes that caches are empty during the first iteration per loop, which contributes to overestimation. As it still demonstrates notable improvements in this scenario, sophisticated cache models may address overestimations more effectively. Moreover, it confirms that loops primarily con-

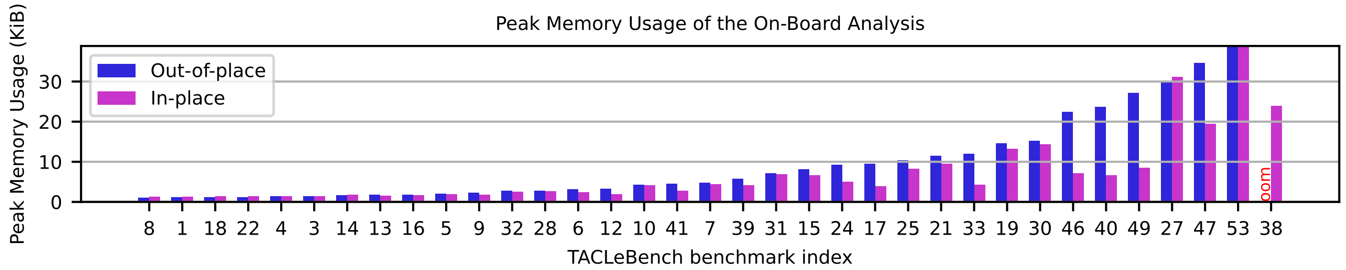


Figure 6: The in-place analysis generally requires less RAM (thus enabling the analysis of *fmref*). In few instances, the creation of temporary copies causes slightly increased memory usage compared to out-of-place. All analysis times remain $\leq$ 26 ms.

tribute to overestimation. Revisiting our prior per-instruction evaluation supports this assumption, as non-control-flow examples exhibit notably lower levels of overestimation.

In conclusion, WASM-WCET achieves self-contained on-device analysis for all but one relevant program, successfully addressing Challenge 2. Moreover, the results are consistent between on-device and offline analyses and allow pinpointing the main sources of overestimation. Load-time interpreter-model specialization and cache awareness significantly reduce high overestimation factors.

C. In-Place Resource Efficiency Comparison

Finally, we assess the resource efficiency of conducting analysis directly on the device. As previously stated, we exclude fourteen TACLeBench cases, where four of them lack sufficient memory for both analysis and execution. As detailed in Table II, eight additional cases are not analyzable on the target device due to memory constraints. Six cases (shown in the table) experience stack overflows during analysis (*oob*) due to insufficient memory, while two (shown) encounter memory allocation failures (*oom*). The remaining four cases (excluded) run out of memory as the static input data specified in the tests exceeds the data segment of the Wasm binary during parsing.

To gain more insights, we monitor peak RAM utilization during the analysis, using Zephyr's memory tracing. We compare two variants of our implementation: one that follows the steps outlined in Section IV-B sequentially, which we call *out-of-place*, and the other that uses the in-place variant presented in Section IV-C. Figure 6 presents the results.

When comparing the in-place and out-of-place variants, we find that for smaller, simpler examples, the in-place variant incurs additional memory overhead. This overhead occurs in binaries that include branches targeting the outermost block. These programs present the worst-case control-structure and require the in-place analysis to maintain the entire CFG, which is equivalent to the out-of-place approach plus temporary copies of BB references during subgraph copying from a frame to its parent. In twenty-five cases, the in-place variant shows improvements over out-of-place, ranging from 2% to 72%. In the specific case of *fmref* (#38), the analysis is only feasible with the in-place variant. The duration to conduct the analysis of a Wasm program on the device ranges from 0.2 to 26.0 ms, which aligns with practical analysis times: data transmission typically requires three-digit millisecond durations, while cycle times reside in the single-digit millisecond range or slightly below. In conclusion, the in-place variant proves advantageous for analyzing more complex files as compared to the out-of-place variant, while slightly increasing memory consumption for some programs. These results confirm that we effectively addressed the challenges of time and memory constraints outlined in Challenge 3.

VI. DISCUSSION & RELATED WORK

Trust in User Annotations & Runtime Checks: In practical applications, a limitation can arise from the need to trust loop bounds in the Wasm binary. However, operators may be enforced to design their programs to be compatible with automatic bound loop-detection, comparable to mechanisms in the Linux kernel, where eBPF rejects non-conformant programs [45]. If annotations are unavoidable, we offer pseudofunctions, which feature the integration of runtime checks. We propose that these checks are natively implemented within the runtime, which introduces a marginal overhead to Wasm's interpretation. The interpreter model considers these checks.

Analysis-Aware Compilation & Systems: WASM-WCET is designed for industrial applications where toolchains optimize code for execution on PLCs. Although OEMs cannot safely rely on their use, operators will likely use these toolchains. Consequently, directly deploying the general-purpose benchmark in Section V-A does not accurately reflect industrial reality; however, it allows pinpointing patterns that contribute to excessive overestimation and offers guidelines for these toolchains to mitigate them. For example, we recommend that compilers avoid multi-level nested loops, identified as a primary source of overestimation, e.g., by unrolling. Furthermore, opcode selection must strike a balance between performance and overestimation. For instance, while substituting multiplication by two with shift-left operations enhances execution speed, it results in higher overestimation in the specific case of WAMR (2.09 compared to 1.68).

Besides analysis-aware compilation, we advocate for interpreters tailored for real-time analysis. This includes implementing distinct handlers for 32- and 64-bit operations to ensure unambiguous interpreter paths for analysis. Additionally, a load-time pre-decoding phase of LEB128 constants, along with adjustments to runtime values and memory encoding, can

eliminate LEB128 handling entirely during execution, thereby reducing overestimation of numerous Wasm operations.

Generally, preserving guarantees entails accepting a certain degree of overestimation. In typical mixed-criticality systems [46], where not all tasks are critical and require conformance with temporal bounds, less critical tasks can utilize this slack time resulting from overestimation.

Alternative Approaches for Tackling Overestimations: Although there are currently no ILP solvers for the strongly resource-constrained devices that we target, recent work [47] investigates hardware accelerators for the Simplex algorithm, laying the foundation for developing such solvers. As our approach does not depend on the timing schema but uses it as the only feasible method, we consider it to be adaptable to IPET. Another approach to reduce overestimation is to combine our interpreter model with the method presented by Zaeske et al. [18], at the cost of assuming that programs can be suspended or deferred at any moment without negative effects. To bound execution time for Wasm, the authors refer to the concept of fuel, also used by other interpreters [48], [49], [50]. The interpreter is allocated a predetermined amount of fuel. Before executing a BB, it estimates the fuel cost of that block to continue execution. For the cost model, they correlate fuel consumption with hardware timings by measuring programs on reference hardware and aligning the results with those on the actual hardware. When integrated with our interpreter model, more precise timings for individual BBs can be provided, reducing overestimation, as neither loops nor conditionals are present during analysis.

Improved Cache Modeling: The proposed cache model is intentionally simplistic, aiming to motivate the investigation of the runtime's state. We aligned the cache size with our benchmarks to ensure the first-miss approach is sufficient, but improved cache modeling may further reduce overestimations. FIFO analysis is generally more complex than other caching strategies, as hits do not alter the replacement order, requiring deeper analysis of hit and miss events and their interplay. Grund and Reineke [51] track upper bounds on hits and misses during abstract interpretation, enabling bidirectional updates, which could be integrated into our timing-schema-based approach by selecting the worst-case bounds along with the worst-case time. Nonetheless, notable overestimation will persist, as [42] assert. They enhance FIFO analysis by quantifying a miss-ratio rather than relying on binary classification, thereby facilitating integration into IPET-based WCET analysis. Integration into our approach is difficult as this implicit method benefits from large control-flow contexts, while the timing schema discards control flow. Alternatively, OEMs may consider adjusting the cache policy.

Analysis of Interpreted Languages: Recent work by Shortt et al. [52] claims to be the first to bound the worst-case cost of Wasm functions. They also base their algorithm on the timing schema and reduce the cyclic CFG to a DAG. Moreover, they provide an implementation for determining loop bounds via symbolic execution during binary parsing, in line with our concept. However, the authors focus solely on the path component of WCET calculation and avoid creating an interpreter model by employing a uniform execution cost of one time unit per instruction. Their overestimations are lower compared to ours; however, our work demonstrates that factors multiply when combining the timing schema with the pessimism inherent to hardware and interpreter models. Significant research on WCET analysis for Java [28], [29], [30] has already been conducted more than twenty years ago. Given that both Wasm and Java are portable bytecodes, the concept of distinguishing between high- and low-level analyses is analogous, as is approximating non-constant instruction costs using linear functions. The three-stage approach of [28] declares the combination of both analyses as the third stage, whereas we emphasize the (unaddressed) contribution of the runtime as the third step. In fact, complying with previously introduced nomenclature, we now present a four-step approach. The contribution of our work is the quantification of the interpreter cost model and applying it to Wasm. Specifically, we address the "overhead associated with translation" (i.e., Δ_i) in [29], which is essential for reliable WCETs. Advancing this line of thought, we demonstrate with WASM-WCET that for Wasm, quantifying this overhead is nontrivial: The overhead is fundamentally contingent upon bounding the loops in the interpreter. Additionally, in our scenario, we avoid trusting pre-calculated (abstract) WCETs—or necessitate their reevaluation—requiring on-device estimation instead of using worst-case execution-frequency vectors.

VII. CONCLUSION

WASM-WCET introduces a novel approach, enabling long-term updatability for safety-critical, resource-constrained devices. By integrating an on-device timing protection mechanism for Wasm modules, we effectively mitigate spatial and temporal interference with the underlying system, while providing portability and reducing dependency on the OEM for updates. Complementing traditional two-stage WCET analysis, we incorporate an additional third stage attributing the interpreter's contribution, and develop the first interpreter cost model for Wasm bytecode, enabling WCET analysis.

We address the unbounded loops within the interpreter's path component and reduce overestimation with our load-time interpreter-model specialization as well as our integration of a software cache model. By implementing a self-contained timing-schema-based method, we tackle the path component of WCET analysis for Wasm. Leveraging Wasm's structured control flow, we improve memory efficiency through our in-place variant, facilitating analysis *on* resource-constrained devices. Our evaluations demonstrate the feasibility, showcasing improvements of up to 98% by the load-time interpreter-model specialization, and up to 82% from our cache model. The analyses conducted on a 128 KiB microcontroller with times of under 26 ms confirm WASM-WCET's practical applicability.

Source code of WASM-WCET:
https://gitos.rrze.fau.de/i4/openaccess/wasm-wcet

References

[1] P. Sandborn, V. Prabhakar, and O. Ahmad, "Forecasting electronic part procurement lifetimes to enable the management of DMSMS obsolescence," *Microelectronics Reliability*, vol. 51, no. 2, pp. 392–399, 2011. DOI: 10.1016/j.microrel.2010.08.005.

[2] T. Ault and T. Bradley, "Risk-based approach for managing obsolescence for automation systems in heavy industries," *Systems Engineering*, vol. 25, no. 6, pp. 551–560, 2022. DOI: 10.1002/sys.21635.

[3] ARC Advisory Group, *Whitepaper: Siemens Process Automation System Migration and Modernization Strategies*, https://assets.new.siemens.com/siemens/assets/api/uuid:0dd88880-2675-4f58-abf5-ad8082c58fd8/migration-whitepaper-en.pdf, 2007.

[4] E. Gregolinska, R. Khanam, F. Lefort, and P. Parthasarathy, *Capturing the true value of Industry 4.0*, https://www.mckinsey.com/capabilities/operations/our-insights/capturing-the-true-value-of-industry-four-point-zero, 2022.

[5] J. Koochaki, J. Bokhorst, H. Wortmann, and W. Klingenberg, "Evaluating condition based maintenance effectiveness for two processes in series," *Journal of Quality in Maintenance Engineering*, vol. 17, no. 4, pp. 398–414, 2011. DOI: 10.1108/13552511111180195.

[6] J. Lee, J. Ni, D. Djurdjanovic, H. Qiu, and H. Liao, "Intelligent prognostics tools and e-maintenance," *Computers in Industry*, vol. 57, no. 6, pp. 476–489, 2006. DOI: 10.1016/j.compind.2006.02.014.

[7] European Parliament, *Regulation (EC) No 178/2002 of the European Parliament and of the Council of 28 January 2002 laying down the general principles and requirements of food law, establishing the European Food Safety Authority and laying down procedures in matters of food safety*, http://data.europa.eu/eli/reg/2002/178/2024-07-01/eng, 2024.

[8] European Commission, *Commission Directive (EU) 2019/1831 of 24 October 2019 establishing a fifth list of indicative occupational exposure limit values pursuant to Council Directive 98/24/EC and amending Commission Directive 2000/39/EC*, http://data.europa.eu/eli/dir/2019/1831/2019-10-31/eng, 2019.

[9] M. Zhang and J. Rantanen, "Improving the law on the prevention and control of occupational diseases in China: An employer-supporting management perspective," *Global Health Journal*, vol. 4, no. 2, pp. 33–41, 2020. DOI: 10.1016/j.glohj.2020.04.004.

[10] E. R. Alphonsus and M. O. Abdullah, "A review on the applications of programmable logic controllers (PLCs)," *Renewable and Sustainable Energy Reviews*, vol. 60, pp. 1185–1205, 2016. DOI: 10.1016/j.rser.2016.01.025.

[11] M. A. Sehr, M. Lohstroh, M. Weber, I. Ugalde, M. Witte, J. Neidig, S. Hoeme, M. Niknami, and E. A. Lee, "Programmable Logic Controllers in the Context of Industry 4.0," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 5, pp. 3523–3533, 2021. DOI: 10.1109/TII.2020.3007764.

[12] R. Liu, L. Garcia, and M. Srivastava, "Aerogel: Lightweight Access Control Framework for WebAssembly-Based Bare-Metal IoT Devices," in *Proceedings of the 6th IEEE/ACM Symposium on Edge Computing (SEC)*, 2021, pp. 94–105. DOI: 10.1145/3453142.3491282.

[13] E. Wen and G. Weber, "Wasmachine: Bring IoT up to Speed with A WebAssembly OS," in *Proceedings of the 18th IEEE International Conference on Pervasive Computing and Communications Workshops and Other Affiliated Events (PerCom Workshops)*, 2020, pp. 1–4. DOI: 10.1109/PerComWorkshops48775.2020.9156135.

[14] G. Peach, R. Pan, Z. Wu, G. Parmer, C. Haster, and L. Cherkasova, "eWASM: Practical Software Fault Isolation for Reliable Embedded Devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3492–3505, 2020. DOI: 10.1109/TCAD.2020.3012647.

[15] B. Li, H. Fan, Y. Gao, and W. Dong, "ThingSpire OS: A WebAssembly-based IoT operating system for cloud-edge integration," in *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2021, pp. 487–488. DOI: 10.1145/3458864.3466910.

[16] J. Ménétrey, M. Pasin, P. Felber, and V. Schiavoni, "Twine: An Embedded Trusted Runtime for WebAssembly," in *Proceedings of the 37th IEEE International Conference on Data Engineering (ICDE)*, 2021, pp. 205–216. DOI: 10.1109/ICDE51399.2021.00025.

[17] B. Li, W. Dong, and Y. Gao, "WiProg: A WebAssembly-based approach to integrated IoT programming," in *Proceedings of the 40th IEEE Conference on Computer Communications (INFOCOM)*, 2021, pp. 1–10. DOI: 10.1109/INFOCOM42981.2021.9488424.

[18] W. Zaeske, S. Friedrich, T. Schubert, and U. Durak, "WebAssembly in Avionics: Decoupling Software from Hardware," in *Proceedings of the 42nd IEEE/AIAA Digital Avionics Systems Conference (DASC)*, 2023, pp. 1–10. DOI: 10.1109/DASC58513.2023.10311207.

[19] C. Petig, *Component Model in Software Defined Vehicles*, WASMCON 2024, Salt Lake City, USA, 2024.

[20] J. Rau and T. Schoppe, *WebAssembly on the Factory Floor: Efficient and Secure Processing of High Velocity Machine Data*, Cloud Native Wasm Day, Paris, France, 2024.

[21] E. Ruppel, *Keynote: When Wasm Meets Cyber-Physical Systems: A Discussion of WebAssembly in Real-Time, Safety-Critical Systems*, WASMCON, Bellevue, Washington, 2023.

[22] C. Woods, *Embedding WASM in the Future*, WASMCON, Bellevue, Washington, 2023.

[23] S. Wallentowitz, B. Kersting, and D. M. Dumitriu, "Potential of WebAssembly for Embedded Systems,"

in *Proceedings of the 11th Mediterranean Conference on Embedded Computing (MECO)*, 2022, pp. 1–4. DOI: 10.1109/MECO55406.2022.9797106.

[24] C. Watt, "Mechanising and verifying the WebAssembly specification," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, 2018, pp. 53–65. DOI: 10.1145/3167082.

[25] C. Watt, M. Trela, P. Lammich, and F. Märkl, "WasmRef-Isabelle: A Verified Monadic Interpreter and Industrial Fuzzing Oracle for WebAssembly," in *Proceedings of the 44th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2023, 110:100–110:123. DOI: 10.1145/3591224.

[26] A. Ramesh, T. Huang, E. Ruppel, D. Dasari, B. Pourmohseni, F. Smirnov, M. Giani, P. Pazzaglia, C. Shelton, N. Pereira, A. Hamann, D. Ziegenbein, and A. Rowe, "Silverline: Lightweight Virtualization and Orchestration of Distributed Systems," in *Proceedings of the 31st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2025, pp. 375–388. DOI: 10.1109/RTAS65571.2025.00042.

[27] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem – overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (ACM TECS)*, vol. 7, no. 3, pp. 1–53, 2008. DOI: 10.1145/1347375.1347389.

[28] I. Bate, G. Bernat, and P. Puschner, "Java virtual-machine support for portable worst-case execution-time analysis," in *Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. (ISORC)*, 2002, pp. 83–90. DOI: 10.1109/ISORC.2002.1003664.

[29] I. Bate, G. Bernat, G. Murphy, and P. Puschner, "Low-level analysis of a portable Java byte code WCET analysis framework," in *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications (RTCSA)*, 2000, pp. 39–46. DOI: 10.1109/RTCSA.2000.896369.

[30] G. Bernat, A. Burns, and A. Wellings, "Portable worst-case execution time analysis using Java Byte Code," in *Proceedings of the 12th Euromicro Conference on Real-Time Systems. (EMRTS)*, 2000, pp. 81–88. DOI: 10.1109/EMRTS.2000.853995.

[31] A. Colin and I. Puaut, "A modular and retargetable framework for tree-based WCET analysis," in *Proceedings 13th Euromicro Conference on Real-Time Systems (ECRTS '01)*, 2001, pp. 37–44. DOI: 10.1109/EMRTS.2001.933995.

[32] WebAssembly Working Group, *WebAssembly Specification 1.0*, https://www.w3.org/TR/wasm-core-1/, 2019.

[33] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and JF Bastien, "Bringing the web up to speed with WebAssembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017, pp. 185–200. DOI: 10.1145/3062341.3062363.

[34] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," in *ACM SIGPLAN Notices*, ACM, vol. 30, 1995, pp. 88–98. DOI: 10.1145/216633.216666.

[35] P. Puschner and A. Schedl, "Computing maximum task execution times: A graph-based approach," *Real-Time Systems*, vol. 13, pp. 67–91, 1997. DOI: 10.1023/A:1007905003094.

[36] AbsInt. "aiT WCET analyzers. "[Online]. Available: https://www.absint.com/ait/.

[37] D. Kästner, M. Pister, S. Wegener, and C. Ferdinand, "TimeWeaver: A tool for hybrid worst-case execution time analysis," in *Proceedings of the 19th International Workshop on Worst-Case Execution Time Analysis (WCET '19)*, 2019, 1:1–1:11. DOI: 10.4230/OASIcs.WCET.2019.1.

[38] W3C WebAssembly Community Group, *WebAssembly Proposal: Multi-Value*, WebAssembly, 2020. Accessed: Oct. 4, 2025. [Online]. Available: https://github.com/WebAssembly/multi-value.

[39] W3C WASI Embedded Special Interest Group, *SIG-Embedded Meeting Notes 2025-08-05*, 2025. Accessed: Oct. 22, 2025. [Online]. Available: https://github.com/bytecodealliance/meetings/blob/main/SIG-Embedded/2025/08-05-Meeting-notes.md.

[40] *Misra-c:2012 – guidelines for the use of the c language in critical systems*, MISRA Consortium, 2013.

[41] Siemens AG, *SIMATIC DP, CPU 1510SP-1 PN for ET 200SP*. [Online]. Available: https://sieportal.siemens.com/su/bmhmN.

[42] N. Guan, X. Yang, M. Lv, and W. Yi, "FIFO cache analysis for WCET estimation: A quantitative approach," in *Proceedings of the 13th Design, Automation & Test in Europe (DATE)*, 2013, pp. 296–301. DOI: 10.7873/DATE.2013.073.

[43] P. Wägemann, T. Distler, C. Eichler, and W. Schroder-Preikschat, "Benchmark Generation for Timing Analysis," in *Proceedings of the 23th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017, pp. 319–330. DOI: 10.1109/RTAS.2017.6.

[44] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener, "TACLeBench: A benchmark collection to support worst-case execution time research," in *Proceedings of the 16th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2016, 2:1–2:10. DOI: 10.4230/OASIcs.WCET.2016.2.

[45] Linux Kernel documentation. “Ebpf verifier.” 2025. [Online]. Available: https://docs.kernel.org/bpf/verifier.html.

[46] A. Burns and R. I. Davis, “A survey of research into mixed criticality systems,” *ACM Computing Surveys (CSUR)*, vol. 50, no. 6, 82:1–82:37, 2017, ISSN: 0360-0300.

[47] M. Bednara, K. Braun, M. Kuchlbauer, and C. Sigwarth, “Accelerating linear programming performance: A hardware-software co-design approach for the simplex method,” in *Proceedings of the 15th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART ’25)*, 2025, pp. 156–166. DOI: 10.1145/3728179.3728180.

[48] Wasmer.io, *Wasmer*, 2025. Accessed: Nov. 7, 2025. [Online]. Available: https://github.com/wasmerio/wasmer.

[49] Wasmi Labs, *Wasmi*, 2025. Accessed: Nov. 7, 2025. [Online]. Available: https://github.com/wasmi-labs/wasmi.

[50] Bytecode Alliance, *Wasmtime*, 2025. Accessed: Nov. 7, 2025. [Online]. Available: https://github.com/bytecodealliance/wasmtime.

[51] D. Grund and J. Reineke, “Abstract Interpretation of FIFO Replacement,” in *Proceedings of the 15th International Static Analysis Symposium (SAS)*, 2009, pp. 120–136. DOI: 10.1007/978-3-642-03237-0_10.

[52] J. Shortt, A. Felty, and A. Somayaji, *Bounding the Execution Cost of WebAssembly Functions*, Preprints of VSTTE 2025, 2025.