WasmBounds: Eliminating Memory Bounds Checks in WebAssembly

Emma Sudo
emmasudo@cs.stanford.edu

Keith Winstein
keithw@cs.stanford.edu

Abstract

This paper presents WasmBounds, a tool that elides redundant memory bounds checks in WebAssembly (Wasm) using abstract interpretation. Although runtimes for Wasm 1.0 on systems with virtual memory have been able to avoid bounds checks by using hardware guard pages, the presence of newer Wasm features (64-bit memories and single-byte page sizes) or an embedded environment generally requires software bounds checks for memory accesses, at the cost of runtime efficiency. WasmBounds shifts this runtime cost to a one-time static analysis phase. The key insight behind WasmBounds is that safe languages such as Rust and Zig already enforce bounds checks on accesses to container types. Although these safety guarantees are not immediately apparent in the compiled Wasm binary, WasmBounds can recover this information with abstract interpretation. By reasoning over an integer interval domain, WasmBounds infers the range of each memory access address and determines whether accesses are provably in bounds. Eliminating these redundant checks yields a roughly 1.2x speedup in our preliminary microbenchmark.

1 Introduction and Motivation

WebAssembly (Wasm) is a binary instruction format that allows applications written in various programming languages to be deployed in a diversity of environments. Wasm is most well known for allowing code to run safely and efficiently in the browser, but it also has applications in serverless and embedded systems.

Wasm's memory safety guarantee ensures that accesses to unallocated memory produce a trap. Unfortunately, Wasm's memory safety comes at a cost. The naive approach to maintaining memory safety is by injecting dynamic bounds checks before every memory access. Besides increasing the instruction count, these runtime checks are costly because they severely hinder the compiler's ability to optimize the program. Nevertheless, Wasm is considered to be efficient because for programs that use a 32-bit address space, the 64-bit host system can reserve 8 GiB of virtual address space (the maximum reachable memory) and mark all pages associated with invalid portions of the memory as inaccessible. This hardware bounds check method incurs almost no runtime overhead.

Hardware bounds checking with virtual memory is not always straightforward because there are cases where the runtime cannot assume a 32-bit address space, a 64-bit host, or 8 GiB of available virtual memory. With the new Memory64 Proposal, Wasm programs can now use a 64-bit indexed address space, which makes hardware bounds checking with a virtual address space more difficult. In addition, the Custom Page Sizes Proposal allows memories to be unaligned with hardware pages, which would prevent hardware bounds checks from being able to mark unallocated Wasm memory as inaccessible. Finally, many embedded environments do not have virtual memory, hindering strategies that uses guard pages. Embedded systems also face resource constraints that make the Custom Page Sizes Proposal necessary. In these situations, Wasm runtimes usually resort to software bounds checking every memory access.

Our tool, WasmBounds, is based on the observation that safe languages already insert runtime bounds checks. For example, Rust and Zig both insert bounds checks before accesses to container types. Duplicating these checks at the Wasm binary level significantly decreases the performance of the program because the checks hinder compiler optimizations. With abstract interpretation, WasmBounds can determine the range of addresses that each load and store can access. Thus, WasmBounds shifts the runtime cost of dynamic bounds checks to a one-time static analysis phase. In our preliminary microbenchmark, we observed a roughly 1.2x speedup after eliminating redundant checks identified by WasmBounds.

This paper proceeds as follows: we first introduce abstract interpretation and describe the design and implementation of WasmBounds. Then, we discuss the preliminary evaluation of WasmBounds and work related to WasmBounds.

2 Abstract Interpretation

Abstract interpretation is a form of static analysis that over-approximates program state after the execution of each instruction [4]. Approximation helps us to avoid exhaustively searching through the entire space of the program [10]. Instead of operating on concrete Wasm values, we modify the semantics instructions to operate over an abstract domain.

2.1 Integer Interval Abstract Domain

In our case, we specifically care about the `i32` values that load and store instructions consume to access memory at that offset. The most appropriate abstract domain to track these offsets is the integer interval domain. More specifically, WasmBounds operates over sets of integer intervals. Sets of intervals made the most sense so that WasmBounds can reason about how conditional branching separates the values that variables can represent into disjoint subsets.

3 Design

At a high level, WasmBounds is an analyzer that takes a Wasm binary as an input and outputs a non-exhaustive list of provably safe memory instruction offsets. This list can be passed to any Wasm runtime to indicate which instructions should not be bounds checked.

3.1 Assumptions

We assume that the input Wasm binary is valid. We also assume that the memory is a fixed size and that the runtime will reject memory grow requests (which is compliant with the Wasm specification).

3.2 Introducing an Example Program

We will introduce the main ideas behind WasmBounds by walking through an example program. Consider the Rust program in Listing 1, which contains a function that calculates the dot product of two static `i32` arrays.

```
1  const N: i32 = 1024;
2
3  static first: [i32; N] = [0; N];
4  static second: [i32; N] = [0; N];
5
6  fn dotproduct() -> i32 {
7    let mut ret = 0;
8    for i in 0..N {
9      ret += first[i] * second[i];
10   }
11   ret
12 }
```

Listing 1: This is a safe dot product program in Rust.

Rust will insert bounds checks at each array access. When the index into the array is greater than or equal to the length of the array, execution will jump to a panic routine. When this Rust program is compiled to Wasm, the output is large and difficult to read. So, we introduce an analogous program written C that mimics the safe behavior of the Rust program. Consider the C program in Listing 2, which contains a function that calculates the dot product of two globally defined int arrays:

```
1  const int N = 1024;
2
3  int first[N];
4  int second[N];
5
6  int get_first(int n) {
7      if (n < N) {
8          return first[n];
9      } else {
10         return 0;
11     }
12 }
13
14 int get_second(int n) {
15     if (n < N) {
16         return second[n];
17     } else {
18         return 0;
19     }
20 }
21
22 int dotproduct(void) {
23   int ret = 0;
24   for (unsigned int i = 0; i < N; ++i) {
25     ret += get_first(i) * get_second(i);
26   }
27   return ret;
28 }
```

Listing 2: This is a safe dot product program in C that is analogous to the Rust program in Listing 1.

We know that the array accesses are always in-bounds because of the safe helpers `get_first` and `get_second`. The fact that the memory accesses are safe is less obvious when we compile the C program to Wasm (Figure 1).

Note that Clang statically determined the number of iterations and the address of both vectors. In addition, Clang inlined the helper functions, optimizing away the checks. We will demonstrate how WasmBounds recovers the safety of the loads on lines 13 and 17, and we will describe how this same approach can identify the safety of the loads in the Rust program in Listing 1 compiled to Wasm.

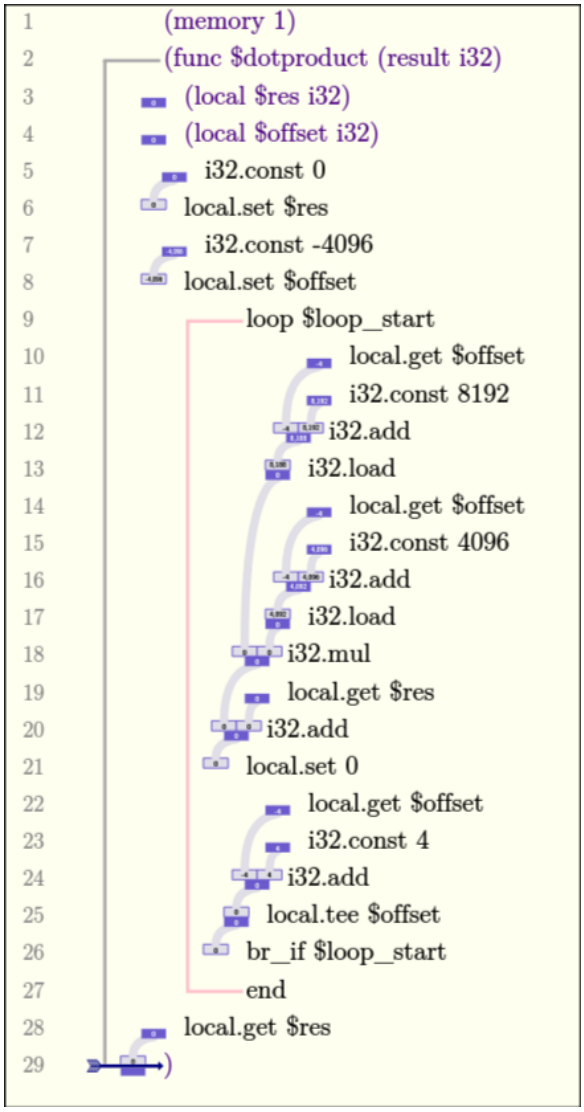

Figure 1: This is a safe dot product program in C compiled to Wasm in the Codillon editor [1]. When a safe program is compiled to a Wasm binary, it is not obvious that each access is in-bounds.

3.3 How does WasmBounds prove that these accesses are safe?

WasmBounds uses its abstract interpreter to identify the range of values that the `i32` popped by the `i32.load` on lines 13 and 17. It does so by keeping track of the value of each variable at each program point as a set of intervals. As we walk through the Wasm in Figure 1, we will be using S_i to denote the set of intervals that the local variable `$offset` can represent between lines i and $i+1$.

During the first iteration, the local variable `$offset` has the value -4096. As a result, the sets of intervals that `$offset` can represent after lines 10 and 14 (when `$offset` gets pushed onto the stack to be used by load instructions), S_{10} and S_{14}, are both $\{[-4096,-4096]\}$. `$offset` gets incremented by 4 on line 25, so $S_{25} = \{[-4092,-4092]\}$. At the `br_if`, the only interval in S_{26} that will flow to program points after the loop is $[0,0]$. All other intervals in S_{26} will lead execution back to the `loop_start` label. So, we split S_{26} into a set of intervals that continue down the program, S^C_{26}, and a set of intervals that go back to the top of the loop, S^L_{26}. At this point $S^C_{26} = \{\}$ and $S^L_{26} = \{[-4092,-4092]\}$.

During the second iteration, we join any intervals from the previous iteration S'_i with the current values produced by the interpreter. S_{10} and S_{14} are now $\{[-4096,-4096],[-4092,-4092]\}$ because $S^{L\,\prime}_{26}$ (the set of intervals at the end of the previous loop iteration) should join with the current value of `$offset` which is still -4096. This is how WasmBounds expresses the fact that after at most 2 iterations of the loop, `$offset` can be -4096 or -4092. The same procedure as the first iteration is followed, resulting in $S_{25} = \{[-4092,-4092],[-4088,-4088]\}$, $S^L_{26} = \{[-4092,-4092],[-4088,-4088]\}$, and $S^C_{26} = \{\}$.

After 1024 iterations, we see that S_{10} and S_{14} are $\{[-4096,-4096],...,[-4,-4]\}$ and $S_{25} = \{[-4092,-4092],...,[0,0]\}$. Since S_{26} now contains $[0,0]$, we can finally add an interval to S^C_{26}. $S^L_{26} = \{[-4092,-4092],...,[-4,-4]\}$ and $S^C_{26} = \{[0,0]\}$.

After 1025 iterations, $S'_i = S_i$ for all program points. This is because only $S^{L\,\prime}_{26}$ joins with the current value after the loop start, so no new intervals get added to S_9. Thus, we have reached a fixed point and we know that these are the final intervals that represent `$offset` at each program point.

Now, we just need to use the WasmBounds interpreter to analyze the set of intervals popped off the stack by each `i32.load`. Because S_{10} and S_{14} end up being $\{[-4096,-4096],...,[-4,-4]\}$, the set of popped intervals is $\{[0,0],...,[8188,8188]\}$. Since the memory is 64 KiB, WasmBounds knows that all of these offsets are in-bounds. The instruction offset of the load on line 13 and 17 will get added to a list of safe instructions.

3.4 Applying WasmBounds to More Programs

The Wasm binary analyzed in Section 3.3 differed from the binary that would have been produced from the analogous Rust program in Listing 1. The Rust-produced binary contains if-else blocks and calls to helper functions.

Extending the WasmBounds interpreter to handle if-else blocks would be similar to the logic that already exists for `br_if`.

Calls to helper functions would "clobber" the global variables and introduce unknown values onto the stack. Although these unknown values could prevent the abstract interpreter from being able to identify a narrow enough set of intervals for the load, calls to external functions do not affect the soundness of the interpreter.

This is all to say that ensuring soundness while extending WasmBounds is as simple as translating the meaning of each new instruction to apply to sets of intervals, but helping WasmBounds maintain precise intervals is nontrivial. After all, WasmBounds would still be sound if it assumed that every `i32` variable ranges from `u32::MIN` to `u32::MAX`.

4 Implementation

WasmBounds is implemented in Rust using `wasm-tools`' `wasmparser` [2]. WasmBounds takes in a Wasm binary and outputs a list of load and store instructions that it can prove are safe. We achieve this through three steps: parsing, initialization, and interpretation.

4.1 Overview

During the parsing phase, we use `wasmparser` to collect information about the memories, global variables, and functions.

During initialization, we construct a control flow graph for each function. Each node represents a program point, and it consists of a list of successor nodes and a way of representing the variable values at that point. For each node, every variable has a set of integer intervals that starts as empty and grows to contain all of the intervals that the variable can represent. The exception is the entry node, where we know that the locals are zero-initialized and globals/parameters start in an unknown state.

During interpretation, we repeatedly step through the function with our abstract interpreter which modifies the Wasm semantics to be compatible with our representation of integers as sets of intervals. At each iteration, we update the nodes with new intervals by taking the union of intervals from the current iteration with the set of intervals from previous iterations. Eventually, the graph reaches a fixed point where additional passes do not change the nodes. At this point, we know the range of values for each variable (even if it is the entire range of 32-bit integers in the worst case), so we can use our interpreter to infer the possible addresses for each load and store instruction. If these addresses are in bounds of the linear memory, we note the offset of the access. These offsets can be passed to any Wasm engine so that the associated check can be elided.

4.2 Abstract WebAssembly Semantics

We translated a subset of Wasm to operate over our abstract domain. Note that the each element of the operand stack is a pair consisting of a set of integer intervals and the expression that set represents.

I32Const(val)
1. Push $(\{[\text{val},\text{val}]\},\text{const})$ onto the stack

I32Add
1. Pop V_1 from the stack
2. Pop V_2 from the stack
3. Create a new set S. For $[l_1,u_1]$ in $V_1[0]$ and $[l_2,u_2]$ in $V_2[0]$, add $[l_1+l_2,u_1+u_2]$ to S
4. Push $(S,\text{Add}(V_1[1],V_2[1]))$ onto the stack

I32Mul
1. Pop V_1 from the stack
2. Pop V_2 from the stack
3. Create a new set P. For $[l_1,u_1]$ in $V_1[0]$ and $[l_2,u_2]$ in $V_2[0]$, add $[l_1+l_2,u_1+u_2]$ to P
4. Push $(P,\text{Mul}(V_1[1],V_2[1]))$ onto the stack

I32Load
1. Pop from the stack
2. Push $([\text{u32::MIN},\text{u32::MAX}],\text{Mem})$ onto the stack

LocalSet(idx)
1. Pop V from the stack
2. Set $S_i = V[0]$ where $i-1$ is the current line number and S is the set of intervals associated with local[idx]

LocalTee(idx)
1. Peek V from the stack
2. Set $S_i = V[0]$ where $i-1$ is the current line number and S is the set of intervals associated with local[idx]

LocalGet(idx)
1. Push $(S_i,\text{local[idx]})$ onto the stack where i is the current line number of the instruction and S is the set of intervals associated with local[idx]

BrIf(label)
1. Pop V from the stack
2. If the expression $V[1]$ does not contain local, branch to label if $V[0]$ does not contain $[0,0]$ and continue execution otherwise. If $V[1]$ contains local and $V[0]$ does not contain $[0,0]$ branch to label. Otherwise, track two sets for that local: the first set contains $[0,0]$ and continues execution, while the second set contains all other intervals in $V[0]$ and branches to label.

4.3 Optimizations

The three-step method described in Section 4.1 has some limitations.

Interpreting the program until fixed-point can be very expensive. One way to address this cost is employing a widening heuristic. We can use our interpreter to express loop conditions in terms of variables. From this expression, we can guess the upper or lower bound on the interval. If the graph does not converge in a set amount of steps, we can assume that the heuristic failed and try another guess.

WasmBounds also struggles with parameters and global variables because it reasons within singular functions. However, WasmBounds can construct conditions on global and parameter values that ensure memory accesses are provably

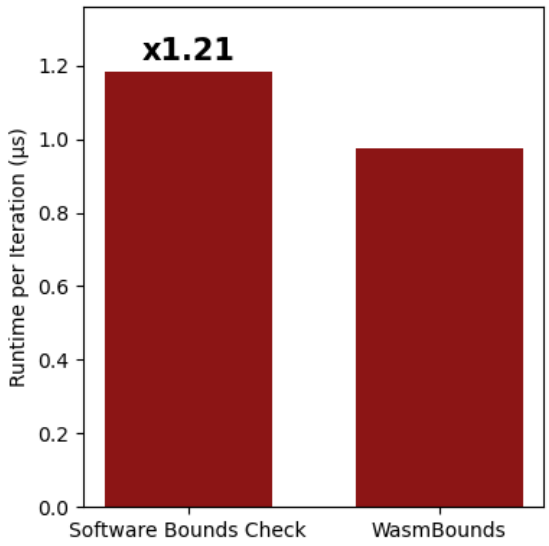

Figure 2: Average runtime per iteration of the dot product microbenchmark with bounds checks and WasmBounds (loads are proven to be safe and checks are elided). WasmBounds is faster by about 21%.

in bounds. Then, we can create a dispatch function where this condition is dynamically checked once and execution is directed to a bounds checked version of the function only if the condition is not met.

5 Evaluation

Since WasmBounds is in its early stages, we performed a preliminary microbenchmark on the program described in Figure 1.

The experiment was run on a machine with 2 AMD EPYC 7702 64-Core Processors and 160 GiB of RAM, running Ubuntu 26.04 LTS (Linux 7.0.0-15-generic).

With the subset of Wasm instructions implemented by WasmBounds so far, WasmBounds was able to identify both the loads on lines 13 and 17 as safe.

We then used `wasm2c` to transpile the Wasm binary back to C and Clang to compile the C to x86 assembly. The baseline software bounds checks both loads while WasmBounds does not. We called the dot product function 1,000,000 times to determine the runtime per iteration. The average runtime per iteration for the baseline was 1.18 μs while the average runtime per iteration for WasmBounds was 0.98 μs. This is a 1.21x speedup. These results are shown in Figure 2.

6 Related Work

WasmBounds follows work that make Wasm sandboxing more efficient and that use abstract interpreation on Wasm.

6.1 Wasm-precheck

Wasm-precheck [8] is an extension of the Wasm specification that includes indexed types to express constraints on values. Like WasmBounds, Wasm-precheck is motivated by the Memory64 Proposal. Where Wasm-precheck differs from WasmBounds, Wasm-precheck requires developers to annotate the Wasm with indexed types. These annotations mean that out-of-bounds accesses are caught during the type checking phase. This approach is more aggressive than that of WasmBounds, which takes as input any Wasm binary and will not output accesses it cannot prove are safe.

6.2 Performant Bounds Checking for 64-Bit WebAssembly

Döllerer and Engelke's "Performant Bounds Checking for 64-Bit WebAssembly" [6] attempts to reduce the cost of software bounds checks using two-level guard pages and x86-64 memory keys. This work can be combined with WasmBounds to make the bounds checks that cannot be elided more efficient.

6.3 Speculative Improvements to Verifiable Bounds Check Elimination

"Speculative Improvements to Verifiable Bounds Check Elimination" [7] is the inspiration for the optimization discussed in Section 4.3. This work describes "speculations," additional constraints that increase the number of accesses that can be proven are safe. The speculations are tested once during runtime and lead to checked or unchecked versions of the code.

6.4 Abstract Interpretation of WebAssembly

Abstract interpretation has been used to statically analyze Wasm [3] [10]. These analyzers focus on using abstract interpretation for dead code elimination, constant propagation, and taint analysis. There are also existing frameworks for abstract definitional interpretation [5] [9]. Together with these work, WasmBounds proves that abstract interpretation is still a relevant technique for achieving a diversity of Wasm static analysis goals.

7 Conclusion

Deterministic traps after out-of-bounds accesses is a core aspect of WebAssembly, but enforcing this guarantee comes at a cost. For the Memory64 and Custom Page Size Proposals, as well as for embedded platforms, the status quo is injecting costly software bounds checks at every memory access. Our tool, WasmBounds, uses abstract interpretation over sets of intervals to identify provably safe accesses so that they can be elided. Our preliminary microbenchmark shows the WasmBounds results in a roughly 1.2x speedup.

References

[1] Codillon — codillon.org. https://codillon.org/. [Accessed 11-06-2026].

[2] GitHub - bytecodealliance/wasm-tools: CLI and Rust libraries for low-level manipulation of WebAssembly modules — github.com. https://github.com/bytecodealliance/wasm-tools. [Accessed 11-06-2026].

[3] BRANDL, K., ERDWEG, S., KEIDEL, S., AND HANSEN, N. Modular abstract definitional interpreters for WebAssembly, July 2023.

[4] COUSOT, P., AND COUSOT, R. Abstract interpretation. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '77* (New York, New York, USA, 1977), ACM Press, pp. 238–252.

[5] DARAIS, D., LABICH, N., NGUYEN, P. C., AND VAN HORN, D. Abstracting definitional interpreters (functional pearl). *Proc. ACM Program. Lang. 1*, ICFP (Aug. 2017), 1–25.

[6] DÖLLERER, L., AND ENGELKE, A. Performant bounds checking for 64-bit WebAssembly. In *Proceedings of the 16th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages* (New York, NY, USA, Oct. 2024), ACM, pp. 23–31.

[7] GAMPE, A., VON RONNE, J., NIEDZIELSKI, D., AND PSARRIS, K. Speculative improvements to verifiable bounds check elimination. In *Proceedings of the 6th international symposium on Principles and practice of programming in Java* (New York, NY, USA, Sept. 2008), ACM, pp. 85–94.

[8] GELLER, A. T., FRANK, J., AND BOWMAN, W. J. Indexed types for a statically safe WebAssembly. *Proc. ACM Program. Lang. 8*, POPL (Jan. 2024), 2395–2424.

[9] KEIDEL, S., AND ERDWEG, S. Sound and reusable components for abstract interpretation. *Proc. ACM Program. Lang. 3*, OOPSLA (Oct. 2019), 1–28.

[10] PACCAMICCIO, M., RAIMONDI, F., AND LORETI, M. Building call graph of WebAssembly programs via abstract semantics.