Performant Bounds Checking for 64-Bit WebAssembly

Lukas Döllerer
Technical University of Munich
Munich, Germany
lukas.doellerer@in.tum.de

Alexis Engelke
Technical University of Munich
Munich, Germany
engelke@in.tum.de

Abstract

WebAssembly is becoming increasingly popular for various use cases due to its high portability, strict and easily enforceable isolation, and its comparably low run-time overhead. For determinism and security, WebAssembly guarantees that accesses to unallocated memory inside the 32-bit address space produce a trap. Typically, runtimes implement this by reserving all addressable WebAssembly memory in the host virtual memory and relying on page faults for out-of-bounds accesses.

To accommodate programs with higher memory requirements, several execution runtimes also implement a 64-bit address space. However, bounds checking solely relying on virtual memory protection cannot be easily extended for 64 bits. Thus, popular runtimes resort to traditional bounds checks in software, which are required frequently and, therefore, incur a substantial run-time overhead.

In this paper, we explore different ways to lower the bounds checking overhead for 64-bit WebAssembly using virtual memory techniques provided by modern hardware. In particular, we implement and analyze approaches using a combination of software checks and virtual memory, using two-level guard pages, and using unprivileged memory protection mechanisms like x86-64 memory keys. Our results show that we can reduce the bounds checking overhead from more than 100% when using software bounds checks to only 12.7% using two-level guard pages.

CCS Concepts: • **Software and its engineering → Runtime environments**; • **Information systems → Web applications**.

Keywords: WebAssembly, Bounds Checking, 64-bit, Virtual Memory, Memory Protection Keys

VMIL '24, October 20, 2024, Pasadena, CA, USA

ACM ISBN 979-8-4007-1213-5/24/10
https://doi.org/10.1145/3689490.3690400

ACM Reference Format:
Lukas Döllerer and Alexis Engelke. 2024. Performant Bounds Checking for 64-Bit WebAssembly. In *Proceedings of the 16th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '24), October 20, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3689490.3690400

1 Introduction

WebAssembly (Wasm) [14] is a bytecode format that allows efficient, portable, and safe execution of untrusted code and was originally designed for web browsers. Compute-intensive parts are written in languages that typically compile to machine code, like C++ or Rust, and are compiled to Wasm bytecode, which is then distributed alongside the website. The Wasm runtime in the browser then verifies and executes the bytecode, very frequently utilizing just-in-time compilation for higher performance.

The safety and portability of Wasm quickly led to adoption outside web browsers for sandboxed code execution, e.g., on embedded IoT devices or the blockchain as smart contracts [5]. Standalone runtimes like Wasmtime [6], Wasmer [31], and WasmEdge [8] also provide access to operating system functionality using the WebAssembly System Interface (WASI) [12], enabling the execution of normal programs without compiling them to native code first. Another application are databases [25, 28], where user-defined functions can be provided as WebAssembly code and, therefore, be integrated safely and efficiently into the query compilation process. Umbra [22, 28] even supports writing entire User-Defined Operators (UDOs) written in Wasm, which can be integrated into query plans and are deeply integrated into the compiled query. For performance, Umbra typically optimizes and compiles Wasm code using LLVM [17].

Wasm provides a single, linear memory that can grow dynamically and permits byte-granular access with 32-bit addresses. For safety and deterministic behavior across Wasm runtimes, memory accesses from Wasm that go beyond the bounds of the allocated memory region are guaranteed to terminate the current Wasm thread (*trap*) [1].

An efficient technique for checking these 32-bit Wasm addresses on a 64-bit host system is to reserve all addressable memory with inaccessible *guard pages*, as shown in Figure 1 and then rely on memory protection features present on all POSIX-compatible systems to detect invalid accesses to memory. The original Wasm whitepaper [14] already proposed this technique.

Figure 1. Memory layout for 3 GiB of accessible Wasm memory. The hatched region represents inaccessible memory, protected from illegal accesses using guard pages.

However, 4 GiB of addressable Wasm memory do not suffice for programs with high memory requirements, leading to a proposal to add 64-bit wide addresses [29], which is already implemented by several Wasm runtimes. Consequentially, the technique to reserve all addressable Wasm memory is no longer possible on 64-bit host systems. Therefore, conforming Wasm runtimes implementing this feature currently resort to software bounds checks for every memory access. Even after using static optimization passes to eliminate unnecessary checks, such checks are still required frequently and significantly increase execution time.

In this paper, we propose multiple bounds checking methods based on virtual memory techniques for 64-bit Wasm runtimes for efficient detection of out-of-bounds accesses. Our main approach is based on traditional guard pages combined with an additional macro guard region, enabling branchless access checks. Furthermore, we also evaluate the usage of memory protection keys, an architectural feature present on x86-64 processors. We implemented our approaches in Umbra's Wasm runtime. Our results show that the use of two-level guard pages allows for bounds checks with an average overhead of 12.7%. Additionally, all presented approaches have a lower overhead than software bounds checks, which have 109.8% run-time overhead compared to execution without bounds checking. Memory protection keys of x86-64 platforms turned out to be not practical for deterministically catching all out-of-bounds accesses.

In summary, the main contributions of this paper are:

- An approach extending guard pages to more than 32-bits involving explicit software checks to ensure that high address bits are clear.
- An approach adding a second level of macro guard pages to traditional (micro-) guard pages to implement two-level guard pages.
- An analysis of why x86-64 memory protection keys are not usable for performant Wasm bounds checking.

2 Background

To utilize hardware features for efficient Wasm out-of-bounds access detection, we use signals to receive the information about detected invalid accesses and C++ exceptions for safe handling of a fault condition in our Wasm runtime.

2.1 POSIX Signals

Memory references that violate memory protection settings typically cause a hardware exception on the CPU core that currently executes the offending instruction. A POSIX-compliant operating system translates this exception into a `SIGSEGV` signal[1] that is delivered synchronously to the currently executed thread.

If the thread's process has previously registered a signal handling function, this function is invoked with information about the signal's origin.

2.2 C++ Exceptions

C++ exceptions are objects that escape normal control flow by unwinding through the current function's call stack until they encounter a `try` block with a suitable `catch` clause that handles the thrown exception object type. If the exception reaches the end of the call stack without being handled, `std::terminate` is executed. There exist multiple possibilities for implementing C++ exception handling. Typically, these are either assisted by the operating system, as in Structured Exception Handling on Windows or use runtime support that reconstructs the call frame by parsing source code annotations, typically in the form of DWARF Call Frame Information (CFI).

On non-Windows systems, DWARF CFI-based approaches are the most popular because they allow for near-zero overhead C++ exception implementations. When the program throws an exception, it activates the unwinder to deliver the exception to the suitable catch clause. To have the exception bubble up the call stack, the unwinder needs to identify the stack frames of function calls in the call stack. This is only possible if all required unwind information was generated and stored in the executable by the compiler.

Besides delivering C++ exception objects to catch clauses, the unwinder also executes cleanup code for functions in the call stack whose frame is discarded, because they are terminated preemptively by the exception. This is extremely relevant for C++ code that defines non-trivial destructors, like `std::unique_ptr` or `std::lock_guard`. [19]

2.3 WebAssembly Memory

The Wasm memory is a linear array of bytes, accessed through `load` and `store` instructions. These calculate the target address by adding a 32-bit integer operand from the stack and a 32-bit integer immediate, which is encoded in the instruction itself. The resulting 33-bit address with a maximum value of $(2^{32}-1)+(2^{32}-1)=2^{33}-2$ describes the starting position of the target bytes in the linear memory array, starting with 0 for the first byte.

[1] For certain invalid memory references, a `SIGBUS` signal may be generated. We refer to both signals when talking about the `SIGSEGV` signal in the following paragraphs.

Similarly, with the memory64 proposal, Wasm memory can be accessed using 65-bit addresses, consisting of a 64-bit address integer from the stack and an encoded 64-bit offset.

The current, non-standardized memory64 feature proposal defines bounds checking for 65-bit addresses to be handled similarly to 33-bit addresses [29]. This implies that 65-bit addresses are neither truncated nor wrapped to fit into 64-bit CPU registers. While this replicates the behavior of 33-bit addresses, it does not seem practical. Therefore, this paper assumes that 65-bit addresses are wrapped to 64-bit addresses according to unsigned integer overflow mechanics.

If the current memory64 proposal is standardized without changes to this behavior, a possible mitigation would be to trap on overflows during address calculation. This is a valid approach because any access to an address greater than $2^{64} - 1$ is always out of bounds [29].

The Wasm memory's initial bounds are set in its declaration. During run-time, the `memory.grow n` instruction increases the memory's upper bound by `n` Wasm pages, each having a size of 64 kiB. Every memory access out of the current Wasm memory bounds is guaranteed to result in a Wasm trap, meaning the immediate termination of the current Wasm runtime thread. [1]

The Wasm specification defines 32-bit, 64-bit, and 128-bit types, which take up 4, 8, and 16 bytes of memory, respectively. Therefore, a memory access to the 128-bit vector type at address a accesses all bytes in the range $[a; a + 15]$. While explicitly checking the bounds of the last byte is possible, this can be avoided by always mapping a guard page behind the last accessible Wasm page to catch partial out-of-bounds accesses.

3 Signal-Based WebAssembly Trap Detection

The general idea of using signals to catch out-of-bounds memory accesses is to reserve all possibly reachable regions of virtual memory but deny access (i.e., map it with `PROT_NONE`). As these page-level protections are enforced for every access by the CPU, using this technique comes at no run-time overhead. We want to rely on such hardware mechanisms as much as possible to optimize for the overwhelmingly common case that an access is in-bounds. On POSIX systems, such violations can be caught using a custom `SIGSEGV` signal handler. The handler first validates that the faulting memory access was indeed caused by accessing one of the reserved memory regions. For debugging, we also verify that the code that caused the memory access, i.e., the instruction address in the signal frame, is part of the compiled Wasm code.

After successful validation, the signal handler has to stop the Wasm execution and hand over control to the runtime environment, which can then report the trap. One standard approach is to use `siglongjmp` inside the signal handler to jump to a function higher up in the stack frame, skipping all Wasm frames in between. Discarding Wasm stack frames like this is generally unproblematic. First, the specification explicitly states that if a memory access causes a trap, the current thread is terminated without any further changes to the runtime state. Second, all resources managed by the Wasm module, e.g., memory, are typically stored in a central place in the runtime state and can easily be discarded from there.

```
(func $wasm_entry
  call runtime_call_with_lock)
```

```
void runtime_call_with_lock() {
  // acquire lock
  const std::lock_guard lock(global_mutex);
  // call wasm code with the lock being held
  wasm_use_locked_object();
  // destructor of std::lock_guard releases lock
}
```

```
(func $wasm_use_locked_object
  ;; Because of some error,
  ;; this function causes a Wasm trap
  trap)
```

Figure 2. Combination of Wasm and C++ code that could lead to leaked resources. Without stack unwinding, the lock is not released on a trap.

However, this is problematic for systems with a deeper integration of Wasm like Umbra. The Wasm code might call a runtime function which, e.g., acquires a lock or allocates some memory and then calls back to another Wasm function that causes a trap — Figure 2 shows an example. If we exit the signal handler with a `siglongjmp`, destructors and other cleanup code of runtime functions will not get executed, causing memory leaks or deadlocks.

C++ exceptions provide a mechanism to unwind the stack while also calling destructors and other cleanup functions along the way. Throwing a C++ exception from a signal handler is not generally safe. However, we ensure the signal is caused by compiled Wasm code. C++ code can only be encountered by the unwinder higher up in the stack frame at function call boundaries, where the compiler expects possible exceptions. Furthermore, while signal handlers are no ordinary function calls but have a special signal frame created by the kernel, unwinders are in many cases able to detect these and unwind through the signal frame, either by detecting the `sigreturn` instruction sequence at the return address (e.g., on AArch64) or by having a custom signal restorer that is annotated with proper unwind information (e.g., on x86-64).

Thus, to simplify integration, we use C++ exceptions in our implementation. Nonetheless, we note that the standardized approach using `sigsetjmp`/`siglongjmp` should work

equally well but requires explicit setup before each entry to compiled `wasm` code, leading to a run-time overhead.

4 Guard Pages

A simple and straightforward extension of the guard page approach beyond the 32-bit address space is to simply increase the guard-paged allocation to 2^n for some suitable n, e.g., a 34-bit address space. Prior to every memory access, we now check in software whether the address is $< 2^n$, i.e., whether any bits above bit n are set. If this is the case, we know that the address points to a byte in memory outside of our mapped region, an out-of-bounds access that should raise a Wasm trap. Otherwise, we can execute the Wasm memory access normally, as an invalid memory reference now is guaranteed to hit a guard page and, therefore, cause a `SIGSEGV`.

A Wasm `memory.grow` instruction now behaves almost identically to how it behaves with 32-bit guard pages: it unprotects the requested pages from the start of the guard region. However, it disallows grow operations beyond the maximum memory size.

This concept is similar to software bounds checks; however, with our approach of fixing a maximum memory size, we do not need to use the exact current memory size because this dynamically changing size is implicitly modeled through the changes in memory protection settings of the guard region. This is especially relevant in a multi-threaded Wasm environment where the current memory size has to be reloaded for software-only bounds checking every check.

On x86-64, however, an immediate operand that tests the upper bits cannot be encoded. There are multiple options to implement this: (a) shift the address to the right and test whether the result is zero (`mov tmpreg, addrreg; shr tmpreg, 34; jnz trap`); (b) load the mask into a register and then use a `test` instruction (`mov tmpreg, 0xfffffffc00000000; test tmpreg, addrreg; jnz trap`); (c) store the large immediate in memory and perform a `test` with a memory operand (`test addrreg, [mask_addr]; jnz trap`). Note that all but the last code sequences require an extra temporary register, which is then unavailable for other code. Although the last option performs an extra memory access, this is almost always a cache hit and does not stall the following (speculative) execution on modern out-of-order processors. We verified this assumption also with micro-benchmarks and the LLVM MCA static code analyzer. This approach also allows us to dynamically grow our maximum Wasm memory size by adjusting the mask stored in memory.

Some other architectures like AArch64 support specifying the address mask as an immediate to the `tst` instruction, removing the need for an additional memory access or register per bounds check. In this implementation, dynamic resizing of the maximum Wasm memory requires recompilation of all compiled code.

5 Two-Level Guard Pages

The approach of larger guard page regions still requires explicit branches to detect out-of-range accesses. To avoid these compare-and-branch code sequences, we instead want to rely on hardware exceptions by adding an extra load instruction that accesses a different memory region based on the upper bits of the address.

Therefore, we add an extra memory region, which we will refer to as the *macro guard region*. This region consists of one 4 kiB host page per 1 GiB of Wasm 64-bit Wasm-addressable memory. A page in the macro guard region is mapped as read-only memory if the corresponding 1 GiB Wasm memory region is allocated in the address space (as a combination of accessible memory and possible micro guard pages). Otherwise, it is mapped as protected and acts as a macro guard page. We refer to this also as *compression*, because several micro guard pages are compressed into a larger macro guard page at a different position in memory.

To avoid using an extra register for the macro guard region's base address, we place the macro guard region immediately before the actual Wasm memory. Further, we arrange the macro guard pages in reverse order to avoid adding a large constant displacement that cannot be encoded as an immediate operand. Figure 3 illustrates the memory layout.

5.1 Bounds Checking

In this scheme, every memory access first performs a load from the macro guard region. This load will trigger an exception if the accessed macro guard page is protected, indicating an out-of-bounds access. If the load succeeds, the actual memory access will either hit accessible memory or a micro guard page.

For 4 kiB host pages (2^{12}) and 1 GiB Wasm memory (2^{30}) per macro guard page, the macro guard page address is computed as follows:

$$macroGuardPage = membase - (addr >> (30 - 12)) - 1$$

To save an instruction on some architectures, the -1 at the end can be avoided by reserving an extra 256 kiB (2^{18}) of guard pages at the end of the Wasm memory.

Figure 4 shows the resulting code. This adds four instructions per memory access on x86-64. However, these have no branches and subsequent code has no dependencies, enabling out-of-order execution to mostly hide this overhead. On AArch64, this causes just two additional instructions due to the availability of three-address instruction formats and an integrated shift in arithmetic instructions.

5.2 Growing Memory

Growing memory within the 1 GiB chunks is no different from standard guard page approaches: the corresponding

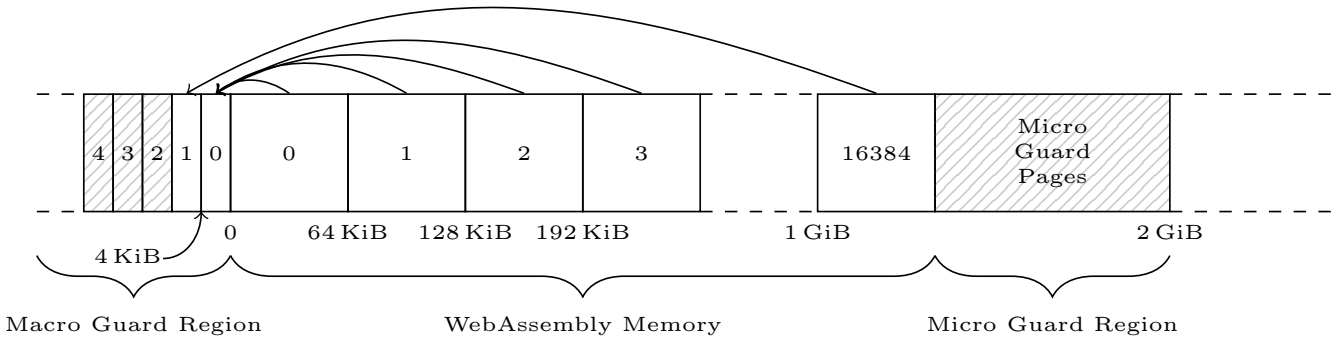

Figure 3. Two-Level Guard Pages with Micro Guard Pages implementation (not to scale). Hatched blocks represent protected memory, unhatched blocks are accessible memory.

```
; rdi = addr, rbx = membase
mov rax, rdi
shr rax, 18
not rax     ; -x-1 == ~x
; macro guard page probe:
movzx eax, byte ptr [rbx + rax]
; actual load:
mov eax, dword ptr [rbx + rdi]
```

```
; x9 = addr, x28 = membase
sub x0, x28, x9, lsr #18
ldrb w0, [x0]     ; macro guard page probe
ldr w0, [x28, x9] ; actual load
```

Figure 4. x86-64 and AArch64 assembly code showing a 32-bit load with the two-level guard page bounds checking scheme.

host pages are mapped as accessible. However, when growing across 1 GiB boundaries, the runtime must allocate micro guard pages for the entire 1 GiB segment before changing the macro guard page from protected to read-only with a call to `mprotect`.

5.3 Memory Consumption

The result of the macro guard page load is not used, and we do not store any data on our macro guard pages. Therefore, the macro guard region does not consume any actual memory because the mapped pages will be backed by the shared zero-page provided by the kernel.

For virtual memory, in our previously stated configuration with 1 GiB memory per macro guard page, the macro guard region has a size of 64 TiB on a host with 4 kiB page size ($\frac{2^{64}}{2^{30}} \cdot 2^{12}$; there are 2^{34} 1 GiB segments, which need one macro guard page of 4 kiB each).

The compression factor can be changed to reduce the virtual memory consumption — this is particularly important for Wasm runtimes that support multiple memories. However, while a larger compression results in a reduced macro guard region size, it also results in an increased virtual memory usage for the uncompressed micro guard pages. For 4 kiB host pages, the configuration with the smallest total virtual memory consumption is a compression size of 256 GiB (2^{38}; 256 GiB macro guard region size + 256 GiB for the first range of micro guard pages). Generally, the compression size for the lowest virtual memory consumption is $\sqrt{2^{64} \cdot \mathit{pagesize}}$. We note that the compression size only impacts virtual memory use but not performance.

5.4 Macro Guard Region Allocation

As macro guard pages will never hold any data and, therefore, will never occupy physical memory space, we map them with the `MAP_NORESERVE` flag. Nonetheless, this approach relies on overcommitting, which is typically allowed by the Linux kernel. However, the kernel can also be configured to disallow overcommitting over a certain limit entirely, even for unreserved mappings with the system setting `vm.overcommit_memory=2`. In this case, a possible workaround for such configurations is to create a file-backed mapping that maps the `/dev/zero` file as shared memory. No swap space will be reserved for the mapping, meaning this method does not require overcommitting at all.

6 Using x86 Memory Protection Keys

The previously described approaches require changes to the compiled Wasm code to enforce memory isolation and, therefore, necessarily have a performance overhead. A different approach is to completely rely on hardware to perform access checks by rendering all non-Wasm memory as inaccessible whenever compiled Wasm code tries to access it. The x86-64 platform provides a feature named Memory Protection Keys (PKeys), which allow restricting access to certain memory regions by writing to a user-space register.

Every page is assigned a PKey (0–15), defaulting to key 0. After an initial setup, read and write access to a page protected by a specific PKey can be inhibited by configuring the user space PKRU register appropriately with the instruction `wrpkru` [15, 24, 30]. Writing to this register is several magnitudes faster than an `mprotect` system call.[2]

To utilize PKeys for Wasm bounds checking, we need to mark all Wasm accessible pages with one PKey *a* and all non-accessible pages with a different PKey *b*. Switching to execution of compiled Wasm code therefore involves removing access rights to all pages marked with PKey *b* through the `wrpkru` instruction.

However, this poses the problem that memory sections like the stack or storage of the runtime system need to be accessed frequently during Wasm execution but must not be accessible directly from compiled Wasm code. This is the case because the Wasm specification guarantees the user that *any* memory accesses outside the bounds of the allocated Wasm memory region result in a trap. This is not efficiently possible for an approach using PKeys, making such an approach inherently non-compliant. Even when accepting this, while the instruction `wrpkru` is substantially faster than a system call, it has a very high execution time and incurs a memory barrier inside the CPU, severely impacting performance when the instruction is executed regularly. Thus, we deem the x86 PKeys feature unusable for implementing or supplementing Wasm bounds checking and do not consider this approach for our evaluation.

7 Experiments

We aim to increase the speed of 64-bit Wasm execution through faster bounds checking methods. Therefore, we implemented all of our approaches in the general purpose Wasm runtime of the Umbra database system [22, 28]. This runtime was initially designed for sandboxed evaluation of user-defined operators during query processing but also offers standalone execution of Wasm code.

7.1 Validation

To assert the valid behavior of our Umbra Wasm runtime implementation, we utilize the official Wasm specification test files. All proposed approaches pass all memory- and trap-related tests and, therefore, fulfill all of the Wasm specification's requirements for memory safety and security.

7.2 Performance

When testing the performance of program execution and compilation, a commonly used benchmark suite is SPEC CPU2017 [4]. While these benchmarks are commonly used and well-known, they are built for native code execution and compilation by full-featured compilers. Therefore, compiling the SPEC CPU2017 benchmarks to Wasm proved to be difficult, given the limited feature set of the available Wasm C and C++ compiler toolchains and limitations enforced by the Wasm specification. Problems in particular include missing `setjmp`/`longjmp`, C++ exceptions, inability to fork and execute child processes, and the unavailability of a Fortran compiler. Nevertheless, we present the compilable subset of the SPEC CPU2017 benchmarks: *505.mcf_r*, *508.namd_r*, *519.lbm_r* and *557.xz_r*. With some modifications to source code and compiler toolchains, more of the benchmarks could be made executable; however, this is out of the scope of this work.

The build and execution challenges of the SPEC CPU2017 benchmarks for Wasm, in addition to the licensing cost, led open source projects like Wasmtime [6] and Emscripten [34] to adopt different, non-standardized benchmark suites [11]. Meanwhile, official Wasm benchmarks primarily test the performance of Wasm in a browser environment with JavaScript interaction[3], which is undesired for measuring standalone execution performance. Therefore, we use our four compiling, standardized SPEC CPU2017 benchmarks.

Additionally, we use a 2D k-Means algorithm [18] benchmark on 10,000 to 20,000,000 data points, implemented by Sichert [28]. It was initially built to compare the execution speed of UDOs in Umbra with other database systems.

7.2.1 Setup. Our benchmarking system uses a single socket, 12-core x86-64 AMD Ryzen 9 7900X CPU, running with a peak frequency of 5.7 GHz. It has 61 GiB of Memory and runs a Linux kernel version 6.2.0-32-generic, built for the Ubuntu 23.04. Benchmarks were compiled using the wasi-sdk-21 toolchain and executed 20 times; the results presented were averaged. Wasm code execution speed highly depends on the host system and architecture. Therefore, it should be noted that the presented measurements are only intended for relative comparisons between the proposed bounds checking methods. Absolute execution times vary between different host machines.

7.2.2 Results. The measured execution times with all bounds checking methods are normalized to execution without bounds checking, denoted as *None*. *32-bit Guard Pages* refers to the standard guard page approach that does not need code changes, *64-bit Guard Pages* refers to the approach described in section 4. *2-lvl Guard Pages* refers to two-level guard page approach (section 5), and *Soft Bounds Check* to software bounds checks as a baseline.

Figure 5 shows that software bounds checks introduce a run-time overhead of up to 242% of the execution time without bounds checks.

[2] A synthetic benchmark executed on our test system showed that an `mprotect` system call is at least 30 times slower than a write operation to a PKey. Additionally, an `mprotect` system call may lead to a *TLB-shootdown*, degrading the performance of the entire process.

[3] E.g. 'webassembly/benchmarks', the official Wasm benchmark repository: https://github.com/WebAssembly/benchmarks (accessed 2024-01-26)

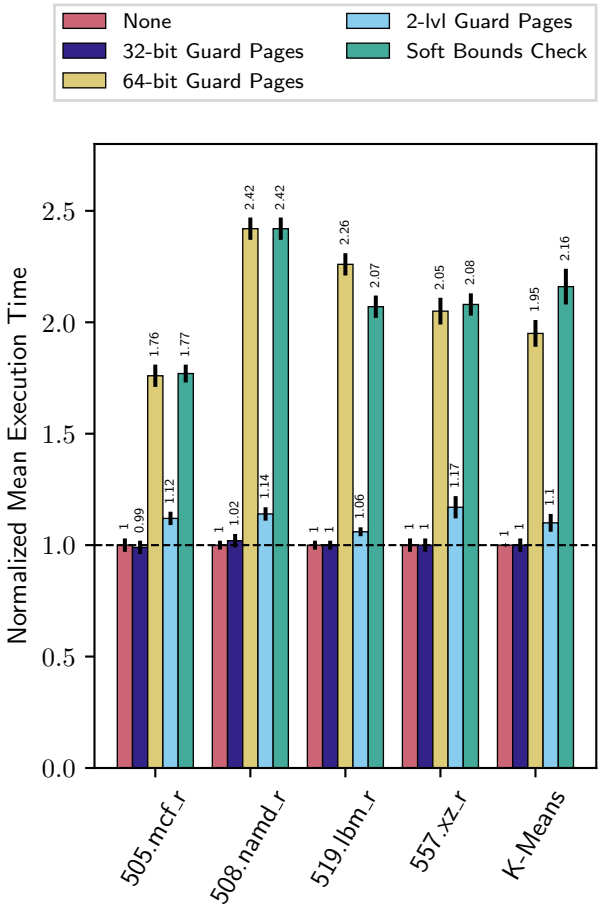

Figure 5. Normalized mean execution times of the compiled SPEC CPU2017 benchmarks and the 2D k-Means algorithm [18]. The results are normalized to the execution time without bounds checks enabled (*None*). Code generation and execution were performed using the Umbra LLVM backend and the Umbra Wasm runtime. (Lower is better)

Guard Pages for 32-bit Wasm execution only add additional unwind information and no executed instructions, leading to a similar execution time as if bounds checks were disabled completely. Their 64-bit equivalent additionally executes a `test`/`jne` instruction pair, including an additional memory access, making it just slightly faster than the software bounds checking method in most benchmarks with a run-time overhead between 76% and 142% of the baseline execution time.

While 64-bit guard pages only perform slightly better in most benchmarks, the two-level guard page approach consistently outperforms software bounds checks. This shows that while the two-level guard pages approach executes more instructions than a software bounds check, eliminating the conditional branch and memory size load substantially lowers the overhead incurred by bounds validation. With a maximum overhead of 17.3%, this approach proves to be a very beneficial strategy, also for larger and more complex applications.

8 Related Work

In order to get an overview of existing methods to decrease the overhead of bounds checking in Wasm execution, we analyzed popular standalone runtimes, like Wasmtime [6], Wasm3 [27], Wasmer [31], WasmEdge [8] and the three major browser runtimes in the JavaScript engines JavaScriptCore [2], V8 [13] and SpiderMonkey [20].

Most open-source Wasm runtimes adopted the Guard Pages bounds checking method for their 32-bit execution modes. A notable exception are SpiderMonkey, Wasm3, and WasmEdge, which always emit software bounds checks.

The memory64 feature proposal introduces larger Wasm memory sizes that require indexing using 64-bit addresses [29]. Therefore, 32-bit Guard pages can no longer be used for out-of-bounds detection. Wasm3, Wasmer, WasmEdge, and JavaScriptCore do not support the memory64 feature proposal or implement bounds checking for 64-bit addresses [33]. All other analyzed Wasm runtimes always emit software bounds checks [9]. However, the V8 team is currently evaluating alternative methods for future versions of their engine.[4]

8.1 Other Methods for Out-Of-Bounds Access Detection

Memory-unsafe programming languages like C and C++ allow programmers to access unallocated memory, leading to vulnerable and error-prone code. Therefore, other research groups already invented various tools to reliably detect unintended out-of-bounds accesses. *Address Sanitizers* track allocated memory regions through binary instrumentation, compiler support, or specialized memory allocator implementations [26]. Depending on the utilized method, they can detect various kinds of invalid memory accesses, often with a granularity of up to a single Byte over bounds. This preciseness typically leads to a high run-time overhead because every load and store operation requires a manual lookup of the accessed address, making the software solutions impractical for production use in a DBMS.

Modern Instruction Set Architecture (ISA) extensions are introduced to reduce the run-time overhead of these approaches by allowing programs to encode additional origin information into memory pointers or facilitate pointer validation in hardware. Existing extensions are, for example,

[4]No official information about this is available at the time of writing. The only information source is the following e-mail from the v8-dev mailing list: https://www.mail-archive.com/v8-dev@googlegroups.com/msg161691.html (accessed 2024-01-30)

SPARC ADI [23], ARM's Memory Tagging Extension for AArch64 [3], Intel's Linear Address Masking (LAM) [16] and CHERI, Capability Hardware Enhanced RISC Instructions [32]. However, these extensions may not be available for the host system. Especially for x86-64 systems, the LAM extension only enables the encoding of additional information into pointers without providing hardware support for validating pointers. Approaches like the ARMv8.5 Memory Tagging Extensions are especially promising for efficient, hardware accelerated bounds checking, as shown by Fink et al. [10], but are also not widely implemented in available processors.

Narayan et al. [21] introduce a custom ISA extension, which implements many of the isolation principles that Wasm provides in software, in hardware. They introduce, among other things, *regions* as an abstraction of Wasm memories at the hardware level with native bounds checking. While their ISA extension solves many of the performance problems related to software-based isolation, it is not yet incorporated into any available processors.

x86-64 CPUs feature debug registers for debuggers to store breakpoint conditions that are automatically checked in hardware. Four of the available eight registers can store addresses that are matched against operands of memory instructions. When an instruction's operand matches the register content, a debug exception is generated by the CPU [15]. Chiueh [7] use this hardware feature to detect out-of-array-bounds accesses in loops. However, the accessed out-of-bounds address must perfectly match the register content, meaning only 4 Bytes of virtual memory can be protected using the debug registers.

9 Conclusion

Out-of-bounds access detection in WebAssembly runtimes is required for safe and secure execution but poses a significant challenge for efficient code execution. Hardware memory protection mechanisms help to reduce the runtime overhead of bounds checks and can even eliminate any speed penalty for 32-bit Wasm execution.

For the most efficient execution of Wasm code with 64-bit addresses, our approach to use two-level guard pages for coarse-grained access checking and micro guard pages for fine-grained, Wasm page-level access control leads to a comparably low overhead of 12.7% on average, which is a substantial improvement from the overhead of software bounds checking of more than 100%.

References

[1] [n. d.]. *WebAssembly Core Specification.* Technical Report. W3C. https://www.w3.org/TR/wasm-core-2/ https://webassembly.github.io/spec/core/_download/WebAssembly.pdf.

[2] Apple Inc. 2024. JavaScriptCore. https://docs.webkit.org/Deep%20Dive/JSC/JavaScriptCore.html

[3] Arm Limited. 2024. *Armv8.5-A Memory Tagging Extension Whitepaper.* Technical Report. Arm Limited. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf

[4] James Bucek, Klaus-Dieter Lange, and Jóakim V. Kistowski. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ACM, Berlin Germany, 41–42. https://doi.org/10.1145/3185768.3185771

[5] Bytecode Alliance. 2023. WebAssembly Micro Runtime. https://bytecodealliance.github.io/wamr.dev/

[6] Bytecode Alliance. 2024. Wasmtime. https://wasmtime.dev/

[7] Tzi-cker Chiueh. 2008. Fast Bounds Checking Using Debug Register. In *High Performance Embedded Architectures and Compilers*, Per Stenström, Michel Dubois, Manolis Katevenis, Rajiv Gupta, and Theo Ungerer (Eds.). Vol. 4917. Springer Berlin Heidelberg, Berlin, Heidelberg, 99–113. https://doi.org/10.1007/978-3-540-77560-7_8

[8] Cloud Native Computing Foundation. 2024. WasmEdge. https://wasmedge.org/

[9] Alex Crichton. 2023. Architecture - Wasmtime. https://docs.wasmtime.dev/contributing-architecture.html#linear-memory

[10] Martin Fink, Dimitrios Stavrakakis, Dennis Sprokholt, Soham Chakraborty, Jan-Erik Ekberg, and Pramod Bhatotia. 2024. Cage: Hardware-Accelerated Safe WebAssembly. https://doi.org/10.48550/arXiv.2408.11456

[11] Nick Fitzgerald. 2021. rfcs/accepted/benchmark-suite.md · bytecodealliance/rfcs. https://github.com/bytecodealliance/rfcs/blob/main/accepted/benchmark-suite.md

[12] Dan Gohman. 2019. wasmtime/docs/WASI-overview.md at main · bytecodealliance/wasmtime. https://github.com/bytecodealliance/wasmtime/blob/main/docs/WASI-overview.md

[13] Google. 2024. V8 JavaScript Engine. https://v8.dev/

[14] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 185–200. https://doi.org/10.1145/3062341.3062363

[15] Intel Corporation. 2023. *Intel 64 and IA-32 Architectures Software Developer's Manual.* Intel Corporation, 2200 Mission College Blvd. Santa Clara, CA 95054-1549 USA. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[16] Intel Corporation. 2023. *Intel Architecture Instruction Set Extensions and Future Features.* 2200 Mission College Blvd. Santa Clara, CA 95054-1549 USA. https://cdrdv2.intel.com/v1/dl/getContent/671368

[17] C. Lattner and V. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, San Jose, CA, USA, 75–86. https://doi.org/10.1109/CGO.2004.1281665

[18] S. Lloyd. 1982. Least Squares Quantization in PCM. *IEEE Transactions on Information Theory* 28, 2 (March 1982), 129–137. https://doi.org/10.1109/TIT.1982.1056489

[19] John McCall, Richard Smith, Jason Merrill, and Tom Honermann. 2012. *C++ ABI for Itanium: Exception Handling.* https://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html

[20] Mozilla Corporation. 2024. SpiderMonkey JavaScript/WebAssembly Engine. https://spidermonkey.dev/

[21] Shravan Narayan, Tal Garfinkel, Mohammadkazem Taram, Joey Rudek, Daniel Moghimi, Evan Johnson, Chris Fallin, Anjo Vahldiek-Oberwagner, Michael LeMay, Ravi Sahita, Dean Tullsen, and Deian Stefan. 2023. Going beyond the Limits of SFI: Flexible and Secure Hardware-Assisted In-Process Isolation with HFI. In *Proceedings of the*

28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 266–281. https://doi.org/10.1145/3582016.3582023

[22] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, the Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf

[23] Oracle. 2022. *Adi - Man Pages Section 7: Standards, Environments, Macros, Character Sets, and Miscellany*. Technical Report. https://docs.oracle.com/cd/E88353_01/html/E37853/adi-7.html

[24] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. 2019. Libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 241–254. https://www.usenix.org/conference/atc19/presentation/park-soyeon

[25] Piotr Sarna. 2022. Wasmtime: Supporting UDFs in ScyllaDB with WebAssembly. https://www.scylladb.com/2022/04/14/wasmtime/

[26] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[27] Volodymyr Shymanskyy and Steven Massey. 2024. Wasm3. Wasm3 Labs. https://github.com/wasm3/wasm3

[28] Moritz-Felipe Sichert. 2024. *Efficient and Safe Integration of User-Defined Operators into Modern Database Systems*. Ph. D. Dissertation. Technische Universität München, Munich. https://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20240111-1713746-1-4

[29] Ben Smith and Andreas Rossberg. 2020. memory64/proposals/memory64/Overview.md at master · WebAssembly/memory64. https://github.com/WebAssembly/memory64/blob/master/proposals/memory64/Overview.md

[30] The Linux man-pages project, Michael Kerrisk, and Alejandro Colomar. 2023-05-03. *mprotect(2) — Linux Manual Page*. https://man7.org/linux/man-pages/man2/mprotect.2.html

[31] Wasmer. 2024. Wasmer. https://wasmer.io

[32] Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Peter G. Neumann. 2019. *An Introduction to CHERI*. Technical Report 941. University of Cambridge - Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom. 43 pages. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-941.pdf

[33] WebAssembly Community Group and Andreas Rossberg. 2024. Feature Extensions - WebAssembly. https://webassembly.org/features/

[34] Alon Zakai. 2011. Emscripten: An LLVM-to-JavaScript Compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*. ACM, Portland Oregon USA, 301–312. https://doi.org/10.1145/2048147.2048224

Received 2024-07-25; accepted 2024-08-21