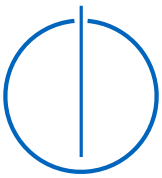
TUM School of Computation,
Information and Technology

Department of Computer Science
Technical University of Munich

Bachelor's Thesis in Informatics

Hardware-Accelerated Bounds Checking for WebAssembly

Lukas Döllerer



TUM School of Computation,
Information and Technology

Department of Computer Science
Technical University of Munich

Bachelor's Thesis in Informatics

Hardwarebeschleunigtes Bounds Checking für WebAssembly

Hardware-Accelerated Bounds Checking for WebAssembly

Author:	Lukas Döllerer
Supervisor:	Prof. Dr. Thomas Neumann
Advisor:	Dr. Alexis Engelke
Submission Date:	15 March 2024

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Ich versichere, dass ich diese Bachelor's Thesis selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Ort, Datum	Lukas Döllerer

Acknowledgments

First, I would like to thank my advisor Dr. Alexis Engelke for his permanent and frequent support, valuable ideas, and foresight. You enabled me to choose a research topic that both sparked my interest and provided relevant outcomes. You guided my research journey through proofreading, brainstorming, and valuable, honest feedback. I also want to thank Dr. Moritz Sichert, without whose preliminary work in this field, this thesis would not have been possible. Overall, I want to thank the TUM chair for database systems and my supervisor, Prof. Dr. Thomas Neumann, for the opportunity to participate in your research and for letting me use your hardware.

Second, I would like to thank my family and friends for giving me support and motivation when I needed it the most. In particular, I want to thank Adrian and Johnny for accompanying me on my academic journey. Without you, my life and studies would certainly not be as successful, but more importantly, both would be way less fun.

Third, I want to thank my mother. Through your love and care, you enabled me to pursue an academic degree in the first place.

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. The Umbra database system allows users to define custom operators using WebAssembly bytecode. These so-called user-defined operators act directly on the stored tuples and are compiled and executed together with the supplied SQL query.
To ensure determinism and security, WebAssembly guarantees that accesses to unallocated memory inside its 32-bit address space produce a trap, preventing illegal access to the database's memory. 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. For 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 easily be extended for 64 bits, making runtimes resort to software bounds checks. These are required frequently and therefore incur a substantial run-time overhead.
In this thesis, we propose 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 shadow memory pages, a combination of software checks and virtual memory, and unprivileged memory protection mechanisms like x86_64 memory keys, in the Umbra WebAssembly runtime.
Our results show that we can significantly reduce the more than 100% bounds checking overhead of software bounds checks to 12.7% using shadow memory techniques for LLVM-optimized WebAssembly machine code, and to 6.8% through a combination of software bounds checks with virtual memory protection for unoptimized machine code.

Contents

1 Introduction

WebAssembly is the web industry's answer to growing performance requirements for code execution in web browsers [21]. JavaScript, a dynamically typed script language, is the de facto standard for rendering websites and interactivity inside the browser's sandbox. But, despite significant development efforts invested by JavaScript runtime manufacturers like Google for over 15 years into the optimization of its execution [7], certain overheads introduced by the dynamic nature of dynamically typed programming languages cannot be completely eliminated [14].

Meanwhile, traditional compilers excel at creating highly optimized machine code from statically typed programming languages for a long time now. They established C and C++ as the foundation of most high-performance applications for over 52 years.

WebAssembly (Wasm) bytecode combines the highly optimized code generation of traditional compilers with the portability and security of a web browser's sandboxed environment. Static, highly optimizable languages like C, C++, or Rust are compiled using compiler frameworks like LLVM. However, instead of emitting machine code, the compiler uses Wasm as the compilation target. Wasm runtimes execute the generated bytecode, either as part of a browser's JavaScript runtime or in a standalone fashion. In particular, all modern web browsers like Chrome, Safari, Microsoft Edge, and Firefox support Wasm execution.

Sandboxed execution of high-performance code is not only of interest for web browsers but also for Database Management Systems (DBMSs) like the DBMS Umbra, a research DBMS that compiles input SQL queries to executable machine code [48]. Data scientists utilize complex algorithms that cannot be easily translated to SQL, tempting them to export data to non-relational analytics tools. This is not only a slow process, but it also comprises all the benefits of a relational DBMS. Therefore, Sichert and Neumann [59] proposed a method to execute user-provided arbitrary code alongside compiled SQL queries. They introduce the User-Defined Operator (UDO), whose logic is written in Wasm code that is dynamically translated to machine code and integrated into Umbra's query execution engine.

In order to ensure the security and safety of Wasm execution, all memory accesses from Wasm code need to be within a reserved memory region, to prevent out-of-bounds access exploits or corruption of the DBMS. The Wasm specification defines a 32-bit address space and guarantees a Wasm trap for every illegal memory access, which leads to the termination of the current Wasm thread. The Wasm whitepaper by Haas et al. [21] already proposes a near-zero overhead solution to 32-bit Wasm execution on 64-bit host systems, which relies on virtual memory protection methods.

For programs with higher memory requirements, a 32-bit address space with a maximum of 4 GiB of addressable memory may not suffice, leading Wasm runtimes to allow the use of 64-bit addresses through the memory64 feature proposal [60]. With an address space as

large as the virtual memory of most host systems, previously employed techniques for the detection of out-of-bounds accesses cannot be used any longer. Therefore, runtimes resort to software bounds checks, which are additional test instructions in the source code that manually check if addresses are within the allocated memory's bounds. These instruction sequences are required frequently, leading to a high run-time overhead.

In this thesis, we propose, implement, and analyze four hardware-accelerated approaches to efficient bounds checking for 64-bit WebAssembly execution in our test system, the Umbra DBMS' Wasm runtime.

Outline

The remainder of this thesis is structured as follows: In Chapter 2, we provide background information about the Wasm bytecode, the Umbra DBMS, UDOs, and the Umbra Wasm runtime. Following, in Chapter 3, we discuss existing techniques for hardware-assisted bounds checking and analyze the bounds checking techniques employed by famous open-source Wasm runtimes. Chapter 4 describes our implementations of the proposed out-of-bounds access detection techniques, which are later analyzed in Chapter 5. We start with the implementation of the commonly used 32-bit bounds checking technique, followed by an upgrade of said technique for 64-bit Wasm. We then explain a technique that relies on Memory Protection Keys, an x86_64 hardware extension. Afterward, we introduce two slightly different shadow memory approaches. Finally, we summarize our results and motivate further research opportunities.

Contributions

Key contributions of this thesis are the description, implementation, and analysis of the following bounds checking methods for 64-bit WebAssembly execution.

- *Guard64*:
  The combination of software bounds checks and a commonly used 32-bit Wasm bounds checking technique.
- *PKeys*:
  Bounds checks using the x86_64 Memory Protection Keys (PKeys) hardware extension.
- *Page Granular Shadow Memory*:
  A shadow memory based bounds checking technique.
- *Compressed Shadow Memory*:
  A variant of the page granular bounds checking requiring less virtual address space.

Additionally, we describe and implement a safe Wasm trap delivery mechanism for Wasm runtimes, based on Linux signals and C++ exceptions.

2 Background

The following section provides an overview of the WebAssembly (Wasm) bytecode format, the Umbra Database Management System (DBMS), User-Defined Operators (UDOs) and the optimized Umbra WebAssembly runtime. We also provide an overview of C++ exception handling and the required stack unwinding process.

2.1 WebAssembly

Web browsers are required to allow the execution of arbitrary, external code for rendering and interactivity of websites while maintaining a safe and secure environment. For a long time, all major browsers supported JavaScript as the only programming language executable inside a browser tab's sandbox. Therefore, JavaScript engines like Mozilla's SpiderMonkey [47], Google's V8 [20], or Apple's JavaScriptCore [4] put much effort into making the interpreted, dynamically typed language run fast through just-in-time (JIT) compilation. However, the language's design limits the potential for optimizations.

As a faster alternative, `asm.js` was introduced by Mozilla [22]. It is a strict subset of ECMAScript 2015, which is already supported out-of-the-box by any browser supporting the execution of JavaScript. It served as a compilation target for C and C++ code. Well-engineered, existing compiler infrastructure like LLVM was used to generate highly optimized, assembly-like JavaScript that browsers could execute faster than handwritten JavaScript. Execution engines were even specifically optimized for the execution of `asm.js`.[1]

Despite the optimization efforts, `asm.js` was still not fast enough to enable computationally complex tasks like video processing or game rendering in browsers [21]. This led engineers from Google, Microsoft, Mozilla, and Apple (Haas et al. [21]) to take the idea further. They introduced a new low-level bytecode called WebAssembly (Wasm). Like `asm.js`, Wasm is an abstract compilation target for compilers. But it is executed in specialized Wasm runtimes, which enables low to no-overhead execution [21].

2.1.1 File Formats

Wasm bytecode can either be stored in a compact binary format or a human-readable text format, commonly having the file extensions `.wasm` and `.wat`, respectively. The Wasm specification uses an abstract syntax to define instructions, data types, and more with a high level of abstraction. The binary format is a dense representation of said abstract syntax, optimized for size and parsing speed. The text format uses S-Expressions [52] to

[1] `https://blog.mozilla.org/futurereleases/2013/11/26/chrome-and-opera-optimize-for-mozilla-pioneered-asm-js/` (accessed 2024-02-04)

render code defined through the specification's abstract syntax in a textual representation for human inspection.[53]

The `.wast` file format is an extension of the Wasm text S-Expression format. It adds "commands" that allow the structuring of a Wasm text file to create Wasm test scripts.[2] The Wasm project supplies a test suite for runtimes to check specification conformance, written using the `.wast` file format.

2.1.2 Design

Wasm is designed as a portable abstraction of modern hardware with a minimal instruction set. A Wasm program binary is called a `module`, which contains the functions, globals, tables, and Wasm memories. Functions contain the actual code in the form of instructions. Instructions are defined to interact with a stack machine; however, because of the strict type system, the stack layout is statically computable so that actual implementations never have to materialize the stack. Even though instructions may consume operands from the current stack, they can also encode operands within themselves. Instructions interact with values. The bytecode specification only allows five value types: signed integer, unsigned integer, and IEEE 754 floating point number types, each in 32-bit and 64-bit length, and a 128-bit wide vector type for SIMD execution.[53]

Wasm programs can access storages called Wasm *memories*, which are linear, contiguous arrays of memory. With the current specification release 2.0, each Wasm module has at most one Wasm memory. It is 8-bit addressable via 32-bit wide addresses, with 0 being the address for the first byte. A feature proposal to the Wasm standard allows the use of 64-bit addresses. However, this is still experimental and not widely supported [60, 69, 53] The Wasm memory is divided into chunks called `pages` with a fixed size of 64 KiB. Modules declare an initial `page` count of their Wasm memory and an optional maximum `page` count. The Wasm memory can be dynamically extended using the `memory.grow x` instruction which extends the Wasm memory by x `pages`.

The current Wasm specification [53] allows two instruction types to access a Wasm memory: `load` and `store` instructions. Both pop a 32-bit integer from the stack and interpret it as the address inside the module's Wasm memory. The instructions themselves additionally encode a 32-bit integer offset in their *memarg*, which is added to the loaded address. When accessing a value of type t at the resulting 33-bit address a in memory, the actually accessed bytes are within the range $[a, a + \text{sizeof}(t) - 1]$, and all bytes need to be within the bounds of the allocated memory region. The largest type defined by the current specification is the 128-bit wide vector type, meaning $\text{sizeof}(t) \leq 16$. It suffices to check whether address $a' = a + \text{sizeof}(t) - 1$ is in bounds, which also covers all other accessed addresses [21]. Following references to the "address" for bounds checking refer to this end address a'.

Some instructions can deviate from normal control flow by producing a trap. This is similar to an exception and causes the execution to immediately halt. Wasm programs cannot handle traps themselves. The specification requires the outside environment to

[2] `https://github.com/WebAssembly/spec/tree/master/interpreter#scripts` (accessed 2024-02-15)

```c
int fac(int* n, int acc)
{
    if (*n <= 1) {
        return acc;
    }
    int new_acc =
        *n * acc;
    (*n)--;
    return
        fac(n, new_acc);
}
```

```
(func $fac (type 2)
    (param i32 i32) (result i32)
    (local i32)
    block  ;; label = @1
        local.get 0
        i32.load
        local.tee 2
        i32.const 2
        i32.lt_s
        br_if 0 (;@1;)
        local.get 0
        local.get 2
        i32.const -1
        i32.add
        i32.store
        local.get 0
        local.get 2
        local.get 1
        i32.mul
        call $fac
        local.set 1
    end
    local.get 1)
```

Figure 2.1: *Comparison between C and Wasm code for calculating the faculty. The Wasm code is in the Wasm text format (`.wat`), described in Section 2.1.1. The standard faculty function was adjusted to work on memory to showcase Wasm memory accesses.*

catch and handle the failure. One notable source for traps are memory accesses outside of the bounds of the allocated memory.[53]

2.1.3 WebAssembly System Interface (WASI)

Wasm was initially designed as a bytecode for execution in web browsers [21]. Therefore, it features a comprehensive JavaScript Application Programming Interface (API) to instantiate Wasm modules, call exported functions and interact with the Wasm memory. Because of the sandboxed nature of JavaScript execution in the browser, Wasm interacts with the outside world only through the API provided by the runtime. This is limited to the web APIs for web browsers, which are also accessible from regular JavaScript. In order to execute Wasm in browser-independent runtimes like Wasmtime [8] or Wasmer [1], the WASI API was introduced by the Wasmtime project. It allows Wasm to access the underlying system in a platform-agnostic manner without the need for a web browser environment. The wasi-sdk project[3] is its leading implementation. It ships the wasi-libc, a partial implementation of the C standard library [9], in combination with a modified version of the LLVM compiler infrastructure framework [40] with Wasm support [19]. The example from Figure 2.2 was compiled using the wasi-sdk clang compiler. The Emscripten project features some support for WebAssembly System Interface (WASI) APIs but mainly focuses on in-browser execution and JavaScript APIs [70]. Therefore, our focus in this thesis is on the wasi-sdk.

[3] `https://github.com/WebAssembly/wasi-sdk` (accessed 2024-01-30)

```c
#include <unistd.h>

int main() {
    write(0, "Hello, World!\n", 14);
}
```

```
(func $main (type 3) (result i32)
    i32.const 0
    i32.const 1024
    i32.const 14
    call $write
    drop
    i32.const 0)

(func $write (type 4)
    (param i32 i32 i32)
    (result i32)
    ;; Omitted wrapper code...
    call $__wasi_fd_write
    ;; Omitted wrapper code...
)

(data $.rodata (i32.const 1024)
    "Hello, World!\0a\00")
```

Figure 2.2: *Comparison between C and wasi-sdk Wasm code for writing a string to the standard output file descriptor using the WASI API.*

2.1.4 Performance

Wasm is a compilation target for code written in languages that allow efficient compilation and static optimization through mature compilers. This gives it a head start in execution speed, making Wasm faster than JavaScript in light applications [61]. However, for heavy applications, like games, the JIT compilers of JavaScript engines give JavaScript code a performance advantage by compiling frequently used routines to native machine code [61]. This optimization was not yet optimized for Wasm execution in 2021 [66]. Haas et al. [21] were able to show in 2017 that Wasm was within 2 times the execution speed of native code execution, for most benchmarks. Since then, browser manufacturers have continued improving their Wasm runtimes. Additionally, multiple feature proposals to the standard try to improve execution speed further, leaving us optimistic for further convergence to native speed.

2.2 Umbra

The Umbra DBMS is a novel disk-based database system developed by Neumann and Freitag [48] at the Technical University of Munich. It is a compiling DBMS, meaning that logical query plans are derived from the SQL input and translated to an Intermediate Representation (IR), similar to the LLVM IR [40]. This IR already contains all operations required to be performed on the stored tuples but in a machine-agnostic format. Umbra then features four code generation and execution back-ends to perform the instructions: An LLVM-based back-end that translates the Umbra IR to actual LLVM IR and uses the existing LLVM compiler infrastructure for optimization passes and code generation. This produces highly optimized code but requires substantial time for compilation. To bridge this startup time, Kersten et al. [39] built the DirectEmit back-end, also known as *Flying Start*, which is a code generator that translates Umbra IR directly to x86_64 machine

code, without optimizations. For debugging purposes, a C back-end that compiles Umbra IR to C code is available. However, it is not used for performance-critical applications. A virtual machine back-end that interprets Umbra IR is also available but superseded by the DirectEmit back-end for execution on x86_64 systems.[48, 39]

2.2.1 Parallel Execution

Umbra's query engine is optimized for parallel execution of SQL queries. Workloads are divided into smaller work packets, so-called *morsels*. These chunks are distributed between worker threads to execute queries in parallel. This morsel-drive query engine makes use of the system's full computational capabilities. [48]

2.2.2 Proxy Classes

Machine code generated by the Umbra runtime is executed as part of the Umbra process and can execute callbacks from the Umbra source code. The translation of these calls and accesses from emitted code is facilitated through *Proxies*. They are C++ wrapper classes that allow generated code to access members and methods of the wrapped C++ classes from the Umbra source code. They provide typed and safe access to underlying C++ data structures from untyped pointers to memory.[39]

2.3 User-Defined Operators

Nowadays, data scientists are required to perform complex algorithms on their datasets. These algorithms often cannot be trivially expressed in SQL because of the language's set-oriented and declarative nature. Therefore, users are tempted to extract data from relational Database Management System (DBMS) into non-relational analytics systems like Tensorflow or Spark, comprising advantages of relational DBMSs. Even though users may program imperative control flow via recursive Common Table Expressions, using the `WITH` keyword, or imperative extensions to the SQL standard and User-Defined Function (UDF), these tend to be either highly complex or rather slow in execution.[59]

DBMSs execute a query written in SQL by performing operations on sets of tuples through operators. Sichert and Neumann [59] introduce the concept of User-Defined Operator (UDO), which they implemented in their research DBMS Umbra and Postgres. UDOs allow users to run custom algorithms inside Umbra's compiling query engine. They are written in a high-level imperative programming language like C or C++, interfacing the compiling DBMS through an easy-to-use Application Programming Interface (API). They are then compiled and integrated into the machine code generated by the DBMS. This method enables imperative algorithm execution with high data throughput and preservation of ACID[4] properties.[59]

The solution proposed by Sichert and Neumann [59] directly integrates the UDO machine code into the code generated by the DBMS. This tight integration leads to near zero-overhead execution but also grants the UDO unrestricted access to the DBMS, posing

[4]Atomicity, Consistency, Isolation, Durability

a security risk and the possibility for users to accidentally corrupt the database's internal state. Additionally, UDOs can only be inlined in code generated by Umbra's LLVM back-end [58]. In his Ph.D. thesis [58], Sichert proposes the usage of a WebAssembly runtime, integrated into the DBMS's query engine, to overcome this hazard and execute arbitrary user-generated code safely and efficiently for all code generation back-ends.

2.4 Umbra WebAssembly Runtime

The Wasm runtime was built into the Umbra DBMS by Sichert to support UDO execution as part of his Ph.D. thesis. In addition to the added safety, using Wasm as an intermediary format for UDOs instead of compiled machine code has the advantage that the UDO's logic is available in Umbra IR which can be optimized and inlined together with other query-related code by all available code generation back-ends.

We first parse the supplied Wasm code from the Wasm binary format to an internal representation for further processing. We then translate this representation to Umbra IR. This process involves mapping the stack-machine-based instructions of Wasm code to the register-machine-based IR operations. The strict Wasm type system enables us to accomplish this translation without a materialized stack at run-time. Sichert describes the translation process in detail in his Ph.D. thesis.

We use the translated code together with the Umbra Wasm runtime for three modes of operation. The first mode allows standalone execution of Wasm programs. It is used for debugging and profiling purposes and features limited WASI support for this purpose. The second mode allows execution of Wasm specification test files to ensure compliance of the runtime with the current specification. We use the third mode for execution of translated UDOs together with other database operators in Umbra's query engine.[58]

2.4.1 Multi-Threading

Wasm UDOs are executed as a part of the morsel-driven Umbra query engine. Therefore, to be reasonably efficient, they require the ability to run in parallel, which needs to be supported by the Wasm runtime. The runtime supports the Threading feature proposal that introduces Wasm memory sharing among multiple Wasm runtime instances and atomic memory access operations [58]. The bounds checking features proposed in this thesis are all designed to work in a multi-threaded environment. However, for ease of implementation and testing, we currently only feature implementations that are safe for single-threaded execution.

2.4.2 WebAssembly Trap Handling Safety

Because of its tight coupling to the Umbra DBMS code generation, the Wasm runtime can also utilize existing features like Proxies. This enables the Wasm runtime to generate code that interacts with C++ objects, like the Wasm runtime object itself, during the execution of the Wasm code. For example, the pointer to the Wasm memory is shared with the Wasm code through such a Proxy. Likewise, complex Wasm instructions like `memory.grow` are implemented through a call to a corresponding C++ function. Therefore,

a pointer to the Wasm runtime instance is always the first parameter of all generated Wasm functions, to facilitate these Proxy calls.

A Proxy call may execute arbitrary C++ code, making proper stack unwinding important even when processing a Wasm trap. The Proxy function may have constructed a C++ object with a non-trivial destructor, which must be called during the deletion of the object [63], e.g. a `std::lock_guard`. Afterward, the Proxy function may have handed execution to another function from translated Wasm code, as shown in Figure 2.3. This seamless integration of Wasm code into C++ code is possible through C++ functor objects wrapping entry points to generated machine code. In the given example, skipping the unwinding of the stack would not call the required destructor of the `std::lock_guard` and, therefore, cause undefined behavior and a lock leak [63].

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

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

```
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();
    // std::lock_guard destructor
        releases lock
}
```

Figure 2.3: *Combination of Wasm and C++ code that could lead to leaked resources. Without stack unwinding, the lock is not released on a trap.*

2.5 Stack Unwinding for C++ Exceptions

C++ exceptions are objects that break normal control flow by *bubbling* up 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. For our target system and compiler, Linux on x86_64 with either gcc-13 [18] or clang-17 [62], this process is carried out by the C++ standard library and runtime, e.g. libstdc++ [16], and an unwinder, like libunwind [56, 44].

After the exception object is allocated by the C++ runtime, a two-phase process is started. In the first phase, the unwinder steps through the stack frames of all functions in the call stack. It calls each function's *personality function* which determines whether the thrown exception object is handled by the concerning function. If this search phase does not yield a valid exception handler, the program is terminated. Otherwise, the second phase starts again at the throw point and unwinds the call stack again. However, now all destructors are called until the matching exception handler is reached. For the code in Figure 2.3, the `std::lock_guard`'s destructor would be called, unlocking the global mutex.

To carry out these two phases, the unwinder needs to identify stack frames, Instruction Pointer (IP), and stack pointer for all functions in the call stack, only based on the register

contents at the time when the exception was thrown. Therefore, it would be unable to properly unwind the call stack for an exception from code executed in a signal handler. Signal handlers are not called by the function causing the signal but by the operating system's kernel. We still want the unwinder to unwind the call stack as if the exception was thrown by the signal-causing function. Therefore, the C standard library glibc [17] generates a signal trampoline for all signal handlers that instructs the unwinder where to find the register contents of the function that raised the signal. An in-depth explanation of this process can be found in Appendix A.6.[46, 67]

Even though the unwinder is technically able to properly process a C++ exception thrown in a signal handler, this process is not allowed by the C++ programming language standard [63]. Exceptions may only interrupt regular function execution at two points: When executing a `throw` operation and during a function call. All other program points may not have enough unwind information for proper stack unwinding and may not be in a defined state for unwinding.

However, Wasm code is guaranteed to never require additional cleanup or destructors outside of the Wasm runtime. A Wasm trap stops execution and discards the runtimes state. Therefore, Wasm code can be interrupted in a defined manner at any Wasm instruction. The unwinding of intermediary Proxy function frames is also defined for C++ exceptions thrown during Wasm execution because all Wasm code calls are function calls. This enables us to safely handle Wasm traps using C++ exceptions thrown from signal handlers without compromising the state of the surrounding Umbra runtime.

3 Related Work

Validating memory operations to only access allocated and assigned memory regions is not only an important part of safe and secure Wasm execution but also plays a crucial part in the security of memory-unsafe languages. We analyzed how memory out-of-bounds accesses are detected in general and how existing Wasm runtimes implement efficient bounds checking.

3.1 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, researchers 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 [55]. 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 [55].

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, SPARC ADI [49], arm's Memory Tagging Extension for AArch64 [5], Intel's Linear Address Masking (LAM) [23] and CHERI, Capability Hardware Enhanced RISC Instructions [68]. 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 acceleration for validating pointers.

x86_64 CPUs feature debug registers to store a debugger's 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 [24]. Chiueh [10] uses this hardware feature to detect out-of-array-bounds accesses in loops with a very low run-time overhead. However, the accessed out-of-bounds address must be a perfect match with the register content, meaning only 4 bytes of virtual memory can be protected using the debug registers.

3.2 Bounds Checking in Other WebAssembly Runtimes

In order to get an overview of existing methods to decrease the overhead of bounds checking in Wasm execution, we analyzed the runtimes shown in Figure 3.1.

Standalone Execution	Wasmtime [8]
	Wasm3 [57]
	Wasmer [1]
	WasmEdge [11]
Browser JavaScript Engines	JavaScriptCore [4]
	V8 [20]
	SpiderMonkey [47]

Figure 3.1: *Analyzed Wasm runtimes.*

The browser JavaScript engines and their integrated Wasm runtimes were selected because they are integrated into three major browser engines: WebKit for the Safari browser, Blink for the Google Chrome browser, and Gecko for the Firefox browser. The standalone Wasm engines analyzed are the most popular open-source projects to date, based on their GitHub star rating.

Haas et al. [21] propose to utilize virtual memory protection techniques for 32-bit Wasm execution on systems with a larger virtual address space, like modern 64-bit systems. We also implement and test this approach, which we call *Guard Pages*. Most other open-source Wasm runtimes also adopted this method. A notable exception are SpiderMonkey, Wasm3, and WasmEdge, which always emit software bounds checks. Wasmtime uses a guard region of 2 GiB after a statically allocated memory region of 4 GiB. All accesses with memarg offsets larger than 2 GiB are therefore always actively bounds checked. Additionally, they always allocate a 2 GiB guard region preceding their main memory to account for possible implementation errors in their compiler back-end CraneLift [13]. Other runtimes like V8 and JavaScriptCore always allocate larger guard regions to fully eliminate software bounds checks for 32-bit Wasm execution.[1,2]

The memory64 feature proposal introduces larger Wasm memory sizes that require indexing using 64-bit addresses [60]. This allows Wasm code to address every byte of the virtual memory space of modern CPUs. Guard pages can therefore not be used for out-of-bounds detection any longer, because they require the allocation of all Wasm-addressable bytes. Wasm3, Wasmer, WasmEdge, and JavaScriptCore do not support the memory64 feature proposal and do not implement bounds checking for 64-bit addresses [69]. All other analyzed Wasm runtimes always emit software bounds checks [13], however, the V8 team is currently evaluating alternative methods for future versions of their engine.[3]

[1] `https://github.com/v8/v8/blob/7fdc2728f2114b083ebbd24710a32fe4574c0c57/src/objects/backing-store.cc#L62` (accessed 2024-03-09)

[2] `https://webkit.org/blog/7691/webassembly/` (accessed 2024-03-09)

[3] 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)

4 Approach

A Wasm runtime has to ensure that all accessed addresses lie within the dynamically changing bounds of the module's Wasm memory. These addresses can be constructed at run-time based on outside parameters, limiting the possibilities for checks during compile-time. Therefore, without our proposed optimizations, a Wasm runtime is required to emit software bounds checks to validate memory instructions.

A software bounds check is implemented as a comparison between the address a' and the current memory size. In a multi-threaded environment, other threads may concurrently execute `memory.grow` instructions, changing the current memory size. Therefore, its value always has to be updated prior to the comparison, leading to an additional memory access. After the comparison, a conditional branch instruction jumps to code that raises a Wasm trap, if the address is larger or equal to the memory size. If the branch is not taken and thus the address lies inside of the allocated Wasm memory, the code that implements the memory access is executed. This control flow layout is beneficial for speculative execution because branch predictors of modern CPUs predict conditional branches to be not taken or use historical data for branching behavior prediction [25]. However, this still leads to a significant number of additional compare and branch instructions, imposing a high run-time overhead.

Out-of-bounds access detection using software bounds checks is a simple and safe approach that works for 32-bit and 64-bit addresses, but it is purely implemented through software, which affects run-time performance. In the Wasm whitepaper, Haas et al. [21] already propose to use memory protection hardware features to perform near-zero overhead memory access validation. Their proposed approach, later explained as *Guard Pages*, is only possible for 32-bit Wasm execution. We propose and implement three distinctive novel bounds checking methods for 64-bit Wasm, which utilize hardware features of modern CPUs and the signal handling mechanism of the Linux kernel in order to improve execution time.

4.1 Prerequisites

The solutions proposed in this thesis can be implemented for multiple platforms and architectures. For our proof-of-concept implementations for the Umbra DBMS, we focused on a system that runs a modern Linux kernel (version ≥ 4.9) on an x86_64 CPU in 64-bit mode, with user-space Memory Protection Key support, capable of executing Umbra. We did not implement our proposed bounds checking methods for the virtual machine code generation back-end, as its behavior is too different from the other back-ends, and it is superseded, for x86_64 systems, by the DirectEmit back-end [39]. Umbra C++ code, together with our implementation, is either compiled using the gcc compiler version 13 [18] or clang 17 [62].

4.2 Signal-Based WebAssembly Trap Detection

The proposed bounds check elision mechanisms all use memory protection features of modern POSIX-compliant CPUs. An out-of-bounds access leads to an invalid memory reference detected by the CPU core that executes the offending instruction. Upon detection, this hardware unit raises a page fault exception [24], which is handled by the Linux kernel [37]. The kernel determines the reason for the exception and sends a `SIGSEGV` signal[1] to the currently executed thread in case of an invalid memory reference. We install a Linux signal handler [33] that instructs the kernel to execute a user space function when encountering this `SIGSEGV` signal.

4.2.1 `SIGSEGV` Source Validation

As described in Section 2.2.2, the Umbra Wasm runtime allows Wasm code to invoke Proxy methods. These methods are implemented in the Umbra source code in C++ and can therefore perform arbitrary tasks. C++ does not enforce memory safety. Therefore, erroneous Umbra Proxy method code can also contain invalid memory references that lead to a `SIGSEGV` signal. To simplify debugging of faulty Proxy methods, we want to separate these actual coding errors from the `SIGSEGV` signals raised through Wasm out-of-bounds memory accesses, whose handling is defined. This is only relevant for debug builds of the Wasm runtime because release builds are assumed not to contain erroneous Umbra source code and do not need to perform source validation for debugging.

The Linux `sigaction` system call allows us to register a signal handling function that, in addition to the signal number, also receives an instance of the `siginfo_t` and `ucontext_t` structs [33]. We use the latter to access the register file at the point in the execution when the signal was raised. Using the value stored in the `RIP` register, we can backtrace the origin of the invalid memory reference.

To acquire a current mapping between virtual addresses and program sections, we access the Linux kernel's `proc` pseudo-filesystem, which gives us access to kernel data structures. The content of the file at `/proc/self/maps` is a string representation of the currently mapped memory regions of the reading process [35]. An example content is shown in Listing A.1.

Using the extracted start and end addresses of the mappings, we can infer the pathname of the mapped code segment where the invalid memory reference originated. In order to determine whether the source segment contains the code generated for the Wasm bytecode, we use the following heuristic: Generated code is either only held in memory, not backed by a source file, and therefore the pathname is empty, or it is written to a file for debugging purposes, in which case the file name starts with the substring "dump".

The `SIGSEGV` signals that do not originate from a code segment fulfilling the heuristic above are *re-thrown*, meaning the current signal handling function for Wasm-generated signals is deregistered, and the `SIGSEGV` signal is manually raised again using the `raise` system call.

[1] For certain invalid memory references, the Linux kernel may instead send a `SIGBUS` signal [37]. Therefore, in our implementation, we treat an occurring `SIGBUS` signal identical to `SIGSEGV` signals. Following references of the `SIGSEGV` signal type imply that `SIGBUS` signals are handled equally.

4.2.2 Controlled Stack Unwinding

Using the signal handler function, we need to terminate the Wasm runtime using a Wasm trap in a controlled manner without interfering with other parts of the host process. This is very important because the runtime is executed as part of the Umbra DBMS' query engine, which must not be interrupted. Additionally, Umbra's Proxy methods allow arbitrary C++ code execution, the dangers of which are further explained in Section 2.4.2. Therefore, proper stack unwinding and `SIGSEGV` handling is crucial for a safe and efficient Wasm runtime integration.

Signal handling for Linux processes is a complex endeavor because even though signal handling functions are defined and implemented similarly to *normal* functions, their behavior is different concerning safety, returning, and exception handling [38]. Therefore, it is often advised to exit the signal handling function's context as soon as possible, for example by using a long jump to a recovery checkpoint [65]. The long jump loads a previously stored execution context, transferring execution from the signal handler to normal program code [34]. However, this resets the stack without unwinding it, again leading to the problems described in Section 2.4.2.

When a Wasm trap is raised, the call stack may contain a mixture of Wasm code and Umbra C++ code from Proxy methods. To comply with the requirements set by the C++ standard for unwinding through C++ code [63], we use the existing C++ runtime's tool for stack unwinding: We `throw` a C++ `WasmError` exception object from the signal handler. The process of C++ exception handling and the safety of throwing an exception from a signal handler is explained in detail in Section 2.5.

The `MightThrowMarker` Umbra IR Instruction

Throwing a C++ exception from the signal handler allows the runtime to unwind the stack until a suitable `catch` clause is encountered if proper unwind information was generated for the signal-causing function. The Umbra code generation back-ends are instructed not to generate unwind information for Umbra IR functions annotated with the behavior flag `Noexcept`. For the LLVM back-end, this is achieved through the LLVM function attribute `nounwind`. Therefore, we need to ensure that no function generated from the Wasm bytecode that could result in an invalid memory reference is annotated as `Noexcept`. For this purpose, we introduce a new pseudo instruction for the Umbra IR called `MightThrowMarker`. Every Wasm memory instruction emits this marker along the actual load or store instruction. During compilation, every function that contains a `MightThrowMarker` instruction cannot be marked `Noexcept`. Therefore, the active code generation back-end is instructed to generate unwind information. Afterward, the `MightThrowMarker` acts as a noop and is not further processed.

Our Wasm runtime finally uses a C++ try-block surrounding the call of the entry function of the code generated from the Wasm bytecode. The catch clause of this block catches any thrown `WasmError` and allows further handling of the Wasm trap outside of the Wasm execution.

4.3 Guard Pages

As already proposed by Haas et al. [21], a Wasm runtime on a 64-bit host system can use virtual memory protection mechanisms to eliminate bounds checks for 32-bit Wasm execution.

$$2^{32} - 1\,(\text{address}) + 2^{32} - 1\,(\text{offset}) + \text{sizeof(largest\_type)} - 1 = 2^{33} + 13 = 8\,\text{GiB} + 13\,\text{B} \tag{4.1}$$

Equation (4.1) shows the calculation of the maximum size of the 32-bit Wasm address space. It is comprised of the maximum 32-bit address, combined with the maximum 32-bit immediate offset and the size of the largest standardized type, the 128-bit vector type.

On a 64-bit host system, the runtime can therefore always map the full 8 GiB + 13 B as the Wasm memory with only the lower, allocated part of the range marked accessible as shown in Figure 4.1. It suffices to add Guard Pages behind the Wasm linear memory because Wasm addresses are interpreted as 64-bit unsigned integer offsets to a Wasm memory base pointer by the Wasm runtime, pointing to Wasm address zero. This means that Wasm code can never access host memory addresses smaller than this base pointer's address. Any memory access to an address in a region marked as inaccessible triggers a processor exception, which is caught and processed into a Wasm trap by the runtime.

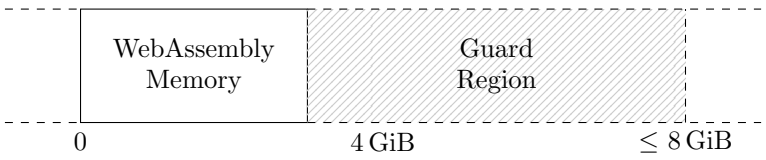


Figure 4.1: *Guard page layout for < 4 GiB of accessible memory.*

The Umbra DBMS, which hosts our Wasm runtime, is only available for 64-bit Linux operating systems. This allows us to employ this near-zero overhead bounds check elision technique. During the compilation of the Wasm code to machine code, we analyze all memory instructions and store the highest encoded offset β. During loading of the compiled Wasm module, we allocate $(4\,\text{GiB} + \beta) \,/\, 2^{12}\,\text{B} + 1$ virtual pages for the Wasm memory, with 2^{12} B being the default x86_64 page size. We use the `mmap` system call with the `MAP_PRIVATE`, `MAP_NORESERVE` and `MAP_ANONYMOUS` flags because the mapping is neither shared nor backed by a file. We set the memory protection to `PROT_NONE`, which instructs the CPU to generate a hardware exception for any attempted read or write operations on the mapped pages.[24, 31]

The `memory.grow` instruction takes the number of additional requested Wasm pages as a parameter. Every Wasm page has a size of 64 KiB, or exactly 16 x86_64 pages. The runtime therefore uses the `mprotect` system call to change the memory protection of all pages in the range

[mem_base_addr+old_mem_size; mem_base_addr+old_mem_size+pages_requested·64 KiB]

to `PROT_READ` and `PROT_WRITE` in order to enable read and write accesses for following Wasm memory operations to the additional requested pages. This converts allocated pages from the *Guard Region* to pages belonging to the accessible Wasm memory. If the maximum memory size is reached, an error code (-1) is returned instead. This maximum size is either defined by the limit encoded in the memory type of the Wasm memory or, if the limit does not specify a maximum memory size, the specification defines a maximum Wasm memory size of 4 GiB.

Valid memory accesses then always operate on unprotected memory. Invalid, out-of-bounds accesses appear on pages that belong to the Wasm runtime but are protected. The x86_64 exceptions triggered by these accesses lead to the Linux `SIGSEGV` signal, which is handled as described in Section 4.2.2.

4.4 Guard64

Using Guard Pages and hardware memory protection features leads to a near-zero overhead out-of-bounds access detection and is the go-to solution for 32-bit Wasm execution. However, with more complex and resource-intensive computational workloads, a maximum of 4 GiB heap memory may not be sufficient for all Wasm use cases. The upcoming memory64 feature proposal therefore extends existing memory operations to allow 64-bit wide memory addresses [60]. This allows Wasm bytecode to address every existing byte in the available virtual address space.[2] Guard Pages are designed to shield mapped pages of the Umbra process against invalid accesses from the Wasm runtime, by moving them out of the Wasm-addressable memory space. This is not possible for 64-bit Wasm addresses, because they allow addressing of the host system's whole virtual address space.

A typical Wasm program does not require enormous amounts of memory, not even code using the memory64 proposal's wider addresses. Therefore, we propose the *Guard64* hybrid bounds checking method for 64-bit Wasm addresses that combines the hardware features of Guard Pages with the 64-bit compatibility of software bounds checks.

We choose a maximum Wasm memory size prior to code execution which is fully mapped, but not reserved. For our implementation, we stick to the size of the 32-bit Wasm address space and, therefore, allocate 4 GiB of memory for a maximum of 4 GiB of available Wasm memory. This memory region serves as our Guard Region, as explained in Section 4.3. A 64-bit Wasm address might point to memory outside this Guard Region if the address is larger than $2^{32} - 1$. Such a larger address must have one of the upper 32 of the 64 available Bits set.

In order to detect such larger addresses, we analyzed two approaches: The first approach, shown in Figure 4.2, copies the address and shifts its bits 32 places to the right, with 32 being the logarithmic size of our allocated memory region. This is done before the actual memory access. The address is inside the bounds of this allocated memory region if the result of the shift is equal to zero. Subsequently, we emit a conditional branch instruction that jumps to code that raises a Wasm trap if the result is not zero. This

[2] Even more than every existing byte. A modern Intel x86_64 CPU typically uses 48-bit wide virtual memory addresses. Therefore, a Wasm memory64 address can address more bytes than our virtual address space contains.

jump is predicted as not taken by modern branch predictors [25], allowing for efficient speculative execution without unnecessary pipeline flushes.

```
(func $wasm_entry (result i32)
    i64.const 0x42
    i32.load)
```

```
mov rax, 0x42
; address now stored in rax
shr rbx, 32
jnz .wasm_trap
; memory access
mov rbx, [.mem_ptr]
mov edx, [rax + rbx]
; result stored in edx
```

Figure 4.2: ***Shift Version:*** *x86_64 assembly pseudo-code (right side) modelling a Guard64 bounds checked memory access using a shift instruction to check the accessed address, generated for the Wasm code on the left side.*

The second approach, shown in Figure 4.3, detects too large addresses by applying the `guard64_mask`, which is stored in memory, to the accessed address through an x86_64 `TEST` instruction, prior to the memory access. The mask's bits are one for all bits that can never be set for any valid Wasm address. For our 4 GiB Wasm memory, these are all bits with an index higher than 31. The address is inside the bounds of the allocated memory region if the bitwise `AND` between the mask and the address, performed by the `TEST`, is equal to zero. The subsequent instructions are equal to the shift-based detection method.

```
(func $wasm_entry (result i32)
    i64.const 0x42
    i32.load)
```

```
mov rax, 0x42
; address now stored in rax
; .guard64_mask:
; 0xFF FF FF FF 00 00 00 00
test rax, [.guard64_mask]
jnz .wasm_trap
; memory access
mov rbx, [.mem_ptr]
mov edx, [rax + rbx]
; result stored in edx
```

Figure 4.3: ***Bitwise AND Version:*** *x86_64 assembly pseudo-code (right side) modeling a Guard64 bounds checked memory access using a test instruction to perform a bitwise AND to check the accessed address, generated for the Wasm code on the left side.*

We implemented both approaches and analyzed their run-time performance, the results are shown in Figure A.6. The `TEST`-based Guard64 implementation outperforms the `shr`-based implementation, as well as plain software bounds checks in all benchmarks for the LLVM and DirectEmit back-end. Therefore, we only consider the `TEST`-based implementation for all further evaluations.

The emitted test and branch sequences only ensure that all memory accesses lie inside of our allocated memory region. They do not guarantee that the addresses actually refer to valid Wasm memory. However, invalid accesses to memory that is not yet unlocked through the `memory.grow` Wasm instruction hit our Guard Pages, violate memory protection settings, generate a hardware exception, and are handled by the mechanism explained in Section 4.3. This mechanism does not incur a run-time overhead, making the test and

branch instructions the only overhead introduced for a Guard64 bounds check.

4.5 Memory Protection Keys

While the Guard64 method already provides hardware-assisted bounds checking for 64-bit Wasm addresses, it still requires additional instructions for all emitted memory accesses. We propose using PKeys, an extension to the x86_64 ISA, to eliminate any overhead of Wasm memory accesses.

Guard Pages utilize POSIX memory protection and hardware exceptions to detect out-of-bounds accesses. The hardware exception is triggered by the CPU for, among other reasons, illegal memory references to protected pages and for memory references to virtual addresses that are not mapped. This means we can also detect out-of-bounds accesses by temporarily protecting all mapped pages that should not be accessible by Wasm code.

This approach poses two significant challenges: First, page protection settings apply to all users of the virtual address space. Other parts of the DBMS process could not access their mapped pages during Wasm execution. This would essentially force query processing in Umbra to run single-threaded as soon as a Wasm UDO is encountered. Second, changing the memory protection settings of a page involves changing the process page table. This is a privileged operation that requires interaction with the kernel to apply the changes. It would have to be repeated for every mapped range of pages, resulting in a 3- to 4-digit number of kernel interactions for every switch to Wasm execution. Both limitations lead to unused system resources and a significant reduction in execution speed, rendering Wasm UDOs unusable.

To overcome these challenges, we propose using PKeys. They provide a mechanism to alter page-based memory protections without changes to the page table. The feature is available for modern x86_64 processors that support the 32-bit wide `PKRU` register for user-mode page PKeys. A PKey is a 4-bit value between 0 and 15 (inclusive). Associating a page with a PKey from user space is possible using the `pkey_mprotect` system call available since Linux 4.9 [36, 32]. It instructs the kernel to write the supplied PKey into the page table entries of the affected pages, making the PKey available for the Memory Management Unit (MMU). By default, all page table entries are protected using PKey 0. After the initial setup, access to a page protected by a PKey is controlled by the corresponding bits in the `PKRU` register. The bit at position $2 \cdot \text{pkey\_number}$ disables any data accesses if set. Write accesses are prohibited if the bit at position $2 \cdot \text{pkey\_number} + 1$ is set. These restrictions apply to all pages protected by the corresponding PKey. Access to the `PKRU` register is granted through the unprivileged `RDPKRU` and `WRPKRU` instructions.[24, 50]

PKeys allow user-space code to change memory protection settings of pages by writing to a register, an operation several magnitudes faster than an `mprotect` system call.[3] Additionally, the `PKRU` register is a CPU register, meaning its content can be different for different threads of the same process [24]. They enable us to quickly disable any access to all non-Wasm memory pages, only for the thread(s) executing a Wasm runtime, without

[3] A synthetic benchmark executed on our test system (described in Section 4.1) 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].

interfering with threads executing other parts of the DBMS.

4.5.1 Locking and Unlocking Umbra Pages

The actual locking and unlocking of the Umbra pages is done by writing to the `PKRU` register using the `WRPKRU` instruction. Therefore, we implemented two custom operations for the Umbra IR called `UnlockUmbraPkey` and `LockUmbraPkey`. The code generation back-ends either use inline assembly (LLVM and C back-ends) or directly emit the `WRPKRU` instruction (DirectEmit back-end). The instruction takes the new value for the `PKRU` register from the `eax` register and requires the `ecx` and `edx` registers to be set to 0 [24].

PKeys are allocated using the `pkey_alloc` system call [36]. The Linux kernel always chooses the lowest unused PKey number for the next allocation.[4] PKey 0 is always allocated by default. We assume that our Wasm runtime is the only user of PKeys in our process, running multiple Wasm runtimes in parallel requires additional synchronization. However, we cannot assume that the first allocated PKey is PKey 1 because the Linux kernel automatically allocates a PKey for the first created execute-only mapping.[5] Therefore, to be able to hard-code the PKey number, we call `pkey_alloc` until we receive PKey number 2.

We hard-code the `PKRU` register values for "disable all access to pages assigned PKey number 2" (0b110000) and "allow all access to pages assigned PKey number 2" (0b000000) into our emitted code. PKey number 0 is never locked, because it is the default PKey, set for all our accessible pages like the ones of the stack segment. All other PKeys are not in use and can be ignored.

4.5.2 Optimization of (Un-)Lock Instruction Sequences

```
define void @wasmFunction_1(...) {
    // ...
    unlockumbrapkey
    %1 = call ptr umbra::wasm::WasmModuleInstance::getFuncrefTableEntry(...)
    lockumbrapkey // <- not required

    // some non-memory Wasm instructions in between...

    unlockumbrapkey // <- not required
    %2 = load int8* %1
    lockumbrapkey
    call indirect void %2
    // ...
}
```

Figure 4.4: *Umbra IR pseudo-code demonstrating a common `LockUmbraPkey`-`UnlockUmbraPkey`-sequence that occurs when performing an indirect function call.*

[4] This is guaranteed by the implementation of our Linux kernel: `https://elixir.bootlin.com/linux/v6.5/source/arch/x86/include/asm/pkeys.h#L100` (accessed 2024-01-29)

[5] `https://elixir.bootlin.com/linux/v6.5/source/arch/x86/mm/pkeys.c#L14` (accessed 2024-02-10)

We emit `UnlockUmbraPkey` and `LockUmbraPkey` Umbra IR instructions around operations that require access to protected memory regions. If multiple such operations are executed in series without Wasm memory accesses in between, unnecessary lock-unlock sequences are emitted, leading to more `WRPKRU` instructions and an execution speed penalty. Therefore, we implemented a custom optimization pass that is executed before the Umbra IR code is handed to the code generation back-end. The pass identifies unlock instructions that follow a lock instruction without memory accesses in between and removes both from the IR code.

Figure 4.4 shows a practical example of where this sequence of lock and unlock instructions might occur in generated code. The `load` operation in line 10 loads the function pointer from the reference acquired from the Wasm function table. It is not a Wasm load instruction and is allowed to access the non-Wasm memory code segment.

4.5.3 Standard Conformance

The Wasm specification guarantees the user that any memory accesses outside the bounds of the allocated Wasm memory region result in a trap. For our bounds checking method, this would mean that every page that is not a Wasm memory page is not accessible using a Wasm memory operation. However, this poses a problem: Our execution stack is used like any other program stack, to store temporary values in stack frames. This memory region is not part of the Wasm memory. It is theoretically possible to disable access to the stack for every memory operation. However, this leads to two `WRPKRU` instructions per memory reference, a much higher run-time cost than just emitting a software bounds check instruction.[6] Therefore, we decided to trade standard conformance and safety for execution speed.

As a result, the pages belonging to the stack are not locked during Wasm execution. Also, the *vsyscall* segment is not locked, because it is the only segment whose memory protection settings we are not allowed to change [32]. Additionally, we need to have some run-time variables that are accessible for the Wasm runtime. To reduce the amount of `WRPKRU` instructions executed, we add an additional read-only page that contains the Wasm memory pointer and current memory size in addition to pointers to the Wasm runtime instance and the WASI runtime instance, which are required by Proxy methods. Right after this read-only page, a section for all global variables is allocated. This section is not protected. Therefore, Wasm code can access and change globals without any page unlocking overhead. All other run-time variables are stored on pages locked for the Wasm code. Proxy methods can only be defined in the Umbra C++ code. Therefore, they are assumed to perform valid memory accesses only. All Umbra pages are unlocked prior to calling a Proxy method and are locked again upon returning from the Proxy. These Proxy methods can therefore also change the content of the read-only page by changing its memory protection settings, which is required, for example, for the `memory.grow` Wasm instruction.

The LLVM and C code generation back-ends define a read-only segment that is loaded together with the generated code and contains, among others, large constants. These

[6] For a measurement of the overhead of `WRPKRU` instructions, refer to Figure 5.6.

segments need to be available during Wasm code execution, which is why they need to be protected with PKey 0. However, while locking all existing Umbra pages during the startup of the Wasm runtime, we cannot identify these segments without additional information about their placement. For the LLVM back-end, we register a callback function to the LLVM C libraries `ObjectLayer` using the `setNotifyLoaded` method. We then use the supplied parameters to identify the loaded segments together with their load addresses and store information about the read-only segments. The C back-end always uses an external C compiler to generate an Executable and Linking Format (ELF) shared library from the generated C code that is then dynamically loaded. We read the underlying ELF file to extract information about the read-only section load offsets. The actual load addresses are calculated using the linked base address acquired using the `dlinfo` function on the dynamically loaded library together with the extracted offsets. The DirectEmit back-end does not place constants into a separate section and we, therefore, do not need to extract any additional information for this code generation back-end.

4.5.4 Restartable Sequences

When locking all Umbra pages that should not be accessible from external Wasm code, we also set the heap segment as inaccessible. This is not problematic for our code, however, our utilized C standard library, glibc [17], expects the heap to be accessible.

Restartable Sequences [12] are implemented by the Linux kernel since version 4.18. They provide an efficient mechanism to manage per-cpu data through a kernel-registered user space object `struct rseq`, which may contain the Restartable Sequence. This sequence is informed about interruptions of the current thread. Such an interruption could, for example, be the preemption for scheduling or a CPU migration. Any running updates to the thread-local object should then be discarded and the update operation restarted. glibc assists programmers by registering such a user space object once during initialization which can then be used for registering custom Restartable Sequences. This `struct rseq` object is placed on the heap.

```
syscall(
    __NR_rseq,
    reinterpret_cast<char*>(__builtin_thread_pointer()) + __rseq_offset,
    __rseq_size,
    RSEQ_FLAG_UNREGISTER,
    RSEQ_SIG
);
```

Listing 4.1: *System call to deregister Restartable Sequence structure of glibc to handle signals without an accessible heap segment.*

When our executed Wasm code performs an out-of-bounds memory access, a `SIGSEGV` signal is supposed to be delivered and our installed signal handler should be called. However, with a registered `struct rseq` object, the kernel first accesses said object to check for an existing Restartable Sequence. With our PKey locked heap, the kernel fails to access the object and sends a `SI_KERNEL SIGSEGV` signal which we cannot recover from. The `SI_KERNEL SIGSEGV` is raised by the kernel, meaning we do not have the required

unwind information to safely throw a C++ exception.

We circumvent this problem by manually deregistering the `struct rseq` object using the system call shown in Listing 4.1.

4.5.5 Preventing Access to Newly Mapped Segments

During Wasm execution, the Umbra process may map new pages. These pages receive the PKey 0 by default, which leaves them unlocked for our Wasm runtime. To prevent this, we implement a wrapper for the `mmap` and `brk` functions provided by the C standard library. We compile the wrapper, which defines functions with the same signatures, to a shared library. We then instruct the run-time loader to load the wrapper library before any other shared library, using the `LD_PRELOAD` environment variable. This allows us to essentially overwrite the function symbols from our subsequently loaded C standard library.

When the `mmap` function is called, our wrapper first calls the original `mmap` function from the glibc with all provided arguments, except the protection flags, which are always set to `PROT_NONE`. After a successful mapping, the requested address range is protected with the requested protection flags and PKey 2 using the `pkey_mprotect` system call. The previously mapped address is later returned. This provides a thread-safe mechanism to never expose newly mapped segments to accesses from our executed Wasm code without changing the ABI for the caller of the `mmap` function. The implementation is shown in Listing A.3.

`sbrk` and `brk` were specified by the POSIX specification to dynamically resize the heap segment. However, both are deprecated and removed since POSIX.1-2001 [30]. Linux still offers both functions, but the `sbrk` function is implemented by glibc using the `brk` system call with internal bookkeeping [30]. We still need to wrap the `sbrk` function, because this internal `brk` call might not be intercepted by our `brk` wrapper function. Usage of `sbrk` or `brk` is discouraged [30], therefore, we only supply a legacy fallback solution for these functions. We apply the correct PKey to the newly allocated region after the corresponding system call, which does not guarantee that the newly mapped segments are never exposed to memory accesses from the Wasm runtime.

An alternative implementation that does not require patching of Linux system calls could use PKey 0 for all non-Wasm pages and PKey 2 for all Wasm pages. During Wasm code execution, we lock all access to pages protected with PKey 0. Locking PKey 0 causes undefined behavior because Linux only defines changing of access rights to PKeys which were acquired through the `pkey_alloc` function, which only returns PKeys larger than 0 [36]. However, it is allowed by the architecture [24] and works for our utilized Linux kernel. Using this trick, we do not need to manually apply the appropriate PKey to newly mapped memory, because the default PKey is PKey 0. We did not implement this strategy because it relies on undefined behavior and might break with future Linux kernel releases.

4.6 Page Granular Shadow Memory

Memory Protection Keys (PKeys) are a new hardware primitive which is not available on all systems running an Umbra DBMS [50]. We therefore came up with a new approach to Wasm bounds checking of 64-bit addresses that uses the same hardware memory protection features required for 32-bit Wasm Guard Pages. The full 64-bit address space is too large to map and protect from invalid accesses. However, a scaled-down mirror image called a *Shadow Memory* can be stored. This concept is similar to the shadow memory employed by memory analysis tools like Google's AddressSanitizer [55]. Every Wasm memory operation first attempts to access a shadow page corresponding to the accessed address. This shadow page is either accessible and the memory operation continues normally, or is protected, and a hardware exception is triggered, leading to a Wasm trap and the subsequent stack unwinding explained in Section 4.2.2.

By downscaling the accessed addresses, we map multiple Wasm memory references to the same x86_64 virtual address. Essentially, we can combine one or more Wasm 64 KiB pages to access a single x86_64 page. Memory protection settings applied to a single x86_64 shadow page therefore apply to one or more larger Wasm pages.

The scaling factor can be chosen freely; however, the smaller the factor, the more virtual address space is lost to shadow memory. A one-to-one mapping of Wasm pages to x86_64 shadow pages means shrinking the 64-bit[7] Wasm address space by a factor of 16. This requires a shadow memory region with a size of 2^{60} B.

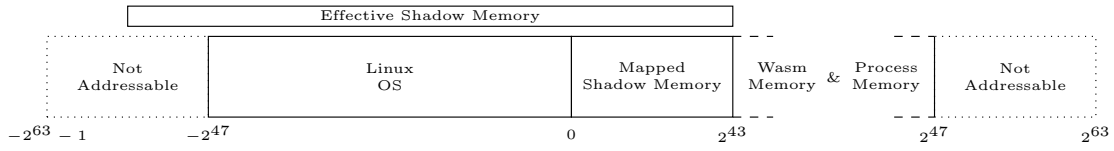


Figure 4.5: *Shadow page layout (not to scale).*

As shown by Serebryany et al. [55], most shadow memory approaches either use a direct scale and offset or table lookups for the address mapping. We optimized this bounds checking technique for speed and, therefore, perform as few lookups and arithmetic operations per memory reference as possible. We utilize a constant scaling factor that is hard-coded into generated code as a logical shift right operation on the accessed Wasm address. These addresses are offsets to the memory pointer, stored in the Wasm runtime. We eliminate the need for an additional shadow memory pointer by placing the shadow pages directly in front of the existing Wasm memory pointer, in reverse order, as shown in Figure 4.5.

Therefore, the address calculation only requires the highest accessed address *last_addr* (Equation (4.3)) to be scaled down by the *shift* factor, subtracted from the *memory_ptr* and decremented by 1 (Equation (4.4)). This *last_addr* is the highest address referenced

[7] An additional 64-bit integer offset is added to the address, making it effectively 65-bit wide. The memory64 proposal [60] does not specify how addresses larger than $2^{64}-1$ should be handled. Therefore, we assume a wrapping mechanic by using unsigned overflow for the address calculation. We may need to emit an additional check on the offset in case future versions of the proposal specify a different behavior.

by a memory operation on the concerning type. The *shift* is statically evaluated and hard-coded into the generated machine code. It is derived from the amount of Wasm pages (*chunk_size*) mapped to one x86_64 *host_page*. We thereby map Wasm addresses 0 to $2^{16} - 1$ to the shadow memory page 0. However, this would interfere with Wasm page 0 if simply subtracted from the memory pointer. We therefore require the additional decrement by 1 in Equation (4.4). This instruction can be replaced with a bitwise negation as shown in Equation (4.5), an optimization that Umbra's compiled code generation back-ends would perform automatically.

$$shift = \log_2 \left(chunk\_size \cdot \frac{wasm\_page\_size}{host\_page\_size} \right) \tag{4.2}$$

$$last\_addr = wasm\_addr + \text{sizeof}(loaded\_type) - 1 \tag{4.3}$$

$$shadow\_address = memory\_ptr - (last\_addr \gg shift) - 1 \tag{4.4}$$

$$shadow\_address = memory\_ptr + \neg(last\_addr \gg shift) \tag{4.5}$$

4.6.1 Shadow Memory Test Code

Each memory operation only needs to validate whether the shadow memory address corresponding to the supplied address is accessible. This requires an operation that only loads a memory reference without interfering with program execution in any other way. For this purpose, we introduce a new instruction to the Umbra IR called *ShadowMemoryTest*. Its operand is the *shadow_address*, which is calculated beforehand through code similar to Equation (4.5).

We evaluated using either an x86_64 `TEST`, `MOV` or `MOVZX` instruction to perform the memory reference test. The `TEST` instruction performs a bitwise AND operation on its two operands, only changing the flag register based on the result [24]. Therefore, it does not interfere with the execution by littering a result register and has a low run-time overhead on success. The `MOV` and `MOVZX` instructions load a value from memory, which would cause an access to our shadow memory page. On a successful access, they read from the uninitialized shadow memory and store the result in a destination register, overwriting its content [24]. However, we require an additional register for calculating our corresponding shadow memory address anyway, which is the register `RDI` in Figure 4.6. Therefore, we do not pollute an additional register if we load our value into this already polluted register.

According to third-party benchmarks [51], some platforms require slightly less μops for executing a `MOV` or `MOVZX` than for a `TEST` instruction. We benchmarked all three possible

```
; address a' in rdx
shr rdi, 4
mov rdx, QWORD PTR [$memory_ptr]
not rdi
test BYTE PTR [rdx + rdi], 0
; ... perform memory access ...
```

```
; address a' in rdx
shr rdi, 4
mov rdx, QWORD PTR [$memory_ptr]
not rdi
mov[zx] edi, BYTE PTR [rdx + rdi]
; ... perform memory access ...
```

Figure 4.6: *Comparison of `TEST`- and `MOV`-based shadow memory access code.*

implementations on our test system and present the results in Appendix A.3. We found that for the DirectEmit back-end, a `TEST` instruction leads to the best performance. For the LLVM back-end, the `MOVZX` instruction, emitted through an LLVM IR load instruction, performs the best in most benchmarks. Therefore, we implement the shadow memory test using a `TEST` for the direct emit back-end and an LLVM IR load for the LLVM back-end. The C back-end, whose performance is not critical, also performs a load operation.

Extending accessible Wasm memory through the `memory.grow` instruction now requires the unprotection of previously inaccessible shadow pages to allow access to the new Wasm pages. Therefore, the implementation always first updates the stored memory size variable to reflect the change in Wasm memory. It then unprotects all shadow memory pages whose mapped Wasm page is now supposed to be accessible by Wasm code. A future implementation could also use the `remap` system call to extend the mapped shadow- and Wasm memory to grow larger than the memory region allocated at initialization.

4.6.2 Shadow Memory Placement

An x86_64 CPU typically uses 4-level paging for virtual address management with a virtual address length of 48-bit [24].[8] The negative portion of this virtual address space[9], i.e., the part where the most significant address bit is 1, is occupied by our Linux kernel [27]. This only leaves 2^{47} B of mappable virtual memory for the Umbra DBMS, the Wasm runtime, the Wasm memory, and the shadow memory.

Unprivileged memory accesses to addresses larger than the maximum virtual user-space memory address $(2^{47} - 1)$ always lead to hardware exceptions and a subsequent `SIGSEGV` signal to our process [24]. We take advantage of this behavior to implement direct mapped shadow memory.

We map our shadow memory region to occupy the lowest end of the virtual address space to allow a shadow memory configuration where every Wasm page is mapped to a dedicated x86_64 shadow page, using integer underflow. The required 2^{60} B of shadow memory are comprised of 2^{43} B of mapped shadow memory in the address range of $[0; 2^{43}[$ and the always unaddressable $2^{60} - 2^{43}$ B at the upper end of our virtual memory address range ($[2^{64} - 2^{60} + 2^{43}; 2^{64}[$), as shown in Figure 4.5. These values will have to be adjusted for future systems with wider virtual addresses. The Linux kernel features a system setting called `vm.mmap_min_addr` that controls the lowest mappable address [28].[10] Therefore, our mapped shadow memory range is $[$`vm.mmap_min_addr`$; 2^{43} -$ `vm.mmap_min_addr`$[$. This allows the Umbra process and the Wasm memory to utilize the remaining 2^{47} B $- 2^{43}$ B $=$ 120 TiB.

Placing this large shadow memory into the lower part of our virtual address space means that no other mappings may exist at addresses lower than 2^{43}. Moving, or remapping, existing memory mappings is complex, because all existing pointers to memory need to be subsequently updated. Therefore, our Wasm runtime maps the shadow memory at

[8] The Intel64 architecture supports virtual addresses of up to 64-bit. 5-level paging with 57-bit wide virtual addresses exists for some server CPU architectures like the AMD EPYC 9004 [2]. Our approach only requires slight adjustments for systems featuring virtual memory addresses wider than 48-bit.

[9] Intel64 48-bit virtual addresses are sign-extended to 64-bit signed "canonical addresses" [24].

[10] The default value is 0. It exists to prevent null pointer dereference and kernel bug exploits [28].

the beginning of the process execution to prevent colliding mappings. Following map requests to `mmap` are automatically placed after the shadow memory and Wasm memory by the kernel. However, this does not solve the problem of mappings that are created before the Umbra process is executed. The compiled Umbra DBMS code is linked to an executable ELF [64] binary file. The stored information about the segments that are loaded for program execution also encodes the virtual address at which the segment must be placed. We instruct the linker using the command line flag `--image-base` to set the base address of the generated ELF file greater than the combined size of the shadow memory and maximum size of the Wasm memory, which is 2^{46} in our implementation.

4.6.3 Relocations and the Code Model

With our loaded binary out of the way, we only face one last problem: relocations. Usually, x86_64 instructions do not allow the encoding of full 64-bit addresses into the instruction itself. Therefore, accesses to such large addresses are normally facilitated by first moving the offset to a 64-bit register and then using this register as an operand. The compiler does not know the virtual addresses of the final executable during compilation. It emits relocations, i.e. placeholders, for these addresses that are later replaced with the actual value by the linker. This means that the compiler does not know how large this address is going to be during compilation.

To assist the compiler, the AMD64 ABI [45] defines code models that restrict the size of address immediate operands. Normally, code is compiled using the "small" code model, which guarantees no address is wider than 32-bit. When using the large shadow memory region, we already have a base address that is at least 43-bit wide. However, instead of using absolute addresses, we can instruct the compiler to only use addresses relative to the current instruction pointer. This makes the code independent of its position in virtual memory and is achieved by using the "Small position independent code model" with the compiler flag `-fPIE` for the compilation of the Umbra executable.

4.6.4 Overcommitting

Memory overcommitting occurs when more virtual memory is requested from the kernel than there is physical memory available. Allowing overcommitting is useful because virtual memory often does not need to be allocated immediately after mapping, or sometimes at all. Our shadow pages are mapped for using the memory protection principles but will never hold any data and, therefore, will never occupy physical memory space. The kernel is informed about this requirement using the `MAP_NORESERVE` flag during the mapping of the shadow memory pages. The Linux kernel can be configured to handle overcommitting using one of three different overcommitting policies set via the `vm.overcommit_memory` system setting. Policy zero is the default, which allows *heuristic overcommitting*. Policy one always allows overcommitting. Both policies allow overcommitting for mappings that are marked as `MAP_NORESERVE`, while policy two always prevents overcommitting over a configurable amount of physical memory. This amount is either set as a ratio using the `vm.overcommit_ratio` setting or an absolute value using the `vm.overcommit_kbytes` setting. Therefore, we recommend to always set the `vm.overcommit_memory` settings to

either policy zero or policy one.[26, 31]

If a system does not allow overcommitting, an alternative solution is to create a file-backed mapping that maps the `/dev/zero` file to memory. No swap space will be reserved for the mapping, meaning this method does not require overcommitting and, therefore, works with `vm.overcommit_memory` set to policy two.

4.7 Compressed Shadow Memory

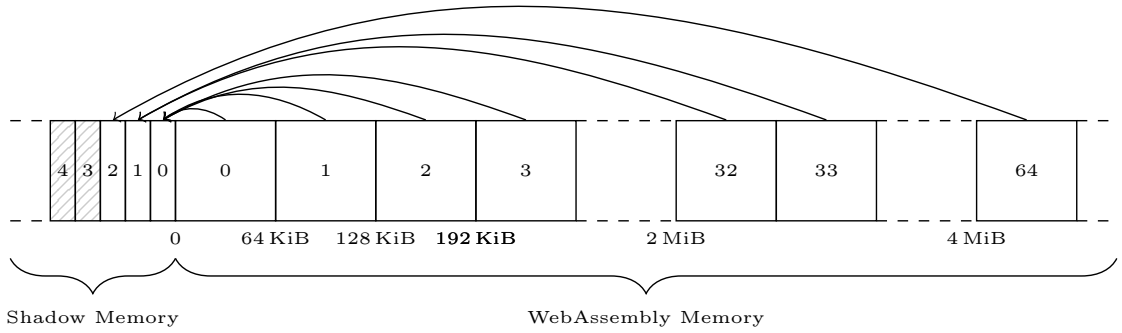


Figure 4.7: *Shadow page layout (not to scale).*

So far, our proposed approach uses a one-to-one mapping of Wasm 64 KiB to x86_64 4 KiB pages for simple and performant out-of-bounds detection, by trading virtual address space for execution speed. However, some applications may require more virtual address space for other parts of the process or, with multiple Wasm runtimes or the multi-memory extension to the Wasm standard [54], might require more than one Wasm memory per process. Therefore, we allow mappings of multiple Wasm pages to a single x86_64 page.

The Umbra Wasm runtime currently does not support the memory64 extension. We therefore implemented and tested this approach for 32-bit Wasm addresses with a scaling factor of 32. This leads to a reduction of 2 MiB of Wasm memory to one x86_64 4 KiB page, i.e., 32 Wasm pages access a single x86_64 page for bounds checks. The whole addressable 33-bit Wasm memory space (32-bit address length + 32-bit offset) then requires $2^{33}/2^{16}/32 + 1 = 4097$ x86_64 pages, an additional 16388 KiB of reserved, but unallocated space. The one additional shadow memory page is required, because Wasm types can span multiple addresses, meaning a memory access to a 32-bit integer at the address $2^{33} - 1$ also accesses the address $2^{33} + 2$. This leads to a shadow memory test to page 4097 of the shadow memory.

For 64-bit Wasm execution, we recommend a larger scaling factor to reduce virtual memory space consumption. Ben Titzer, one of the co-founders of Wasm, proposes to shift all Wasm 64-bit addresses bit 24 to the right to access their shadow page.[11] With our approach, this combines 64 GiB of Wasm memory to a single x86_64 shadow page,

[11] `https://github.com/WebAssembly/design/issues/1221#issuecomment-407742985` (accessed 2024-02-03)

which may lead to performance deficits for Wasm memories smaller than 64 GiB (see below). Therefore, we propose a smaller scaling factor: a shift of 18, grouping together 1 GiB of Wasm memory to a single x86_64 shadow page. This requires a shadow memory size of 2^{34} x86_64 pages or 64 TiB, of which 2^{16} B do not need to be mapped if the shadow memory is moved to the lowest part of the virtual address space.

4.7.1 Handling Sub-Shadow Memory Page Growth

With the shadow memory bounds checking approach, memory protection rules may be applied at a granularity of multiple Wasm pages, e.g. 32 pages in our implementation. This means that applications growing their memory at amounts that are not multiples of 32 Wasm pages pose a problem to our runtime. The Wasm specification clearly states that a `memory.grow` n operation may either fail, or allocate **exactly** n additional 64 KiB Wasm pages on the default Wasm memory. Also, an out-of-bounds memory access is guaranteed to cause a Wasm trap. Therefore, we cannot always allocate memory in multiples of 32 pages, regardless of the exact n, similar to implementations of `malloc`. A legal option is to have the `memory.grow` n operation always fail for every n where n is not a multiple of 32. The specification marks the instruction as non-deterministic, allowing it to fail in other cases than resource exhaustion. However, even though this would be conformant to the specification, it is unexpected by most programs and would render the Wasm runtime rather unusable.

To handle memory growth smaller than the shadow page chunk size, we propose two solutions: *Successful Segfaults* and *Micro Guard Pages*. For the final implementation that is later analyzed, we decided to only use Micro Guard Pages, as it both reduced code complexity and significantly increased execution speed.

Successful Segfaults

The Successful Segfaults implementation handles sub-shadow memory page growth by adding an additional check to the installed signal handler (Section 4.2.2). The `siginfo_t` struct, supplied as a parameter to the signal handling function, has a field `si_addr` containing the address of the failed memory reference. We use this address (*f_addr*) together with the memory start pointer from the Wasm runtime (*memory_ptr*) and the memory size (*memory_size*) to check if the failed memory reference actually referred to a valid Wasm memory page whose shadow page just was not unprotected yet or not. Therefore, we first calculate the shadow page offset and apply the shift from Equation (4.2) to the left, essentially reversing Equation (4.4).

$$offset = memory\_ptr - f\_addr - 1 \tag{4.6}$$

$$wasm\_offset \approx offset \ll shift \tag{4.7}$$

The original calculation is not bijective, meaning that this yields an approximated *wasm_offset*. However, this offset is guaranteed to be within the same Wasm page as the actually accessed offset. It is compared with the *memory_size* to check for an out-of-bounds access. If *wasm_offset* is larger than *memory_size*, then the normal routine described in

Section 4.2.2 is continued, and a Wasm trap is raised. Otherwise, the access was within the allocated Wasm memory, and execution can continue. This poses a problem: The `SIGSEGV` that led to the execution of the signal handler interrupted execution of the generated `TEST` instruction before the IP was increased. Returning from the signal handler would cause the CPU to execute the same `TEST` instruction again, leading to another `SIGSEGV`. To prevent this infinite loop, we manipulate the IP register to point to the instruction just after the generated `TEST` instruction to continue execution normally after returning from the signal handler. The IP is accessed through the `ucontext_t` struct pointer passed to the handler. If we know the exact length of our emitted `TEST` opcode in bytes, we know the value by which the IP needs to be advanced.

For this purpose, we always emit a deterministic, hard-coded opcode for the *ShadowMemoryTest* instruction when using the compressed shadow memory bounds checking method. With a constant size of the instruction code, we are able to set the IP to the address immediately after the `TEST` instruction without first disassembling the binary. For our current implementation, we choose the simplest version of the instruction, a `TEST [rax], 0`, which has the opcode `0xf6 0x00 0x00`. This can be further optimized in future versions by encoding part of the shadow memory address calculation in the opcode.

Therefore, we know that the length of the emitted shadow memory test instructions is 3 byte. After having incremented the IP from the `ucontext_t` struct by 3, we return from the signal handler and have the operating system return execution to our Wasm runtime.

Patching wasi-libc and LLVM

Most programs do not allocate memory in chunks of exactly 32 Wasm pages. When using the Successful Segfaults implementation, this causes every memory access to an address that is not within a fully allocated 2 MiB range of memory to raise a `SIGSEGV` and return through the signal handler, a process with a huge time penalty. This increases the execution time of the benchmarks (Section 5.2) so much that they did not finish in a reasonable amount of time and had to be terminated.

Fortunately, users do not typically write Wasm bytecode by hand. With Wasm being a compilation target, most user code is written in a high-level language like C and therefore uses the C standard library for heap memory management, i.e., it uses malloc. As described in Section 2.1.3, C code can be compiled to Wasm code that accesses the kernel through WASI APIs using the *wasi-sdk*. This Software Development Kit (SDK) supplies a *wasi-libc* that implements many parts of the C standard library to work without access to the system API through WASI APIs. It is very likely that all code executed in the Umbra Wasm runtime is compiled using the wasi-sdk, and therefore, all `memory.grow` instructions originate from the wasi-libc. It is, of course, not impossible for the user to emit custom Wasm instructions through the use of compiler builtins (e.g., `__builtin_wasm_memory_grow`) or inline assembly. However, this is not the typical use case and, therefore, not part of the performance optimization evaluation.

Even the wasi-libc actually only uses the `memory.grow` instruction for one single implementation: the `sbrk` system call wrapper. Even though the `sbrk` system call was removed in POSIX.1-2001 [30], it is still in use in the wasi-libc because `mmap` cannot be implemented using WASI. Besides possible usage by user code, the `sbrk` function is

only used by wasi-libc's `malloc` implementation, `dlmalloc` [41]. Instead of adjusting the `sbrk` function to only allocate memory in chunks of 32 Wasm pages, we opted for the less intrusive method of changing allocation parameters of `dlmalloc`. This leaves `sbrk` intact, giving users fine-grained control over memory allocation, while the default heap allocator `malloc` allocates Wasm memory in 32 Wasm pages at a time. This was achieved by defining the `DEFAULT_GRANULARITY` to be $32 \cdot 2^{16}$.

This change forces all generated Wasm `memory.grow` instructions to allocate in multiples of 32 Wasm pages. However, it does not influence the initial page count, set in the memory type definition of the Wasm module. If this is not a multiple of 32, it offsets all subsequent allocations and, therefore, reduces the efficiency of the runtime. This initial page count is defined by `wasm-ld`, the Wasm linker. It can be explicitly set using the `--initial-memory` flag, defaulting to the minimal required heap size, which was 2 Wasm pages for our test binaries. We altered the implementation of the `wasm-ld` linker to only accept explicit initial memory values that are multiples of 32 Wasm pages and also aligned the default value to this multiple. This involves changing the source code of the Wasm library of LLVM's linker `lld`.

After recompiling the patched wasi-libc and LLVM project, the patched wasi-sdk can be used in the same way as the stock SDK. Through these changes, all Wasm memory operations only access addresses within Wasm pages whose shadow page is fully available to the Wasm runtime and therefore unprotected. Without hand-crafted `memory.grow` instructions, all encountered `SIGSEGV` memory protection signals are the result of out-of-bounds accesses and lead to Wasm traps. This allocation in junks of 32 Wasm pages significantly increases the performance of this bounds checking method, enabling us to benchmark it.

Micro Guard Pages

The Successful Segfaults implementation under-unprotects shadow memory pages. Therefore, a valid Wasm memory access might fail and raise a `SIGSEGV` signal, only because its corresponding shadow memory page is not yet unprotected.

On the contrary, the Micro Guard Pages implementation over-unprotects shadow memory pages. Even if some Wasm pages covered by the last shadow memory page are not available to the Wasm code yet, we already unprotect that last shadow memory page. Without any further mitigations, some invalid Wasm memory accesses would not lead to a Wasm trap and could even access other pages of the Umbra process, which is forbidden by the Wasm specification. To prevent such invalid accesses, we always allocate the full Wasm page chunk that maps to the last shadow memory page. Therefore, in our implementation, we always expand the Wasm memory in multiples of 32 Wasm pages. All pages that should be accessible after the `memory.grow` instruction are marked accessible using the `mprotect` system call with the `PROT_READ` and `PROT_WRITE` flags. All other pages are marked inaccessible using the `PROT_NONE` flag. This concept is very similar to Guard Pages introduced in Section 4.3, however with a much smaller Guard Region. For our implementation, this Guard Region is always smaller than 2 MiB.

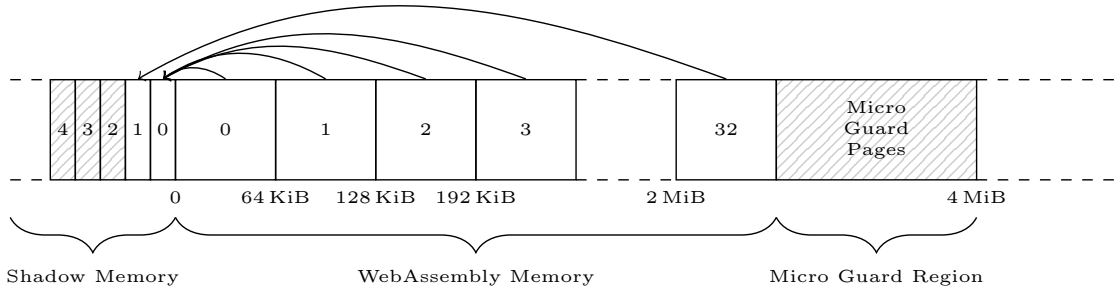


Figure 4.8: *Compressed Shadow Memory with Micro Guard Pages implementation (not to scale).*

The Micro Guard Pages concept is shown in Figure 4.8. Hatched blocks represent protected memory; unhatched blocks are accessible by Wasm code. The latest `memory.grow` operation extended the Wasm memory to a size of 33 Wasm pages. For our implementation, where 32 Wasm pages are combined to access one shadow memory page, this amounts to a fully covered shadow memory page (page 0) and a partially covered shadow page (page 1). With Micro Guard Pages, we unprotect shadow memory page 1 and create the shown Micro Guard Region, a protected memory region that covers all Wasm addresses whose corresponding shadow memory page 1 is accessible but which are not made available yet.

Subsequent `memory.grow` calls first decrease the size of the Micro Guard Region to make more Wasm pages accessible. When the next chunk of Wasm pages is reached, a new Micro Guard Region is mapped.

5 Evaluation

Our goal is to increase the execution speed of our Umbra Wasm runtime through faster bounds checking methods. Therefore, we put our approaches to the test to quantify potential speedups and discover further potential for optimization.

5.1 Validation

We utilize end-to-end tests for our implementations to validate correct Wasm runtime behavior. Creating a single Wasm file that tests multiple edge cases is not possible though, because a successful out-of-bounds access detection results in a Wasm trap and a shutdown of the runtime. The specification test functionality of the Umbra Wasm runtime System Under Test (SUT) therefore allows to test for fault behavior and trap instructions without a termination of the runtime using the `assert_trap` keyword. We use this mechanism together with the test files from the official WebAssembly specification tests suite[1] to assert that our changes successfully detect out-of-bounds accesses and do not negatively impact other parts of the Wasm execution. The Umbra Wasm runtime is not fully compatible with the specification. Therefore, we only use previously succeeding test files for this regression testing.

We also extended the provided tests to check offsets encoded in the memarg parameter of the memory operation. This is required because the specification tests only check for the detection of out-of-bounds addresses and not for memarg offsets causing the accesses outside of the allocated memory's bounds. These `.wast` files are first compiled to a binary format using the reference Wasm interpreter implementation [21], which is then interpreted, compiled and executed by the Umbra Wasm runtime. The resulting files are used for testing the runtime with every supported combination of code generation back-ends and bounds checking methods.

5.2 Benchmark Selection

When it comes to testing the performance of program execution and compilation, a commonly used benchmark suite is SPEC CPU® 2017 [6]. The Standard Performance Evaluation Corporation (SPEC) bundles and curates real-world, computationally intensive programs in this suite. The programs, written in C, C++, or Fortran, are grouped into four groups: SPECspeed® 2017 Integer, SPECspeed® 2017 Floating Point, SPECrate® 2017 Integer, and SPECrate® 2017 Floating Point.

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

[1] `https://github.com/WebAssembly/testsuite` (accessed 2024-02-03)

CPU® 2017 benchmarks to Wasm proves to be difficult, given the limited feature-set of the wasi-sdk C and C++ compiler and limitations enforced by the Wasm specification. We were able to compile and execute four different benchmarks:

- `505.mcf_r`, a C-program for scheduling public mass transport.[2]
- `508.namd_r`, part of a biomolecular system simulator.[3]
- `519.lbm_r` and `619.lbm_s`, a fluid simulator.[4]
- `557.xz_r`, a compression utility.[5]

A comprehensive list of all benchmarks included in the SPEC CPU® 2017 benchmark suite with their corresponding errors that occurred during compilation or execution is shown in Table A.1.

These build and execution challenges of the SPEC CPU® 2017 benchmarks for Wasm, in addition to the licensing cost and the problem of long run-times, while containing duplicate workloads [42], lead open source projects like Wasmtime and Emscripten to adopt different benchmark suites. Many Wasm-specific benchmarks are built to test the performance of Wasm in a browser environment with JavaScript interaction.[6] However, we are only interested in standalone execution performance, like the Wasmtime project. They built their custom benchmarking suite called "Sightglass"[7], comprised of unstandardized, randomly assorted programs [15].

Therefore, we use the four compiling SPEC CPU® 2017 benchmark programs for our further evaluations because there is currently no other standardized benchmarking suite available for Wasm. These programs are all used to process and generate data, making them relevant workloads for potential UDOs. Additionally, we use a 2D k-Means algorithm [43] benchmark implemented by Sichert [58], which he originally built for comparing the execution speed of UDOs in Umbra with other database systems for his Ph.D. thesis. We only use this benchmark for the overall performance evaluation, because it is tightly integrated into the Umbra runtime, making modifications and specialized execution challenging.

5.3 Setup

All benchmarks were executed on a machine with 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 is running a Linux kernel version 6.2.0-32-generic, built for the 23.04 Ubuntu release "Lunar Lobster". The benchmarks themselves were executed in a low-overhead containerized environment, built from an Ubuntu 22.04 LTS image, and managed using

[2] `https://www.spec.org/cpu2017/Docs/benchmarks/505.mcf_r.html` (accessed 2024-01-26)

[3] `https://www.spec.org/cpu2017/Docs/benchmarks/508.namd_r.html` (accessed 2024-01-26)

[4] `https://www.spec.org/cpu2017/Docs/benchmarks/619.lbm_s.html` (accessed 2024-01-26)

[5] `https://www.spec.org/cpu2017/Docs/benchmarks/557.xz_r.html` (accessed 2024-01-26)

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

[7] `https://github.com/bytecodealliance/sightglass` (accessed 2024-01-26)

the podman program. The machine was not shared with other users during benchmark execution. However, to further reduce the skew of different system characteristics between benchmarks, we performed all measurements multiple times and averaged the results. Still, Wasm code execution speed is highly dependent 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 and absolute execution times may vary heavily between different host machines.

5.4 Results

Figure 5.1 shows the comparison between our implemented bounds checking methods for a real-world workload in a database setting. We focus on the pure execution time of the benchmark because the compilation and validation of the Wasm code are executed together with the surrounding query and are therefore skewed by the overall run-time overhead of the Umbra DBMS query compilation. The SQL query for defining and executing the UDO is shown in Listing A.2.

The used bounds checking methods are the following:

- **None**: Completely disables all bounds checks. While this makes the runtime very unsafe, we can still execute our benchmark, because bounds checks are not required for successful execution of valid Wasm programs. This execution time serves as a baseline for the other execution times.

- **Guard Pages**: Commonly used bounds checking method for 32-bit Wasm, described in Section 4.3.

- **Guard64**: 64-bit Guard Pages, as described in Section 4.4.

- **PG Shadow Memory**: **P**age **G**ranular Shadow Memory, the method explained in Section 4.6.

- **C Shadow Memory**: **C**ompressed Shadow Memory, a version of the Shadow Memory method that consumes less virtual address space, explained in Section 4.7. To circumvent the sub-shadow page growth problems described in Section 4.7.1, we utilize the *Micro Guard Pages* solution presented in the same section.

- **PKeys**: Bounds checks using x86_64 Memory Protection Keys, explained in Section 4.5.

- **Software Bounds Check**: Instructions comparing the accessed address to the memory size are emitted directly into the generated machine code for bounds checking.

- **Debug**: Bounds checking method that performs a call to a Proxy method (Section 2.2) implemented in C++ code which then performs the software bounds check.

5.4.1 UDO Bounds Check Performance

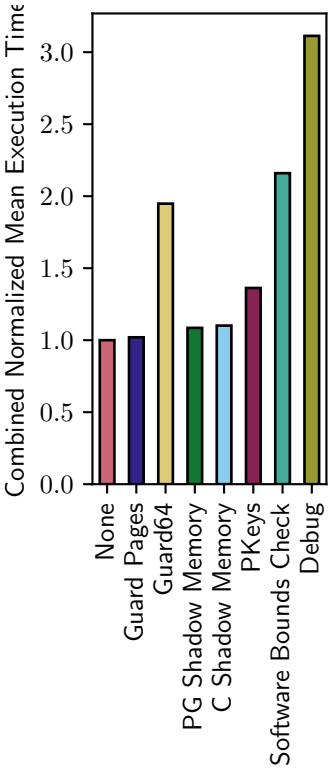


Figure 5.1: *Mean, combined execution times from the graph on the right side. The x-axis shows the utilized bounds checking method. (Lower is better)*

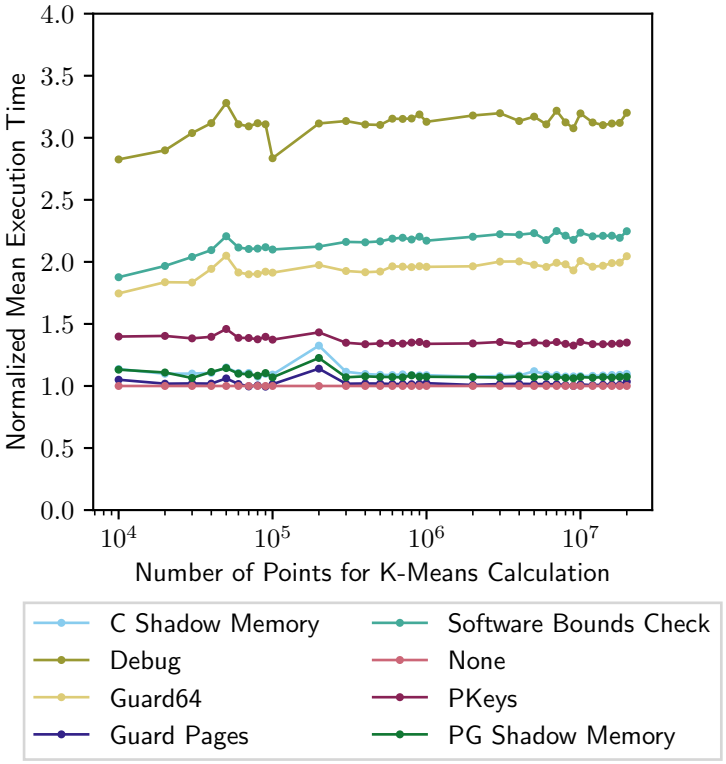


Figure 5.2: *Mean execution times of the 2D k-Means algorithm [43] on 10,000 to 20,000,000 data points, normalized relative to the execution time without bounds checks enabled ("None"). The algorithm was implemented as a UDO for the Umbra DBMS and was executed using the Umbra Wasm runtime. Code generation was performed via the LLVM back-end. (Lower is better)*

As expected and visible in Figure 5.1, the debug bounds checking method has the highest run-time overhead. Every debug bounds check first needs to load the required C++ Proxy method function pointer which requires additional memory accesses, making it slower than the software bounds check method, which performs the same computation. This active method, which relies on bounds checking routines directly emitted to machine code, still poses a significant run-time overhead compared to our proposed solutions.

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 requires a test instruction and, therefore, an additional memory load, making it only slightly faster than software bounds checks with an average run-time overhead of about 94.8% of the baseline execution time.

PKeys perform significantly better for this benchmark with a run-time overhead of 36.2% on average. Both proposed approaches that use a shadow memory for bounds checking perform even better, with the page granular version introducing an overhead of

8.5% on average and the compressed version an average overhead of 10.1% compared to the execution time without bounds checks.

Software bounds checks, conversely, pose a run-time overhead of approximately 87.6% to 124.9%. Figure 5.2 shows where this variance originates. For a small number of points used for the k-Means algorithm, the overall execution time is very short ($\approx 2.3\,\mu s$). In comparison, the compilation time of the LLVM framework alone is almost a magnitude larger. We conclude that other effects dominate run-time performance at these short execution times. Only after 60.000 input points, the measured overhead of the bounds checking methods seems to stabilize. Figure 5.1 shows the mean of the normalized overhead of all numbers of input points and, therefore, is less prone to these artifacts of short execution timespans.

This stabilization shown in Figure 5.2 tells us that for longer-running programs, the overhead introduced by bounds checking scales proportional to the execution time of the entire code. This is very important to ensure that our proposed methods are also suitable for computationally expensive and long-running workloads.

Interestingly, the PKey and shadow memory methods impose a similar overhead for all input sizes, even though their run-time overhead depends on different features of the code. Shadow memory tests are emitted for every memory reference, the execution time penalty should therefore always be proportional to the number of memory accesses. This proportionality is also attested by Figure 5.2. However, the overhead of the PKey method comes from accesses to restricted memory areas. It appears that the number of these accesses in our test program is also proportional to the amount of memory accesses executed. This is not guaranteed, as shown in Figure 5.3.

5.4.2 SPEC CPU® Benchmark Bounds Check Performance

In Figure 5.3, the first thing to notice is that the PKey method forms a clear outlier for the `505.mcf_r` benchmark. This is not caused by the significantly higher execution time of the `WRPKRU` instruction but by a high number of required `WRPKRU` instructions, as shown in Figure 5.6. The discussion of the possible reasons for this high overhead can be found in Section 5.4.3.

Apart from this outlier, we can confirm for the LLVM back-end with Figure 5.3 that our proposed solutions outperform software bounds checking techniques in conventional benchmarks by far. 32-bit execution is up to 2.4 times faster with Guard Pages than with software bounds checks. For 64-bit Wasm code, the Shadow Memory methods only introduce a run-time overhead of 6.4% to 18.0% compared to execution without bounds checks.

The DirectEmit back-end performs software bounds checks more efficiently than the LLVM back-end. However, for 32-bit Wasm code, the Guard Pages approach still wins clearly with zero-overhead execution, with all small deviations from 1 being likely due to imperfections in the reproducibility of the measurements. For 64-bit execution, the Guard64 method also outperforms the fast software bounds checks for all benchmarks.

It is important to keep in mind that all shown values are normalized to their baseline execution time required to execute the benchmark without bounds checks. For the `505.mcf_r` benchmark, the baseline execution time of the LLVM back-end is 9.1 s, of the

C back-end 29.7 s and 12.7 s of the DirectEmit back-end.

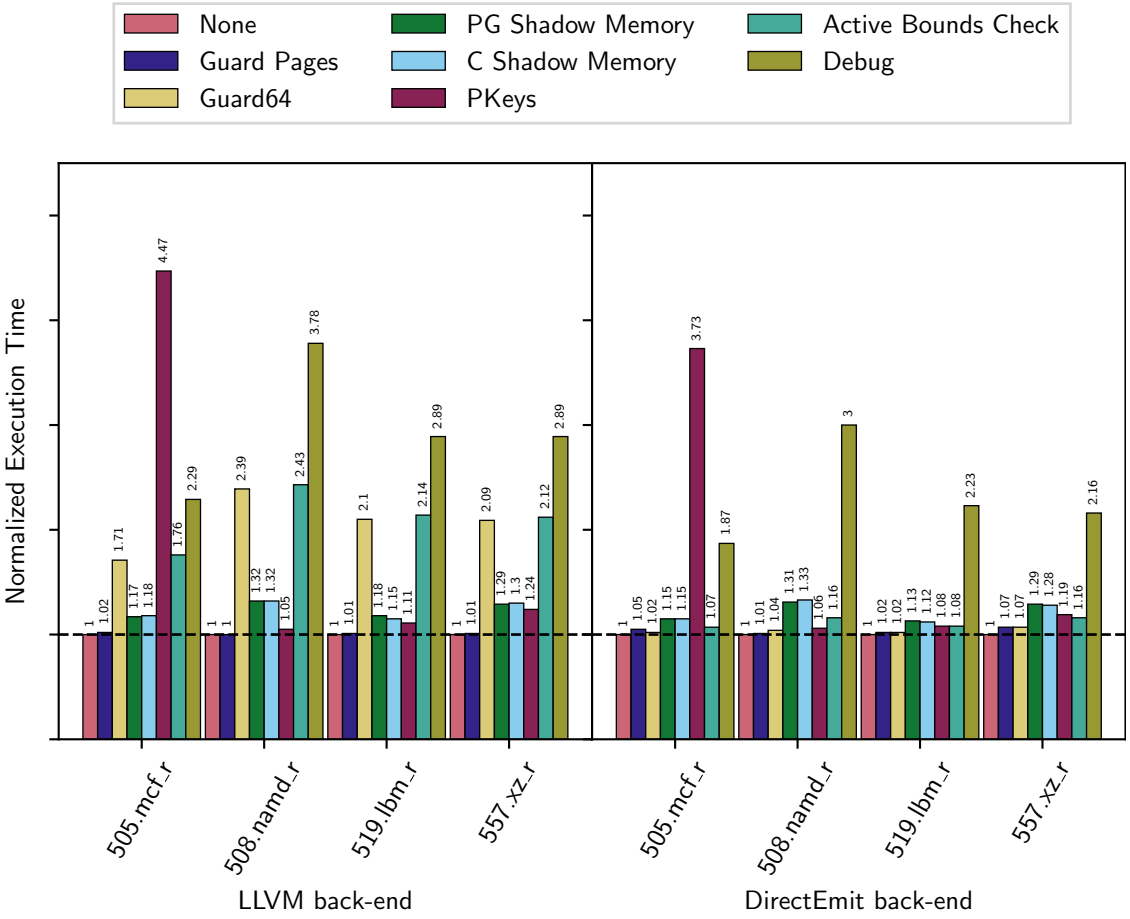


Figure 5.3: *Normalized execution times of the compiled SPEC CPU® 2017 benchmarks, normalized to the baseline execution time without bounds checks ("None"), executed by the Umbra Wasm runtime with the bounds checking mechanisms noted on the x-axis. The utilized code generation back-end is written beneath the graphs. (Lower is better)*

In Figure 5.4, we can see the compilation overhead of Wasm execution in our Umbra runtime using the LLVM code generation back-end. This compilation time does not include the time required for decoding, validation, and translation of the Wasm bytecode to Umbra IR. We chose not to include those durations because they are not heavily influenced by the used bounds checking method and, in comparison to the compilation times, are diminishing. We also only show the compilation times for the LLVM back-end. The C back-end is only used for debugging and the compilation times therefore do not matter for real query execution. The DirectEmit back-end was built to achieve very low compilation overhead and only requires between 3.5 ms to 70.4 ms, a time negligible compared to LLVM's 389.4 ms to 7872.3 ms compilation time.

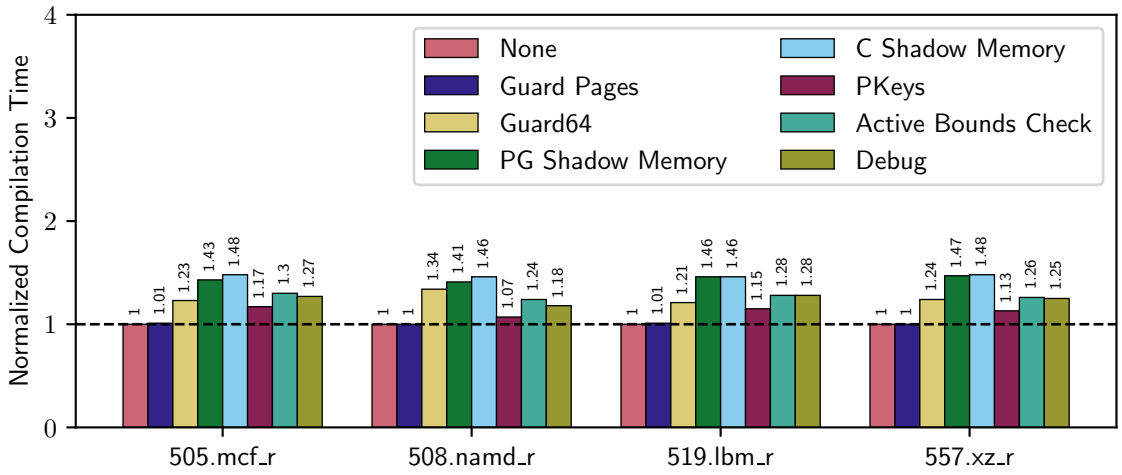


Figure 5.4: *Umbra LLVM back-end compilation time of the SPEC CPU® 2017 benchmarks with different bounds checking mechanisms. (Lower is better)*

We can see that all bounds checking methods that emit additional instructions also increase the required compilation time. Software bounds checks and shadow memory tests increase compilation times notably. Extracting a correlation of emitted bounds check instructions and the required compilation time is difficult because of the complex internals of LLVM compiler infrastructure. However, we know that Guard64 and software bounds checks are translated to similar LLVM IR which is reflected by similar compilation times for both methods.

While the Guard Pages method does not introduce additional instructions, it may require more unwind information to be generated, signaled using the `MightThrowMarker` Umbra IR instruction explained in Section 4.2.2. Therefore, it is interesting that the overhead introduced through additional annotations is insignificant and is not reflected in an increased compilation time.

We conclude from this Figure that while no available bounds checking method has a very high compile-time overhead, they all introduce slight delays in the compilation.

5.4.3 PKey Page Locking Overhead

A drawback of the out-of-bounds detection using memory protection keys is that we always need to write to the `PKRU` register in order to access any page that regular Wasm code cannot access. This is required for all Proxy method calls and changes to security-relevant runtime stores like the current recursion depth counter.

The results presented in Figure 5.5 were collected by injecting additional counting instruction sequences into the code generated by the LLVM back-end that incremented counter variables for every execution of the corresponding instruction or code segment. While this alters the execution speeds, it does not alter the program flow and, therefore, the instruction counts reflect the number of executed instructions during regular program execution.

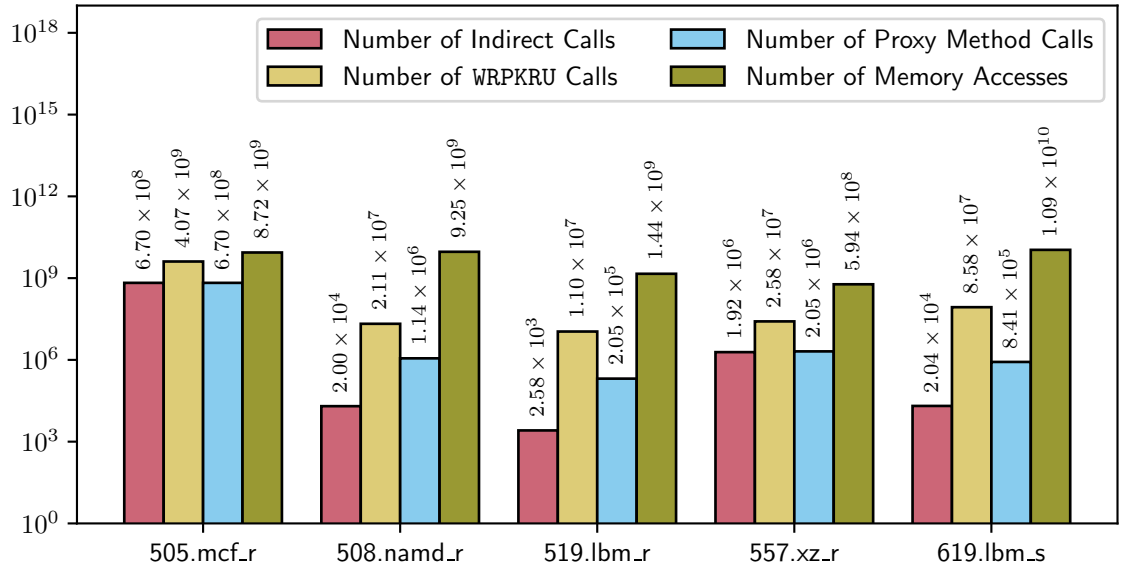


Figure 5.5: *Number of executed instances of selected operations. The measurements were taken through counting instructions injected into machine code generated by Umbra's LLVM back-end. All compiled benchmarks used the PKey bounds checking method.*

Benchmark	# of `WRPKRU` instruction calls	CPU Cycles per `WRPKRU` instruction call
`505.mcf_r`	4,071,029,758	45.36
`508.namd_r`	21,100,610	37.66
`557.xz_r`	25,840,336	54.50
`519.lbm_r`	10,978,604	61.10
`619.lbm_s`	85,791,428	54.66

Figure 5.6: *Comparison of additional CPU cycles and additional instructions required for `WRPKRU` operations during end-to-end execution of SPEC CPU® 2017 benchmarks using PKey bounds checking and the Umbra LLVM code generation back-end.*

To calculate the per-instruction cycle counts shown in Figure 5.6, we used the `perf` performance analysis tools for Linux [29] to count the CPU cycles required for the execution of the benchmark with and without emitting `WRPKRU` instructions. We assume that the difference between both measurements reflects additional CPU cycles introduced by the `WRPKRU` instructions itself and other overheads caused by the execution of the instruction.

The expected latency of the `WRPKRU` instruction measured by third-party benchmarks is 40 CPU cycles and 7 μops for the AMD Zen 4 architecture, the CPU architecture of our test system [51]. Additionally, execution of the `WRPKRU` instruction requires the `eax`, `ecx` and `edx` registers, meaning more register moves and potential spilling of values to the stack. Furthermore, the `WRPKRU` instruction never executes speculatively and memory accesses affected by the instruction are guaranteed to execute after the instruction completes [24]. This can lead to more pipeline stalls and inefficient out-of-order execution around the instruction, amounting to more CPU cycles required for the program execution. Together,

this explains why the required cycles per emitted `WRPKRU` instruction call are so different between benchmarks and are overall higher than the expected latency of the instruction itself.

Based on these findings, it is evident that the high overhead of the PKey bounds checking method for the `505.mcf_r` program seen in Figure 5.3 is the result of a large number of `PKRU` register writes, in comparison to the other benchmarks. It is not caused by an exceptionally slow execution of the PKey locking instruction itself. Manual inspection of the generated code revealed that the `505.mcf_r` benchmark contains a large number of indirect function calls, visible in Figure 5.5, which require a lookup in the Wasm function reference table. This lookup is implemented using a Proxy method in the Umbra Wasm runtime and a pointer dereference, reading otherwise protected memory. Therefore, it always requires prior unlocking and subsequent locking of the Umbra pages, leading to the execution of two `WRPKRU` instructions per indirect function call.

An optimization to circumvent this overhead is to move Wasm function references to a read-only page and implement the indirect function call lookup using a custom Umbra IR instruction. This removes the overhead of the Proxy method call and the two emitted `WRPKRU` instructions.

6 Summary

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 run-time overhead of bounds checks and can even completely eliminate any speed penalty for 32-bit Wasm execution. However, not every execution platform may feature the required hardware extensions. Therefore, we think a hybrid system for Wasm bounds checking is best suited. A Wasm runtime that is equipped with all proposed bounds checking methods and the capability of emitting software bounds checks as a fallback solution. This approach ensures the highest achievable run-time speed for any execution platform while maintaining compatibility, even with legacy and future systems.

Outlook

The presented bounds checking methods can be further improved in several aspects. The most important feature that is not implemented yet is thread safety. Single-threaded execution of UDOs in Umbra's query engine does not scale with modern hardware. Umbra's Wasm runtime therefore implements the "Threading" feature proposal, but the introduced out-of-bounds detection methods are not compatible yet. While most of the implementation only misses proper synchronization between threads, implementing the trap behavior and stack unwinding (Section 4.2.2) for multiple threads requires careful consideration.

When performing out-of-bounds detection using software bounds checks or one of our shadow memory approaches, we first calculate the highest accessed address, based on the access size. A more performant implementation could always operate on the actually accessed address, saving an addition operation in the generated machine code. However, accesses to addresses close to the memory bounds could then lead to undetected out-of-memory accesses if the accessed memory region is only partially contained in the allocated Wasm memory. We would solve this problem by always appending a single guard page to the end of our allocated Wasm memory to detect invalid accesses close to the memory bounds.

The idea of shadow memory bounds checking can be extended to divide the shadow memory into multiple stages. Each stage's table is indexed by a range of bits of the address and either determines the address of the next stage or rules the address to be invalid. This approach, which is similar to multi-level paging, trades address lookup time for a reduction of the shadow memory size and, therefore, consumption of the virtual address space.

The shadow memory method can also be extended to mappings smaller than the size of an x86_64 page. A shadow memory test instruction could read a pointer from the shadow

memory address which is pointing to a valid, mapped memory location or is a null pointer. Therefore, an address lookup in the shadow memory turns into a load and subsequent `TEST` instruction, requiring an additional register. This approach also trades execution time and register space for a reduction of the shadow memory size.

A Appendix

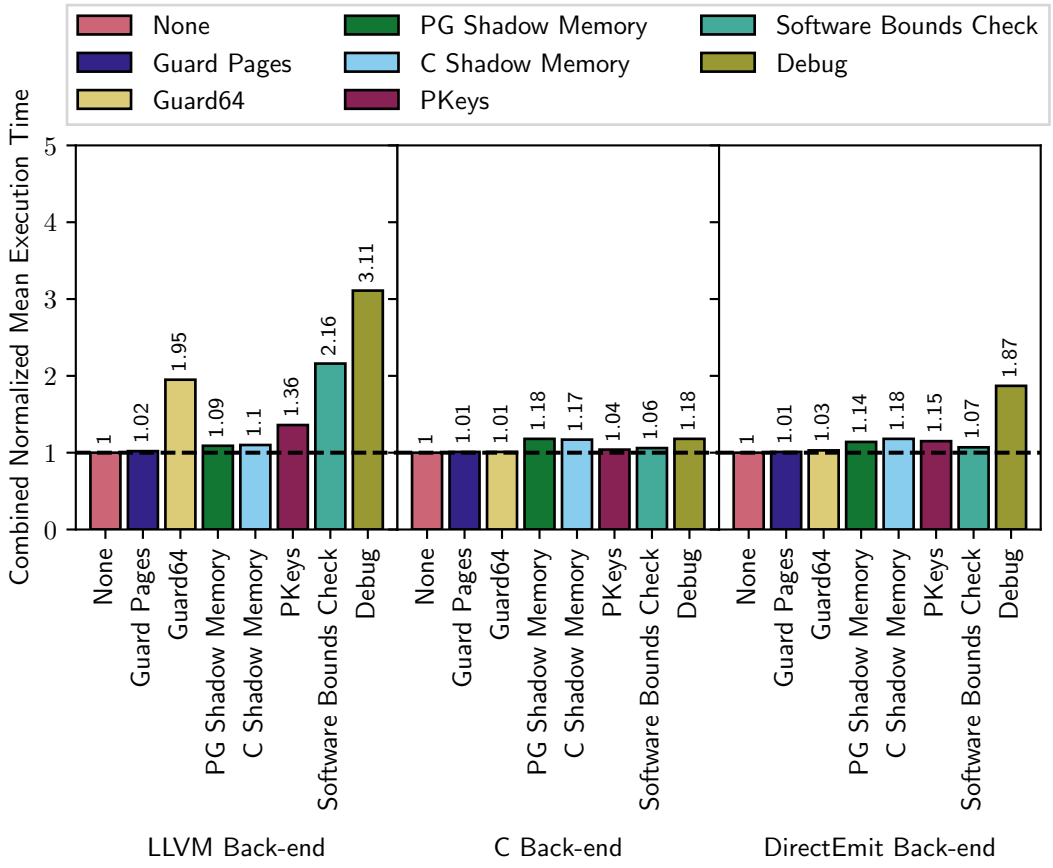


Figure A.1: *Combined normalized execution times of the 2D k-Means algorithm as a Wasm UDO. Values are normalized to execution without bounds checks ("None"). The x-axis shows the utilized bounds checking method. (Lower is better)*

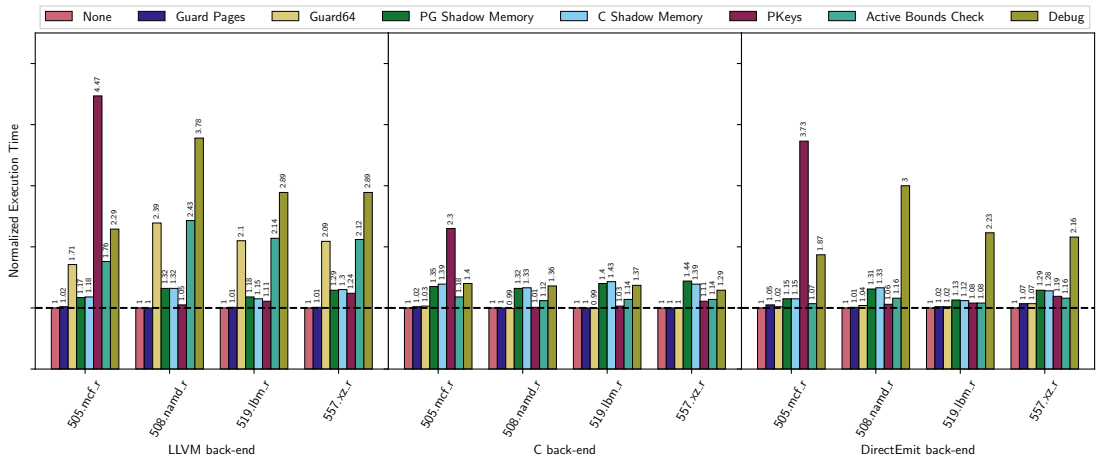


Figure A.2: *Combined normalized execution times of the SPEC CPU® 2017 benchmarks annotated on the x-axis. Values are normalized to execution without bounds checks ("None"). (Lower is better)*

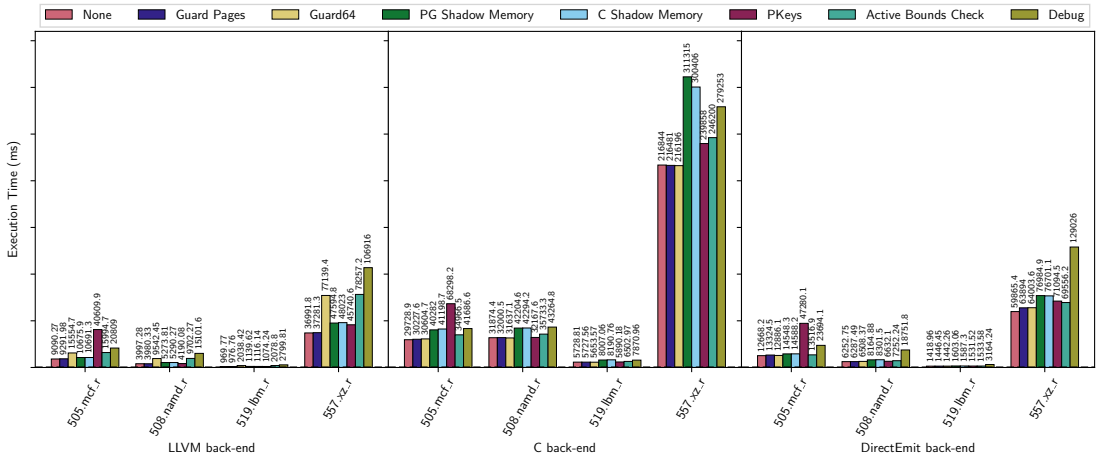


Figure A.3: *Combined absolute execution times of the SPEC CPU® 2017 benchmarks annotated on the x-axis. (Lower is better)*

```
[start address] [end address] [permissions] [offset] [device] [inode] [pathname]
55c791671000-55c791673000 r--p 00000000 00:23 590996 /usr/bin/cat
55c791673000-55c791677000 r-xp 00002000 00:23 590996 /usr/bin/cat
55c791677000-55c791679000 r--p 00006000 00:23 590996 /usr/bin/cat
55c791679000-55c79167a000 r--p 00007000 00:23 590996 /usr/bin/cat
55c79167a000-55c79167b000 rw-p 00008000 00:23 590996 /usr/bin/cat
55c792b3d000-55c792b5e000 rw-p 00000000 00:00 0 [heap]
7f4a7de00000-7f4a7de26000 r--p 00000000 00:23 968841 /usr/lib64/libc.so.6
7f4a7de26000-7f4a7df92000 r-xp 00026000 00:23 968841 /usr/lib64/libc.so.6
7f4a7df92000-7f4a7dfe7000 r--p 00192000 00:23 968841 /usr/lib64/libc.so.6
7f4a7dfe7000-7f4a7dfeb000 r--p 001e6000 00:23 968841 /usr/lib64/libc.so.6
7f4a7dfeb000-7f4a7dff3000 rw-p 001ea000 00:23 968841 /usr/lib64/libc.so.6
7f4a7e0f1000-7f4a7e0f3000 rw-p 00000000 00:00 0
7f4a7e0f3000-7f4a7e0f4000 r--p 00000000 00:23 968840 /usr/lib64/ld-linux-x86-64.so
    .2
7f4a7e0f4000-7f4a7e11b000 r-xp 00001000 00:23 968840 /usr/lib64/ld-linux-x86-64.so
    .2
7f4a7e11b000-7f4a7e125000 r--p 00028000 00:23 968840 /usr/lib64/ld-linux-x86-64.so
    .2
7f4a7e125000-7f4a7e127000 r--p 00032000 00:23 968840 /usr/lib64/ld-linux-x86-64.so
    .2
7f4a7e127000-7f4a7e129000 rw-p 00034000 00:23 968840 /usr/lib64/ld-linux-x86-64.so
    .2
7fff5bdd5000-7fff5bdf6000 rw-p 00000000 00:00 0 [stack]
7fff5bdf7000-7fff5bdfb000 r--p 00000000 00:00 0 [vvar]
7fff5bdfb000-7fff5bdfd000 r-xp 00000000 00:00 0 [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0 [vsyscall]
```

Listing A.1: *Example output of* `cat /proc/self/maps` *(shortened, first line added manually).*

A.1 UDO Definition and Execution

```
CREATE FUNCTION wasm_udo_kmeans(table) RETURNS TABLE LANGUAGE 'udo-wasm' SECURITY
    DEFINER AS 'udo_kmeans.wasm', 'KMeans';

WITH data AS (SELECT x, y, CAST(cluster_id AS bigint) AS payload FROM
    points_70000000) SELECT "clusterId", COUNT(*) FROM wasm_udo_kmeans(TABLE (
    SELECT * FROM data)) GROUP BY "clusterId";
```

Listing A.2: *SQL code for definition and execution of UDO. Table* `points_70000000` *was created beforehand and contains randomly generated triples of coordinates in 2D space with their* `clusterId`*.*

A.2 `mmap` Patch

```
#define _GNU_SOURCE

#include <stdio.h>
#include <dlfcn.h>
#include <sys/mman.h>

static void* (*real_mmap)(void* addr, size_t length, int prot, int flags, int fd,
    off_t offset) = NULL;

static void mmap_init(void)
{
   real_mmap = dlsym(RTLD_NEXT, "mmap");
   if (NULL == real_mmap) {
      fprintf(stderr, "Error in 'dlsym': %s\n", dlerror());
   }
}

void* mmap(void* addr, size_t length, int prot, int flags, int fd, off_t offset)
{
   if (real_mmap == NULL) {
      mmap_init();
   }

   void* p = real_mmap(addr, length, PROT_NONE, flags, fd, offset);
   if (p == MAP_FAILED)
      return MAP_FAILED;

   // Umbra pkey hardcoded to 1
   if (pkey_mprotect(p, length, prot, 1) != 0)
      return -1;
   return p;
}
```

Listing A.3: *Wrapper function for the C standard library function* `mmap` *to always assign PKey 1 to newly mapped segments. This code was originally written by Piotr Praszmo (*`https://stackoverflow.com/a/6083624`*, accessed 2024-02-03) and later altered by us.*

A.3 Evaluation of Shadow Memory Test Instructions

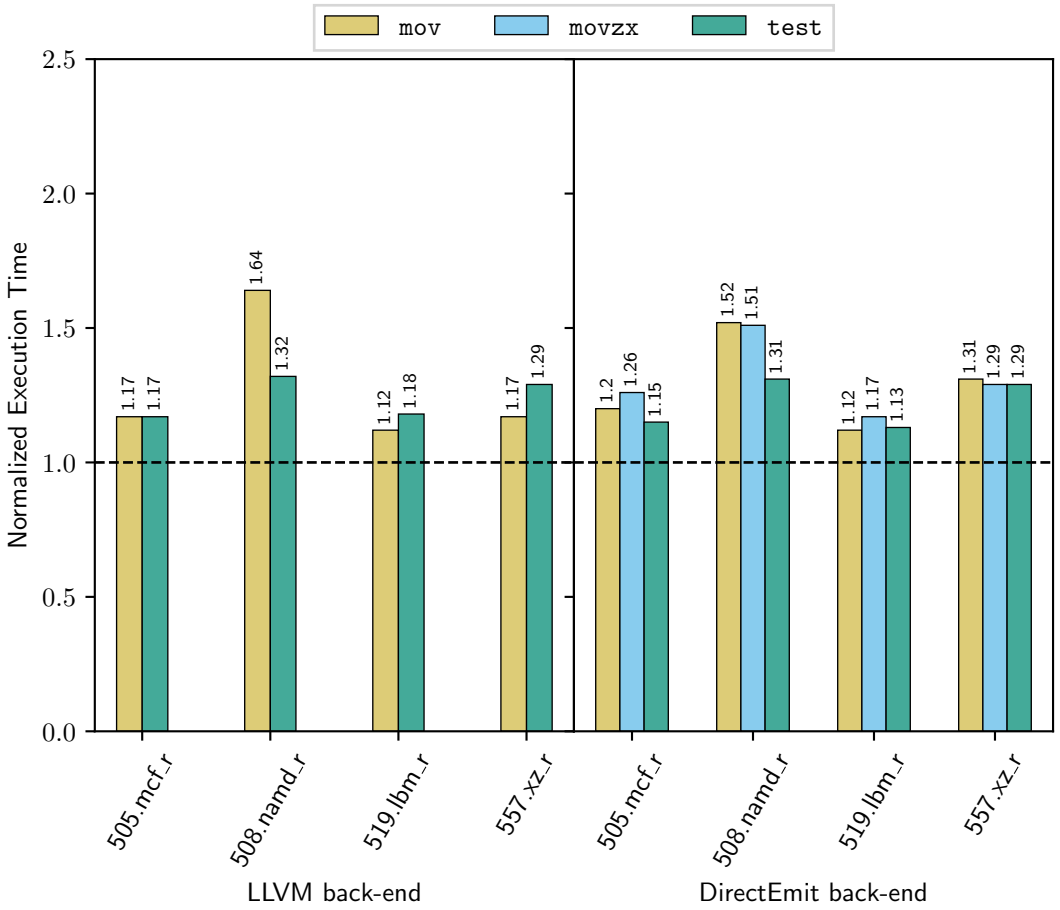


Figure A.4: *Comparison of different x86_64 assembly instructions used for the shadow memory test of the* ***Page Granular Shadow Memory*** *bounds checking technique.*

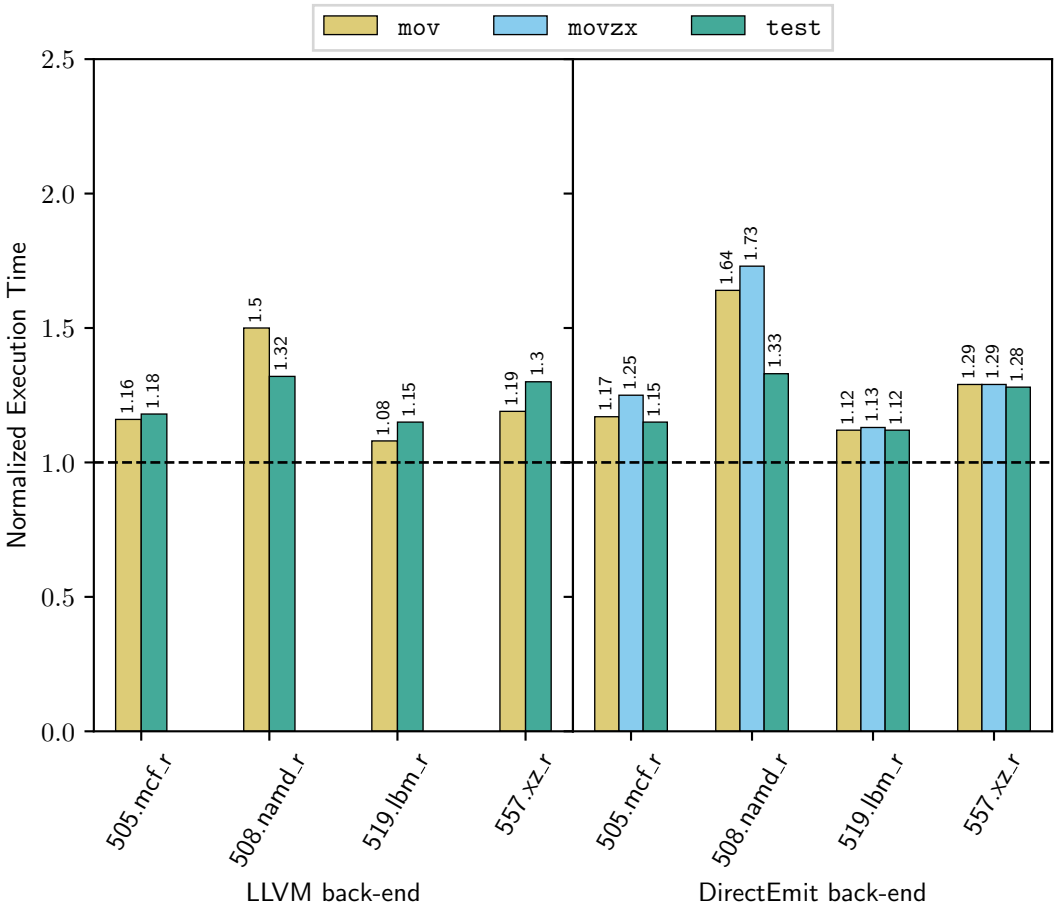


Figure A.5: *Comparison of different x86_64 assembly instructions used for the shadow memory test of the* ***Compressed Shadow Memory*** *bounds checking technique.*

A.4 Evaluation of Guard64 Test Instructions

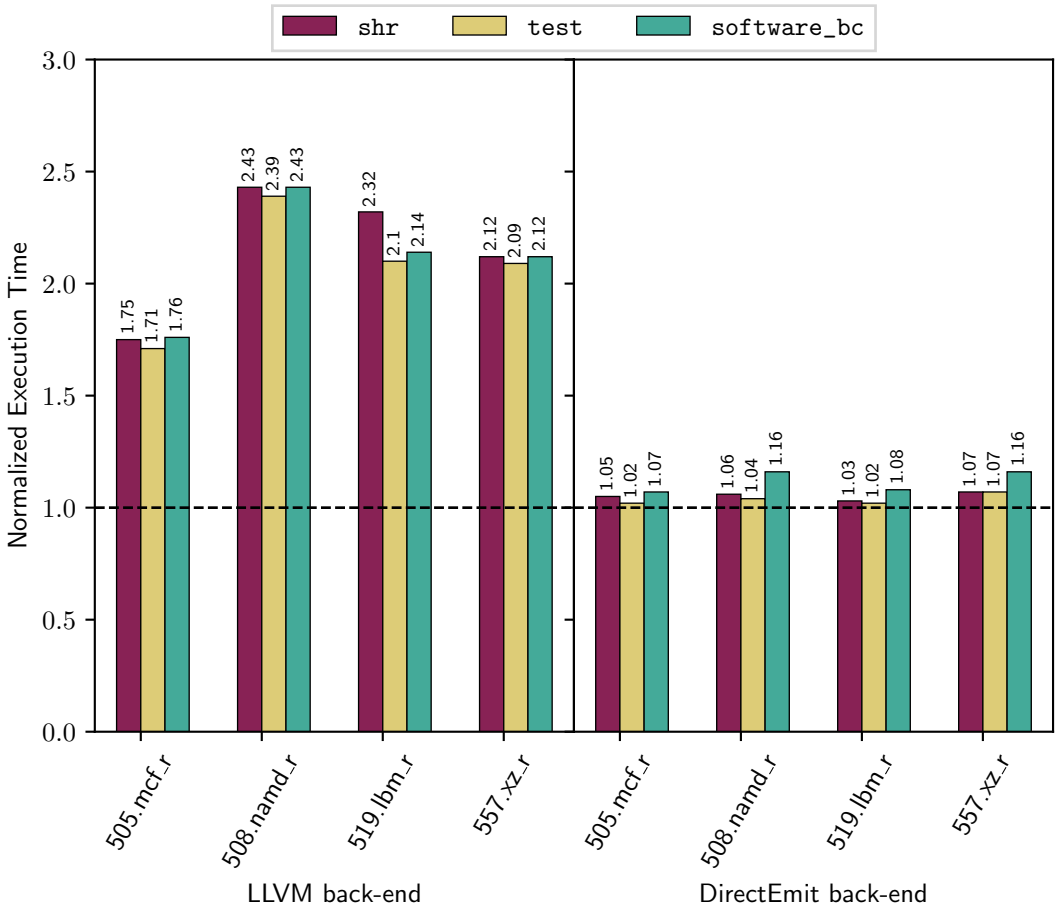


Figure A.6: *Comparison of different x86_64 assembly instructions used for the address test of the Guard64 bounds checking technique. The* `shr` *technique uses a shift to identify whether upper bits of the address are set. The* `test` *technique does so with a* `test` *instruction.* `software_bc` *is a reference value comparing the cost of the Guard64 bounds check with the cost of a software bounds check.*

A.5 SPEC CPU® Benchmark Wasm Runtime Compatibility

Benchmark Name	Usable	Comment
intspeed		
600.perlbench	✗	Missing Header: setjmp
602.gcc_s	✗	Missing Header: sys/wait

620.omnetpp	✗	Missing Header: setjmp
648.exchange2_s	✗	wasi-sdk cannot compile Fortran
641.leela_s	✗	Requires C++ exception support
625.x264_s	✗	Linker error: Duplicate symbol `cfgparams`
631.deepsjeng_s	✗	Runtime Error: Out of memory allocating hashtables
623.xalancbmk_s	✗	Missing Header: 'linux/limits.h'
intrate		
500.perlbench_r 502.gcc_r 520.omnetpp_r 523.xalancbmk_r 525.x264_r 541.leela_r 548.exchange2_r	✗	Equivalent to **intspeed**
505.mcf_r	✓	Fixed for Umbra Wasm by implementing WASI function `path_open`
531.deepsjeng_r	✗	Runtime Error: Recursion limit reached
557.xz_r	✓	
999.specrand_ir	✓	
fpspeed		
603.bwaves_s	✗	wasi-sdk cannot compile Fortran
607.cactuBSSN_s	✗	Missing Header: 'netdb.h'
619.lbm_s	✓	Fixed for Umbra Wasm by implementing WASI function `path_filestat_get`
621.wrf_s	✗	Linker error: Call to undeclared function 'dup2'
627.cam4_s	✗	wasi-sdk cannot compile Fortran
628.pop2_s	✗	wasi-sdk cannot compile Fortran
638.imagick_s	✗	missing sys/wait
644.nab_s	✗	Non-thread-local declaration of 'errno' follows thread-local declaration
649.fotonik3d_s	✗	wasi-sdk cannot compile Fortran
654.roms_s	✗	wasi-sdk cannot compile Fortran
996.specrand_fs	✓	
fprate		
503.bwaves_r 507.cactuBSSN_r 521.wrf_r 527.cam4_r 538.imagick_r 544.nab_r 549.fotonik3d_r 554.roms_r	✗	Equivalent to **fpspeed**

508.namd_r	✓	Works, but computes incorrect result because of Wasm floating point handling
510.parest_r	✗	Requires C++ exception support
511.povray_r	✗	Requires C++ exception support
519.lbm_r	✓	Fixed for Umbra Wasm by implementing WASI function `path_filestat_get`
526.blender_r	✗	missing sys/wait
997.specrand_fr	✓	

Table A.1: *Analyzed compatibility of SPEC CPU® benchmarks with the Umbra Wasm runtime and wasi-sdk.*

A.6 Signal Handler Unwinding in glibc and libc

Unwinders like `libunwind` for Linux use unwind information in the `DWARF` format and the exception-throwing function's current register states to backtrace stack frames in that function's call stack. In x86_64, the `call` assembly instruction is used to execute another function. Besides setting the IP to the address of the function's first instruction, it also saves the address of the instruction directly after the `call` instruction to the stack. This address is used by the unwinder to backtrace the call chain.

We focus on handling of `SIGSEGV` signals in the following because this is the only signal type required for generating Wasm traps for out-of-bounds accesses. This signal is a synchronous signal [37], meaning execution is blocked until the signal is handled. It is raised during the execution of a faulting instruction. We register our signal handler using the `sigaction` standard library function which in turn uses the `rt_sigaction` system call to inform the Linux kernel about the installed signal handler. The `SA_SIGINFO` flag allows us to receive additional run-time information. Therefore, we receive an instance of the `ucontext_t` structure that contains the register state at the point in execution where the signal was raised.

Compared to a normal function call, signal handlers are not reached via a call instruction and no return address is pushed to the stack. Upon registering a signal, the Linux kernel creates a new stack frame on the user-space stack below the red-zone, which will be used for the signal handler's execution.[1] When the signal handler function is called, the return address on the stack is not the address of the next instruction of the signal-causing function, but the start address of a function called the "signal trampoline". For the x86_64 architecture, this function is stored in the `sa_restorer` field of the `sigaction` structure used for registering the signal.[2,3] This function is provided by the C runtime and executes the `sigreturn` system call, handing execution back to the kernel for cleanup and normal

[1] `https://elixir.bootlin.com/linux/v6.5/source/arch/x86/kernel/signal.c#L207` (accessed 2024-01-28)

[2] Using the `sa_restorer` field is generally not required for all architectures. However, for x86_64, it is required and automatically set by the C standard library through its `sigaction` implementation.

[3] `https://elixir.bootlin.com/linux/v6.5/source/arch/x86/kernel/signal_64.c#L188` (accessed 2028-01-28)

process resumption.

This poses a problem to the unwinder: The return address on the stack is not the actual address of the place where the current function, the signal handler, was called. The required call address is stored in the `ucontext_t` structure. Our C library's developers were aware of this problem. They annotate the "signal trampoline", called `restore_rt`, with additional DWARF information stored in the `.eh_frame` section of the generated binary.[4] This information for the `restore_rt` function from LLVMs `libc` is shown in Listing A.4. glibc uses the same mechanism.

```
$ llvm-dwarfdump -eh-frame /lib64/libc.so.6

0000258c 00000010 00000000 CIE
  Format:                DWARF32
  Version:               1
  Augmentation:          "zRS"
  Code alignment factor: 1
  Data alignment factor: -8
  Return address column: 16
  Augmentation data:     1B

  DW_CFA_nop:
  DW_CFA_nop:


000025a0 00000078 00000018 FDE cie=0000258c pc=0003f18f...0003f199
  Format:       DWARF32
  DW_CFA_def_cfa_expression: DW_OP_breg7 RSP+160, DW_OP_deref
  DW_CFA_expression: R8 DW_OP_breg7 RSP+40
  DW_CFA_expression: R9 DW_OP_breg7 RSP+48
  [...]
  DW_CFA_expression: RSP DW_OP_breg7 RSP+160
  DW_CFA_expression: RIP DW_OP_breg7 RSP+168
  DW_CFA_nop:
  DW_CFA_nop:

  0x3f18f: CFA=DW_OP_breg7 RSP+160, DW_OP_deref: RAX=[DW_OP_breg7 RSP+144], RDX=[
      DW_OP_breg7 RSP+136], RCX=[DW_OP_breg7 RSP+152], RBX=[DW_OP_breg7 RSP+128],
      RSI=[DW_OP_breg7 RSP+112], RDI=[DW_OP_breg7 RSP+104], RBP=[DW_OP_breg7 RSP
      +120], RSP=[DW_OP_breg7 RSP+160], R8=[DW_OP_breg7 RSP+40], R9=[DW_OP_breg7
      RSP+48], R10=[DW_OP_breg7 RSP+56], R11=[DW_OP_breg7 RSP+64], R12=[
      DW_OP_breg7 RSP+72], R13=[DW_OP_breg7 RSP+80], R14=[DW_OP_breg7 RSP+88], R15
      =[DW_OP_breg7 RSP+96], RIP=[DW_OP_breg7 RSP+168]
```

Listing A.4: *DWARF debugging information entry for `restore_rt` function from LLVM's `libc` C standard library.*

The interesting parts are the `DW_CFA_expression` statements. These use the `DW_OP_breg7` which reads the value of the stack pointer register and adds the following signed offset to said register. The C standard library `libc` uses this mechanism to tell the unwinder where to find the actual contents of the mapped registers. All these offsets are the offsets of the stored registers in the `ucontext_t` structure[5] which is still on the

[4] `https://github.com/bminor/glibc/blob/ae49a7b29acc184b03c2a6bd6ac01b5e08efd54f/sysdeps/unix/sysv/linux/x86_64/libc_sigaction.c#L71` (accessed 2024-01-28)

[5] `https://github.com/bminor/glibc/blob/ae49a7b29acc184b03c2a6bd6ac01b5e08efd54f/sysdeps/unix/sysv/linux/x86_64/ucontext_i.sym#L16` (accessed 2024-28-01)

user-space stack frame placed by the kernel before entering the signal handler. The unwinder uses this additional debugging information to recover the register state at the point in execution when the signal was raised. Afterward, it continues unwinding as if the exception from the signal handler was thrown at the point of the instruction that caused the `SIGSEGV` signal.

Therefore, stack unwinding for exceptions thrown in signal handlers registered by the glibc or libc libraries is safe and guaranteed to behave identically to an exception thrown at the instruction that caused the signal to be raised.

List of Figures

List of Tables

Listings

Bibliography

[1] Wasmer. Wasmer Inc. URL `https://wasmer.io`. Accessed on 2024-01-30.

[2] *High Performance Toolchain: Compilers, Libraries & Profilers Tuning Guide for AMD EPYC™ 9004 Processors*. Advanced Micro Devices, Inc., 2023. URL `https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/tuning-guides/58020-epyc-9004-tg-high-perf-toolchain.pdf`. Accessed on 2024-02-02.

[3] N. Amit and VMware Research. Optimizing the TLB Shootdown Algorithm with Page Access Tracking. *Proceedings of the 2017 USENIX Annual Technical Conference*, 2017.

[4] Apple Inc. JavaScriptCore. URL `https://docs.webkit.org/Deep%20Dive/JSC/JavaScriptCore.html`. Accessed on 2024-01-30.

[5] Arm Limited. Armv8.5-A Memory Tagging Extension Whitepaper. Technical report, Arm Limited. URL `https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf`. Accessed on 2024-02-09.

[6] J. Bucek, K.-D. Lange, and J. V. Kistowski. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 41–42, Berlin Germany, 2018. ACM. ISBN 978-1-4503-5629-9. doi: 10.1145/3185768.3185771.

[7] M. Bynens. Celebrating 10 years of V8, Sept. 2018. URL `https://v8.dev/blog/10-years`. Accessed on 2024-02-15.

[8] Bytecode Alliance. Wasmtime. URL `https://wasmtime.dev/`. Accessed on 2024-01-30.

[9] C Standards Committee - ISO/IEC JTC1/SC22/WG14. International Standard ISO/IEC 9899:2018(E) – Programming Language C. Standard, International Organization for Standardization, Geneva, CH, 2018.

[10] T. Chiueh. Fast Bounds Checking Using Debug Register. In *High performance embedded architectures and compilers*, pages 99–113, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-77560-7. doi: 10.1007/978-3-540-77560-7_8.

[11] Cloud Native Computing Foundation. WasmEdge. URL `https://wasmedge.org/`. Accessed on 2024-01-30.

[12] J. Corbet. Restartable sequences in glibc. *LWN.net*, Jan. 2022. URL `https://lwn.net/Articles/883104/`. Accessed on 2024-03-09.

[13] A. Crichton. Architecture - Wasmtime, 2023. URL `https://docs.wasmtime.dev/contributing-architecture.html#linear-memory`. Accessed on 2024-01-10.

[14] G. Dot, A. Martinez, and A. Gonzalez. Analysis and Optimization of Engines for Dynamically Typed Languages. In *2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 41–48, Florianopolis, Brazil, Oct. 2015. IEEE. ISBN 978-1-4673-8011-9. doi: 10.1109/SBAC-PAD.2015.20.

[15] N. Fitzgerald. rfcs/accepted/benchmark-suite.md. Technical report, bytecodealliance, Jan. 2021. URL `https://github.com/bytecodealliance/rfcs/blob/main/accepted/benchmark-suite.md`. Accessed on 2023-11-05.

[16] Free Software Foundation Inc. The GNU C++ Library. URL `https://gcc.gnu.org/onlinedocs/libstdc++/`. Accessed on 2024-02-19.

[17] GNU C Library Project and Free Software Foundation, Inc. The GNU C Library. URL `https://sourceware.org/glibc/`. Accessed on 2024-02-19.

[18] GNU Project and Free Software Foundation Inc. GCC, the GNU Compiler Collection. URL `https://gcc.gnu.org/`. Accessed on 2024-02-19.

[19] D. Gohman. WASI: WebAssembly System Interface. Technical report, bytecodealliance, 2019. URL `https://github.com/bytecodealliance/wasmtime/blob/main/docs/WASI-overview.md`. Accessed on 2024-01-10.

[20] Google. V8 JavaScript Engine. URL `https://v8.dev/`. Accessed on 2024-01-30.

[21] 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, pages 185–200, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.3062363.

[22] D. Herman, L. Wagner, and A. Zakai. asm.js - Working Draft. Technical report, Mozilla, Aug. 2014. URL `http://asmjs.org/spec/latest/`. Accessed on 2024-01-06.

[23] *Intel Architecture Instruction Set Extensions and Future Features*. Intel Corporation, 2200 Mission College Blvd. Santa Clara, CA 95054-1549 USA, Dec. 2023.

[24] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, 2200 Mission College Blvd. Santa Clara, CA 95054-1549 USA, Sept. 2023.

[25] *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation, 2200 Mission College Blvd. Santa Clara, CA 95054-1549 USA, Jan. 2024.

[26] *Overcommit Accounting*. The kernel development community, Oct. 2023. URL `https://docs.kernel.org/mm/overcommit-accounting.html`. Accessed on 2024-01-21.

[27] *Memory Management*. The kernel development community, Mar. 2023. URL `https://docs.kernel.org/arch/x86/x86_64/mm.html`. Accessed on 2024-02-02.

[28] *Documentation for /proc/sys/vm/*. The kernel development community, Oct. 2023. URL `https://docs.kernel.org/admin-guide/sysctl/vm.html`. Accessed on 2024-02-02.

[29] M. Kerrisk and A. Colomar. *perf(1)*. The Linux man-pages project, 2022. URL `https://man7.org/linux/man-pages/man1/perf.1.html`. Accessed on 2024-02-13.

[30] M. Kerrisk and A. Colomar. *brk(2)*. The Linux man-pages project, 2023. URL `https://man7.org/linux/man-pages/man2/brk.2.html`. Accessed on 2024-01-29.

[31] M. Kerrisk and A. Colomar. *mmap(2)*. The Linux man-pages project, 2023. URL `https://man7.org/linux/man-pages/man2/mmap.2.html`. Accessed on 2024-01-29.

[32] M. Kerrisk and A. Colomar. *mprotect(2)*. The Linux man-pages project, 2023. URL `https://man7.org/linux/man-pages/man2/mprotect.2.html`. Accessed on 2024-01-29.

[33] M. Kerrisk and A. Colomar. *sigaction(2)*. The Linux man-pages project, 2023. URL `https://man7.org/linux/man-pages/man2/sigaction.2.html`. Accessed on 2024-01-29.

[34] M. Kerrisk and A. Colomar. *setjmp(3)*. The Linux man-pages project, 2023. URL `https://man7.org/linux/man-pages/man3/longjmp.3.html`. Accessed on 2024-01-30.

[35] M. Kerrisk and A. Colomar. *proc(5)*. The Linux man-pages project, 2023. URL `https://man7.org/linux/man-pages/man5/proc.5.html`. Accessed on 2024-01-29.

[36] M. Kerrisk and A. Colomar. *pkeys(7)*. The Linux man-pages project, 2023. URL `https://man7.org/linux/man-pages/man7/pkeys.7.html`. Accessed on 2024-01-29.

[37] M. Kerrisk and A. Colomar. *signal(7)*. The Linux man-pages project, 2023. URL `https://man7.org/linux/man-pages/man7/signal.7.html`. Accessed on 2024-01-29.

[38] M. Kerrisk and A. Colomar. *signal-safety(7)*. The Linux man-pages project, 2023. URL `https://man7.org/linux/man-pages/man7/signal-safety.7.html`. Accessed on 2024-01-29.

[39] T. Kersten, V. Leis, and T. Neumann. Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra. *The VLDB Journal*, 30(5):883–905, Sept. 2021. ISSN 1066-8888, 0949-877X. doi: 10.1007/s00778-020-00643-4.

[40] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, San Jose, CA, USA, 2004. IEEE. ISBN 978-0-7695-2102-2. doi: 10.1109/CGO.2004.1281665.

[41] D. Lea. A Memory Allocator. Technical report, 2000. URL `https://gee.cs.oswego.edu/dl/html/malloc.html`. Accessed on 2024-01-17.

[42] A. Limaye and T. Adegbija. A Workload Characterization of the SPEC CPU2017 Benchmark Suite. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 149–158, Belfast, Apr. 2018. IEEE. ISBN 978-1-5386-5010-3. doi: 10.1109/ISPASS.2018.00028.

[43] S. Lloyd. Least Squares Quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, Mar. 1982. ISSN 0018-9448. doi: 10.1109/TIT.1982.1056489.

[44] LLVM Project. libunwind. URL `https://github.com/llvm/llvm-project/tree/33c6b77d2a18862fb5b16160ef9d600382e93f19/libunwind`. Accessed on 2024-01-28.

[45] H. Lu, M. Matz, M. Girkar, J. Hubicka, A. Jaeger, and M. Mitchell. System V Application Binary Interface AMD64 Architecture Processor Supplement, Jan. 2024.

[46] J. McCall, R. Smith, J. Merrill, and T. Honermann. *C++ ABI for Itanium: Exception Handling*, 2012. URL `https://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html`. Accessed on 2024-01-28.

[47] Mozilla Corporation. SpiderMonkey JavaScript/WebAssembly Engine. URL `https://spidermonkey.dev/`. Accessed on 2024-02-14.

[48] T. Neumann and M. J. Freitag. 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*, 2020.

[49] *adi (7)*. Oracle, 2022. URL `https://docs.oracle.com/cd/E88353_01/html/E37853/adi-7.html`. Accessed on 2024-02-09.

[50] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 241–254, Renton, WA, July 2019. USENIX Association. ISBN 978-1-939133-03-8.

[51] Real-Time and Embedded Systems Lab. uops.info. URL `https://uops.info/table.html`. Accessed on 2024-02-18.

[52] R. Rivest and D. Eastlake. S-expressions. Internet-draft draft-rivest-sexp-03, Internet Engineering Task Force / Internet Engineering Task Force, Aug. 2023. URL `https://datatracker.ietf.org/doc/draft-rivest-sexp/03/`. Accessed on 2024-03-09.

[53] A. Rossberg. WebAssembly Core Specification, Apr. 2022. URL `https://www.w3.org/TR/wasm-core-2/`. Accessed on 2024-02-19.

[54] A. Rossberg and A. Nelson. Multiple Memories for Wasm, Feb. 2020. URL `https://github.com/WebAssembly/multi-memory/blob/c2c00d81cbc1a9a4a8c2411bebbb7151d8cc492b/proposals/multi-memory/Overview.md`. Accessed on 2024-02-03.

[55] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*, pages 309–318, Boston, MA, June 2012. USENIX Association.

[56] A. Sharma, D. Watson, and D. Mosberger-Tang. The libunwind project. URL `https://savannah.nongnu.org/projects/libunwind/`. Accessed on 2024-01-28.

[57] V. Shymanskyy and S. Massey. wasm3. Wasm3 Labs. URL `https://github.com/wasm3/wasm3`. Accessed on 2024-01-30.

[58] M. Sichert. *Efficient and Safe Integration of User-Defined Operators into Modern Database Systems*. PhD thesis, Technische Universität München, Munich, 2024. URL `https://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20240111-1713746-1-4`. Accessed on 2024-02-19.

[59] M. Sichert and T. Neumann. User-Defined Operators: Efficiently Integrating Custom Algorithms into Modern Databases. *Proceedings of the VLDB Endowment*, 15(5): 1119–1131, Jan. 2022. ISSN 2150-8097. doi: 10.14778/3510397.3510408.

[60] B. Smith, A. Rossberg, and S. Clegg. Memory64, Feb. 2024. URL `https://github.com/WebAssembly/memory64/blob/bcf4834fe72f1227eb2f1ec46c107f16fac94d9e/proposals/memory64/Overview.md`. Accessed on 2024-01-10.

[61] J. W. Sunarto, A. Quincy, F. S. Maheswari, Q. D. A. Hafizh, M. G. Tjandrasubrata, and M. H. Widianto. A Systematic Review of WebAssembly VS Javascript Performance Comparison. In *2023 International Conference on Information Management and Technology (ICIMTech)*, pages 241–246, Malang, Indonesia, Aug. 2023. IEEE. ISBN 9798350326093. doi: 10.1109/ICIMTech59029.2023.10277917.

[62] The Clang Team. Clang C Language Family Frontend for LLVM. URL `https://clang.llvm.org/`. Accessed on 2024-01-28.

[63] The Standard C++ Foundation. International Standard ISO/IEC 14882:2020(E) – Programming Language C++, Dec. 2020.

[64] TIS Committee. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification, May 1995. URL `https://refspecs.linuxbase.org/elf/elf.pdf`. Accessed on 2024-02-03.

[65] K. C. Wang. Signals and Signal Processing. In *Systems Programming in Unix/Linux*, pages 205–219. Springer International Publishing, Cham, 2018. ISBN 978-3-319-92429-8. doi: 10.1007/978-3-319-92429-8_6.

[66] W. Wang. Empowering Web Applications with WebAssembly: Are We There Yet? In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1301–1305, Melbourne, Australia, Nov. 2021. IEEE. ISBN 978-1-66540-337-5. doi: 10.1109/ASE51524.2021.9678831.

[67] D. Watson, A. Sharma, and D. Mosberger-Tang. *libunwind documentation*. The libunwind project. URL `https://www.nongnu.org/libunwind/docs.html`. Accessed on 2024-01-28.

[68] R. N. M. Watson, S. W. Moore, P. Sewell, and P. G. Neumann. An Introduction to CHERI. Technical Report 941, University of Cambridge - Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, 2019. URL `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-941.pdf`. Accessed on 2024-02-19.

[69] WebAssembly Community Group. Feature Extensions - WebAssembly. URL `https://webassembly.org/features/`. Accessed on 2024-01-10.

[70] A. Zakai. Emscripten: An LLVM-to-JavaScript Compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 301–312, Portland Oregon USA, Oct. 2011. ACM. ISBN 978-1-4503-0942-4. doi: 10.1145/2048147.2048224.