Analyzing Redundant Memory Bounds Checks for Wasm2c

Allen Jue and David Lu

December 2024

1 Abstract

WebAssembly (Wasm) is a popular, low-level language that allows for near native performance, while offering security and portability guarantees on the web. Wasm uses software memory bounds checking to mitigate memory vulnerabilities. However, these software mitigations introduce performance overheads. In this paper, we create a dynamic analysis tool that identifies redundant memory bounds checks generated by wasm2c. We use simple heuristics to identify an upper bound of 6.54% of redundant memory checks in the Dhrystone benchmark. This preliminary study demonstrates the promise of creating a static anlysis tool for Wasm binaries and the potential for using wasm2c as an intermediary for optimizing Wasm binaries.

2 Introduction/Project Goals

WebAssembly (Wasm) is an increasingly popular binary instruction format designed to be a compilation target for high-performance programming languages, enabling efficient and portable deployments on the web for client and server applications [15, 8]. Typically, compute-intensive workloads are written in low-level languages like C++ or Rust and then compiled to a Wasm binary that is also deployed on the website. The binary is run in a separate runtime, which is responsible for verifying and maintaining the security guarantees provided by Wasm.

Once compiled, Wasm binaries are in bytecode format, and are often difficult to debug and analyze[12]. To help bridge the gap between WebAssembly's binary format and traditional development workflows, the WebAssembly Toolkit (abbreviated as wabt) provides a multitude of tools to help developers. One such tool is wasm2c, which decompiles Wasm binaries into more readable C source and header files, giving developers a larger range of tools they can use to profile performance, debug errors, or perform static analysis. As the number of languages that use Wasm as a compile target increases and the Wasm ecosystem matures, the importance of these decompilation tools to help understand and optimize WebAssembly code continues to grow.

A core feature of Wasm is its security and speed, which is done through sandboxed isolation of Wasm runtime instances, providing a safety property of Software Fault Isolation (SFI)[6]. Wasm instance memory is a single, linear array of bytes that can be grown dynamically as needed. For safety, upon accessing memory outside the Wasm accessible region the associated thread will trap, or terminate. This is implemented either as a software bounds check or via guard pages, where memory outside of a Wasm instance's linear memory is marked as inaccessible [9, 6].

Contributions. We make the following contributions:

- We created a tool to identify and measure the number of redundant memory bounds checks in C modules produced by wasm2c.
- Using this tool, we analyzed a diverse set of synthetic programs derived from a wasm2c-converted Dhrystone WebAssembly binary to evaluate redundancy in memory bounds checks.
- We demonstrated the feasibility of converting the C modules generated by wasm2c back into functional WebAssembly binaries.

3 Previously Discussed Security Concerns, CVE, etc.

Guard pages have traditionally been an efficient method of determining out-of-bounds memory accesses. In contrast to software bounds checks, they work instead by reserving addressable memory as inaccessible, which will trigger a page fault from the built-in memory subsystem. This page fault can then be used to determine out-of-bounds accesses, removing the need to perform a software check. In 32-bit systems, the maximum memory is 4GiB, and 32-bit pointers can only go up to 4GiB, making the memory protection (usually through page faults on the unmapped memory) a good indicator of an out-of-bounds access. However, as of the memory64 proposal, which details the move to support memory sizes larger than 2^{32} bits, reserving all addressable Wasm memory is no longer possible on a 64-bit host system. Therefore, 64-bit systems typically fall back to the more expensive bounds checks for memory accesses[4, 3].

This fallback suffers from heavy performance overhead, and can incur anywhere from 20% to 220% overhead depending on the workload for common operations, such as Cholesky decomposition and GEMM. However, on some system configurations such as Arm/Wasmtime, a performance overhead of as much as 650% may be incurred [14].

It should also be noted that bounds checks and guard pages can suffer from implementation bugs that can be exploited to mount spectre-style attacks [7]. One such example was CVE-2023-26489, where an erroneous code lowering rule in Cranelift's x86_64 backend allowed malicious WASM instances to access memory outside the sandbox[1]. However, this lies outside the scope of our project, as our work is more concerned about optimizing redundant bounds checks.

4 Solution

Static analysis tools are powerful programs that can identify bugs and sites of optimizations. However, creating a static analyzer can be a monumental task, and it may be an inefficient venture if the effect size of a static analyzer is insignificant. In this paper, we developed a dynamic analysis tool that provides an optimistic limit of the redundant bounds checks that can be potentially detected in C code generated by wasm2c with an ideal static analysis tool.

Wasm2c preserves the functionality of the original Wasm binary, and it allows us to analyze the Wasm code as C, an organized, higher-level language. If the amount of memory bounds checks is a non-trivial amount, we argue that it is worthwhile to create a static analysis tool or edit the wasm2c transpiler to ignore generating code for unnecessary memory bounds checks. To accomplish this, we utilize runtime heuristics that align with information that a static analysis tool would have. In particular, we consider 3 heuristics:

1. Functions that are 1 frame away from each other (caller-callee relationships)
2. Functions with redundant memory checks
3. Functions that pass parameters to each other

When analyzing functions that are one frame away from each other, we are ensuring that the dynamic analysis is focused and computationally feasible by limiting scope to immediate function relationships. If we look at methods with memory bounds checks, which occur at every load and store, we can identify sources for potentially redundant memory checks. By focusing on functions that pass parameters to each other, we are essentially imitating dataflow tracking through function arguments. In wasm, all data types will be u32 or u64 integers. We consider u32 parameters, as our memory layout supports u32 pointers. These pointers appear to be verified in a caller function and redundantly verified in the callee function. We consider these three heuristics because they have a low opportunity cost for implementation and offer significant insight into program memory access behavior.

To gauge the performance benefits of dynamic analysis for Wasm programs, we instrument the Dhrystone benchmark. Dhrystone is a synthetic computing benchmark that tests programs meant to mimic common processor usages. Analyzing common processor usages will allow us to generalize our results to real-world Wasm binaries. Creating a dynamic analysis tool also has the added benefit of identifying realistic optimizations when exploring control flow and memory access patterns. We evaluate instrumenting memory accesses with heuristics (1+2) and compare that with heuristics (1+2+3). With heuristics (1+2), we identify all pointers that are redundantly verified temporally, and we can compare that with heuristics (1+2+3) to determine the difference between the ideal and the realistic.

5 Implementation

5.1 Computer Specifications

We use a combination of UT Lab computers and personal computers to implement the end-to-end process of instrumentation.

Linux Machines (Lab Computers)

- Operating System: Ubuntu Linux
- Architecture: x86_64

Personal Machines

- Operating System: MacOS Sequoia 15.1.1 (24B91)
- Architecture: x86_64

5.2 Satisfying Redundant Memory Bounds Check Heuristics

To satisfy heuristic (1), we begin with creating a call graph that determines caller-callee relationships. We use GNU CFlow that takes in the C source file that is generated by wasm2c and outputs a call graph. The call graph that is generated visualizes function signatures and caller-callee relationships. We are interested in functions that pass parameters to each other, as they are potentially pointers.

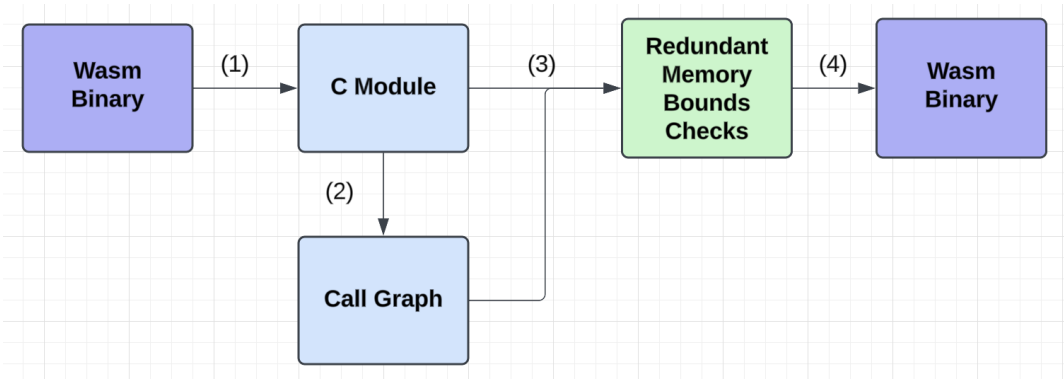

Figure 1: (1) We start by generating instrumented C source and header files with a modified wasm2c. (2) We use GNU Cflow to create a call graph from the generated C module. (3) Using the call graph as an input, we run the instrumented C module and that will save the memory accesses to an output file and display the number of redundant memory bounds checks. (4) We convert the generated C module back to Wasm.

We created a Python script that parses the call graph and outputs the results into a simplified call graph that has one caller separated by a comma separated list of callees per line. When running the instrumented code, the we first instantiate a call graph, and this allows us to determine caller-callee relationships efficiently.

To satisfy heuristic (2), we need to determine if a memory check has been called recently. We rely on tracking every memory access in a `MemoryInfo` struct. The struct stores the pointer that has been validated, what function it was last validated in, and how many times it has been deemed redundantly validated. Prior to every load or store, we check if the pointer being accessed has been verified in the same function or is a callee of the function that last verified it.

To satisfy heuristic (3), we need to determine if the current pointer that is being accessed is a parameter that was validated in the caller function. Heuristic (2) satisfies that the pointer was verified in the caller function. However, it could be the case that this pointer is the product of arithmetic operations or a parameter assignment to a local variable. Performing pointer arithmetic on a pointer introduces additional complexity. This can obscure the relationship of the pointer. We are interested in the case where the program creates a copy of the parameter and uses the copy.

Ensuring that the value of the pointer remains within the bounds of wasm's linear memory can be determined with static analysis techniques like symbolic execution, but that is out of scope for this paper. Instead, we create a map that stores parameter values. During every functions' `FUNC_PROLOGUE`, these parameter values are stored with a composite key formed by the function name, variable name, and current stack depth. Without an efficient key generating method, we found this could create a bottleneck in the dynamic analysis tool and prevent any meaningful programs from being run. We use FNV-1a, a non-cryptographic hash function that is able to take in these inputs to create a key. FNV-1a is designed to be fast and the likelihood of collisions is rare [2]. This is a coarse approximation of tracking pointer variables, which may introduce false positives (higher number of redundant checks). We conjecture that the likelihood of accessing a pointer equal to the parameter but was not passed through the parameter is relatively rare, but this remains to be verified. Whenever a pointer is accessed, we can check if the pointer is equal to one of the parameters. During the `FUNC_EPILOGUE`,

the parameters that were stored are freed.

Finally, we instrument all memory accesses to increment a global counter for memory accesses and sum the redundant bounds checks for all pointers to determine the total number of redundant bounds checks.

5.3 Emscripten

As a final step, we demonstrate that it is possible to convert C code generated by wasm2c back into Wasm binaries with Emscripten. We created two simple Wasm programs, fibonacci and matrix multiplication. Little to no changes were made to the transpiled C modules, and the programs performed correctly when converted back to wasm. This highlights the flexible nature of wasm2c as a transpiler. It generates code that can be analyzed and instrumented at a high level.

6 Evaluation

Criterion 1+2	
Clean rechecks	552,080,105
Total memory checks	675,178,741
Percentage repeated	**81.77%**
Overhead	**5,917%**

Criterion 1+2+3	
Clean rechecks	21,770,370
Total memory checks	332,778,455
Percentage repeated	**6.54%**
Overhead	**9972%**

Figure 2: Dhrystone metrics

The memory instrumentation introduces overheads of 5,917% to 9,972% when running the analysis tool with heuristics (1+2) and heuristics (1+2+3), respectively. This is slightly misleading, as this profiling tool need only be run when initially determining opportunities for optimization. Moreover, the time complexity of the profiling tool empirically appears to remain linear with a larger constant.

81.77% and 6.54% of memory bounds checks are redundant for heuristics (1+2) and heuristics (1+2+3), respectively. The staggering number of redundant memory bounds checks for heuristics (1+2) makes sense, as the memory operations tend to be performed on a subset of the same 8GiB sandbox. However, this number drops when adding heuristic (3) to a more modest 6.54% and increased the overhead by 68.53%. This demonstrates that considering parameter values is the most stringent heuristic. 6.54% is a non-trivial amount of redundant memory bounds checks, and the overhead refines the results into identifiable memory bounds checks. These are the low-hanging fruit that do not require a complex static analysis tool to detect.

7 Related Work

Memory safety remains an important goal in security, as papers like Softbound and CET provide software enhancements that can offer memory safety[10, 11]. These are similar to Wasm's practice of inserting conditional branches into its code to validate every memory access. SIMBER is a tool that removes redundant memory bound checks for SoftBound via statistical inference[16]. Rather than statically analyzing possible redundant accesses, it uses previous runs to statistically determine redundant checks with 0 false positives. WPBound optimizes SoftBound by removing redundant memory bounds checks for for-loops[13]. Memory accesses and their associated ranges can be determined by their weakest precondition, which decreased the average runtime overhead from 77% to 47%. There have been existing projects that work on removing redundant bounds checks in Wasm binaries statically and dynamically[5]. Our work differs in that we are analyzing redundant memory accesses in the C code generated by wasm2c.

Though supporting guard pages in 64-bit systems has typically been a difficult task, recently there have been proposals to make them usable for 64-bit systems through implementations like two-level guard pages, which combine traditional guard pages with an additional macro guard region, enabling branchless access checks[3]. Preliminary results have shown drastic improvement over software-based bounds checks, incurring a maximum overhead of 17.3%. It is worth noting that this approach is implemented in Umbra's Wasm runtime, which differs from the environment our work is tested in, and a majority of runtimes still rely on software-based bounds checks.

Other approaches have included using built-in hardware features to help relieve the overhead incurred by these checks. CAGE is a hardware-accelerated WASM toolchain leveraging ARM's MTE and PAC hardware extensions to provide both spatial and temporal memory safety issues for unmodified C/C++ programs. To improve sandboxing performance, it attempts to offload the bounds checks to MTE hardware, incurring runtime and memory overheads under 5.8% and 3.7% respectively while accelerating Wasm's sandboxing mechanisms by over 5.1%. However, as is the case with many hardware-accelerated approaches, this approach still requires a software-based fallback on unsupported hardware [4].

8 Future Work

8.1 Dynamic Analysis Optimizations

The dynamic analysis tool is limited by its coarse heuristics. Currently, we only analyze pointers passed through parameters in caller-callee relationships. We plan to incorporate analyzing pointers returned by back edges in the program's call graph. Our current approach is prone to false positives for heuristic (3). This occurs because we are not performing comprehensive variable tracking—focusing instead on the runtime values of pointers. By extending the tool to track variable assignments and apply symbolic execution, it may be possible to refine the analysis and further reduce false positives. Moreover, testing on real-world Wasm binaries, as opposed to synthetic programs, could provide a more accurate assessment of the tool's effectiveness in practical scenarios.

8.2 Static Analysis Tools

Despite these opportunities for improvement, there are diminishing returns in creating a more detailed dynamic analysis tool. Synthetic programs suggest that the upper bound of redundant memory checks

is approximately 6.54%. Consequently, the next logical step would be to implement a static analysis tool capable of tracking parameter values, analyzing caller-callee relationships, and guiding the elimination of excess memory bounds checks during code generation.

8.3 Creating a Transpilation Pipeline

We have only converted relatively simple programs, such as Fibonacci computations and matrix multiplication, between Wasm and C using tools like wasm2c and Emscripten. This pipeline demonstrates the feasibility of intermediary optimizations between Wasm and C. It offers opportunities to extend its applicability to more complex and diverse programs. As the number of Wasm binaries continues to grow in the wild, such workflows can play a crucial role in the Wasm ecosystem's maturation. By simplifying and improving the transpilation process, we can support the adoption of Wasm across a broader range of applications and industries.

9 Conclusion

This paper explores the potential of identifying and mitigating redundant memory bounds checks in WebAssembly binaries through a dynamic analysis approach. By analyzing the Dhrystone benchmark with our tool, we demonstrate that up to 6.54% of memory bounds checks may be unnecessary. This indicates an opportunity for optimization with a static analysis tool. Such a tool could significantly enhance the efficiency of transpiled Wasm code while maintaining security guarantees, providing benefits for both developers and the broader Wasm ecosystem. Future work will involve applying these techniques to real-world Wasm binaries to better understand their impact and refine the transpilation pipeline to offer seamless optimization.

References

[1] CVE-2023-26489. Available from NIST National Vulnerability Database, CVE-ID CVE-2023-26489, March 2023.

[2] Stack Exchange Community. Which hashing algorithm is best for uniqueness and speed? https://softwareengineering.stackexchange.com/questions/49550/which-hashing-algorithm-is-best-for-uniqueness-and-speed/145633145633, Jan. 2011. Accessed: 2024-12-04.

[3] Lukas Döllerer and Alexis Engelke. Performant bounds checking for 64-bit webassembly. In *Proceedings of the 16th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, VMIL '24, page 23–31, 2024.

[4] Martin Fink, Dimitrios Stavrakakis, Dennis Sprokholt, Soham Chakraborty, Jan-Erik Ekberg, and Pramod Bhatotia. Cage: Hardware-accelerated safe webassembly, 2024.

[5] Firefox. Bugzilla 1282618: Wasm: Implement a simple redundant bounds check elimination pass. https://bugzilla.mozilla.org/show$_b$ug.cgi?id = 1282618, $June$2016.

[6] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. *SIGPLAN Not.*, 52(6):185–200, June 2017.

[7] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.

[8] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. New kid on the web: A study on the prevalence of webassembly in the wild. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 23–42, 2019.

[9] n.a. Webassembly specification. https://webassembly.github.io/spec/core/, Dec. 2024.

[10] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Softbound: highly compatible and complete spatial memory safety for c. *SIGPLAN Notices*, 44(6):245–258, 2009.

[11] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Cets: Compiler enforced temporal safety for c. *SIGPLAN Notices*, 45(8):31–40, 2010.

[12] Nikita Solodkov and Brayton Noll. The state of webassembly 2023. Technical report, Cloud Native Computing Foundation, 2023.

[13] Y. Sui, D. Ye, Y. Su, and J. Xue. Eliminating redundant bounds checks in dynamic buffer overflow detection using weakest preconditions. *IEEE Transactions on Reliability*, 65(4):1682–1699, Dec. 2016.

[14] Raven Szewczyk, Kimberley Stonehouse, Antonio Barbalace, and Tom Spink. Leaps and bounds: Analyzing webassembly's performance with a focus on bounds checking. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*, pages 256–268, 2022.

[15] WebAssembly. Webassembly. https://webassembly.org/, Oct. 2024. Accessed: 2024-10-27.

[16] H. Xue, Y. Chen, F. Yao, Y. Li, T. Lan, and G. Venkataramani. Simber: Eliminating redundant memory bound checks via statistical inference. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 413–426, 2017.