SafeRace: Assessing and Addressing WebGPU Memory Safety in the Presence of Data Races

REESE LEVINE, University of California at Santa Cruz, USA
ASHLEY LEE, University of California at Santa Cruz, USA
NEHA ABBAS, University of California at Santa Cruz, USA
KYLE LITTLE, University of California at Santa Cruz, USA
TYLER SORENSEN, Microsoft, USA and University of California at Santa Cruz, USA

In untrusted execution environments such as web browsers, code from remote sources is regularly executed. To harden these environments against attacks, constituent programming languages and their implementations must uphold certain safety properties, such as memory safety. These properties must be maintained across the entire compilation stack, which may include intermediate languages that do not provide the same safety guarantees. Any case where properties are not preserved could lead to a serious security vulnerability.

In this work, we identify a *specification vulnerability* in the WebGPU Shading Language (WGSL) where code with data races can be compiled to intermediate representations in which an optimizing compiler could legitimately remove memory safety guardrails. To address this, we present SafeRace, a collection of threat assessments and specification proposals across the WGSL execution stack. While our threat assessment showed that this vulnerability does not appear to be exploitable on current systems, it creates a "ticking time bomb", especially as compilers in this area are rapidly evolving. Given this, we introduce the SafeRace Memory Safety Guarantee (SMSG), two components that preserve memory safety in the WGSL execution stack even in the presence of data races. The first component specifies that program slices contributing to memory indexing must be race free and is implemented via a compiler pass for WGSL programs. The second component is a requirement on intermediate representations that limits the effects of data races so that they cannot impact race-free program slices. While the first component is not currently possible to apply to all WGSL programs due to limitations on how some data types can be accessed, we show that existing language constructs are sufficient to implement this component with minimal performance overhead on many existing important WebGPU applications. We test the second component by performing a fuzzing campaign of 81 hours across 21 compilation stacks; our results show violations on only one (likely buggy) machine, thus providing evidence that lower-level GPU frameworks could relatively straightforwardly support this constraint. Finally, our assessments discovered GPU memory isolation vulnerabilities in Apple and AMD GPUs, as well as a security-critical miscompilation of WGSL in a pre-release version of Firefox.

CCS Concepts: • **Security and privacy** → *Web application security*; • **Software and its engineering** → **Semantics**; **Compilers**; • **Computing methodologies** → **Parallel programming languages**.

Additional Key Words and Phrases: data races, memory safety, GPU programming languages

ACM Reference Format:
Reese Levine, Ashley Lee, Neha Abbas, Kyle Little, and Tyler Sorensen. 2025. SafeRace: Assessing and Addressing WebGPU Memory Safety in the Presence of Data Races. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 297 (October 2025), 29 pages. https://doi.org/10.1145/3763075

Authors' Contact Information: Reese Levine, University of California at Santa Cruz, Santa Cruz, USA, relevine@ucsc.edu; Ashley Lee, University of California at Santa Cruz, Santa Cruz, USA, allilee@ucsc.edu; Neha Abbas, University of California at Santa Cruz, Santa Cruz, USA, neabbas@ucsc.edu; Kyle Little, University of California at Santa Cruz, Santa Cruz, USA, kyle.little@utah.edu; Tyler Sorensen, Microsoft, Seattle, USA and University of California at Santa Cruz, Santa Cruz, USA, tyler.sorensen@ucsc.edu.

ACM 2475-1421/2025/10-ART297
https://doi.org/10.1145/3763075

```
i: array<u32, 1>, m: array<u32, 4>
```

Thread 0	Thread 1
a: `i[0] = 2`	c: `i[0] = 4`
`... // some computation`	
b: `let r0 = m[min(i[0], 3)]`	

(a) Program with a data race on `i[0]`, which is used to index into `m`.

```
i: array<u32, 1>, m: array<u32, 4>
```

Thread 0	Thread 1
a: `i[0] = 2`	c: `i[0] = 4`
`var x = 2`	
b: `let r0 = m[min(x, 3)]`	

(b) Program with a data race on `i[0]`, with `x` used to index into `m`.

Fig. 1. WGSL compilers insert dynamic bounds-checks (shown in red) to provide memory safety. We show that under current specifications the bounds-checks can be optimized away in either program, leading to potential memory safety violations.

1 Introduction

A programming language is safe with respect to some property if it guarantees that property will hold throughout the entire compilation process, and thus, during execution. One important property is *memory safety*, which prevents behaviors such as out-of-bounds and use-after-free accesses. This property is especially important as some of the most notorious exploits and vulnerabilities found in recent decades are related to memory safety [17–19]; furthermore, large-scale corporate studies have found that the majority of all vulnerabilities are related to memory safety [8, 11].

Memory isolation vulnerabilities, in which a process can accesses the memory of another process, are one class of vulnerabilities that can occur when memory safety is violated. The risk of memory isolation vulnerabilities has led to the development of memory-safe languages, including languages such as JavaScript and WebAssembly (Wasm) for web browsers. At the same time, runtimes, e.g., operating systems, have developed protections to maintain memory isolation, e.g., exceptions and segmentation faults, even when executing programs written in memory-unsafe languages like C++.

However, due to their lean (and performance oriented) implementations, GPUs might not have these defensive runtimes, nor other mechanisms that provide memory protection. Furthermore, because essentially all modern consumer devices allow multiple processes to share access to the GPU, these devices are increasingly receiving attention in the security community; including the discovery of several notable cases of memory isolation vulnerabilities [65, 79]. Running GPU programs in web browsers amplifies security risks even further, as web pages can execute untrusted code and thus, may maliciously try to access data from other tabs or applications.

Despite these risks, GPUs enable many important applications on the web, such as AI where a single GPU can now run LLM inference [24, 77, 92]. WebGPU, the successor to WebGL, is a framework that brings first-class support for general-purpose GPU computation to browsers. WebGPU programs are written in a high-level language, WGSL, which is documented to enforce memory safety. However, this guarantee is not straightforward, as WGSL is not directly translated into GPU machine code; instead, WGSL programs are translated to lower-level native GPU languages before being compiled to vendor-specific (and sometimes closed-source) GPU machine code. Therefore, WGSL relies on lower-level compilers and runtimes to maintain safety properties during optimization passes and execution.

If a valid implementation of a programming language specification can break a language's safety property, we say that there exists a *specification vulnerability*. This work identifies a *data race specification vulnerability* (DRSV) in WGSL and discusses how it can be assessed and addressed.

1.1 WGSL Data Race Specification Vulnerability

In many languages, e.g., C and C++, data races cause undefined behavior, as they can lead to extremely counterintuitive outcomes. However, because these behaviors are not desirable in security-sensitive contexts (e.g., browsers), languages such as JavaScript and Wasm have defined data races, albeit with very permissive semantics [84, 85]. Similarly, WGSL utilizes compiler passes and safe libraries to avoid many types of undefined behavior. However, the languages which WGSL targets are *not* free of undefined behavior, and in particular follow C/C++ in giving data races undefined behavior. A data race occurs when multiple threads concurrently access the same memory location without proper synchronization. Modern programming languages have adopted special primitives, e.g., *atomic* in C++, and *volatile* in Java, to safely signal when a memory location may be accessed by multiple threads, with their precise semantics defined in formal memory models [5, 56].

WGSL derives its memory model from C++, with memory accesses specified as either atomic or non-atomic. Figure 1a shows a WGSL program with two arrays of unsigned integers, `i` and `m`, and two threads performing non-atomic accesses to these arrays. This program contains behaviors that, in isolation, are undefined in some languages: (1) because memory locations are non-atomic and there is no synchronization, there are data races between (a, c) and (b, c), and (2) while the race technically makes the program undefined, a possible natural compilation could allow for an out-of-bounds read if b reads the value c wrote to `i[0]`. The potential out-of-bounds read is a memory safety violation which, in turn, could lead to a vulnerability. The highlighted code in operation b, i.e., `min(..., 3)`, shows a check that WGSL compilers include to enforce memory safety. However, WGSL compilers currently contain no special protections against data races.

The Problem: Optimizing Away Bounds-Checks. Even with the bounds-check inserted, a compiler for an unsafe lower-level language can optimize the code in a way that inserts a "time-of-check to time-of-use" (TOCTOU) [2] memory safety violation. Specifically, a compiler for an unsafe language is free to assume programs do not contain data races. Therefore, when only looking at the operations in thread 0, the compiler may notice that `i[0]` is set to 2 and remains unchanged until it is loaded in b. Based on a range analysis, it can determine that the call to `min` is unnecessary and remove it, leading to an out-of-bounds access if `i[0]` reads 4 in b. While it is unlikely a real implementation would perform the problematic optimization on this particular program, in Sec. 4.1 we describe a more realistic program that could trigger this optimization.

The program in Fig. 1a shows straightforwardly how data races can lead to vulnerabilities, but memory safety issues can occur due to other surprising behaviors allowed by data races. In particular, Fig. 1b shows a WGSL program which contains the same data race between a and c, but now the access to `m` in c depends on a separate variable `x`. Prior work shows how reasonable implementations might set `x` to 4–for example if the compiler notices that both `x` and `i[0]` contain the same value (2), and decides to not allocate a register for `x`, reloading `i[0]` instead and causing a potential out-of-bounds access [26].

To recap, WGSL code, i.e., Fig. 1 without the `min`, is written by application developers, and may contain out-of-bounds accesses and data races. To avoid out-of-bounds accesses, WGSL compilers insert bounds-checks when generating lower-level native GPU code. The WGSL specification also mandates that if a data race occurs, memory accesses should be restricted to in-bounds memory in order to maintain memory safety [83, Sec. 2.2], but WGSL compilers do not include any special protections against data races. However, native GPU languages currently give undefined semantics to data races, allowing the optimizations described above when compiling to vendor-specific machine code. This mismatch between the WGSL specification and the native language guarantees means that the bounds-checks can be removed despite the fact that they should not be to maintain memory safety, creating the WGSL DRSV.

The Solution: Constraining Optimizations. The optimizations described above must be disallowed so that the bounds-checks remain, i.e., to maintain memory safety. In Fig. 1a, this can be achieved by specifying that all accesses to `i[0]` be *atomic*. Atomic accesses signal to the compiler that multiple threads might modify `i[0]`, preventing it from making the assumption that `i[0]` remains unchanged between *a* and *b*. In Fig. 1b, the compiler must be prevented from using the value in `i[0]` to refer to `x`. While this could be accomplished by again making accesses to `i[0]` atomic, extending this principle to full programs would require making *every* memory access atomic, causing unacceptable performance decreases for many applications. Rather, we propose two properties which ensure memory safety violations do not occur due to data races while making a minimal set of accesses atomic. We call these properties the SafeRace Memory Safety Guarantee (SMSG).

The first property of SMSG specifies that slices of programs which contribute to memory indexing must be race-free, and can be applied in Fig. 1a by making accesses to `i[0]` atomic. In more complex code, multiple accesses may contribute to the calculation of the index into `m`. In these cases, it is not necessary to do the complete index calculation atomically, i.e., by loading all memory and performing arithmetic indivisibly. Rather, it is sufficient to ensure that each sub-access which contributes to the calculation is done atomically, using a backward slicing algorithm which considers all control, index, and data dependencies of an access. In Sec. 5, we show formally how this avoids data races on the full index calculation and ensures memory safety.

The second property of SMSG specifies that data races cannot affect the behavior of race-free program slices, preventing the data race on `i[0]` from affecting the index into `m` in Fig. 1b. The first property can be implemented as a compiler pass, while the second property must be adopted by intermediate representations which WGSL targets. In Sec. 5 we formally define both properties, and in Sec. 6 and Sec. 7 we evaluate the performance and practical impacts of adopting them.

1.2 SafeRace: Threat Assessments and Specification Proposals

Given the potential for attackers to exploit the specification vulnerability outlined above, we present SafeRace, a thorough investigation into how data races can affect memory safety properties when compiling WGSL (a safe language) to lower-level unsafe languages. We perform threat assessments that test native frameworks and WebGPU implementations for memory isolation vulnerabilities and introduce specification proposals which ensure WGSL's memory safety in the presence of data races. Our results provide a framework which can inform the development of other languages with similar compositional designs.

Threat Assessment: Bottom-Up (Sec. 3). First, we assess the potential for programs with simple memory safety violations, e.g., out-of-bounds accesses without bounds-checks, to cause vulnerabilities. We conduct our assessment in two native GPU frameworks: Vulkan, which runs on Linux and Android operating systems, and Metal, which runs on macOS and iOS. We test 11 GPUs from 6 vendors and discover two new memory isolation vulnerabilities, one due to uninitialized memory on AMD GPUs and the other due to out-of-bounds accesses on Apple GPUs.

Threat Assessment: Top-Down (Sec. 4). We develop a fuzzer, WGSLMemSmith, based on WGSL-Smith [47], to test WGSL implementations for the DRSV, i.e., the data race specification vulnerability described above. WGSLMemSmith generates random programs that contain potential memory safety violations due to combinations of data races and out-of-bounds accesses in order to determine whether WGSL's DRSV can be triggered by existing compilers. We run WGSLMemSmith on 21 unique compilation stacks, i.e., combination of browser, operating system, and GPU vendor, and show that the specification vulnerability is either not implemented or is unlikely to be triggered.

Despite our negative result, the vulnerability remains a ticking time bomb, motivating the development of the specification proposals introduced below. Additionally, our fuzzing finds a

simpler memory isolation vulnerability due to missing bounds-checks in a pre-release version of Firefox; this vulnerability has been confirmed and fixed by Mozilla developers.

SafeRace Memory Safety Guarantee (Sec. 5). To fix the WGSL DRSV, we introduce the SafeRace Memory Safety Guarantee (SMSG), a contract between a higher-level memory-safe language and lower-level unsafe languages. SMSG consists of two properties: (1) program slices which contribute to memory indexing, e.g., events *a*, *b*, and *c* in Fig. 1a, must be race-free, and (2) data races cannot affect the behavior of race-free slices of programs.

Property 1 can be enforced by an initial pass in WGSL compilers which ensures a program slice is race-free by translating some non-atomic memory accesses into atomic accesses. We implement Property 1 in Tint, the compiler for WGSL programs included in Chromium-based browsers. Some data types in WGSL, e.g., floats, cannot currently be accessed atomically, so our implementation cannot apply to all possible WGSL programs, e.g., a program with a conditional on floating point values to determine the index into a memory region. However, our experimental results (Sec. 6) show that for a set of important benchmarks, no data types which cannot be accessed atomically are affected by Property 1. We also show that Property 1 has negligible performance impact on many existing WebGPU applications, including AI inference and general compute applications, with the only observed impact being a 10% performance overhead in a sorting GPU kernel. Property 2 ensures that data races do not affect race-free slices of programs and must be incorporated into the frameworks targeted by WGSL, i.e., currently Vulkan, Metal, and DirectX. We augment WGSLMemSmith with the ability to perform metamorphic fuzzing between programs with and without data races and find violations of Property 2 on only one (likely buggy) machine, providing evidence that these frameworks could relatively straightforwardly adopt this constraint (Sec. 7).

Applications Beyond WebGPU (Sec. 8). While the main body of this work focuses on WGSL, the ideas expressed in our threat assessments and specification proposals apply to other parallel programming languages. An increasing number of modern languages are being developed that strive to maintain memory safety through combinations of features like bounds-checked accesses and ownership of data. At the same time, there is a proliferation of tools that can transpile one language to another, and new processors and accelerators with diverse architectures are being developed. This increases the likelihood that layered compilation stacks will become more prevalent, where safety properties must be evaluated across source and target languages.

Contributions. In summary, our contributions are:

- A bottom-up threat assessment that shows memory safety violations can straightforwardly lead to memory isolation failures in many GPU frameworks; resulting in the discovery of two new GPU isolation vulnerabilities (Sec. 3).
- A top-down threat assessment using WGSLMemSmith to evaluate whether the WGSL DRSV is currently exploitable. We show that the DRSV is likely *not* exploitable today, while also discovering a memory isolation vulnerability in a pre-release version of Firefox (Sec. 4).
- The SafeRace Memory Safety Guarantee (SMSG), two properties that when combined provide memory safety in the presence of data races in WGSL programs (Sec. 5).
- An implementation of SMSG Property 1 as a WGSL compiler pass and an evaluation that shows it negligibly impacts the performance of a set of WebGPU applications (Sec. 6).
- A metamorphic fuzzing approach implemented in WGSLMemSmith which finds that SMSG Property 2 is empirically supported by most systems we tested (Sec. 7).

Our artifact for this work is publicly available [51]. As part of responsible disclosure, we have presented this work to the W3C WebGPU working group and reported our vulnerabilities to the affected vendors.

2 Background

2.1 GPU Execution Model

GPU frameworks consist of a host API that executes on the CPU and a programming (or *shading*) language which executes on the GPU. Applications allocate resources, transfer data, and schedule execution using the host API, while shaders are dispatched to run on the GPU asynchronously. We focus on *compute* shaders in this paper, which perform general-purpose computation on GPUs. GPU execution models are hierarchical, with several execution scopes and memory address spaces. Below, we outline the execution scopes defined in WebGPU and how they relate to different address spaces. While this list is not exhaustive (especially when considering vendor-specific frameworks, such as CUDA), it is sufficient for this work.

- *Thread*: The base unit of computation, executing a stream of instructions[1]. The *private* address space is local to a single thread.
- *Workgroup*: Threads are organized into sets called workgroups (called thread blocks in CUDA). All threads in a workgroup share access to the *workgroup* address space (called shared memory in CUDA). The size of this memory is fixed at shader launch time.
- *Device*: The widest scope of execution on a single GPU, consisting of all threads that execute as part of the same shader dispatch, i.e., the grid of threads. All threads can access device memory, which CUDA calls global memory and WebGPU calls the *storage* address space, through fixed-length buffers which are allocated and initialized before execution starts.

WebGPU. WebGPU is a high-level GPU framework meant for execution in the browser. To support portable execution, it must target several lower-level frameworks (and their shader languages): DirectX (HLSL) on Windows [60], Vulkan (SPIR-V) on Linux and Android [37], and Metal (MSL) on macOS and iOS [3]. There are currently three major runtimes being developed for WebGPU: Dawn [1] for Chromium based browsers, wgpu [55] for Firefox, and a WebKit implementation for Safari [87]. Some browsers, e.g., Chrome, run GPU commands from all tabs on one process, foregoing any process isolation properties provided by operating systems [67].

2.2 Memory Consistency Models and Data Races

A memory consistency model (MCM) defines the allowed behaviors of shared memory parallel programs. Both hardware models [32, 64] and language models [5, 56] have been developed, with language models also specifying what constitutes a data race. In the WGSL MCM, memory locations are either atomic or non-atomic. Memory locations cannot be type cast, so it is not possible to non-atomically access an atomic location, like it would be in other languages, e.g., C++. Memory operations access sets of 8-bit memory locations. A data race occurs under the following conditions: two threads perform non-atomic operations on the same set of memory locations, with at least one being a write and without proper synchronization. On the other hand, threads are allowed to concurrently perform atomic operations on the same set of memory locations without causing data races. While prior work has shown how *mixed-size* non-aligned accesses may lead to data races on atomic locations [33], WebGPU's memory layout restrictions ensures this cannot occur.

Atomic operations and synchronization operations, e.g., workgroup barriers, can be used to constrain allowed behaviors and avoid data races. However, data races in WGSL are a global property, so a race *anywhere* in a program poses risks. Therefore, in this work, we focus only on synchronization-free fragments of programs and analyze memory safety in the presence of non-atomic operations and non-synchronizing atomic operations. Non-synchronizing atomic operations are called *relaxed* atomic operations and are available in WGSL and its backend targets.

[1]In WebGPU, a thread is called an *invocation*, but we use *thread* as it is the more general term in parallel programming.

Prior work has developed semantics that encompasses the common use cases of relaxed atomics and proved the soundness of compiler transformations like merging or removing atomic operations [78, 81]. In particular, the semantics of relaxed atomics specify that an operation must happen indivisibly, i.e., it cannot be split into separate operations. Additionally, the value of a relaxed atomic load may not be assumed, prohibiting optimizations such as a rematerializing a load from memory if the compiler has already optimized based on some value for that load. These operations and restrictions apply in both WGSL and the languages that it targets.

Some languages, e.g., C++, also include *volatile* accesses, which must be treated by compilers as having visible side-effects. However, concurrent volatile accesses are still considered a data race (and hence are undefined), and other arguments against using volatile to solve concurrency problems exist [5]. Further, WGSL lacks volatile, and SPIR-V disallows it under the Vulkan memory model, so in this work we use atomic accesses to avoid data races.

2.3 Sources of Memory Safety Violations

WGSL shaders can include behaviors that are normally undefined in low-level languages, including data races, uninitialized or out-of-bounds accesses, and undefined arithmetic. If left unchecked, these operations can lead to unexpected outcomes, e.g., crashes or security vulnerabilities. To mitigate these issues, WebGPU explicitly provides well-defined semantics to many of these behaviors through straightforward compiler transformations, e.g., always initializing variables and inserting checks to avoid undefined arithmetic. Other behaviors, including data races and out-of-bounds accesses, are classified as *dynamic errors*. If a dynamic error occurs during shader execution, WebGPU specifies that memory accesses should affect only memory allocated for that shader [83, Sec. 7.3].

Out-Of-Bounds Accesses. WGSL ensures memory safety by either (1) relying on underlying framework guarantees; (2) preemptively clamping memory indexes; or, (3) inserting dynamic checks that avoid executing the memory access if it would occur out-of-bounds. For example, DirectX guarantees that out-of-bounds reads on device memory in HLSL must return 0, while out-of-bounds writes are discarded [58]. Vulkan provides an optional feature called *robust buffer access* that enables the same protections as DirectX on device memory in SPIR-V at some cost to performance [39]. However, both DirectX [59] and Vulkan [38, Sec. 3.52.8] leave the behavior of out-of-bounds accesses in workgroup and private memory undefined. Metal, by not describing their behavior, implicitly leaves all out-of-bounds accesses undefined [42].

In a program free of data races, the combination of underlying guarantees and inserted bounds-checks are enough to provide memory safety. However, the presence of data races raises additional complications. In fact, one of the possible outcomes of clamping an out-of-bounds access *is a data race*. For example, consider a program where thread 0 writes to index 4 of an array A of size 5, while thread 1 erroneously tries to read index 5 of A. If out-of-bounds accesses are avoided by clamping indexes to the size of A, thread 1 will now read index 4 of A, creating a data race with thread 0's write to index 4. Therefore, it is necessary to consider the effects of data races on all programs with potential out-of-bounds accesses.

Data Races. Beyond classifying data races as dynamic errors, the WGSL specification is ambiguous about how they affect program behavior, a fatal flaw due to data races causing undefined behavior in underlying frameworks. Vulkan is the only one of the three native frameworks with a publicly available formal memory model, which enforces that programs must be free of data races to be well-defined [37, App. B]. Metal is based on the C++14 specification, which specifies that data races lead to undefined behavior [15]. DirectX does not describe the consequences of data races at all, implicitly leaving their behavior undefined.

Table 1. Results from our bottom-up threat assessment. We find two new vulnerabilities, highlighted in red, both which affect memory isolation when applications are run in different processes.

			Memory Region:	**Device**		**Phys. Storage**		**Workgroup**	
Framework	**OS**	**Device**	**Process:**	*same*	*diff*	*same*	*diff*	*same*	*diff*
Vulkan	Linux	AMD Radeon RX 7900 XT		✓	✓	✣	✓	✗	✗
		AMD Ryzen 7 5700G		✗	✗	✣	✓	✓	✓
		Intel Arc A770 Graphics		✓	✓	✣	✓	✓	✓
		Intel UHD Graphics 770		✓	✓	✣	✓	✓	✓
		NVIDIA GeForce RTX 4070		✓	✓	✣	✓	✣/✗	✓
	Android	Arm Mali-G710		✗	✓	✗	✓	✓	✓
		Qualcomm Adreno 610		✓	✓	–	–	✗	✓
		Qualcomm Adreno 740		✓	✓	–	–	✓	✗
Metal	macOS	Intel Iris Plus Graphics		✓	✓	–	–	✓	✓
		Apple M2		✣	✣	–	–	✗	✗
		Apple M3		✣	✓	–	–	✓	✓

✓= No Vulnerability ✣= Out-of-bounds Access Vulnerability ✗= Uninitialized Memory Vulnerability

3 Threat Assessment: Bottom-Up

We first motivate memory safety in WGSL by performing a bottom-up threat assessment; that is, we aim to explore whether memory safety violations lead to memory isolation vulnerabilities in WGSL target languages. As GPUs can be utilized by multiple concurrent processes on the same machine, any memory isolation failure that leaks data between processes is cause for concern.

Our assessment covers two frameworks and three operating systems, Vulkan (Linux and Android) and Metal (macOS), as well as three address spaces exposed by WebGPU: device, workgroup, and private memory. Since many browsers run WGSL programs from different tabs on the same process, we test for memory isolation both when GPU programs are running in different processes and in the same process. This assessment runs two programs concurrently—a victim and an attacker—and tests for memory isolation failures due to uninitialized memory and out-of-bounds accesses.

The victim allocates a block of memory, with the size dependent on the type of memory being tested. Then, the victim launches many GPU threads, with each thread assigned a set of memory locations within the block of memory. Each thread alternately writes a canary value and a secret value to even and odd memory locations, respectively. Writing a canary allows us to avoid false positives where the attacker spuriously reads the secret value and tests whether the attacker is able to read consecutive out-of-bounds memory values. The victim's memory is continually re-allocated, either automatically in the case of private and workgroup memory or manually in the case of device memory. The victim runs indefinitely until manually stopped.

Like the victim, the attacker allocates a block of memory, and assigns each thread a set of memory locations. However, the memory locations assigned to each thread of the attacker may be either in or out-of-bounds of the allocated block of memory. The attacker threads iterate over their assigned memory locations and search for the canary, which if found in-bounds indicates that the allocated memory was previously used by the victim and not cleared properly, or if found out-of-bounds indicates that the attacker is able to read real data outside its allocated block. If the canary is found, the attacker records the value of the next memory location and the results are analyzed to determine whether the secret value was found.

Generally, GPU memory references cannot be manipulated using integer arithmetic, e.g., like pointers in C-like languages, which restricts the ability to write certain algorithms, e.g., arbitrary tree structures. However, Vulkan has introduced a new type of memory, *physical storage buffers*, allowing applications to pass a virtual 64-bit address directly to a shader that can be dereferenced and manipulated similarly to a pointer. The memory safety implications of physical storage buffers

is not well understood, so we assess both device memory using traditional buffer bindings and physical storage buffers in Vulkan.

3.1 Evaluation

Table 1 shows the results of our threat assessment. Overall, we test 11 GPUs from 6 vendors using 2 frameworks and 3 operating systems. Every GPU except the Intel devices show a memory isolation failure. We do not show results from private memory, as we observed no failures in this region.

3.1.1 Same-Process Mode. This mode follows the setup in many browsers, where GPU programs are run in the same process. Failures observed in this mode are not considered vulnerabilities by vendors, as memory isolation is not guaranteed between applications in the same process. However, in the browser these failures would allow an attacker to read data from another tab if bounds-checks were optimized away, highlighting the danger posed by WGSL's specification vulnerability.

Both the AMD Ryzen 7 5700G, an integrated GPU, and the Arm Mali-G710, a mobile GPU, had uninitialized device memory isolation failures, while the Apple M2/M3 GPUs had out-of-bounds access isolation failures. Every device we tested that supports physical storage buffers also had failures. While WebGPU does not support physical storage buffers, our results underscore the need for caution when adopting new GPU features. In workgroup memory, the AMD Radeon RX 7900 XT, NVIDIA GeForce RTX 4070, Qualcomm Adreno 710, and Apple M2 had failures. Memory isolation failures due to uninitialized workgroup memory was reported by prior work [79], which ran tests in a different-process mode. Our results show that some GPUs which are susceptible to this attack in the different-process mode also are susceptible in the same-process mode. The NVIDIA GPU is the only device which had failures due to out-of-bounds accesses in workgroup memory.

3.1.2 Different-Process Mode. We also test memory isolation of GPU frameworks in a different-process mode, as an attacker running code from the browser might also attempt to steal data from other applications running on the same machine. In device memory, we observe failures on two GPUs: the AMD Ryzen 7 5700G and the Apple M2. These failures can be exploited by a malicious process on the same machine to leak data from victim applications or construct covert channels. We describe each of these vulnerabilities below. Additionally, we observe uninitialized workgroup memory isolation failures on the AMD Radeon RX 7900 XT, Qualcomm Adreno 740, and Apple M2, replicating results from previous work [79].

AMD Uninitialized Memory Vulnerability. This vulnerability allows an attacker to read uninitialized memory which can contain data written from another process, if the attacker uses AMD's open-source Mesa drivers. We confirmed that multiple AMD-integrated GPUs are affected, both the Ryzen 7 5700G in our study and a Ryzen 5 5600H. This vulnerability was confirmed by AMD, assigned a CVE [21], and a patch for the AMD drivers was released.

Apple M2 Out-Of-Bounds Access Vulnerability. This vulnerability allows an attacker to access data from another process by performing out-of-bounds accesses. The extent of this vulnerability is limited—only reliably leaking one 32-bit value before driver or hardware protections seemed to kick in (perhaps in response to a previous vulnerability that allowed arbitrary access to out-of-bounds memory [20]). Nevertheless, we show that the remaining vulnerability is enough to construct a reliable (albeit slow) covert channel between two processes. This vulnerability, which affects Apple M1 and M2 GPUs, has been reported to and confirmed by Apple.

4 Threat Assessment: Top-Down

Our top-down threat assessment aims to determine whether the WGSL data race specification vulnerability (DRSV) is exploitable today. We perform this assessment through a large-scale fuzzing

```
int min(int a,int b) {return (b<a)?b:a;}

void f(int *restrict idx, int *restrict data, int *restrict output) {
    if (*data == 0)
        *idx = 17;
    else
        *idx = 19;
    if (*idx < 100)
        *output = data[min(*idx, 99)];
}
```

Listing 1. Extension of Fig. 1 showing a C program which exemplifies WGSL's data race specification vulnerability. Pointers are restricted to avoid aliasing, as in WGSL.

```
  %data_0 = load i32, ptr %data, align 4
  %data_is_0 = icmp eq i32 %data_0, 0
  %idx_0 = select i1 %data_is_0, i32 17, i32 19
  store i32 %idx_0, ptr %idx, align 4
  %idx_0_ext = zext nneg i32 %idx_0 to i64
  %data_ptr = getelementptr inbounds i32, ptr %data, i64 %idx_0_ext
  %data_val = load i32, ptr %data_ptr, align 4
  store i32 %data_val, ptr %output, align 4
```

Listing 2. Compiler optimizations can turn the code in List. 1 into this LLVM intermediate representation.

campaign with WGSLMemSmith, an extension of the WGSLSmith fuzzer [47] which we developed. The design of WGSLMemSmith is informed by understanding how compilers might optimize away bounds-checks like the one in Fig. 1. Therefore, we first extend the example from Fig. 1 to examine how valid compiler passes in real implementations might introduce memory safety violations, then show how WGSLMemSmith generates *patterns* which could trigger the specification vulnerability.

4.1 Extended Example

Listing 1 shows C code that implements the same out-of-bounds protections as the GPU example in Fig. 1. We use C here because it allows us to utilize the LLVM compiler project to easily explore optimization passes. In contrast, optimizing compilers for GPUs are either closed source or don't provide tools that allow fine-grained control of optimizations in an easily configurable way. Also, while there are differences in intermediate representations built specifically for GPUs, the concepts necessary to understand the specification vulnerability in WGSL can be explored in LLVM.

In this example, assume that the length of the `data` array is 100. The code in red, i.e., the `min` check, would be added by a compiler for a memory-safe language. The rest of the code would be written by an application developer (in the memory-safe language). In a single-threaded context, there is no possibility of an out-of-bounds access in this code, but in code running on a parallel processor a separate thread might create a data race on `*idx`, writing a value greater than 99.

The LLVM code in List. 2 shows the results of emitting LLVM from the C code using `clang` and running the following LLVM optimization passes using `opt`: `inline`, `mem2reg`, `early-cse`, `instcombine`, and `simplifycfg`. In the resulting code, a register `%idx_0` is allocated that holds the results of the conditional check on the value of `ptr %data` (List. 2 lines 1-3), which corresponds to the conditional on `data` in List. 1 lines 4-7. The value of `%idx_0` is stored to `ptr %idx` (List. 2 line 4), which corresponds to the two stores to `idx` in List. 1 lines 5 and 7. Instead of reloading the

```
Thread 0
a: i[0] = 2
b: let r0 = m[i[0]]
```

(a) Basic Pattern. Thread 1 races on `i[0]`, writing an out-of-bounds value, e.g., 4.

```
Thread 0
i[0] = 500000
let r0 = m[1 + i[0]]
```

(b) Undefined Arithmetic Pattern. Thread 1 writes a value that causes overflow, e.g., 2 billion.

```
Thread 0
i[0] = 0
var x = 0
if (i[0] > 0) x = 4
let r0 = m[x]
```

(c) Control-Flow Pattern. Thread 1 writes a value that causes the conditional to execute, e.g., 1.

Fig. 2. The three types of patterns that WGSLMemSmith generates. Here, `i` is a signed 32-bit integer array of size 1 and `m` is an unsigned 32-bit integer array of size 4 in either device, workgroup, or private memory.

value of `ptr %idx`, the value of `%idx_0` is used (extended to an `i64`) to index into `ptr %data` since the compiler assumes the value of `ptr %idx` has not changed (List. 2 lines 5-6). This also allows the compiler to provably remove the `min` check in List. 1 line 9 when loading from `ptr %data_ptr` (List. 2 line 7). Note that the conditional in List. 1 line 8 is also optimized away in List. 2.

In this example, the compiled code is still safe, as the compiler uses `%idx_0` instead of loading from `ptr %idx`. However, in a more complex program, extra computation could cause register pressure, leading to `%idx_0` being spilled to memory before being used to index into `ptr %data`. A valid optimization could then rematerialize `%idx_0` from `ptr %idx`, saving a stack slot[2]. Given a data race on `ptr %idx`, this value might be larger than the size of `data`, causing an out-of-bounds access. As far as we know, this optimization is not in LLVM, although it has been proposed [46]. This exploration shows that if the DRSV in WGSL is exploitable, it is likely due to programs that are susceptible to these types of optimizations. We now describe the base shaders WGSLMemSmith generates and then show how patterns which might trigger these optimizations are incorporated into the shaders.

4.2 Shader Generation

WGSLMemSmith generates shaders in a single pass. It currently supports a subset of WGSL types and expressions, consisting of assignment statements, integer arithmetic, conditionals, and loops. Assignment statements have a variable number of operands in their expressions—by default between 1-3. Optionally, a setting *register pressure* can be turned on, which increases the number of operands in expressions (up to 20) and prioritizes using local variables in order to increase the number of registers the compiler might allocate. Overall, there are 16 parameters when generating shaders that can be modified at runtime. This allows us to run comprehensive fuzzing campaigns that randomly generate shaders with different parameters at each iteration.

Along with the base shader, WGSLMemSmith generates racy *patterns* that seek to induce compiler optimizations that could trigger the WGSL DRSV. Patterns are based on templates instantiated during the generation process, with the three types of patterns shown in Fig. 2. For example, the code in Fig. 2a is an example of a **Basic** pattern, including potential data races and out-of-bounds accesses. As part of pattern generation, conditionals like the ones in lines 4-6 of List. 1 are included, with condition and assignment values randomly chosen. Patterns can be nested in conditionals, loops, or even inside other patterns themselves. Along with basic patterns, WGSLMemSmith generates patterns with other characteristics:

[2]This same optimization was shown to be problematic when bounding certain data race behaviors in prior work [26].

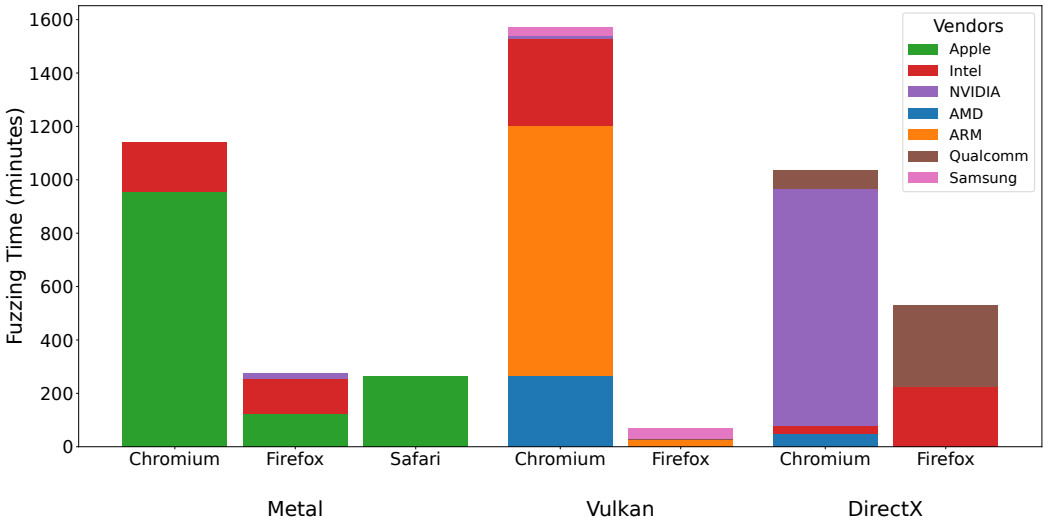

Fig. 3. WGSLMemSmith fuzzing time broken down by compilation stack.

- **Memory Region.** WGSLMemSmith generates patterns which perform potential out-of-bounds accesses on device, workgroup and private memory, as different memory regions have different semantics and as such, might each expose different behavior.
- **Undefined Arithmetic.** Following C++, native GPU languages leave the behavior of signed integer overflow, division/modulo by zero, and division/modulo of the most negative integer by -1 as undefined. WGSL compilers include checks to avoid undefined behavior when performing arithmetic. However, a compiler could determine these checks are unnecessary and remove them (e.g., through range analysis). Figure 2b shows an example of integer overflow, where the calculation of the index into `m` can overflow due to the data race if the arithmetic checks inserted by WGSL compilers are optimized away.
- **Control Flow.** The calculation of an index into memory can be affected by control instructions. If any memory locations that determine the outcome of the conditional participate in a data race, then it is possible that the calculated index may be out-of-bounds. For example, Fig. 2c shows an example where a data race could cause the variable `x` to be assigned to 4, even though the compiler determines that the conditional will not execute.

Inspired by previous work that use parallel strategies for memory model testing on GPUs [50], WGSLMemSmith generates programs that run thousands of threads in parallel, with all threads executing patterns and racing with other threads on memory locations used to calculate indexes. Due to non-determinism in multi-threaded execution, each generated shader runs for 3 iterations to increase the chance of observing vulnerabilities.

4.3 Evaluation

We developed a website which runs shaders generated by WGSLMemSmith through WebGPU's JavaScript API and utilized it to test 21 unique compilation stacks. Overall, WGSLMemSmith ran for over 81 hours and tens of millions of unique thread executions, with Fig. 3 breaking down the testing time on different compilation stacks. While we ran fuzzing on both the Chrome and Edge web browsers, we combine these results in our report as they are both built on the Chromium open-source browser and rely on the same underlying WebGPU implementation.

Table 2. Percentage of shaders showing memory isolation failures on an Apple M3/M2 GPU with bounds-checks disabled.

	Memory					
	Private		Workgroup		Device	
Pattern	**M3**	**M2**	**M3**	**M2**	**M3**	**M2**
Basic:	4.5%	0.9%	4.5%	3.3%	37.3%	18.6%
UndefArith:	5.4%	4.3%	3.0%	3.7%	3.6%	3.1%
ControlFlow:	8.2%	6.1%	1.0%	0.0%	37.8%	20.4%

4.3.1 Testing for Vulnerabilities. Our fuzzing campaign led to a negative result–data races did not cause memory isolation vulnerabilities due to bounds-checks being optimized away. While this is not a formal proof of the absence of such optimizations, it shows that exploiting the WGSL specification vulnerability is at the least very difficult, and compilers just might not implement the types of optimizations that could cause it today. Despite the lack of observable security concerns, the specification vulnerability is still a ticking time bomb for WGSL's memory safety, and there is no guarantee that compilers won't implement it in the future.

While testing a pre-release version of Firefox on a laptop running Windows with a Qualcomm GPU, we observed non-zero values when reading from out-of-bounds indexes in workgroup memory. Our investigation showed that the Firefox WGSL compiler was not adding bounds-checks correctly for workgroup and private memory. This vulnerability was confirmed by Mozilla privately and fixed in its implementation of WebGPU [90]. As WebGPU was not available in Firefox's release browser at the time of our discovery, this vulnerability luckily did not impact most users.

4.3.2 Mutation Testing. To provide more confidence in our testing, we perform mutation testing [22] where we analyze the ability of WGSLMemSmith to observe memory isolation failures in the absence of bounds-checks. These could be missing due to compiler bugs, e.g., the Firefox vulnerability described in Sec. 4.3.1, or unsound optimizations in the presence of data races, e.g., the examples in Fig. 1 and List. 1. We utilize a flag in the Chromium browser to disable bounds-checks on two GPUs, an Apple M2 and M3. As shown in Sec. 3.1, when programs written in Apple's native Metal shading language with memory safety violations were run, these GPUs revealed memory isolation failures. Therefore, we would expect our fuzzing to catch these failures with bounds-checks disabled. For each mutation testing trial, WGSLMemSmith is configured to produce one of the three types of patterns from Fig. 2, i.e., **Basic**, **UndefArith**, and **ControlFlow**:

Each pattern type is then combined with each memory region, private, workgroup, or device, and shaders are generated and run for five minutes on each GPU. The results in Tab. 2 show the percentage of generated shaders that led to memory isolation failures on each GPU. Every combination except control flow dependence and workgroup memory on the M2 showed isolation failures, validating that WGSLMemSmith can observe failures (in most cases) due to data races if bounds-checks are removed.

4.3.3 Register Pressure. We analyze whether the shaders generated by WGSLMemSmith are causing register pressure on various GPUs, as we discussed in Sec. 4.1 how this could lead to dangerous optimizations. We utilize a Vulkan extension, *pipeline executable properties*, which allow developers to analyze statistics about shaders. We first randomly generate a shader with WGSLMemSmith using the same limits as in our fuzzing campaign. Next, we use the Tint WGSL

compiler to generate a SPIR-V shader, set up a short program that accepts the shader as input, compiles it, and returns the shader statistics, which we report for three GPUs below.

Intel Arc A770 Graphics. There are three relevant statistics reported by Vulkan on this GPU: max live registers, spill count, and fill count. To the best of our knowledge, spill count and fill count measure the number of register spills and re-load instructions which the compiler adds to the shader. For the shader we test, the max live registers is reported as 331, the spill count as 683, and the fill count as 1952, showing that register pressure is indeed occurring.

AMD Radeon RX 7900 XT. Several relevant statistics are reported by Vulkan on this GPU: compiler VGPR/SGPRs, used VGPR/SGPRs, and scratch memory usage. VGPR (SGPR respectively) stands for vector (scalar) general purpose registers, which are used when each thread in a SIMD unit operate on different (the same) data. For the shader we test, the compiler reports 256 available VGPRs and 106 available SGPRs, and uses 256 VGPRs as 37 SGPRs. Our current implementation of WGSLMemSmith focuses on maximizing registers per thread which explains why SGPRs are not maximized—future versions of WGSLMemSmith might be able to address this deficiency. We observe that the final statistic, scratch memory usage is 0 when VGPRs are not maximized, and increases when they are (for our test shader, the value is 496 bytes). Therefore, we can infer that our test shader is causing register spilling on this GPU.

NVIDIA GeForce RTX 4070. The only statistic reported by Vulkan on this GPU is used register count—for our test shader, this value is reported as 255. This number matches the reported maximum number of registers per thread on NVIDIA's Ada GPU architectures [14], meaning that the shader is at least utilizing all available registers.

5 The SafeRace Memory Safety Guarantee

Our threat assessments, along with vulnerabilities found in prior work [40, 48, 65, 79], show that GPUs memory isolation vulnerabilities continue to be a serious concern. And although the specification vulnerability in WGSL does not currently seem exploitable, the optimizations which would make it so are valid and could legitimately be implemented at any time by GPU compilers. Therefore, we introduce the SafeRace Memory Safety Guarantee (SMSG), which fixes the WGSL DRSV by ensuring that bounds-checks are robust in the presence of data races.

Consider a WGSL program P_u, which may contain unsafe behaviors. When P_u is compiled to lower-level languages, it is transformed into P_s by inserting checks to guard against unsafe behavior, e.g., `min` in Fig. 1. However, as described in Sec. 1.1 and Sec. 4.3.1, WGSL compilers include no special protections against data races and may in fact introduce data races. Compiler transformations in the lower-level language can transform P_s into a new program P'_s. Under current specifications, if either P_u or P_s contain data races, P'_s may contain memory safety violations, e.g., Fig. 1 with the `min` check removed. To address this, SMSG consists of two properties which when combined ensure that P_u is safe from the DRSV. This in turn ensures that in the absence of other undefined behavior (or compiler bugs), valid transformations from P_s to P'_s are also safe.

- **Property 1: Race-Free Indexing Slices:** Program slices which contribute to memory indexing must be race-free.
- **Property 2: Data Race Non-Interference:** In the absence of other undefined behavior, data races cannot affect the behaviors of race-free slices of programs.

We now formally describe a logic for constructing program slices which contribute to memory indexing and walk through a couple examples of how to apply this logic.

Table 3. SafeRace Memory Logic

Program	P	The program under analysis
Memory Scope	$R_i^{p\|w\|d}$	Private/workgroup/device memory region
Memory Type	$R_i^{\{p\|w\|d\}na}$	Non-atomic memory
	$R_i^{\{w\|d\}a}$	Atomic memory
Memory Accesses	$A_x^{\{r\|w\|i\}}$	Read/write/initialization access
	$A_x.R_i$	Memory region accessed
	$A_x.I$	Index dependencies
	$A_x.C$	Control dependencies
	$A_x.D$	Data dependencies
	$A_x.S$	Slice: $(A_x.I \cup A_x.C \cup A_x.D)+$
	Applying SMSG	
Property 1 (RaceFreeSlice)	$\forall A_y \in (A_x.I \cup A_x.C)$: $\texttt{mkAtomic}(A_y.R_i) \wedge$ $\forall A_z \in A_y.S$: $\texttt{mkAtomic}(A_z.R_i)$	
Property 2 (RaceInterference)	$\forall A_x, A_y \in P.A : A_x \neq A_y \wedge A_y \notin A_x.S \implies$ $\texttt{bhvrs}(A_x) \mid P.A \equiv \texttt{bhvrs}(A_x) \mid P.A \setminus A_y$	

5.1 A Memory Logic for WGSL

Table 3 shows the constructs used to reason about memory safety in WGSL and the languages it targets. WGSL provides a more restricted programming model than other low-level languages, e.g., C++, which we take advantage of to construct our memory logic. A program P operates on data which is either constant, e.g., literals and program constants, or in disjoint memory regions R[3]. A memory region's name is denoted using a subscript, e.g., R_i is the memory region referred to by name i. Memory regions can either be private variables/arrays (R_i^p), or shared workgroup/device memory (R_i^w/R_i^d). Shared memory regions can either be non-atomic ($R_i^{\{w|d\}na}$) or atomic ($R_i^{\{w|d\}a}$).

A memory region is a fixed-size set of 8-bit memory locations. The type of all memory locations in a region is the same, with casting between atomic and non-atomic references to the region prohibited. The set of static memory accesses within a program, i.e., those present in the source code, is represented by A. Read and write accesses are represented by A^r/A^w, respectively. Declarations of input buffers to a program are represented as a special *initialization* access type A^i. Each memory access is given a unique id x which we denote using a subscript on A. Each access A_x has several fields, namely: $A_x.R_i$ denotes the memory region accessed; $A_x.I$ are the index dependencies, i.e., the set of accesses which potentially contribute to the calculation of the index into A_x; $A_x.C$ are the accesses on which A_x is control-dependent; and $A_x.D$ are the accesses on which A_x is potentially data-dependent, which for a read access A_x^r are the set of accesses it may read from, and for a write access A_x^w are the set of accesses which determine the value written.

Because our approach is based on static analysis, we say that an access A_x is *potentially* dependent on an access A_y if there exists an assignment of values, e.g., due to data races or conflicting atomic accesses, to other accesses in the program that causes A_x to be dependent on A_y. Intuitively, this means conditionals based on shared memory read accesses may resolve to any branch and assuming a write access with an index dependency on a shared memory read access may write

[3]Writable memory bound to a shader cannot alias in WebGPU [82, Sec. 14.1].

```
1  var<device>      a: array<i32, 1>;  A_0^i : {R_a^dna}
2  var<workgroup>   b: array<i32, 1>;  A_1^i : {R_b^wna}
3  var<workgroup>   c: array<i32, 1>;  A_2^i : {R_c^wna}
4  var<device>      d: array<i32,10>;  A_3^i : {R_d^dna}
5  fn shader() {
6    a[0]=5;          A_4^w : {R_a}
7    b[0]=1;          A_5^w : {R_b}
8    var x:i32=3;     A_6^w : {R_x^pna}
9    if(a[0]>5)       A_7^r : {R_a, D = {A_4^w}}
10     x=100;         A_8^w : {R_x, C = {A_7^r}}
11   c[0]=b[0];       A_9^r : {R_b, D = {A_5^w}}  A_10^w : {R_c, D = {A_9^r}}
12                              A_11^r : {R_x, D = {A_8^w, A_6^w}}
13                              A_12^r : {R_c, D = {A_10^w}}
14                              A_13^r : {R_d, I = {A_11^r, A_12^r}}
15   var r=d[x+1+c[0]]);  A_14^w : {R_r^p, D = {A_13^r}}
16 }
```

Listing 3. WGSL pseudocode illustrating how to reason using our memory logic.

to any location in a memory region R_i. This ensures that any calculation of A_x's dependencies is sound, but potentially an over-approximation.

The transitive closure of the union of $A_x.I$, $A_x.C$, and $A_x.D$ is called the access *slice*, or $A_x.S$. A slice represents all memory accesses which may affect A_x, including its reachability, the memory locations it accesses, the value stored if $A_x \in A^w$, and the value it returns if $A_x \in A^r$. $A_x.S$, along with its constituent sets, can be obtained via static program slicing based on a data-flow and control-flow analysis [89].

The definition of an access now allows us to precisely define how to reason about programs with SMSG. Property 1 can be enforced by a compiler in a safe high-level language via the rule RaceFreeSlice in Tab. 3. This rule specifies that given an access A_x, for all accesses A_y in $(A_x.I \cup A_x.C)$, if $A_y.R_i$ is a shared memory region it must be made atomic. Additionally, for all accesses A_z in the transitive closure of $A_y.S$, if $A_z.R_i$ is a shared memory region, it also must also be made atomic (note that `mkAtomic()` is a no-op for private memory regions and that when applying RaceFreeSlice to A_x, $A_x.D$ is not included because the data dependencies of A_x do not determine the memory locations in R_i A_x accesses). Intuitively, the program slice for each index and control dependency of an access is now race-free as all contributing memory accesses have been made atomic, preventing the optimizations discussed in Sec. 1.1 and Sec. 4.1.

Property 2, which must be adopted by the memory-safe language *and* the languages in its compilation stack, specifies that if a memory access A_y is not in $A_x.S$, it cannot affect the result of A_x even if it participates in a data race. Therefore, when running a program P, the possible behaviors of A_x, i.e., $\texttt{bhvrs}(A_x)$, with A_y included in P must be equivalent to the possible behaviors of A_x when running P with A_y removed.

5.2 Applying SMSG

First, we show how SMSG can be applied in the simple examples in Fig. 1, and then move on to a more complex example which includes control flow and multiple types of dependencies.

Simple Example. In both programs in Fig. 1, there are two memory regions which we assume are device-scoped—`i` (R_i^{dna}) and `m` (R_m^{dna})—and one private memory region—`x` (R_x^p). In Fig. 1a, the access to `m` is index-dependent on the access to `i[0]`, so applying RaceFreeSlice means compiling R_i^{dna}

to R_i^{da}, thereby making all accesses to `i[0]` atomic. Because accesses are now atomic, the compiler cannot assume that `i[0]`'s value won't change between accesses, and thus, cannot erroneously remove the bounds-check. In Fig. 1b, R_x^p is race-free as it is a private memory region, and applying RaceInterference means that the data race on R_i^{dna} cannot affect it.

Larger Example. Listing 3 shows an example of how to apply this logic to a program P_u. There are six memory regions in P_u: R_a^{dna}, R_b^{wna}, R_c^{wna}, R_d^{dna}, R_x^p, and R_r^p. All memory events, including the declaration of buffers on lines 1-4, are represented as memory accesses. Only non-empty base fields are shown, with the type of a memory region omitted after its first reference.

Accesses and their dependencies are shown either above or on the line they reference, so the accesses on lines 12-15 all correspond to the code in line 15. Working backwards from line 15, A_{13}^r is index-dependent on A_{11}^r and A_{12}^r. A_{11}^r is potentially data-dependent on A_6^w and A_8^w, while A_8^w is control-dependent on A_7^r. Meanwhile, A_{12}^r is data-dependent on A_{10}^w, which in turn is data-dependent on A_9^r. At this point, every memory region besides R_d and R_r in P_u is contained in accesses in $A_{13}^r.S$.

Assuming the absence of data races, which a compiler for an unsafe language can do since all memory regions are non-atomic, all bounds-checks in this example could be provably removed as the index on line 15 evaluates to 5. However, data races may cause these values to change in ways that break memory safety. For example, a race on `c[0]` or `b[0]` could cause their values to increase even after the compiler has removed the bounds check, as Fig. 1 and List. 1 showed. Similarly, a race on `a[0]` could cause the code in the conditional to execute despite the compiler optimizing based on the determination that it is dead code[4].

When compiling from $P_u \longrightarrow P_s$, applying Property 1 via RaceFreeSlice means compiling memory regions as follows: $R_a^{dna} \longrightarrow R_a^{da}$, $R_b^{wna} \longrightarrow R_b^{wa}$, and $R_c^{wna} \longrightarrow R_c^{wa}$. Legal transformations $P_s \longrightarrow P_s'$ cannot assume the value of atomic memory locations, prohibiting optimizations that both remove the bounds check and rematerialize from the shared memory regions in accesses in $A_{13}^r.S$. The other memory region in accesses in $A_{13}^r.S$, R_x^p, is not made to shared memory, RaceInterference ensures that its value cannot be affected by data races elsewhere in the program.

5.3 SMSG Soundness

We now argue that the rules for applying SMSG ensure that data races cannot lead to memory safety violations. Consider a program P_u which contains a memory access A_x that may attempt to read out-of-bounds memory. In an initial compilation from $P_u \longrightarrow P_s$, e.g., WGSL to a lower-level language, A_x is compiled to A_x', which is guaranteed to not read out-of-bounds due to checks added by the compiler. Now, assume a valid transformation T in the lower-level language compiles $A_x' \xrightarrow{T} A_x''$, and that A_x'' no longer contains bounds-checks and leads to a memory safety violation.

If Property 1 is applied correctly, there are no data races in B, the set of memory accesses in $A_x'.I$ and $A_x'.D$ and in their slices. Therefore, any analysis the compiler performs to remove the bounds-check in A_x'' while considering the values of memory accesses in B is safe. The only other source of data races in P_s are accesses not in B, which cannot affect the behaviors of accesses in B according to Property 2. As B is exactly the set of accesses that contribute to the calculation of A_x''s (and therefore A_x'''s) index, A_x' and A_x'' are safe. This means that if T is a valid transformation, it must have introduced the memory safety violation due to behaviors other than data races, validating our claim that P_s is safe in the presence of data races.

[4]Since it is dead code, a compiler might decide to eliminate it, in which case R_a does not need to be made atomic. However, GPUs execute in a single-program multiple-data manner, so dead code in one thread may not be dead in another and a compiler may decide to leave it in place.

Complete Memory Safety in WebGPU. Enforcing SMSG ensures that data races no longer lead to globally undefined behavior or memory safety violations due to out-of-bounds accesses. However, to provide complete memory safety, WebGPU needs to guarantee that no possible WGSL program can lead to undefined behavior in its lower-level language targets. The current WGSL specification avoids undefined behavior by classifying its common sources, including data races, as dynamic errors, but as we have shown this is not necessarily sufficient. Therefore, we analyze the other sources of dynamic errors mentioned in the specification, which are: aliased writable memory bound to a shader [83, Sec. 7.3], mismatched memory layouts [83, Sec. 13.4], unbounded loops [83, Sec. 9.4.3], and an erroneous graphics-specific vertex value [83, Sec. 12.3].

Non-aliasing of memory is enforced by the WebGPU host API [82, Sec. 14.1], and while mismatched memory layouts between the host/shaders can affect correctness of programs, the values in memory do not change the analysis used to apply SMSG. We do not consider graphics pipelines, leaving unbounded loops as the last source of dynamic errors. Unlike in languages like C++, where unbounded loops with side effects are useful for many tasks, GPU languages are generally not designed for long-running programs. In fact, most GPU languages currently lack a well-specified forward progress model [80], so even unbounded loops with side effects often lead to crashes or other issues. The lack of support for unbounded loops has impacts on memory safety, and open-source WGSL compilers insert some code in response to the fact that the Metal compiler might optimize away bounds-checks due to considering infinite loops without side effects undefined behavior [35]. While so far this approach seems be successful, the lack of specificity around the behavior of unbounded loops means that we currently only apply SMSG guarantees to source programs that are free of behaviors like unbounded loops.

5.4 Other Memory Safety Approaches

There are several other approaches which could be used to avoid the removal of bounds-checks and help maintain memory safety in WGSL. The first, and simplest, is to simply enforce that all memory accesses are made atomic when compiling from WGSL to lower-level GPU languages, which would remove data races from all programs. However, even though the `relaxed` memory order semantics for atomics which we use are less costly than more heavyweight memory orders like `acquire`/`release` and sequentially consistent, our results in Sec. 6.1 show that in cases where some accesses must be made relaxed atomic operations, there is already a performance decrease. Making every access atomic likely would cause unacceptable performance for many applications.

Another option is LLVM's `unordered` atomic ordering [54, Atomic Memory Ordering Constraints] for all accesses, which provides weaker semantics than `relaxed` but prevents rematerializing loads and is used by LLVM to match Java's memory model. However, the languages which WGSL targets, e.g., SPIR-V, Metal, and HLSL, do not all explicitly support unordered atomics yet. To implement this solution, the changes in GPU compiler implementations and the performance of using unordered atomics would need to be carefully studied and tested. Additionally, SMSG can be integrated with unordered atomics, as Property 1 can translate access dependencies to unordered atomics instead of relaxed atomics, while Property 2 is weaker than unordered atomic guarantees.

A different style of approach is to integrate some form of "unoptimizable checks" into WGSL and the languages it targets. For example, LLVM has a concept of *linkage types* [54, Linkage Types], including types like *weak* which prevent a compiler from removing unreferenced global variables and functions. If the `min` function is declared using this attribute, the compiler would therefore not be able to remove it. While this approach is promising, unoptimizable checks would require a large amount of coordination and implementation work across WGSL and the languages it targets. Also, to our knowledge, the guarantees offered by existing unoptimizable checks like linkage types are not formally defined in relation to existing sources of undefined behavior such as data races. On the

other hand, SMSG is a novel solution to memory safety in the presence of data races using existing language constructs that are preserved across all target languages, i.e., relaxed atomic accesses.

6 Implementing SMSG Property 1

We implemented Property 1 as a pass in Tint, the WGSL compiler included in the Dawn WebGPU implementation used by Chromium. While Tint is not an optimizing compiler, it does include significant infrastructure for performing passes on WGSL programs, e.g., inserting bounds-checks and ensuring arithmetic expressions are well defined. Tint converts WGSL programs into an intermediate representation (IR) in static single-assignment (SSA) form and includes utilities for traversing and modifying the IR.

Our pass consists of two phases: a backwards slicing phase and a forward conversion phase. In the slicing phase, the RaceFreeSlice rule is applied to compute the sets of accesses which contribute to an access A_x. In the conversion phase, non-atomic memory regions are converted to atomic regions, and every usage of these regions is updated to atomically load or store to them. Our implementation accounts for nested types, e.g.., arrays and structs, handles control flow, e.g., loops and branches, and performs complete interprocedural analysis. In the process of writing our implementation, we encountered several constructs that either provided opportunities for optimization or exposed limitations in WGSL. We now detail these constructs as they provide interesting avenues for future WGSL development.

Read-Only Regions. All WGSL memory regions have an *access mode*, either `read`, `read_write`, or `write`. During our calculation of Property 1, if we encountered a read-only region, we did not convert it to atomic, because by definition reads cannot race with each other in the absence of a write. However, in some cases it is possible that even if a region is marked as `read_write`, it is only read to during execution of the shader. While not implemented, regions which are only read from, even if marked `read_write`, do not need to be accessed atomically.

Limitation on Atomic Types. WGSL currently only allows two atomic types: unsigned and signed 32-bit values. This has implications for both the completeness and performance of our implementation of Property 1. For example, one element of a `vec2` may be included in the index dependency of an access, but since a `vec2` cannot be made atomic, we cannot convert it and ensure it won't participate in a data race. Similarly, if a floating point value is part of the control dependency of an access, the region the floating point is loaded from can't be converted. Therefore, it is currently not possible to implement Property 1 completely in WGSL. When our implementation encounters a data type that cannot be accessed atomically in WGSL, it is flagged so that users of the compiler pass can be made aware of the issue. For example, safety critical applications may decide to discard flagged shaders and perform the computation on the CPU instead. In the future, WGSL could add support for atomic floats and other data types, following the example set by C++, which implements an atomic type template which can be applied as long as certain *named requirements* are satisfied [16].

The limitations on atomic types also impact the performance of shaders translated by our pass. In WGSL, atomic types are not *constructible*, i.e., able to be loaded/stored directly, passed into functions, or created outside of the declarations of buffer types. If a struct or sized array which contains a type converted by our pass is loaded into a private variable, a function loads each field of the struct or loops over the array and loads each element. It is not possible to directly load a runtime-sized array, creating an upper bound on the number of accesses which must be made atomic. WGSL's ability to support atomic types is dependent on their implementation in native GPU languages. These languages are slowly adding support for more atomic types, so future enhancements to WGSL can increase the coverage of SMSG Property 1. For full completeness, however, it may be required to limit the types which can contribute to memory accesses.

Table 4. Impact of applying SMSG Property 1 to WebGPU benchmarks running on an Apple M3.

Benchmark	Shaders	Impacted Shaders	Impacted Accesses	Pass Timing (ms)	Runtime Overhead
WebLLM DeepSeek Qwen 7B [70]	73	4	24	62.47	0.0%
WebLLM Llama 3.2 1B [70]	63	1	24	53.78	0.0%
ONNX Runtime (Whisper) [88]	60	0	0	8.67	0.0%
ONNX Runtime (Phi-3) [91]	19	0	0	3.77	0.0%
MediaPipe Gemma2 2B [36]	72	0	0	5.17	0.0%
FFT Ocean Demo [4]	9	0	0	0.11	0.0%
Game Of Life [86]	1	0	0	0.08	0.0%
Compute Boids [86]	1	0	0	0.03	0.0%
Bitonic Sort [86]	3	1	10	0.34	9.8%
MSM [23]	4	3	7	23.43	3.0%

Targeting Different Backends. As described in Sec. 2.3, both DirectX and Vulkan provide guarantees that operations on device memory will not execute out-of-bounds, while Metal does not. It may seem then that we do not need to apply SMSG Property 1 to device memory accesses in DirectX and Vulkan. However, these guarantees are more analogous to runtime protections in operating systems, rather than features of the languages themselves. This is because data races in both HLSL and SPIR-V programs are undefined, meaning that memory safety violations could potentially occur in racy programs. In some sense, these could also be considered specification vulnerabilities of DirectX and Vulkan, motivating future work on the development of these frameworks. For our purposes, this means that we apply SMSG Property 1 to all memory operations on all backends.

6.1 Evaluation

While the goal of SMSG is to prevent malicious actors from exploiting memory safety violations in WGSL programs, its guarantees apply to all WGSL programs. This means that even in programs that are not malicious, the SMSG pass analyzes memory accesses and translates dependencies into atomic operations. Therefore, we conducted a study to evaluate the cost of SMSG Property 1 in terms of both compilation time and GPU performance.

6.1.1 Benchmarks. Our implementation of Property 1 in Tint allows us to build a custom version of Chromium on an Apple M3 and run our analysis on a set of WebGPU applications, detailed in Tab. 4. As WebGPU is still in active development and is not yet fully available on all web browsers, this set of applications consists mostly of demonstrations. We believe these applications provide a good benchmark of how WebGPU can be utilized by browser applications.

The first five benchmarks in Tab. 4 are AI applications. **WebLLM DeepSeek Qwen 7B** and **WebLLM Llama 3.2 1B** utilize WebLLM [70] to run LLM inference. Under the hood, WebLLM relies on Apache TVM [34], specifically TVM's Web runtime, and supports many models. We chose two models for our benchmark: the Llama-3.2-1B-Instruct generative model, and the DeepSeek-R1-Distill-Qwen-7B reasoning model. **ONNX Runtime (Phi-3)** and **ONNX Runtime (Whisper)** use ONNX Runtime Web [25] through transformers.js [41], which allows machine learning models to be deployed and run in browsers using WebGPU. The **Phi-3** benchmark runs inference using the Phi-3-mini-4k-instruct model, while the **Whisper** benchmark runs speech recognition with OpenAI's Whisper Large V3 Turbo. Finally, **MediaPipe Gemma2 2B** uses Google's MediaPipe suite to run LLM inference with the Gemma2 2B model.

The other benchmarks consist of demonstrations of WebGPU's compute abilities. Three, **Bitonic Sort**, **Game of Life**, and **Compute Boids**, are part of the WebGPU Samples maintained by the

official WebGPU working group [86], **FFT Ocean Demo** shows how to use compute shaders to render complex ocean waves as part of the Babylon.js library for 3D web graphics [4], and **MSM** is a WebGPU implementation of multi-scalar multiplication submitted to the 2023 ZPrize competition [95]. This last benchmark relies on sparse-matrix multiplication techniques, an area where complex index dependencies may be more common than in dense computations.

6.1.2 Compilation Cost. Unlike a language like CUDA which is often precompiled to GPU assembly code, WGSL shaders are translated to native GPU languages and vendor-specific machine code at runtime. To mitigate the cost of runtime compilation, WebGPU implementations and native GPU frameworks utilize layered and non-standardized compilation processes, often involving asynchronous operations, just-in-time (JIT) compilation, and caching components. This complex structure makes it difficult to measure the timing of the full compilation pipeline. Therefore, in Tab. 4, rather than reporting the time taken by our pass as a percentage overhead of the full compilation process, we instead simply report the raw time (in the **Pass Timing** column) it takes to run our pass on the shaders in each benchmark.

Compared against only the time taken to run the existing passes in the `tint` compiler, the addition of our pass adds a large overhead (up to 9x for the slowest-compiling shader in the **MSM** benchmark). However, these pre-existing passes are generally much simpler, e.g., doing a sequential scan of a shader and adding bounds-checks to memory accesses, while our pass requires relatively complex dependency analysis and program traversal. The timing of passes in the `tint` compiler also does not include the time it takes to parse WGSL programs into an IR and write them out to native GPU programs, nor the time taken by native language compilers to generate machine code, meaning that the time taken to run all passes is a small part of the overall compilation and runtime process. For example, on the **WebLLM DeepSeek Qwen 7B** benchmark, the one-time cost of our pass summed across all shaders was only 62.47 ms, while each shader ran dozens to hundreds of times for a total of several seconds during a *single* inference response, with typical LLM interactions usually consisting of many inferences.

6.1.3 GPU Performance Cost. Atomic operations constrain compiler optimizations, and thus, can negatively impact performance [10, 74]. Therefore, it is likely that any atomic operations added to satisfy SMSG Property 1 could lead to reduced application performance. However, as shown in Tab. 4 in the **Runtime Overhead** column, on the majority of the benchmarks we tested SMSG Property 1 did not apply to any shaders. We now discuss the four benchmarks where it did apply.

While the two WebLLM model benchmarks differed in the number of shaders used by the program, they both compile a shader called `apply_penalty_inplace_kernel` which requires making four memory regions atomic, affecting 24 accesses in the program. However, we have received confirmation from the developers of WebLLM that these shaders are not yet used during LLM inference. Additionally, the accesses that must be made atomic in this shader are floating point accesses, and as mentioned in Sec. 6 WGSL does not yet allow atomic access to floating point data types. Therefore, if WebLLM starts using this shader, we can no longer guarantee memory safety in this application, at least until WGSL begins to support atomic accesses to more data types.

The **Bitonic Sort** benchmark does require making one workgroup buffer memory region atomic, affecting 10 accesses in one shader. To complete one full sort, using default parameters, this shader runs 91 times. In order to get precise GPU timing information, we extended Dawn's WebGPU implementation to automatically record GPU timestamps for every compute shader. We ran the full sort three times each with our pass enabled and disabled on an Apple M3 GPU, and calculated the average time across the 91 runs of the shader. On average, each shader run took between .04-.09 ms, for a total of 3.6-8.2 ms. With our pass enabled, average runtime of the shaders increased by 9.8%, from .064 ms per shader to .071 ms. We performed the same analysis on the **MSM** benchmark;

applying SMSG Property 1 to these shaders required making 7 accesses atomic. The average shader runtime of this benchmark increased from .99 seconds to 1.02 seconds, or 3%.

To conclude, these results show that only ~2% of the shaders in our benchmarks are affected by SMSG Property 1. The benchmarks consist of important WebGPU applications in several domains, including LLM inference and general compute algorithms, and are either generated by state-of-the-art tensor compilers, e.g., TVM [34], or handwritten by expert GPU developers. Despite this, many of the shaders used by these applications do not include the complex indirect memory accesses that require applying transformations to satisfy SMSG Property 1, validating the efficacy of SMSG. For example, matrix multiplication is generally written using simple index calculations taken from combinations of thread identifiers and constants, and none of the matrix multiplications in our LLM benchmarks require any transformation. On the other hand, it is possible that some applications, e.g., bitonic sort, will experience reduced performance due to the safety requirements of SMSG Property 1. We also show that in this set of benchmarks, data types which cannot be accessed atomically in WGSL are generally not used in index dependence calculations, with the only exception being the WebLLM shader which may be utilized in the future. All browser-based languages must make tradeoffs between performance and safety, and we believe our results provide evidence that the performance overhead of adopting SMSG Property 1 is within acceptable limits.

7 Testing SMSG Property 2

While SMSG Property 1 can be implemented by WGSL compilers, Property 2 must be adopted by the languages WGSL targets. This may seem like a thorny problem; many languages, including C++ and GPU languages, leave the behavior of data races undefined to avoid having to reason about constraints like the one required by Property 2. However, previous work on data races has shown how to bound the behavior of data races to provide a local data-race-free (Local DRF) property and proven the soundness of many existing compiler optimizations given this property [26].

The Local DRF property is stronger than SMSG Property 2. It prevents a data race on one memory location x causing an unrelated memory location y to read a value written to x, what has been called *bounding data races in space*. It also prevents the effects of data races from leaking across synchronization points, e.g., barriers, called *bounding data races in time*. In particular, bounding data races in time requires preserving load-store ordering, which has been shown to have relatively low cost on CPUs. However, the cost of load-store reordering has not been studied on GPUs, where it is very commonly observed [49]. Still, prior work showed that preserving load-store ordering is not necessary to bound data races in space, while proving that many existing compiler transformations are compatible with this property [26], and therefore, SMSG Property 2.

To test whether SMSG Property 2 is maintained by current GPU compilers, we augmented the WGSLMemSmith fuzzer using metamorphic fuzzing. When generating shaders, variables and memory locations are split into *safe* and *unsafe* sets. Safe memory locations are read-only or written to by only one thread, while unsafe memory locations may be read or written to by any thread. Safe local variables are only used in combination with safe memory locations, while unsafe variables can be combined in expressions with unsafe memory locations. Intuitively, this process creates race-free slices of programs, as well as slices that may contain races. During fuzzing, we run two shaders, one with only race-free slices and the other with both race-free and racy slices.

The race-free shader is run first and the values in memory at the end of execution are saved. Since this shader contains no data races or sharing of memory between threads, the results are deterministic. The racy shader, which contains the exact statements in the safe shader along with statements that operate on unsafe memory locations and may contain data races, is run next. Then, the values in safe memory locations at the end of the racy shader execution are compared with the

values from the race-free shader execution—if the racy shader results differ, the addition of data races may have caused the differences, violating SMSG Property 2.

7.1 Evaluation

As we use the same shaders for both the metamorphic fuzzing and testing for SMSG Property 1, our evaluation consists of the same set of compilation stacks and testing time shown in Fig. 3. On most compilation stacks we tested, we found that data races did not effect race-free slices of programs. This was true for all compilation stacks using Vulkan and DirectX as the WebGPU backend, as well as Metal backends targeting Apple M-series GPUs. On DirectX backends, data races caused unexpected values in unsafe memory locations, i.e., values that could not be explained by any writes to those memory locations, but this behavior did not affect race-free slices of the programs.

On one machine, an older MacBook with an Intel SoC, running the racy shader caused memory in race-free slices of the shader to change. We built a browser-based reducer using fuzzing reduction techniques [69, 94] and determined that the mismatch between shader outputs in the shaders which we reduced were due to an unrelated issue that manifests in the presence of certain specific loop constructs and occurs when running a single-threaded shader containing no data races on the GPU. We believe this issue is a compiler/driver bug and have reported our findings to Apple.

Our observations show that SMSG Property 2 empirically holds in every WebGPU implementation we have tested except on one (likely buggy) machine, which should ease its adoption by GPU languages. Besides memory safety, SMSG Property 2 also allows more compositional reasoning about program behaviors, opening up the door to other future optimizations and safety properties.

8 Applications Beyond WebGPU

The proposals in this paper are designed to address WGSL's specification vulnerability, but our results can also inform the general development of safe languages which target parallel processors and seek to provide memory safety and reasonable semantics for data races. Our work has potential lessons and impacts for three different areas: existing browser-based languages, existing native languages, and future development of languages that require compositional safety reasoning.

8.1 Browser-Based Languages

The other two main languages used in the browser, JavaScript and Wasm, both require bounds-checks for memory safety and constrain the effects of data races, including specifying that data races must be bound in space. Unlike WGSL, the semantics of these languages can also be fully controlled by an implementation. For example, the V8 JavaScript and Wasm engine controls the optimization phases of compilation and directly generates machine code, avoiding intermediate languages with undefined behavior. In some cases, these engines implement bounds-checks by relying on operating systems/processors to fault when memory is accessed out-of-bounds [85], which as we show cannot be guaranteed by GPU runtimes.

Even so, the problems associated with optimizations on bounds-checks and data races must be considered by these languages. JavaScript and Wasm engines may target architectures without built-in memory protections, so must not implement optimizations which both remove bounds-checks and rematerialize loads. Additionally, as described in Sec. 4.2, dead code may execute due to data races in control dependencies, so optimizations on those blocks must be carefully checked.

8.2 Native Languages

Some native languages, such as Java and Go, give weak semantics to data races and control their compilation stacks, similar to the browser-based languages. Rust takes a different approach, using its ownership system to avoid races and memory safety violations in Safe Rust. However, Unsafe

Rust, along with other unsafe languages like C++, do not provide the same guarantees. A large body of research exists to retrofit unsafe languages with memory protections, including runtime analyses like Address Sanitizer [75] and pointer augmentation like Checked C [71]. However, undefined behavior, e.g., due to data races, makes it difficult to reason about these protections' soundness.

In contrast, if a native language were to adopt SMSG Property 2 and apply SMSG Property 1 to indirect accesses via its compiler, it would be able to provide more robust memory safety guarantees. In general, SMSG would be difficult to apply to large, feature-rich C++ codebases spanning many files and utilizing complex language features, as the Property 1 compiler pass needs access to the full program slice to compute the full access dependency graph and cannot easily handle features like pointer aliasing, dynamic allocations, and linkage types. However, in safety-critical settings, e.g., automotive, there is already precedent, e.g., MISRA-C++ [61], for restricted subsets of languages where Property 1 could more easily apply. For example, a C++ dialect where all pointers are restricted to not alias, memory locations are not accessed both atomically and non-atomically, and mixed-size accesses are restricted would correspond almost exactly to WGSL, in which case SMSG could apply directly. We believe this is a promising avenue of future research, with the tools and proposals in this paper providing a framework for formally defining such a language dialect.

8.3 Compositional Language Design

WGSL is somewhat unique in that it both needs to enforce memory safety and relies on intermediate languages which are inherently weaker with regards to these protections, motivating the development of SMSG. However, we argue that going forward, this type of compositional language design will become more prevalent. The post-Dennard Scaling landscape has led to the development of numerous types of architectures and associated software stacks, while frameworks like LLVM and MLIR are paving the way for increasingly connected pathways from one language to another.

As shown in this paper, understanding the interactions between languages and their safety guarantees is complex. Our approach blends a mix of more formal language proposals and extensive empirical evaluation. We hope our results motivate future research that seeks to clearly specify behaviors that have classically been left undefined, such as data races, making it easier to compose language semantics and enable performant and safe applications.

9 Related Work

Detecting and Reasoning About Data Races. Prior work on data races has tried to classify races as either "benign" or "destructive" [31, 63], or argued that there are no benign races [9] and shown how races can break basic assumptions of program behavior [26]. There is a large line of research on detecting and avoiding races, both statically [7, 29, 66] and dynamically [30, 73, 76].

Researchers have also developed tools to statically analyze GPU programs for data races [6, 13, 52], but these tools are not complete and may report false positives, which hamper adoption by industry [43, 62, 72]. More recently, a sound and partially-complete static data race checker for GPUs has been developed [53], and a GPU programming language which prohibits data races and out-of-bounds accesses by construction has been proposed [45]. Dynamic data race detection on the GPU has also shown promise, finding new data races without false positives [44]. In this work, we do not attempt to detect or remove data races; instead, we focus on providing guarantees that data races will not compromise the security of a programming language. In fact, SMSG Property 1 compiler passes could be further optimized using static data race detection techniques, as shaders statically determined to be data-race-free do not need any transformations to maintain memory safety.

Memory Isolation. Memory isolation on GPUs has been extensively studied—CUDA was shown to leak global and register data [65], leftover global memory from NVIDIA and AMD GPUs could be used to infer webpages visited by a user [48], leftover shared and register memory could be used to reconstruct inference output of neural networks [68, 79], and out-of-bounds accesses on NVIDIA GPUs could be used to overwrite threads' local memory and degrade output of neural networks [40]. All this prior work also has focused on reverse engineering low level details of GPU assembly, handwriting shaders to exhibit issues, or combinations of both to reveal specific, one-off vulnerabilities. In contrast, we present a set of properties that guarantee memory isolation in a high level language, WGSL, by preserving memory safety through the compilation stack.

Compiler Testing. Compiler fuzzing has been widely studied, e.g., finding bugs in C compilers [93] and GPU compilers for languages like OpenGL [27], SPIR-V [28], and even WGSL [47]. These approaches are either based on differential testing [57], comparing outputs of multiple compilers, or metamorphic testing [12], where programs are changed in ways that should not affect their semantics and results are compared. Another feature of these approaches is that they explicitly seek to avoid undefined behavior, constructing programs that are guaranteed to not contain undefined behaviors or adding transformations that remove undefined behaviors at the source level. Unlike other approaches, we explicitly include constrained sources of undefined behavior, e.g., data races, out-of-bounds accesses, and undefined arithmetic, to test the security properties of compilers.

10 Conclusion

In this paper, we showed that WebGPU currently contains a specification vulnerability where valid compiler transformations could introduce memory safety violations due to data races. We show through our bottom-up threat assessment that such violations could readily be turned into memory isolation vulnerabilities. To address this, we provide the SafeRace Memory Safety Guarantee (SMSG), a set of specification proposals that preserves memory safety in the presence of data races. We show that this can be done with relatively little overhead or changes to current frameworks, enabling browsers to more safely take advantage of GPU acceleration and providing a framework for future development of safe languages.

Data Availability Statement

Resources and information for reproducing the results in this work are available openly [51].

Acknowledgments

We thank all the reviewers of this paper for their detailed and helpful feedback; this work is more rigorous and precise because of them. We would like to thank Trail of Bits for helping with our vulnerability disclosures, as well as the product security teams at Apple and especially AMD for promptly responding and working with us to address the issues we found. We also appreciate the quick responses and fixes by the Firefox and wgpu developers after we reported the bounds-checking issue to them. We thank the members of the W3C WebGPU working group and the Khronos Memory Model Task sub-group for allowing us to present our work to them and for providing valuable feedback. We also specifically thank Faith Ekstrand, who provided information on using the Vulkan pipeline executable properties extension, and Natalie Vock, who helped diagnose and fix the AMD security issue. This work was supported by a gift from Google and an NDSEG fellowship. This material is based upon work supported by the National Science Foundation under Award No. 2239400. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

References

[1] 2024. Dawn: A WebGPU implementation. https://dawn.googlesource.com/dawn.

[2] R. Abbott, J. Chin, Jed Donnelley, W. Konigsford, S. Tokubo, and D. Webb. 1976. Security analysis and enhancements of computer operating systems. (1976). https://nvlpubs.nist.gov/nistpubs/Legacy/IR/nbsir76-1041.pdf

[3] Apple Inc. 2025. Metal. https://developer.apple.com/documentation/metal/.

[4] Babylon.js Team. 2025. Babylon.js FFT ocean demo. https://playground.babylonjs.com/?webgpu#YX6IB8#758

[5] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. https://doi.org/10.1145/1926385.1926394

[6] Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. 2012. GPUVerify: a verifier for GPU kernels. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. https://doi.org/10.1145/2384616.2384625

[7] Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2018. RacerD: compositional static race detection. *Proc. ACM Program. Lang.* (2018). https://doi.org/10.1145/3276514

[8] Google Security Blog. 2019. Queue hardening enhancements. https://security.googleblog.com/2019/05/queue-hardening-enhancements.html.

[9] Hans-J. Boehm. 2011. How to miscompile programs with "benign" data races. In *3rd USENIX Workshop on Hot Topics in Parallelism*. https://dl.acm.org/doi/10.5555/2001252.2001255

[10] Brandon Alexander Burtchell and Martin Burtscher. 2024. Characterizing CUDA and OpenMP synchronization primitives. In *2024 IEEE International Symposium on Workload Characterization*. https://userweb.cs.txstate.edu/~burtscher/papers/iiswc24b.pdf

[11] Microsoft Security Response Center. 2019. We need a safer systems programming language. https://msrc.microsoft.com/blog/2019/07/we-need-a-safer-systems-programming-language/.

[12] Tsong Yueh Chen, Shing-Chi Cheung, and Siu-Ming Yiu. 1998. Metamorphic testing: a new approach for generating next test cases. (1998). https://doi.org/10.48550/arXiv.2002.12543

[13] Tiago Cogumbreiro, Julien Lange, Dennis Liew Zhen Rong, and Hannah Zicarelli. 2021. Checking data-race freedom of GPU kernels, compositionally. In *Computer Aided Verification: 33rd International Conference*. https://doi.org/10.1007/978-3-030-81685-8_19

[14] NVIDIA Corporation. 2024. *CUDA Ada GPU architecture tuning guide*. NVIDIA Corporation. https://docs.nvidia.com/cuda/ada-tuning-guide/index.html.

[15] cppreference.com. 2024. Multithreading - C++ Language Documentation. https://en.cppreference.com/w/cpp/language/multithread

[16] cppreference.com. 2025. std::atomic - C++ Reference. https://en.cppreference.com/w/cpp/atomic/atomic

[17] NIST National Vulnerability Database. 2002. CVE-2002-0649. https://nvd.nist.gov/vuln/detail/CVE-2002-0649.

[18] NIST National Vulnerability Database. 2014. CVE-2014-0160 (Heartbleed). https://nvd.nist.gov/vuln/detail/cve-2014-0160.

[19] NIST National Vulnerability Database. 2016. CVE-2016-4655. https://nvd.nist.gov/vuln/detail/CVE-2016-4655.

[20] NIST National Vulnerability Database. 2022. CVE-2022-32947. https://nvd.nist.gov/vuln/detail/CVE-2022-32947.

[21] NIST National Vulnerability Database. 2025. CVE-2024-36353 (AMD Leftover Global Memory). https://nvd.nist.gov/vuln/detail/CVE-2024-36353.

[22] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. 1978. Hints on test data selection: help for the practicing programmer. *Computer* (1978). https://doi.org/10.1109/C-M.1978.218136

[23] Tal Derei and Koh Wei Jie. 2023. webgpu-msm-bls12-377: WebGPU MSM implementation for BLS12-377 curve (ZPrize 2023). https://github.com/td-kwj-zp2023/webgpu-msm-bls12-377.

[24] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. LLM.int8(): 8-bit matrix multiplication for transformers at scale. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*. https://doi.org/10.48550/arXiv.2208.07339

[25] ONNX Runtime developers. 2025. Using WebGPU with ONNX Runtime. https://onnxruntime.ai/docs/tutorials/web/ep-webgpu.html.

[26] Stephen Dolan, KC Sivaramakrishnan, and Anil Madhavapeddy. 2018. Bounding data races in space and time. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/3192366.3192421

[27] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated testing of graphics shader compilers. *Proc. ACM Program. Lang.* (2017). https://doi.org/10.1145/3133917

[28] Alastair F. Donaldson, Paul Thomson, Vasyl Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpiński. 2021. Test-case reduction and deduplication almost for free with transformation-based compiler testing. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. https:

//doi.org/10.1145/3453483.3454092
[29] Cormac Flanagan and Stephen N. Freund. 2001. Detecting race conditions in large programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. https://doi.org/10.1145/379605.379687
[30] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: efficient and precise dynamic race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/1542476.1542490
[31] Cormac Flanagan and Stephen N. Freund. 2010. Adversarial memory for detecting destructive races. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/1806596.1806625
[32] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. https://doi.org/10.1145/2837614.2837615
[33] Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. 2017. Mixed-size concurrency: ARM, POWER, C/C++11, and SC. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. https://doi.org/10.1145/3009837.3009839
[34] Apache Software Foundation. [n. d.]. Apache TVM: Open Deep Learning Compiler Stack. https://github.com/apache/tvm.
[35] gfx-rs Developers. 2025. wgpu Issue #4972: support for buffer mapping in chrome WebGPU. https://github.com/gfx-rs/wgpu/issues/4972 Accessed: 2025-03-12.
[36] Google MediaPipe Studio. 2025. MediaPipe Studio: LLM inference demo. https://mediapipe-studio.webapps.google.com/studio/demo/llm_inference
[37] Khronos Group. [n. d.]. Vulkan 1.3 Specification. https://registry.khronos.org/vulkan/specs/1.3-extensions/html/vkspec.html.
[38] Khronos Group. 2024. SPIR-V Unified Specification. https://registry.khronos.org/SPIR-V/specs/unified1/SPIRV.html#_memory_instructions. Accessed: 2024-10-07.
[39] Khronos Group. 2024. Vulkan Guide - Robustness. https://docs.vulkan.org/guide/latest/robustness.html
[40] Yanan Guo, Zhenkai Zhang, and Jun Yang. 2024. GPU memory exploitation for fun and profit. In *33rd USENIX Security Symposium (USENIX Security 24)*. https://www.usenix.org/system/files/usenixsecurity24-guo-yanan.pdf
[41] Hugging Face. 2025. Transformers.js documentation. https://huggingface.co/docs/transformers.js/en/index
[42] Apple Inc. [n. d.]. Metal shading language specification. https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf.
[43] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *2013 35th International Conference on Software Engineering*. https://doi.org/10.1109/ICSE.2013.6606613
[44] Aditya K. Kamath and Arkaprava Basu. 2021. iGUARD: In-GPU Advanced Race Detection. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. Association for Computing Machinery. https://doi.org/10.1145/3477132.3483545
[45] Bastian Köpcke, Sergei Gorlatch, and Michel Steuwer. 2024. Descend: a safe GPU systems programming language. *Proc. ACM Program. Lang.* (2024). https://doi.org/10.1145/3656411
[46] Chris Lattner. 2012. LLVM memory use markers. https://www.nondot.org/sabre/LLVMNotes/MemoryUseMarkers.txt
[47] Bastien Lecoeur, Hasan Mohsin, and Alastair F. Donaldson. 2023. Program reconditioning: avoiding undefined behaviour when finding and reducing compiler bugs. *Proc. ACM Program. Lang.* (2023). https://doi.org/10.1145/3591294
[48] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. 2014. Stealing webpages rendered on your browser by exploiting GPU vulnerabilities. In *2014 IEEE Symposium on Security and Privacy*. https://doi.org/10.1109/SP.2014.9
[49] Reese Levine, Mingun Cho, Devon McKee, Andrew Quinn, and Tyler Sorensen. 2023. GPUHarbor: testing GPU memory consistency at large (experience paper). In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. https://doi.org/10.1145/3597926.3598095
[50] Reese Levine, Tianhao Guo, Mingun Cho, Alan Baker, Raph Levien, David Neto, Andrew Quinn, and Tyler Sorensen. 2023. MC Mutants: evaluating and improving testing for memory consistency specifications. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. https://doi.org/10.1145/3575693.3575750
[51] Reese Levine, Ashley Lee, Neha Abbas, Kyle Little, and Tyler Sorensen. 2025. Artifact for SafeRace: assessing and addressing WebGPU memory safety in the presence of data races. https://doi.org/10.5281/zenodo.16915241
[52] Guodong Li and Ganesh Gopalakrishnan. 2010. Scalable SMT-based verification of GPU kernel functions. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*. https://doi.org/10.1145/1882291.1882320

[53] Dennis Liew, Tiago Cogumbreiro, and Julien Lange. 2024. Sound and partially-complete static analysis of data-races in GPU programs. *Proc. ACM Program. Lang.* (2024). https://doi.org/10.1145/3689797

[54] LLVM Project. 2025. LLVM Language Reference Manual. https://llvm.org/docs/LangRef.html.

[55] Rust Graphics Mages. [n. d.]. wgpu. https://github.com/gfx-rs/wgpu.

[56] Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. https://doi.org/10.1145/1040305.1040336

[57] William M. McKeeman. 1998. Differential testing for software. *Digit. Tech. J.* (1998). https://api.semanticscholar.org/CorpusID:14018070

[58] Microsoft. 2024. Direct3D 11 Advanced Stages - Compute Shader Access. https://learn.microsoft.com/en-us/windows/win32/direct3d11/direct3d-11-advanced-stages-cs-access

[59] Microsoft. 2024. ld_raw (sm5 - asm). https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/ld-raw--sm5---asm-.

[60] Microsoft. 2025. DirectX specifications. https://microsoft.github.io/DirectX-Specs/.

[61] MISRA Consortium. 2023. MISRA C++: guidelines for the use of C++ in critical systems. https://www.misra.org.uk/misra-c-plus-plus/. Originally published June 5, 2008; latest edition released October 2023. Accessed 2025-07-24.

[62] Marcus Nachtigall, Lisa Nguyen Quang Do, and Eric Bodden. 2019. Explaining Static Analysis - A Perspective. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop*. https://doi.org/10.1109/ASEW.2019.00023

[63] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. 2007. Automatically classifying benign and harmful data races using replay analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/1250734.1250738

[64] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 memory model: x86-TSO. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*. https://doi.org/10.1007/978-3-642-03359-9_27

[65] Roberto Di Pietro, Flavio Lombardi, and Antonio Villani. 2016. CUDA leaks: a detailed hack for CUDA and a (partial) fix. *ACM Trans. Embed. Comput. Syst.* (2016). https://doi.org/10.1145/2801153

[66] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. 2011. LOCKSMITH: Practical static race detection for C. *ACM Trans. Program. Lang. Syst.* (2011). https://doi.org/10.1145/1889997.1890000

[67] Chromium Project. 2023. WebGPU technical report. https://chromium.googlesource.com/chromium/src/+/main/docs/security/research/graphics/webgpu_technical_report.md.

[68] Frederik Dermot Pustelnik, Xhani Marvin Saß, and Jean-Pierre Seifert. 2024. Whispering pixels: exploiting uninitialized register accesses in modern GPUs. (2024). https://doi.org/10.48550/arXiv.2401.08881

[69] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/2254064.2254104

[70] Charlie F. Ruan, Yucheng Qin, Xun Zhou, Ruihang Lai, Hongyi Jin, Yixin Dong, Bohan Hou, Meng-Shiun Yu, Yiyan Zhai, Sudeep Agarwal, Hangrui Cao, Siyuan Feng, and Tianqi Chen. 2024. WebLLM: a high-performance in-browser LLM inference engine. arXiv:2412.15803 [cs.LG] https://arxiv.org/abs/2412.15803

[71] Andrew Ruef, Leonidas Lampropoulos, Ian Sweet, David Tarditi, and Michael Hicks. 2019. Achieving Safety Incrementally with Checked C. In *Principles of security and trust*. https://doi.org/10.1007/978-3-030-17138-4_4

[72] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from building static analysis tools at Google. *Commun. ACM* (2018). https://doi.org/10.1145/3188720

[73] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.* (1997). https://doi.org/10.1145/265924.265927

[74] Hermann Schweizer, Maciej Besta, and Torsten Hoefler. 2015. Evaluating the cost of atomic operations on modern architectures. In *2015 International Conference on Parallel Architecture and Compilation*. https://doi.org/10.1109/PACT.2015.24

[75] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: a fast address sanity checker. In *USENIX ATC 2012*. https://www.usenix.org/conference/usenixfederatedconferencesweek/addresssanitizer-fast-address-sanity-checker

[76] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*. https://doi.org/10.1145/1791194.1791203

[77] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y. Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E. Gonzalez, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. FlexGen: high-throughput generative inference of large language models with a single GPU. (2023). https://doi.org/10.48550/arXiv.2303.06865

[78] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. 2017. Chasing away RAts: semantics and evaluation for relaxed atomics on heterogeneous systems. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. https://doi.org/10.1145/3079856.3080206

[79] Tyler Sorensen and Heidy Khlaaf. 2024. LeftoverLocals: listening to LLM responses through leaked GPU local memory. (2024). https://doi.org/10.48550/arXiv.2401.16603
[80] Tyler Sorensen, Lucas F. Salvador, Harmit Raval, Hugues Evrard, John Wickerson, Margaret Martonosi, and Alastair F. Donaldson. 2021. Specifying and testing GPU workgroup progress models. *Proc. ACM Program. Lang.* OOPSLA (2021). https://doi.org/10.1145/3485508
[81] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. https://doi.org/10.1145/2676726.2676995
[82] W3C. [n. d.]. WebGPU. https://www.w3.org/TR/webgpu/.
[83] W3C. 2024. WebGPU Shading Language (WGSL). https://www.w3.org/TR/WGSL/. Accessed: 2024-10-08.
[84] Conrad Watt, Christopher Pulte, Anton Podkopaev, Guillaume Barbier, Stephen Dolan, Shaked Flur, Jean Pichon-Pharabod, and Shu-yu Guo. 2020. Repairing and mechanising the JavaScript relaxed memory model. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/3385412.3385973
[85] Conrad Watt, Andreas Rossberg, and Jean Pichon-Pharabod. 2019. Weakening WebAssembly. *Proc. ACM Program. Lang.* (2019). https://doi.org/10.1145/3360559
[86] WebGPU Samples. 2025. WebGPU Samples. https://webgpu.github.io/webgpu-samples/
[87] WebKit Contributors. 2025. WebGPU Source in WebKit. https://github.com/WebKit/WebKit/tree/main/Source/WebGPU.
[88] webml-community. 2025. Whisper Large V3 Turbo WebGPU. https://huggingface.co/spaces/webml-community/whisper-large-v3-turbo-webgpu
[89] Mark Weiser. 1984. Program slicing. *IEEE Transactions on Software Engineering* (1984). https://doi.org/10.1109/TSE.1984.5010248
[90] wgpu contributors. 2024. Add support for restricting indexing to avoid OOB accesses. https://github.com/gfx-rs/wgpu/pull/6431
[91] Xenova. 2025. Experimental Phi-3 WebGPU. https://huggingface.co/spaces/Xenova/experimental-phi3-webgpu
[92] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2023. SmoothQuant: accurate and efficient post-training quantization for large language models. In *Proceedings of the 40th International Conference on Machine Learning*. https://doi.org/10.48550/arXiv.2211.10438
[93] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/1993498.1993532
[94] A. Zeller and R. Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* (2002). https://doi.org/10.1109/32.988498
[95] ZPrize Initiative. 2025. ZPrize: accelerating the future of zero-knowledge cryptography. https://www.zprize.io/. Accessed on 2025-07-24.

Received 2025-03-25