Dynamic Buffer Overflow Detection for GPGPUs

Christopher Erb
AMD Research
Advanced Micro Devices, Inc.
Austin, TX, USA
Christopher.Erb@amd.com

Mike Collins
AMD Research
Advanced Micro Devices, Inc.
Austin, TX, USA
Mike.Collins@amd.com

Joseph L. Greathouse
AMD Research
Advanced Micro Devices, Inc.
Austin, TX, USA
Joseph.Greathouse@amd.com

Abstract
Buffer overflows are a common source of program crashes, data corruption, and security problems. In this work, we demonstrate that GPU-based workloads can also cause buffer overflows, a problem that was traditionally ignored because CPUs and GPUs had separate memory spaces. Modern GPUs share virtual, and sometimes physical, memory with CPUs, meaning that GPU-based buffer overflows are capable of producing the same program crashes, data corruption, and security problems as CPU-based overflows. While there are many tools to find buffer overflows in CPU-based applications, the shift towards GPU-enhanced programs has expanded the problem beyond their capabilities.

This paper describes a tool that uses canaries to detect buffer overflows caused by GPGPU kernels. It wraps OpenCL™ API calls and alerts users to any kernel that writes outside of a memory buffer. We study a variety of optimizations, including using the GPU to perform the canary checks, which allow our tool to run at near application speeds. The resulting runtime overhead, which scales with the number of buffers used by the kernel, is 14% across 175 applications in 16 GPU benchmark suites. We use the same benchmarks to demonstrate its efficacy by finding a total of 13 buffer overflows in 7 benchmarks.

1. Introduction
Buffer overflows are a common software problem with a long history [7]; famous security attacks such as the Morris Worm, Code Red, and Slammer were all predicated on this error. By allowing accesses outside of the “correct” region of memory, buffer overflows can lead to program crashes, data corruption, and security breaches [53]. Owing to this long history in CPU-based applications, numerous tools have been built to find and stop buffer overflows [10, 12, 29, 42, 45, 58–60, 65, 68, 74].

In contrast, little attention has been paid to buffer overflows on GPUs. While GPU programs are just as susceptible to memory bugs as CPU programs, the following differences have led developers to incorrectly ignore the problem:
1. GPU and CPU memory has traditionally been separated, making it difficult for GPUs to corrupt the CPU memory.
2. GPU programs rarely use pointers or make function calls, making visible crashes from buffer overflows less likely.
3. GPU memory is often allocated in a less dense manner than CPU memory, so overflows from one buffer will not necessarily corrupt useful data in another.

Unfortunately, these neither prevent overflows nor protect against their effects. Miele, for instance, recently demonstrated that GPU-based buffer overflows can lead to remote GPU code execution [54], as did Di et al. [26].

Additionally, GPUs can access CPU memory over interconnects such as PCIe®, and standards such as Heterogeneous System Architecture (HSA) allow virtual memory sharing between CPUs and GPUs [13, 40]. Such tight integration allows GPU overflows to easily corrupt CPU data.

There currently exist few tools to help catch GPU buffer overflows. Oclgrind can instrument OpenCL™ programs with checks to find such errors, but it causes runtime overheads of up to $300 \times$ [61]. Techniques from fast CPU-based detectors like Electric Fence [60] and StackGuard [21] are difficult to add to closed-source vendor runtimes; they require changes to virtual memory managers or compilers.

Towards this end, this paper describes the design of a runtime buffer overflow detector for OpenCL GPU programs. By catching function calls that allocate OpenCL memory buffers, we can add canary regions outside of the buffers. After an OpenCL kernel completes, our tool checks these canary regions to see if the kernel wrote beyond its buffers’ limits. If so, our tool alerts the user to this problem.

We limit the overhead of these analyses by tuning our canary checks and switching them between the CPU and GPU. Our tool causes a mean slowdown of only 14% across 175 programs in 16 OpenCL benchmark suites. We use the same benchmarks to demonstrate its efficacy by finding a total of 13 buffer overflows in 7 of the programs.

In total, this paper makes the following contributions:
• We describe the design of the first canary-based OpenCL buffer overflow detector. It is also the first to work with OpenCL 2.0 shared virtual memory buffers.
• We detail techniques, such as using GPU kernels to check canary values for overflows, which limit our tool to an average slowdown of only 14%.
• We show that our tool finds real problems by detecting and fixing 13 buffer overflows in 7 real benchmarks, many of which have not been observed by other tools.
2. Background

This section includes background information for this work. Section 2.1 discusses buffer overflows and tools to find them. Section 2.2 describes GPU memories, and Section 2.3 discusses how OpenCL™ 2.0 presents this memory to software.

2.1 Buffer Overflows

Buffer overflows are software errors that result from accessing data beyond the limits of a buffer. An example of a buffer overflow (perhaps most famously described by Aleph One [6]) is illustrated in Figures 1(a) and 1(b). Here, an array and the function’s return address are placed next to one another. Copying too many values into the array will overwrite the address, allowing an attacker to take control of the application. Overflows can also cause silent data corruption, memory leaks, and crashes, among other problems.

Numerous tools have been built to catch buffer overflows in CPU programs. Heavyweight tools like Valgrind [58] add extra checks to validate each memory access, but their overheads mean that they are used sparingly by developers.

This work therefore focuses on more lightweight techniques. In particular, as illustrated in Figure 1(c), we place canary values outside of buffers that are susceptible to overflows. The values are later checked and, if they have been overwritten, the tool reports a buffer overflow.

Previous tools have used canaries to check for overflows in CPU programs. StackGuard [21] uses canaries to protect the stack, while ContraPolice can protect heap structures [45]. Both of these tools add canary checks at various points in the application. Along similar lines, Electric Fence [60] adds canary pages around heap structures and protects the canaries using the virtual memory system; a page fault therefore indicates a buffer overflow. While these tools are useful for finding CPU-based buffer overflows, they do not work for GPU memory.

2.2 GPU Memory

CPUs and GPUs have traditionally had separate memory spaces, so overflows on the GPU would not corrupt CPU data. In addition, because GPUs rarely performed tasks like dereferencing pointers, corrupted GPU buffers rarely caused crashes. GPU memory managers often pack data less densely than CPU managers, making it harder for an overflow to corrupt other buffers [50]. As such, GPU buffer overflows were often ignored or left undetected, and there were few tools built to find them.

While this may imply that buffer overflows are a minor problem on GPUs, this is not the case. Miele recently demonstrated that buffer overflows on the GPU could be used to inject code that could allow attackers to take control of the GPU’s operation [54]. Beyond hijacking GPU control flow, the move towards tightly integrated heterogeneous systems means that GPUs can also corrupt CPU memory by directly accessing CPU buffers over interconnects like PCIe®.

This can happen in OpenCL applications that allocate buffers with flags such as CL_MEM_ALLOC_HOST_PTR or CL_MEM_USE_HOST_PTR [3]. Similarly, fine-grained shared virtual memory buffers are stored in this manner in AMD’s OpenCL 2.0 implementation [8]. Margiolas and O’Boyle took advantage of these techniques to reduce DMA transfers between the CPU and GPU memories [52]. They only moved buffers into the GPU’s memory if they would be accessed frequently enough to justify the transfer costs. As such, they designed an automated system that may leave some CPU buffers susceptible to GPU buffer overflows.

New chip designs bring advancements that will exacerbate this problem. Single-chip heterogeneous SoCs are now available from companies such as AMD [46], Intel [28], and Nvidia [27] that allow CPUs and GPUs to share the same physical memory. Both AMD and Intel allow these devices to share virtual memory, as well. Similarly, HSA compliant devices share virtual memory between the CPU and accelerators [13, 70]. This means that GPU buffer overflows will become more problematic in the future.

2.3 OpenCL Memory Buffers

This work focuses on buffer overflows caused by OpenCL™ kernels running on GPUs. As such, this section details the type of buffers used in OpenCL kernels.

Stack Values Many OpenCL implementations do not put stack variables into memory, instead preferring to allocate them entirely in registers. In addition, analyzing these variables requires modifying the OpenCL compiler, which is often a proprietary part of a vendor’s software stack. As such, our buffer overflow detector does not analyze stack values.

Local Memory Local memory is often allocated into on-chip scratchpad memories at kernel invocation time. Because this memory is not shared with the DRAM buffers, attempting to access values outside of the allocated region often causes GPU kernels to crash immediately. As such, our tool does not search for local memory overflows.

Global cl_mem Buffers These memory buffers are allocated by the host using functions such as clCreateBuffer. By default, these buffers are allocated into the GPU’s memory and cannot contain pointers. However, as mentioned in

![Figure 1. Example of a buffer overflow. (a) and (b) show how copying too much data into a buffer can corrupt neighboring variables. (c) shows a canary value after the buffer; if this canary changes, a buffer overflow has occurred.](image-url)
Section 2.2, they can be forced into the CPU’s memory. Our tool watches for buffer overflows in these regions by wrapping calls to functions that create cl_mem buffers and expanding the requested size to include our canary regions.

Global cl_mem Images Images are multi-dimensional buffers created using functions such as clCreateImage2D (before OpenCL 1.2) and clCreateImage (OpenCL 1.2+). Images are like cl_mem buffers, except that it is possible for an application to overflow one dimension of the image without writing past the “end” of the buffer (the final dimension). To enable discovery of these overflows, our tool expands each dimension of an image with canary regions.

Sub-Buffers A sub-buffer is created by calling the function clCreateSubBuffer, which takes a reference to an existing cl_mem buffer and returns a cl_mem object that points into the middle of the original buffer. Because this sub-buffer is within a memory region that has already been allocated, our tool cannot expand this buffer with canary regions at sub-buffer creation time. Instead, our tool creates a shadow copy of this buffer, as further described in Section 3.

Coarse-grained SVM Shared virtual memory (SVM) is a feature of OpenCL 2.0 that allows buffers that reside in the GPU’s memory to contain pointers into their buffer and into other SVM buffers. These regions are allocated with the clSVMAlloc function. Coarse-grained SVM buffers must be mapped into the CPU’s memory in order to access them on the CPU, and this generally copies the data. Once the buffer is mapped on the CPU, the pointers it contains are still valid.

Our tool watches for buffer overflows in these regions by wrapping calls to clSVMAlloc and expanding the requested allocation size to include our canary regions.

Fine-grained SVM Like coarse-grained SVM, these buffers can contain pointers that are valid on both the CPU and the GPU. However, fine-grained SVM buffers need not be mapped and copied in order to access them from the CPU. AMD’s OpenCL runtime enables this by storing them in host memory and allowing the GPU to access this CPU memory region [8]. Our tool watches for buffer overflows in these regions by wrapping calls to clSVMAlloc and expanding the requested allocation size to include our canary regions.

Fine-grained System SVM These are pointers to traditional CPU memory, such as heap data returned from malloc. Because the allocation of these buffers does not go through any OpenCL APIs, and because most modern discrete GPUs do not support fine-grained SVM, our detector does not analyze this type of memory.

GPUs that do support these regions (such as the integrated systems discussed in Section 2.2) do so by sharing virtual memory between the CPU and GPU. As such, CPU-based buffer overflow detection mechanisms such as Electric Fence [60] would work for fine-grained system SVM. A GPU buffer overflow to such a region would cause a GPU page fault [73], meaning that our tool would not be required.

3. Design of a Buffer Overflow Detector

Our buffer overflow detector uses canary values to detect whether a GPU kernel has written past the end of any global memory regions. This is akin to tools like StackGuard [21], ContraPolice [45], and Electric Fence [60]. In such systems, the canary values beyond the end of a buffer are periodically checked to ensure that no buffer overflow has happened.

StackGuard and ContraPolice add the canary checks into the application, requiring it or the system libraries to be recompiled. Electric Fence protects canary pages using the virtual memory system, and writing into the canary region will cause a page fault. This is done by dynamically linking against Electric Fence and replacing calls to functions like malloc, which does not require recompilation.

Like Electric Fence, our tool works on unmodified programs. We accomplish this using the Unix LD_PRELOAD mechanism to create an OpenCL™ wrapper [62] (though similar library injection techniques can also be performed on the Windows® operating system [75]).

Our wrapper catches OpenCL API calls that allocate global memory and expands the requested sizes to include canary regions that are initialized with known patterns. We then wrap calls that set GPU kernel arguments in order to keep track of which buffers to check. Finally, we wrap the function call that launches GPU kernels, and, after the kernel completes, we check the canary values from any buffer it could access. If a canary value has changed, a bug in the kernel has caused a buffer overflow.

We check the canary values after the kernel completes because we cannot catch canary writes in the kernel (like StackGuard) or with memory protection (like Electric Fence). This limits the types of errors our tool finds, since there is a time when a corrupted value could be used before the canaries are checked. An attacker could avoid our checks by taking control of the GPU before we get a chance to see the canary values and could even reset the canary values to avoid detection. As such, our buffer overflow detector offers no security guarantees. Nonetheless, like the lightweight bounds checking technique by Hasabnis et al., this tool can still find useful problems [39]. As we demonstrate in Section 6, our technique is useful as a debugging tool.

Our tool’s three wrapper mechanisms are illustrated in Figure 2. The following sections detail a simple version of our system to make the explanation easier. Section 4 discusses performance optimizations.

3.1 Buffer Creation APIs

As Figure 2(a) shows, we wrap buffer creation APIs such as clSVMAlloc, clCreateBuffer, and clCreateImage. We then extend the buffers created by these functions and record meta-data about them. For all buffers whose creation does not use an existing allocation as a base, the size of the buffer is increased by the length of a canary region (8 KB in our studies), which is initialized with a static data pattern.
The flag `CL_MEM_USE_HOST_PTR` adds some difficulty to this scheme, since it allows previously allocated CPU memory to be used as an OpenCL buffer. We are unable to resize this region because we cannot update all of the user’s pointers to it. To work around this problem, we create an extended shadow copy of the buffer whenever we are about to execute a kernel. The copy is extended with canaries, and, upon completion of the kernel, the useful data in the shadow buffer is copied back to the original host memory. This is valid because the OpenCL standard allows implementations to cache the buffer contents of host pointer regions. Our tool caches them until after our canary check completes.

We use shadow copies more generally to solve problems arising from buffer creation using pointers to existing allocations. Similar to using the `CL_MEM_USE_HOST_PTR` flag, subbuffers are references to previously allocated buffers. Images may also be created with a previously existing buffer or image. In these cases, because our tool cannot update all references to the original data, a shadow copy is used.

### 3.2 Setting Kernel Arguments

Figure 2(b) shows how we catch the calls that are used to assign arguments to OpenCL kernels, `clSetKernelArg` and `c1SetKernelArgSVMPointer`. Wrapping these allows us to maintain a list of the buffers that each kernel can access.

This list is used when the kernel is launched to know which buffers we should check for overflows. For each global memory buffer argument, we keep a list of buffer sizes, canary values, and pointers to the buffers’ meta-data. In addition, this bookkeeping allows us to know if any SVM buffers are accessible from this kernel. As an optimization, we check for identical arguments; if two arguments use the same buffer, we only check the canary values once.

### 3.3 Kernel Enqueues

Figure 2(c) illustrates how we wrap the kernel launch function, `clEnqueueNDRangeKernel`. This is where buffer overflow detection takes place. The detector first analyzes the kernel’s arguments. Kernels with no global buffers cannot cause overflows and are run like normal.

If, however, there are `cl_mem` buffers passed to the kernel, we must verify that these buffers’ canary regions were not perturbed by the kernel. If any of the buffers were allocated without a canary region (e.g. `CL_MEM_USE_HOST_PTR` was used), the wrapper makes temporary shadow copies that contain enough space for our canary region and assign them as kernel arguments. We then launch the kernel.

While this kernel is executing, we enqueue a checker that will execute immediately after the original kernel finishes. This checker will verify the canaries of all the buffers that the kernel could have accessed.

SVM buffers add extra complexity, because we are unable to tell which will be accessed solely by looking at the kernel’s arguments. SVM regions can contain pointers to other SVM regions, so if any argument to a kernel is to an SVM buffer, it is possible to access all other SVM buffers in the application. As such, if any of the kernel’s arguments give access to an SVM buffer, the canary regions for all SVM buffers in the application must be verified.

Checking image canaries also adds complexity. In order to detect overflows in any dimension (e.g. writing past the end of a row in a 2D image), we allocate a canary region per dimension. As such, the number of canaries depends on both the number of dimensions and their size. We therefore read the canaries hierarchically from the image into a one-dimensional array. This flattened collection of all image canaries is fed into a checker kernel along with a buffer that contains the end point for each image.

Should the verification function find an overflow, a debug message is printed to the screen and, optionally, execution is halted. The debug message, shown later in Figure 11, shows the kernel name, the argument name, and where in the canary region the first corruption happened. We are able to obtain the function argument’s name by using the `clGetKernelArgInfo` function, since we know the argument index of the overflowed buffer. This does not work for SVM regions that are not passed as kernel arguments.

Finally, any shadow copy buffers are copied back to their CPU memory regions and the application continues.
3.4 API Checking

We also perform simple checks for functions that operate directly on cl_mem objects (e.g. clEnqueueWriteBuffer). These quickly compare the inputs for the operation with the cl_mem’s instantiated size. This identifies overflows not caused by kernels and prevents us from later finding them with a checker kernel and misattributing the error.

4. Accelerating Buffer Overflow Detection

Section 3 described our buffer overflow detector, while this section describes techniques to increase its performance.

We use a GPU microbenchmark with a variable number of buffers to test these techniques. Our detector will check each buffer’s canary values after the work kernel (which does no real work) ends. We then record time taken to perform the canary checks, allowing us to test the overheads of our tool at a variety of configurations. More buffers will lead to more checker overhead, because there are more checks to perform.

**CPU vs GPU Checkers** Intuitively, GPUs should excel at the parallel task of checking canary values. However, GPUs must amortize kernel launch overheads and need a significant amount of parallel work to fill their hardware resources. As such, we compared the overheads of checking canary values on both the CPU and the GPU as we vary the number of buffers (and thus the number of canaries).

For the CPU checker, our clEnqueueNDRangeKernel wrapper asynchronously enqueues a command to read the canary regions back to the host after the kernel finishes. A call to clWaitForEvents allows us to wait until this read completes, and a single CPU thread then checks the canaries.

For the GPU checker, our wrapper launches dependent checker kernel(s) immediately after the work kernel. The GPU checker will begin execution after the work kernel ends, and it will check the canary values in parallel.

**Checking multiple buffers per kernel** The simplest GPU canary checker uses one kernel per buffer, where the kernel’s arguments are the buffer to check and the offset to the canary region. This results in poor GPU utilization since each buffer only has a few thousand canary values to check.

A slightly more complicated solution uses a single kernel that takes a variable number of buffers in a fixed number of kernel arguments. We accomplish this by copying the canary regions from all of the buffers into a single buffer. Afterwards a single parallel kernel can check the entries from many buffers at once, leading to better GPU utilization.

**Utilizing SVM Pointers** Like cl_mem buffers, the canaries in SVM regions can also be checked using either one kernel per buffer or a single kernel that checks copies of the canaries for all buffers. Alternatively, it is possible to create a buffer of SVM pointers, each of which points to the beginning of a canary region in the original SVM buffer. This allows the checker kernel to directly read the SVM regions’ canary values without requiring any extra copies.

**Cleaning Modified Canaries** We must reset any corrupted canary values before subsequent kernel iterations to avoid falsely declaring more buffer overflows. This is faster for the checkers that directly check the original buffer (like the SVM-pointer method), since they can immediately write over the modified canaries. For checkers that use canary copies, we use an asynchronous clEnqueueFill<X> to reset the canary regions after they have been copied.

**Asynchronous Checking** Stalling the CPUs until the work kernel and canary checks complete can cause significant overhead. First, the application itself may have CPU work that can take place while the work kernel is running; adding synchronous canary checks will eliminate this parallelism. In addition, launching GPU checker kernels synchronously can expose dozens of microseconds of launch overhead.

To prevent this, we asynchronously launch the GPU checker kernels and use OpenCL™ events to force them to wait on the work kernels. We then launch a thread that waits on the checker’s completion event in order to print any debug messages. The checker’s event is returned to the application so that waiting on the work kernel will also wait on the checker. Calls to clGetEventProfilingInfo, however, return the profiling information for the worker kernel.

**Overheads of Checking Canaries** The overheads of these techniques across various numbers of buffers are compared in Figures 3-5. The overhead of the GPU checkers is split into two parts: the added time spent on the host arranging canary regions and launching kernels, and the added time spent in the checker kernels. We note that applications which asynchronously launch their work kernels could perform other useful work in parallel to these checks.

Figure 3 shows the overheads of checking cl_mem buffers. For small numbers of buffers, and consequently small data sets and transfers, checking on the CPU results in less overhead. In these situations, the GPU has little work to do, and amortizing the launch overheads is more difficult.

As more buffers are added, using a single GPU checker for all of the buffers results in less overhead. The time spent marshaling the canary values increases along with the number of buffers, but the checker kernel time increase more slowly, since the GPU can check many canaries in parallel. The difference in kernel times between using one kernel per buffer and using one kernel for all buffers demonstrates the benefit of running many canary checks in parallel.

Figure 4 shows a similar test for SVM buffers. The CPU checker is always slower here because, it must first copy canaries into a smaller SVM before mapping that to the host. A GPU check can instead use SVM copies or pointers and remain relatively unaffected by data movement and mapping.

Passing an array of SVM pointers to the checker kernel (rather than marshaling the canaries themselves) leads to lower host-side overheads. The most efficient way to check SVM buffers is therefore using one kernel for all canaries and passing it an array of pointers to the canary regions.
The previous sections showed that GPU canary checks can increase performance through parallelism and reduced bus transfers. These tests, however, did not include the cost of compiling the GPU checker kernel, which must be paid once each time the application is run. Figures 6-8 show the checker costs, including compilation time, as we repeatedly call the work kernel.

Figure 3. Time to check cl\_mem buffers. Using one checker kernel per buffer on the GPU suffers from underutilization, so the time spent in the kernels quickly increases.

Figure 4. Time to check SVM buffers. The CPU overhead is higher here because we must copy a subsection of the SVM to a shorter SVM to map it back to the host. Single-buffer GPU checks still suffer from underutilization. Using SVM pointers speeds up the multi-buffer GPU check.

Figure 5. Time to check images. Images have many canaries due to their multi-dimensional nature, so their total check time is high. GPU checks eliminate bus transfer times.

Figure 6. Total checker runtime across iterations checking 8 cl\_mem buffers. The large kernel compilation time for GPU checks takes many iterations to amortize.

Figure 7. Total checker runtime across iterations checking 8 SVM buffers. The GPU-based SVM checks are much faster, so it takes less time to amortize the checker kernel compilation overheads.

Figure 8. Total checker runtime across iterations checking 8 256×256 images. The host overheads make it easy for GPU kernels to amortize their initial compilation time.

Figure 6 shows that it takes more than 1000 kernel invocations before the compilation overhead is amortized when checking cl\_mem buffers on the GPU. Because of this, we fall back to performing all canary checks for cl\_mem buffers on the CPU. While some programs may run thousands of iterations, the benefits of GPU checks would only slowly reach those implied by Figure 3. Dynamically switching between CPU and GPU checks based on the number of observed iterations is an interesting direction for future study.

Figures 7 and 8 show how many iterations it takes to reach those implied by Figure 3. Dynamically switching between CPU and GPU checks based on the number of observed iterations is an interesting direction for future study.

The previous sections showed that GPU canary checks can increase performance through parallelism and reduced bus transfers. These tests, however, did not include the cost of compiling the GPU checker kernel, which must be paid once each time the application is run. Figures 6-8 show the checker costs, including compilation time, as we repeatedly call the work kernel.

Amortizing Kernel Compilation

The previous sections showed that GPU canary checks can increase performance through parallelism and reduced bus transfers. These tests, however, did not include the cost of compiling the GPU checker kernel, which must be paid once each time the application is run. Figures 6-8 show the checker costs, including compilation time, as we repeatedly call the work kernel.

Figure 6 shows that it takes more than 1000 kernel invocations before the compilation overhead is amortized when checking cl\_mem buffers on the GPU. Because of this, we fall back to performing all canary checks for cl\_mem buffers on the CPU. While some programs may run thousands of iterations, the benefits of GPU checks would only slowly reach those implied by Figure 3. Dynamically switching between CPU and GPU checks based on the number of observed iterations is an interesting direction for future study.

Figures 7 and 8 show how many iterations it takes to reach those implied by Figure 3. Dynamically switching between CPU and GPU checks based on the number of observed iterations is an interesting direction for future study.

The previous sections showed that GPU canary checks can increase performance through parallelism and reduced bus transfers. These tests, however, did not include the cost of compiling the GPU checker kernel, which must be paid once each time the application is run. Figures 6-8 show the checker costs, including compilation time, as we repeatedly call the work kernel.

Amortizing Kernel Compilation

The previous sections showed that GPU canary checks can increase performance through parallelism and reduced bus transfers. These tests, however, did not include the cost of compiling the GPU checker kernel, which must be paid once each time the application is run. Figures 6-8 show the checker costs, including compilation time, as we repeatedly call the work kernel.
5. Benchmark Analysis

This section reports the overheads caused by running an application under our buffer overflow detector. We describe our experimental setup and benchmarks in Sections 5.1, detail the memory overheads caused by our tool in Section 5.2, and show performance overheads in Section 5.3.

5.1 Experimental Setup

All of our experiments were performed on a system with a 3.7 GHz AMD A10-7850K CPU, 32 GB of DDR3-1866, and an AMD FirePro™ W9100 discrete GPU. The GPU’s core runs at 930 MHz, it has 320 GB/s of memory bandwidth to its 16 GB of GDDR5 memory, and it is connected to the CPU over a 3rd Generation PCIe® x8 connection.

The system ran Ubuntu 14.04.4 LTS and version 15.30.3 of the AMD Catalyst™ graphics drivers (fglrx). We used the AMD APP SDK v3.0 as our OpenCL™ runtime.

We ran 175 benchmarks from 16 open source OpenCL benchmark suites: the AMD APP SDK, FinanceBench [36], GPU-STREAM [23], Hetero-Mark [55], CloverLeaf and TeaLeaf from Mantevo [51], NAS Parallel Benchmarks [67], OpenDwarfs [32], Pannotia [16], Parboil [72], Phoronix [47], PolyBench/ACC [35], Rodinia [15], SHOC [22], StreamMR [30], ViennaCL [66], and the exascale proxy applications CoMD, LULESH, SNAP, and XSbench [1].

We ran two series of experiments. The first measured the memory overhead caused by extending OpenCL buffer allocations with 8 KB of canaries. We use this to show that the memory overhead of our tool is manageable and will not break most applications. The second set of experiments measured the change in wall-clock time between running these applications alone and with our buffer overflow detector. Because of the large number of benchmarks, these results are shown as the geometric mean of all benchmarks within each suite, as well as the geometric mean across all benchmarks.

5.2 Application Memory Overheads

Figure 9 shows the maximum amount of OpenCL™ buffer space added by using our tool. Figure 9(a) sorts all of the benchmarks from lowest to highest overhead, while Figure 9(b) focuses on those with overheads higher than 30%.

The geometric mean of this overhead across all 175 benchmarks is 16%, though it is often much less than 1%. The median overhead is 0.1%. Nevertheless, some of the benchmarks see very high overheads because the canary regions are a fixed 8 KB, while buffer allocation may be small. Our detector can also use some additional internal buffers to store things like arrays of SVM canary pointers. As shown in Figure 9(b), these overheads can reach almost 1000×.

For example, the SHOC benchmark md5hash allocates three cl_mem buffers that are 4, 8, and 16 bytes, respectively. Each of these buffers is then extended with 8 KB of canaries, increasing the aggregate buffer size by 24 KB, or 878× the initial 28-byte allocation.

In addition to the relative overheads illustrated with bars, Figure 9(b) lists the absolute amount of OpenCL buffer space that our detector adds. In general, the programs that have large relative overheads have small absolute overheads because the buffer space used by the application is small. This implies that our tool will rarely cause major memory pressure issues that will prevent applications from running.

Nevertheless, our 8 KB canary size was somewhat arbitrarily chosen based on our GPU’s maximum workgroup size (256), the length of a double (8 bytes), and width vector width recommended for older AMD GPUs (4). We believe there is future research in sizing canary regions to maximize error coverage while minimizing memory overheads.

5.3 Application Performance Overheads

Our detector checks the canaries for each buffer that a kernel can access, and the canary regions are a fixed size. As such, the runtime of our checker should scale linearly with the number of buffers and kernel invocations. In contrast, the relative overhead of our detector depends on the runtime of the kernels, since short kernels will make the checker time more prominent. As such, it is useful to analyze the performance overheads on real programs.

Figure 10 shows the runtime overhead of our tool on the 175 OpenCL™ enhanced applications described in the previous sections. We divide the benchmarks into their 16 respective suites in order to improve legibility. The blue bars show the geometric mean of the runtime overheads within a suite, while the upper and lower bars show the highest and lowest runtime overheads within that suite, respectively.
The worst overhead caused by our tool is a 4.8× slowdown in the proxy application SNAP MPI [24]. This overhead is caused by a significant number of very small kernels that are launched synchronously. The kernels zero_buffer and sweep_plane take an average of 400 µs and 230 µs, respectively, and are called repeatedly. sweep_plane also uses 15 cl_mem buffers. As such, the execution time of the checkers is greater than the runtime of the real kernels. In addition, both kernels are executed synchronously; very soon after they launch, all other CPU work stops to wait for the kernel to complete. The execution time of the check is therefore almost fully exposed as overhead.

Despite this, our detector rarely caused overheads greater than 50%; only 10 applications saw more than 50% slowdown. This typically occurred because, like SNAP_MPI, the program used short kernels or frequent synchronization. Short kernels do not last long enough to amortize the cost of canary checking, and synchronization prevents our detector from overlapping checks with unrelated CPU work.

In rare situations, our tool caused the application to run slightly faster. This was due to secondary effects on the host, such as link order and memory layout changes [56]. For instance, Rodinia’s hotspot3D consistently wrote its output files faster when using our tool, yielding a 7% speedup.

The final category, “ALL,” shows the geometric mean, maximum, and minimum overheads across all 175 benchmarks. Our tool causes an average slowdown of 14% across these benchmarks, with a median overhead of just 6%. We believe that this level of performance will allow our tool to be used in continuous integration systems, nightly and regression tests, and other development situations that preclude more heavyweight tools.

Looking forward, GPU applications are moving towards more asynchronous operation [2] and launching further work from within kernels [37, 43, 64]. Both of these would help amortize or hide the costs of our canary checker. As such, we believe that the overheads seen by our detector will decrease on future workloads.

6. Buffer Overflows Detected

An important test of software analysis tools is whether they can find real problems, so this section details the overflows that we found in our benchmarks. When our tool detects an overflow, it emits information about the kernel that caused the overflow and how far past the end of the buffer it wrote, as shown in Figure 11. This can help pinpoint the problem, but finding and fixing the root cause is still a manual process.

In aggregate, our tool found buffer overflows in 13 kernels across 7 programs and returned no false positives. These errors are shown in Table 1, which explains what problems we found when writing patches or workarounds.

The error in Parboil’s mri-gridding occurs because the sizes of two output buffers, keys_o and values_o, are based on the number of input elements, n. The splitRearrange kernel can cause an overflow because it assumes their lengths are evenly divisible by four. A fix for this is to allocate the buffers based on \((((n + 3)/4) * 4)\). This error may have also been found by the authors of Oclgrind when they claimed that their detector “has been used to identify bugs in real OpenCL applications ... including ... Parboil” [61].

In the StreamMR benchmarks kmeans and wordcount, some output buffers are allocated based on previous kernel results. The copyerHashToArray kernels write contiguous data to these buffers from all 64 work items in each workgroup, but the host does not ensure their sizes are evenly divisible by 64. In addition, these kernels sometimes use an uninitialized variable to access their hash tables. Both of these can result in buffer overflows and are easily fixed.

Hetero-Mark’s kmeans allocates the features_swap buffer based on npoints. The number of work items for the kmeans_swap kernel is set to be ≥npoints and evenly divide by 256. The kernel does not check the length of the buffer, so work items beyond npoints write past the end of the array based on their ID. This error is derived from a known problem in Rodinia’s kmeans, and it can be fixed (as in Rodinia 3.1) by adding a length check into the kernel.

Hetero-Mark’s sv allocates multiple buffers based on sizeInBytes, a product of m_len and n_len. The kernels sv_init_velocities, sv_compute0, sv_computel, and sv_update0 access these buffers using a variety of bad offset calculations. For instance sv_compute0 will attempt to write to z[n_len * m_len + m_len], which will result in a buffer overflow. We could not verify our fixes to each kernel’s miscalculations, since we are not the application’s authors. A workaround is to increase the size of sizeInBytes.

![Figure 10. Normalized runtime when running buffer overflow detection.](image-url)
Table 1. Overview of the errors found by our tool. We found buffer overflows in 13 separate kernels across 7 benchmarks.

<table>
<thead>
<tr>
<th>Suite</th>
<th>Benchmark</th>
<th>Kernel</th>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parboil [72]</td>
<td>mri-gridding</td>
<td>splitRearrange</td>
<td>keys_o and values_o are not allocated enough space.</td>
</tr>
<tr>
<td>StreamMR [30]</td>
<td>kmeans</td>
<td>copyerHashToArray</td>
<td>outputKeys, outputVals, and keyValOffsets are too small.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wordcount</td>
<td>copyerHashToArray</td>
</tr>
<tr>
<td>Hetero-Mark [55]</td>
<td>OpenCL 1.2 kmeans</td>
<td>kmeans_swap</td>
<td>Incorrect range check in threads writing to feature_swap.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sw, compute0</td>
<td>Bad SizeInBytes causes cu, cv, and z to be too small.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sw, compute1</td>
<td>Bad SizeInBytes causes u.next and v.next to be too small.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sw_update0</td>
<td>Bad SizeInBytes causes cu, cv, and z to be too small.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sw, init_velocities</td>
<td>Bad SizeInBytes causes u and v to be too small.</td>
</tr>
<tr>
<td>Hetero-Mark [55]</td>
<td>OpenCL 2.0 kmeans</td>
<td>kmeans_swap</td>
<td>Incorrect range check in threads writing to feature_swap.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sw, compute0</td>
<td>Bad SizeInBytes causes cu, cv, and z to be too small.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sw, compute1</td>
<td>Bad SizeInBytes causes u.next and v.next to be too small.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sw_update0</td>
<td>Bad SizeInBytes causes cu, cv, and z to be too small.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sw, init_velocities</td>
<td>Bad SizeInBytes causes u and v to be too small.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sw, update0</td>
<td>Bad SizeInBytes causes cu, cv, and z to be too small.</td>
</tr>
</tbody>
</table>

7. Related Work

This section discusses works related to detecting buffer overflows. Section 7.1 describes CPU-based detection tools, and Section 7.2 describes other analysis tools for GPUs.

7.1 CPU Buffer Overflow Detection

Perhaps the most popular memory analysis tool is the dynamic binary instrumentation engine Valgrind [58]. Its Memcheck tool searches for memory errors such as buffer overflows [69]. While this can find many problems, its runtime overhead (dozens of times slowdown) limits the situations where it can be used. Similar open source and commercial tools have roughly the same limitations [14, 42].

Compile-time instrumentation tools like AddressSanitizer [68], Baggy Bounds Checking [9], and SoftBound [57] can perform checks with overheads of roughly 10%-2× [68]. More traditionally, StackGuard (which inspired techniques used in GCC) inserts canaries before critical stack values and adds checks to verify them before they can cause security problems [21]. Our tool also uses canary values, but we do not require recompilation. Additionally, because our checks take place after the kernel completes, we cannot offer the same security guarantees as these inline checks.

The desire to further reduce overheads led to hardware-supported bounds checking, as in HardBound [25] and Intel’s MPX [41]. Our GPU buffer overflow detector has the benefit of requiring no added hardware support.

Electric Fence [60], and related tools like DUMA [10], catch calls to malloc and add protected canary pages around the allocated memory. Writing to one of these canary pages will result in a page fault that will eventually crash the program. Our tool similarly wraps allocation calls to create canary regions, but it does not utilize virtual memory to catch overflows. Systems that allow the GPU to share a coherent virtual memory with the CPU will be able to use such techniques, but many current GPUs do not share the full virtual memory space with the CPU and do not allow the canary regions to be protected in this way.

7.2 GPU Analysis Tools

Ours is not the first GPU debugging tool, nor is it the first to search for GPU buffer overflows. This section compares other GPU analysis tools to our buffer overflow detector.

Oclgrind Oclgrind is an OpenCL™ device simulator that, like Valgrind for CPU applications, can be used to build analysis tools for OpenCL kernels [61]. Like Valgrind, one of the tools that comes prepackaged with Oclgrind is a memory access checker. Oclgrind presents itself to the kernel as a CPU device, however, which limits its ability to be automatically run on some applications; the majority of our benchmarks would need manual modifications to run in Oclgrind.

While requiring manual intervention adds some difficulty, the primary limitation of Oclgrind is its execution overhead. Because it simulates OpenCL devices on the CPU, it adds extra analysis overheads and also runs the original kernel much slower. We tested a subset of our benchmarks and found that Oclgrind ran them up to 300× slower than native execution. This aligns with the authors’ claim of running “typically a couple of orders of magnitude slower than a regular CPU implementation.” Compared to our tool’s 14%, 300× slowdowns severely limit what Oclgrind can test.
The authors measured GPU overheads of processes outside of their targets, but adds runtime overheads contained within verified memory regions. This prevents kernel memory accesses so that they will always be made to runtime slowdowns; the CUDA-MEMCHECK likely uses the compiler to add checks into the kernel. This would lead to runtime slowdowns; the CUDA-MEMCHECK manual claims that “applications run much slower under CUDA-MEMCHECK tools,” and the authors of Cudagrind measured this slowdown to be roughly 120% [11]. In addition, as mentioned for the WebCL Validator, this tool would need to instrument the CUDA APIs in some way in order to know the limits of the allocated buffers. Finally, unlike our tool, CUDA-MEMCHECK only works on GPUs from Nvidia.

8. Conclusions and Future Work

This work introduced a GPU buffer overflow detector for OpenCL™ kernels. We demonstrated that buffer overflows can happen in GPU kernels and detailed the design of a canary-based tool to automatically find these problems. Our tool wraps OpenCL API calls in order to catch buffer allocations, expand each buffer, and insert canary values after it. We then catch kernel argument assignments in order to know which buffers are vulnerable to overflow in a kernel. By catching kernel invocations, we can check the buffers’ canary values after the kernel finishes to detect overflows. We demonstrated how to accelerate these tests by asynchronously checking the canary values on the GPU. This technique scales with the number of buffers, and it leads to an average overhead of only 14% across 175 applications.

Looking forward, there are a number of future directions to take this work. Our tool does not work on fine-grained system SVM, which allows GPUs to coherently access data using pointers allocated from CPU functions like malloc [49]. Such systems could detect overflows using tools like Electric Fence [60], since they rely on virtual memory protection mechanisms or page migration to work [38]. It would also be useful to add compiler-focused features to our tool. Our buffer meta-data could be passed as arguments to the kernel and, like the WebCL Validator [44] or CUDA-MEMCHECK [34], the compiler could add checks within the kernel to detect buffer overflows. This could pinpoint problems within the kernel and detect overflows in local and shared memory at the expense of vendor neutrality.

Finally, while this work focused on GPUs, more accelerators are being introduced into heterogeneous systems [4, 5, 17–20, 63, 76]. Building broadly useful tools that work across these accelerators will be an important area of work.

The tool described in this work is available at: https://github.com/GPUOpen-ProfessionalCompute-Tools/clARMOR

Acknowledgments

We would like to thank Arkaprava Basu, Leonardo Piga, and the anonymous reviewers for providing comments and suggestions that improved the final version of this paper. Thanks also to Michael LeBeane for his naming recommendations. AMD, the AMD Arrow logo, AMD Catalyst, FirePro, and combinations thereof are trademarks of Advanced Micro Devices, Inc. OpenCL is a trademark of Apple Inc. used by permission by Khronos. PCIe is a registered trademark of PCI-SIG Corporation. Windows is a registered trademark of Microsoft Corporation. Other names used herein are for identification purposes only and may be trademarks of their respective companies.
References


[38] M. Harris. CUDA 8 and Beyond. Presented at the GPU Technology Conf., 2016.


