INDEX

Future Work

7.1 Improvements to ASan
7.2 Enhancements to Valgrind

Conclusion

Introduction

1.1 Background

Memory safety is crucial for stable and secure computing systems. Bugs like buffer overflows, use-after-free vulnerabilities, and null pointer dereferences can cause crashes, data corruption, and even remote code execution attacks. However, C and C++ have a major drawback in memory management. These languages require manual memory allocation and deallocation, making them prone to vulnerabilities. Addressing these issues is essential to prevent exploits and safeguard user data.

Memory-related vulnerabilities in C and C++ code have long been a concern in software development. Various tools and techniques have been developed to mitigate these vulnerabilities, including memory safety mechanisms. These mechanisms detect and prevent memory-related vulnerabilities, such as buffer overflows and memory errors, at runtime. Popular tools like AddressSanitizer (ASan) and Msan are used for this purpose.

By employing memory safety mechanisms, programmers can minimize the risk of memory-related vulnerabilities in their code, enhancing reliability, security, and maintainability. Despite the challenges posed by memory management in C and C++, addressing these vulnerabilities is critical for developing secure and robust software systems.

Various security mechanisms have been developed to mitigate memory bugs in operating systems. However, they often come with trade-offs in terms of performance overhead, complexity, and compatibility. AddressSanitizer (ASan) and Valgrind are two prominent security mechanisms that aim to enhance memory safety in operating systems.

ASan is a runtime tool developed by Google that focuses on detecting memory bugs, such as buffer overflows and use-after-free errors. It operates by instrumenting the application's code during runtime, adding checks and additional metadata to each memory access. Valgrind uses a technique known as dynamic binary instrumentation (DBI) to analyze and monitor the execution of programs. It operates by running the target program in a virtual execution environment, often referred to as a "sandbox" or "virtual machine."

Despite their significance, there is a need to evaluate and compare the practicality and benefits of ASan and Valgrind in the context of an operating system project. This comparison will shed light on their respective strengths, weaknesses, performance characteristics and effectiveness in mitigating memory bugs.

1.2 Problem Statement

The problem we aim to address in this project is the prevalent occurrence of memory bugs in operating systems and the need for effective security mechanisms to mitigate them. These bugs can be exploited by attackers to gain unauthorized access, execute arbitrary code or cause system crashes, leading to service disruptions and potential data breaches.

Traditional approaches to ensuring memory safety, such as static analysis and manual code reviews, have limitations in detecting and preventing memory bugs, particularly in large and complex operating system projects. Moreover, these methods often require significant time and effort from developers and they cannot guarantee complete elimination of memory bugs.

By addressing memory bugs effectively, we can minimize the risks associated with software vulnerabilities and bolster the overall security posture of operating systems.

1.3 Objective

The objective of this project is to propose and implement a compelling solution that leverages existing security mechanisms to enhance memory safety in an operating system context. Various security mechanisms have been developed to mitigate memory bugs in operating systems. However, they often come with trade-offs in terms of performance overhead, complexity, and compatibility. AddressSanitizer (ASan) and Valgrind are two prominent security mechanisms that aim to enhance memory safety in operating systems.

By comparing ASan and Valgrind, we aim to evaluate their practicality, benefits and limitations, providing insights for developers and system administrators to make informed decisions about the most suitable security mechanism for their projects.

Our specific goals include:

- **Evaluate Performance:** Measure and compare the performance overhead of ASan and Valgrind when integrated into an operating system project. Assess the impact of these mechanisms on system resources, such as execution time and memory usage.

- **Assess Effectiveness:** Determine the effectiveness of ASan and Valgrind in detecting and mitigating memory bugs commonly found in operating systems. Analyze their ability to identify different types of vulnerabilities, such as buffer overflows, use-after-free errors, and null pointer dereferences.

- **Explore Limitations:** Identify and understand the limitations and challenges associated with ASan and Valgrind in an operating system context. Investigate scenarios where these mechanisms may exhibit false positives or false negatives, and evaluate their impact on system security and stability.

- **Practicality and Benefits:** Assess the practicality and benefits of integrating ASan and Valgrind into an operating system project. Consider factors such as ease of integration, compatibility with existing codebases, and the level of effort required for implementation and maintenance.

2. ASan (AddressSanitizer)

ASan (AddressSanitizer) is a runtime tool developed by Google that detects memory bugs, such as buffer overflows and use-after-free errors, in an operating system project. It operates by instrumenting the application's code during runtime, adding checks and metadata to each memory access. During the instrumentation process, ASan replaces the memory allocation and deallocation functions, such as malloc() and free(), with its own versions. Refer fig 1. These modified functions allocate additional memory to store metadata alongside each allocated object.

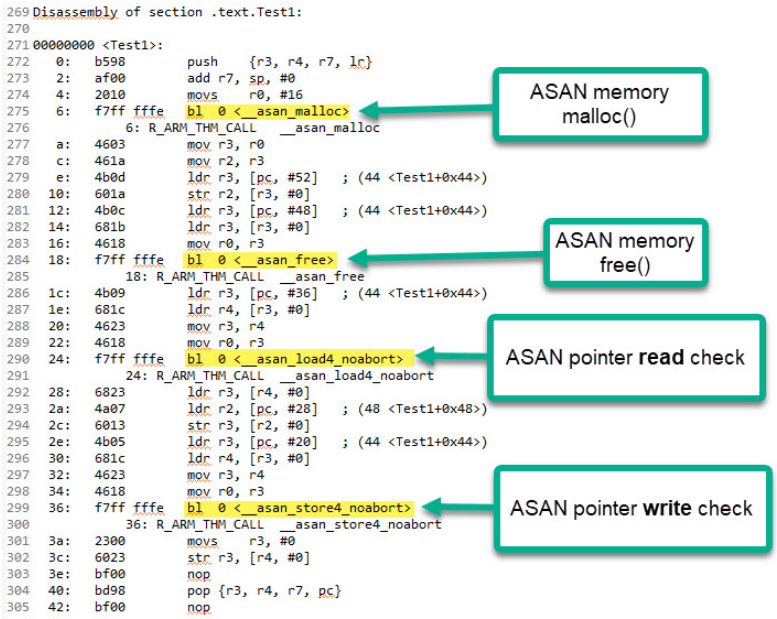

Fig 1. ASAN instrumenting the code for each pointer access and calling ASAN libraries routine to check if that read or write access through a pointer is valid

ASan can be integrated into an operating system project by compiling the code with the appropriate compiler flags and linking against the ASan runtime library.

3. Valgrind

Valgrind is a powerful open-source framework for debugging and profiling applications. It provides a suite of tools that help identify memory leaks, detect memory errors, profile performance, and analyze threading issues in programs. Valgrind operates by dynamically instrumenting the executable and running it in a virtual environment, allowing detailed analysis of memory operations and program behavior. Its ease of use and extensive capabilities make Valgrind a popular choice for developers and software testers seeking to improve the reliability and performance of their applications.

Valgrind has a variety of tools in its Tool Suite. Examples of tools include Memcheck, Cachegrind, Massif, Helgrind, etc. They are used for memory-management problems, cache profiling, heap profiling and thread debugging respectively. Each tool is used for a specific purpose. For our experiment, we are using " Valgrind Memcheck", which is a memory error detector. It detects common problems in C and C++ like memory leaks, using undefined values, overrunning and underrunning heap blocks, accessing already freed memory, overlapping source and destination pointers in "memcpy", fishy argument values (negative values in malloc) etc. For example,

4. Experimental Setup

4.1 Benchmark Selection - Parsec

The experimental setup involved conducting benchmark tests using the Parsec benchmark suite to compare the performance and effectiveness of ASan and Valgrind. Six out of 10 benchmarks were executed on a Linux operating system.
To evaluate ASan and Valgrind, they were compiled and instrumented separately. The Parsec benchmark suites were then executed and relevant metrics such as execution time and memory usage were measured.

Following benchmarks were used in evaluation:

1. **Blackscholes**: Blackscholes calculates prices for a portfolio of European options using Black-Scholes partial differential equation.The program divides the portfolio into several units, which is equal to the number of threads, and processes them concurrently. Inputs vary from 1 option(test) to = 10,000,000(native)

2. **Vips:** VASARI Image Processing System. The benchmark includes fundamental image operations like affine transformation and one convolution. The VASARI Image Processing System fuses all image operations to construct an image transformation pipeline that can operate on subsets of an image. VIPS used memory mapped I/O and the sizes of input range from 256x288(test) to 18000x18000(native)

3. **Fluidanimate:** Fluidanimate is used to simulate an incompressible fluid for interactive animation purposes. At every time step, fluidanimate executes 5 kernels: Rebuild spatial index,

Compute densities, Compute forces, Handle collisions with scene geometry, update positions of particles. Input sets range from 5000 particles, 1 frame(test) to 500,000, 500 frames(native).

4. **Streamcluster:** Solves the online clustering problem. Continuously produced data has to be organized under real-time conditions. The program spends most of its time evaluating the gain of opening a new center. This operation uses a parallelization scheme which employs static partitioning of data points. The program is memory bound for low-dimensional data and becomes increasingly computationally intensive as the dimensionality increases.

5. **Raytrace:** The raytrace application is a workload that renders a 3D scene using a technique called ray tracing, which can create photorealistic images by tracing the path of light. Ray tracing is computationally intensive but can produce effects like reflections and shadows that are difficult to incorporate into other rendering methods. The raytrace benchmark program uses a version of ray tracing optimized for speed rather than realism.

6. **Canneal:** The simulated annealing algorithm used by canneal is a heuristic optimization method that is inspired by the annealing process in metallurgy. Canneal is a demanding benchmark because it has high memory access requirements due to the large size of the netlist data structure. To handle this requirement, the benchmark is designed to be parallelized across multiple threads and the threads communicate by updating a global routing cost value.

4.2 Test Environment

Operating System: Ubuntu 22.04 LTS
Hardware: 8.0 GiB Memory, Intel Core i5-5200U CPU @ 2.20GHz x 4, 240GB disk capacity
Compilers: gcc (Ubuntu 11.3.0-1ubuntu~22.04) 11.3.0
Network Configuration: Standalone machine
Test cases: Six Parsec benchmarks

5 Results

In this section, we present the performance metrics obtained by running the Parsec benchmark suite with multiple memory sanitizers, including ASan and Msan. The performance evaluation aimed to assess the impact of these sanitizers on the execution time of the benchmarks.

5.1 Execution Time

Benchmark	Baseline Time (min)	Asan Time (min)	Time Factor	Valgrind Time (min)	Time Factor
Blackscholes	1.44	1.65	14.5	51.53	3478.47
Vips	1.50	3.32	121.3	36.14	2309.33
Fluidanimate	5.33	16.44	208.4	120.93	2168.86
Streamcluster	5.99	10.74	79.3	69.20	1055.26
raytrace	2.39	24.87	940.6	53.16	2124.27
canneal	2.84	3.73	31.3	12.48	339.44

Table 1 summarizes the performance metrics w.r.t. Execution time for each memory sanitizer across the Parsec benchmarks.

5.2 Bugs

These are the some of the common bugs we are trying to detech using ASan and Valgrind:

1. **Out-of-bound stack access**: Out of bounds stack access refers to accessing memory locations outside the bounds of the allocated stack. By accessing the memory beyond the bounds of the stack, the program crashes or behaves unexpectedly. Stack overflow is an example of an out-of-bounds stack access. When a program writes data beyond the allocated stack, a stack overflow occurs.
2. **Memory leaks:** Memory leaks occur when a program allocates memory but does not free it after usage. Such leaks occur when there is a bug in the program that prevents the memory from being freed.
3. **Buffer overflows:** Buffer overflows occur when a program writes data beyond the allocated buffer, leading to overwrites on adjacent memory locations. Buffer overflows are caused by programming errors like not checking the size of input data or copying data into a fixed-size buffer without any bound checks.
4. **Use-after-free errors:** Use-after-free errors occur when a programming error causes the code to access memory that has already been freed, leading to unpredictable behaviour, crashes, or security vulnerabilities.
5. **Uninitialized memory use:** Uninitialized memory user occurs when a program accesses uninitialized memory, i.e., memory that has not been properly initialized. Such bugs can lead to garbage values being accessed and can lead to unpredictable behavior and errors.
6. **Detecting data races:** Data races occur in multithreaded programs, where two or more threads access the same memory location concurrently, and at least one of them is a

write operation. When this happens, the order of the accesses depends on which thread accesses the memory location first. This leads to incorrect reads and writes.

6 Analysis

In the case of the **Vips** and **fluidanimate** benchmarks, there is a increase of 121.3% and 208.4% in execution times when using ASan compared to the baseline. This can be attributed to several factors that contribute to the additional time taken by ASan during the memory sanitization process. Since ASan's instrumentation process adds additional instructions and checks to each memory access, an extra computational overhead is introduced. As a result, the Vips and fluidanimate, which involve a large number of memory accesses, experience increased execution time due to the added ASan instrumentation.

When evaluated against **Raytrace** benchmark, ASan results in the highest time increase, with a staggering 940.6%. It should be noted that Raytrace involves a significant number of floating-point operations and these operations could be time-consuming. ASan's additional checks for each memory access, including those involving floating-point calculations, added to the computational overhead and further increased the execution time.

The performance impact of AddressSanitizer (ASan) on the **BlackScholes**, **Streamcluster** and **canneal** benchmark when compared to other benchmarks is comparatively low. BlackScholes, Stream-cluster and canneal exhibit memory access patterns that align well with ASan's instrumentation and detection mechanisms, resulting in a smaller performance impact.

Different benchmarks may have distinct levels of optimization applied, and specific optimizations can help alleviate the performance impact of ASan. The **Streamcluster** benchmark is more compliant to certain optimizations that mitigate the impact of ASan compared to other benchmarks.

ASan and Valgrind are dynamic analysis tools for detecting memory-based errors but have subtle differences in their working. ASan is a specific tool which detects memory errors like buffer overflows, uninitialized reads, etc by adding extra instrumentation to the code. On the other hand, Valgrind is a tool used for various types of analyses including memcheck (detects memory-management related errors primarily in C and C++ programs), Cachegrind (cache profiler), Helgrind (detects data races in multithreaded programs), Massif (heap profiler), etc. Therefore, ASan is a more specific tool while Valgrind has a variety of debugging and profiling tools. Valgrind has a significantly higher overhead than ASan because of this general purpose nature. Therefore, it slows down the execution by a factor of 5-10. ASan is relatively lighter. Valgrind emulates a virtual machine that executes the program, while ASan inserts runtime checks into the already compiled code. This makes Valgrind more efficient at detecting a range of errors which ASan cannot detect. Such examples include the detection of uninitialized stack memory or uninitialized heap memory allocated by calloc.

To conclude, Valgrind is a more general purpose and heavyweight tool, whereas ASan is relatively lightweight and is easily integrated into a developer's workflow.

References:

1. The PARSEC Benchmark Suite: Characterization and Architectural Implications (link)
2. Valgrind (link)
3. Memcheck (link)
4. ASan (link)
5. ASan code: (link)
6. Blackscholes (link)
7. Vips: (link)
8. Fluidanimate (link)
9. Streamcluster (link)
10. Raytrace: (link)

---

1. Bienia, C., Kumar, S., Singh, J. P., & Li, K. (2008). The PARSEC Benchmark Suite: Characterization and Architectural Implications. *Princeton University Technical Report TR-811-08*.
2. Valgrind https://valgrind.org/info/about.html
3. Memcheck https://valgrind.org/info/tools.html#memcheck