Reducing the Security Risks of C and C++ Programming Languages with Code Sanitizers

University of Turku
Department of Computing
Master of Science (Tech) Thesis
Cyber Security
May 2024
Fatih Uzunoğlu

Supervisors:
Seppo Virtanen
Tahir Mohammad
Jaakko Järvi

UNIVERSITY OF TURKU
Department of Computing

FATIH UZUNOĞLU: Reducing the Security Risks of C and C++ Programming Languages with Code Sanitizers

Master of Science (Tech) Thesis, 54 p., 2 app. p.
Cyber Security
May 2024

C and C++ programming languages have certain characteristics that can act like a double-edged sword. Having direct access to memory allows programmers to implement linked-lists and do pointer arithmetic operations but at the same time it also allows *buffer overflows*. Or, having vast amount of *undefined behaviors* allows the compilers to optimize the code to the full extent, but it also causes major vulnerabilities such as *double free* (*i.e.* trying to release part of memory that was already released). Code sanitizers, first introduced with the *Address Sanitizer* by Google in 2012, offer a flexible code weakness mitigation run-time mechanism.

In this thesis, the major code sanitizers were tested on popular open-source software with regard to their effectivenesses on finding bugs that have the potential of compromising security. It is also discussed if they can be employed in production (release mode), and if they can be an alternative to porting code into "secure" languages. It is found out that code sanitizers in certain cases can be used in production in order to reduce the attack surface. Furthermore, it is found out that if disk size is not scarce, *Undefined Behavior Sanitizer* can be utilized in production without noticeable impact on the performance or memory usage even though the increase of program size it induces has the potential of spoiling spatial locality for the CPU cache mechanisms.

It should be also noted that during experimentation, Undefined Behavior Sanitizer was able to find numerous undefined behaviors in an open-source *Minecraft* clone game named *Minetest*. One notable undefined behavior found was thought to cause behavioral divergence between Intel and ARM architectures due to casting negative floating-point number into unsigned integer. The patches containing the fixes which were merged into the upstream repositories are included in this thesis.

The research conducted while writing this thesis suggests that code sanitizers do not necessarily need to be limited being used in debugging sessions, as they are shown to be effective in eliminating the top *Common Weakness Enumeration* (CWE™) entries while having a modest overhead. The tests done on popular projects and real-world scenarios hint that code sanitizers deserve more attention from software developers.

Keywords: C, C++, Security, Code Sanitizers, LLVM Clang, GNU GCC, ASan, TSan, MSan, UBSan, SafeCLib, CheckedC, Valgrind

Contents

Appendices

1 Introduction

The saying "With great power comes great responsibility." applies to different fields. In the context of programming languages, it can be said that while it is possible to do a certain operation in one language, it might not be possible to do in a different language.

Computer science is a world of trade-offs. When it comes to programming, it may be considered fair to make use of certain kinds of practices in order to lessen the stress of debugging and improve maintenance. For example, *garbage collection* is a known mechanism that is provided built-in in several programming languages which relieves the programmers of managing objects' lifetimes.

In the same context, some programming languages provide semantics or features that allow low level operations which may be dangerous if not utilized properly. Manual management of memory enables more effective usage of memory, since the programmer may in most cases determine the lifetime of objects, but at the same time it may cause certain vulnerabilities if care is not taken.

The 2023 Common Weakness Enumeration (CWE™) Top 25 Most Dangerous Software Weaknesses list[1] shows that in 2023, the first three most common weaknesses were all memory related. In other words, these listed weaknesses (*use-after-free*, *buffer overflow*, *out-of-bounds write*) were all a consequence of certain programming languages having no built-in checks for memory operations and automatic

[1] `https://cwe.mitre.org/top25/archive/2023/2023_kev_list.html`

life-cycle management.

Although higher level programming languages such as *Python* have been rising in popularity in recent years [1], programming languages that are often considered low-level still have a big share in software development. If great responsibility can be provided, these so-called low level languages may shine in production regarding performance and efficiency. Especially certain fields, such as *embedded systems* and *game development* require using low level languages due to performance and resource constraints.

There can be many programming languages listed as low-level, but usually C and C++ are the main two programming languages that are known to be used when a low-level language is to be utilized. C, invented in the 1970s and C++, invented in 1985 followed the norms of that era. For example, it was not a big concern of not providing a garbage collection mechanism.

1.1 Problem Statement

Both C and C++ programming languages are considered unsafe. However, they are still pretty much relevant in the software industry due to reasons such as being allowed to do low level operations for improved performance and efficiency.

Although C++ initially improved many aspects of C, these improvements were not associated with the security aspect. For example, C++ provided *classes* but it did not provide a mechanism to handle pointers more securely [2] (until the *C++11* standard). Still, even though C++ with the newer standards has been providing more and more features that are beneficial for security, it still allows to do operations that might be considered dangerous. To summarize, the power C and C++ employ requires programmers to be cautious. A carelessly implemented feature may end up causing tremendous cost for companies.

In this thesis, code sanitizers are evaluated with regard to their capabilities of

finding coding errors and bugs that have the potential of compromising software security.

1.2 Research Questions

To address the issues defined in Section 1.1, the following research questions are defined:

1. Are code sanitizers in the current state effective in eliminating vulnerabilities?
2. How do code sanitizers in the current state compare to alternative solutions?
3. Are code sanitizers in the current state feasible to use in production with regard to their gradually improved overheads?

1.3 Research Objectives

To address the research questions defined in Section 1.2, the following research objectives are identified:

1. Investigate the current state of effectivenesses of the major code sanitizers in detecting security issues with regard to the top weaknesses designated in the Common Weakness Enumeration (CWE™) list.
2. Investigate the advantages and disadvantages of the major code sanitizers, and reach a conclusion on in which ways they are the most useful and can be a better alternative compared to other solutions used to eliminate security risks.
3. Benchmark the major code sanitizers on popular open-source projects to find out the current size, memory, and performance overheads of code sanitizers in order to justify using them in production.

```
struct fat_pointer {
  void* value; // Original pointer value
  void* base; // Base address of the intended referent
  size_t size; // Size of the referent
}
```

Listing 1.1: Instrumented pointer (fat pointer) [4].

1.4 Utility of Code Sanitizers

The concept of *Code Sanitizers* was introduced first as *Address Sanitizer* by *Google*. With code sanitizers, weaknesses may be detected during run-time automatically and relevant action may be taken.

Code sanitizers work by instrumenting program code so that extra steps are performed at run-time by the CPU. For example, Address Sanitizer works by using *shadow memory* to detect memory related problems [3]. Some code sanitizers use *fat pointers* which allows adding metadata to regular pointers [4]. Being able to associate metadata to pointers makes it possible to, for example, detect buffer over-read by storing the size of the pointed object (Listing 1.1).

So far, code sanitizers have been exclusively used in debugging sessions. However, as this thesis shows, they might be useful in certain other areas with subject to compromises. These areas are:

1. Utilizing code sanitizers for large code bases, instead of porting code into "secure" languages.

2. Utilizing code sanitizers in production, as long as the overhead can be tolerated.

If the overhead of code sanitizers is low, it may not be necessary to spend time and effort on porting legacy code -which is an ongoing trend-, but rather the code sanitizer may be employed in production. According to research [5] [6], it may be even possible to *de-bloat* existing code sanitizers, which further hints the feasibility

of employment of code sanitizers in production.

1.5 Organization of the Thesis

The structure of this thesis is as follows.

The findings on the literature regarding code sanitizers are described in Chapter 2. The literature suggests that code sanitizers are proven to be effective in order to eliminate security issues as well as other problems (*e.g.* memory leaks).

Both C and C++ programming languages suffer from the top weaknesses listed in the Common Weakness Enumeration (CWE™) list, and these are detailed in Chapter 3. Code sanitizers that target these weaknesses are then listed and addressed in Chapter 4.

Then, an elaborated discussion regarding the feasibility of code sanitizers is made in Chapter 5 based on their promises and the data collected by testing them.

Lastly, a conclusion which suggests utilizing code sanitizers more in software development is made in Chapter 6 and one Appendix A is enclosed which provides additional insight on code sanitizers by showing their outputs when they detect a problem in the code.

2 Literature Review

Code sanitizer, as a concept, was first introduced by *Google* in 2012 with *Address Sanitizer*. Since then, there have been different types of code sanitizers developed, such as *Undefined Behavior Sanitizer*[1], *Thread Sanitizer* [7], and *Memory Sanitizer* [8]. Most of the sanitizers are orthogonal regarding their functionalities, as they cover different weaknesses.

According to the research done by *Wagner* et al. [9], an approach of controlled security is possible by tweaking the program to maximize security within a given overhead rate. Their research employs utilization of code sanitizers. Based on this, it can be said that production code that runs fast enough can be employed with security functions. Increase of security can practically compensate the overhead, since the overhead would not be able to be realistically observed by the user.

NSan, which is a floating-point numerical sanitizer, follows a novel approach in order to detect floating-point issues with high performance, as introduced by *Courbet* in 2021 [10]. When existing code sanitizers are observed, it can be seen that although performance is not a target for many, there have been performance improvements made. This fact hints that over time it may be wise to re-evaluate deploying code sanitizers in production.

It should be noted that not all code sanitizers aim detecting spots that are a concern for security. For example, *Rotem* et al. introduces *Warrior1* in their paper

[1] `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`

"Warrior1: A Performance Sanitizer for C++" [11], which is a sanitizer that detects performance antipatterns.

Systematization of Knowledge (SoK) titled "SoK: Sanitizing for Security" prepared by *Song* et al. [4] covers the most important sanitizers that can be employed for increasing security with examples. These include the Address Sanitizer, Thread Sanitizer, Memory Sanitizer, and Undefined Behavior Sanitizer which are focused in this thesis.

Although not necessarily supported by all hardware configurations, it is possible to utilize certain hardware functionality to implement sanitizers. For example, *Hardware-assisted Address Sanitizer* (HWASAN)[2] is a variant of Address Sanitizer that makes use of *Address Tagging* available on `AArch64` (*ARM64*). According to the research titled "Memory Tagging and how it improves C/C++ memory safety" done by *Serebryany* et al. [12], a fully hardware assisted memory tagging (in heap-only mode) could have near-zero overhead except for very *malloc*-intensive applications, where the overhead would still remain at moderate levels.

Hardware assisted code sanitizing remains to be a feature exclusive to certain architecture(s) as of 2024. However, given the benefits, it can be expected that the support for other architectures, is going to gradually improve given the popularity.

However, memory tagging based address sanitizing has also weaknesses. According to the same research paper [12], the precision of buffer overflow detection with Hardware-assisted Address Sanitizer is not as high as with the regular Address Sanitizer. The precision increases as the *tag granule* decreases.

Consider Listing 2.1. The issue is that in `AArch64`, memory locations are tagged by adding four bits of metadata to each 16 bytes of physical memory, which is called as tag granule [13], and overflowing within the tag granule goes undetected.

[2] `https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html`

```
uint8_t *p = (uint8_t*)malloc(10);
p[12] = 0; // will NOT be detected by HWASAN in AArch64.
p[12] = 0; // will be detected by ASAN.
```

Listing 2.1: Buffer overflow detection and 16-byte tag granule.

3 Security Risks of C and C++ Programming Languages

Certain aspects of both C and C++ programming languages enable them to be useful in areas unlike higher level languages. For example, having direct access to memory is not a trivial thing in many languages. Also, the greater amount of undefined behaviors allows compilers to optimize code more.

In this chapter, the major risks of C and C++ programming languages are described. Since code sanitizers target these risks, it is thought to be beneficial to describe them in the thesis.

3.1 Memory-related risks

Both C and C++ allow direct access to memory. Direct memory pointers allow implementing certain features trivially. For example, a *linked list* is often implemented using pointers. In programming languages where there are no *raw* pointers, such as *Rust* (safe mode) or *Python*, it is harder to efficiently implement a linked list.

At the same time, pointers can be dangerous if not used carefully. For this reason, there are *smart pointers* that patch some shortcomings of raw pointers. Smart pointers are a C++-only feature, which were defined in the 2011 standard of the language [14] and have since been used widely in the software industry due to their benefits.

Still, there are cases that a raw pointer is a better choice to be used. Or, there might be a case that efforts of porting raw pointers into smart pointers in legacy code are not feasible. Code sanitizers may be helpful to employ in these situations as having direct access to the memory make it possible to cause buffer overflows, such as *Out-of-bounds Write* and *Out-of-bounds Read*.

3.1.1 CWE-787: Out-of-bounds Write

Out-of-bounds write (CWE-787[1]) is a situation in which memory is written outside intended boundaries. According to the 2023 Common Weakness Enumeration (CWE™) Top 25 Most Dangerous Software Weaknesses list, Out-of-bounds write is the **top weakness** in 2023.

It is possible to do code execution with out-of-bounds write because the program in memory can be arbitrarily modified. This is especially a great risk for server-client architecture because if the server contains out-of-bounds write weakness, a client may target the weakness and make the server behave in a way that might compromise the security of the whole system. Therefore, it can be stated that the code execution vulnerability aspect of this weakness makes it a very dangerous side effect of memory access in C and C++ programming languages. Fortunately, Address Sanitizer has the ability to eliminate this issue.

Out-of-bounds write is simple to demonstrate. In Listing 3.1, the size constant is incorrectly interpreted which leads to the weakness.

3.1.2 CWE-125: Out-of-bounds Read

Out-of-bounds read (CWE-125[2]) is a situation in which memory is read out of intended boundaries. According to the 2023 Common Weakness Enumeration (CWE™)

[1] `https://cwe.mitre.org/data/definitions/787.html`

[2] `https://cwe.mitre.org/data/definitions/125.html`

```c
#include <string.h>
#include <assert.h>

// A function shifts every character except
// the trailing NULL character by one:
void shiftString(char* const string) {
    assert(string);
    // Here, `<=` is used instead of `<`
    // leading to writing memory beyond
    // the string's boundaries:
    for (size_t i = 0; i <= strlen(string); ++i) {
        ++(string[i]);
    }
    // This function will corrupt the memory
    // each time it is used.
}
```

Listing 3.1: Out-of-bounds write in C.

Top 25 Most Dangerous Software Weaknesses list, Out-of-bounds read is the **7th top weakness** in 2023.

Although considerably less dangerous than out-of-bound write, out-of-bound read may also cause significant vulnerabilities. For example, sensitive data residing in memory can be exposed with such vulnerability, causing information leakage. Again, Address Sanitizer may be employed to prevent these kinds of vulnerabilities.

An example of the weakness is using string functions where the input string does not end with a trailing null character.

3.1.3 CWE-416: Use After Free

Use-after-free (CWE-416[3]) refers to a situation where referencing of a memory location occurs after it was freed. Such weakness might crash the program or cause code execution, which makes it as dangerous as CWE-787 (Out-of-bounds Write, Section 3.1.1).

Usually, pointers are set to `NULL` (C) or *nullptr* (C++) after the object they point to are released from the memory (freed). Then, with proper checks it is possible to

[3] `https://cwe.mitre.org/data/definitions/416.html`

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

char* string = NULL;

// A function which prints the global `string`
void consume() {
    if (string) {
        // By the time string is checked, it might be already
        // freed, causing Use After Free condition.
        printf("%s", string);
    }
}

// A function which updates the global `string`
void update(const char* const input) {
    free(string);
    if (input) {
        // clone the input string (allocate memory)
        string = strdup(input);
        assert(string);
    } else {
        string = NULL;
    }
}

// If `consume()` and `update()` are called in parallel
// such as different threads call these functions,
// then it is possible that a TOCTOU issue would trigger
// Use After Free.
```

Listing 3.2: TOCTOU induced use after free in C.

avoid this vulnerability. However, this might not be feasible in all cases.

For example, a *TOCTOU* (Time-of-check Time-of-use) (CWE-367[4]) condition might make it possible for a pointer to be slipped from a check. A data race between different threads might cause TOCTOU induced Use After Free. In Listing 3.2, it can be seen that it is possible to use a pointer that has already been invalidated after the check due to parallel code execution.

[4] https://cwe.mitre.org/data/definitions/367.html

3.1.4 CWE-415: Double Free

Double free (CWE-415[5]) refers to a situation where a portion of memory is being freed where it was already freed.

Double free usually corrupts the program memory, which means that the program would not run as intended afterwards.

3.1.5 CWE-476: NULL Pointer Dereference

Null pointer dereference (CWE-476[6]) is a situation where a pointer pointing to a "null" address is dereferenced.

Null pointer dereference can be considered a "safer" weakness, because it usually only leads to program crash rather than code execution.

This weakness is often caused by forgetting an *if* statement or an *assertion*.

3.2 Insecure Features

C, being invented in 1972, contains some features that are considered to be unsafe. Although some of these features are deprecated by the new iterations of the language, some are not. These include the following (the list is not exhaustive):

1. `free()` does not automatically set the pointer to `NULL`. This may lead to use after free condition if the programmer is not careful.

2. Implicit type casting may cause integer overflows, which may be hard to detect.

3. Constant being non-default allows overriding variables that are meant to be constant, if the programmer forgets to put the `const` qualifier.

[5] `https://cwe.mitre.org/data/definitions/415.html`

[6] `https://cwe.mitre.org/data/definitions/476.html`

Since C++ carries C's legacy, most of the insecure aspects are carried over for compatibility reasons.

3.3 Undefined and Implementation Defined Behaviors

The general standpoint of both C and C++ standards can be considered in a way that everything is undefined except it is necessary for something to be defined.

Implementation defined behavior refers to a situation where a behavior might differ between different kinds of implementations. For example, different toolchains may act differently.

The reason for allowing both undefined and implementation defined behavior is usually optimization. Compilers have the ability to optimize code because the programmer cannot expect undefined behaviors to behave in a particular way.

However, if the programmer is not careful enough, undefined behaviors may cause vulnerabilities. Undefined Behavior Sanitizer aims to detect the undefined behaviors residing in program code.

4 Major Code Sanitizers

In this thesis, only the most common sanitizers that come by default with the major toolchains (*LLVM Clang*, *GNU GCC*, and *MSVC*) are covered. It should be noted that since the major code sanitizers are backed by Google, the LLVM project provides the ultimate support for code sanitizers since it is also backed by Google. For example, Memory Sanitizer (MSan) is not supported by the GNU toolchain.

Sanitizers are tools designed to spot certain weaknesses at run-time. A dynamic approach for detecting bugs makes it feasible to find more bugs compared to a static approach (static code analyzers).

The major code sanitizers have been available in major toolchains by default. That is, in order to make use of code sanitizers no additional installation needs to be made. However, code sanitizers are an *opt-in* preference due to their impact on the performance so certain toolchain flags must be set in order to enable them.

4.1 Address Sanitizer (ASan)

Address Sanitizer (*abbr.* ASan) was the first public code sanitizer introduced by Google in 2012. It is used to sanitize address related problems in C and C++ code.

Address Sanitizer is reported to be able to detect the following conditions[1]. It can be configured to target a specific condition rather than its default broad range of detection.

[1] `https://github.com/google/sanitizers/wiki/AddressSanitizer`

1. Use after free/return/scope (Discussed in Section 3.1.3).
2. Heap buffer overflow (Discussed in Sections 3.1.1, 3.1.2).
3. Stack buffer overflow (Discussed in Sections 3.1.1, 3.1.2).
4. Global buffer overflow (Discussed in Sections 3.1.1, 3.1.2).
5. Initialization order bugs (Discussed in Section 4.1.1).
6. Memory leaks. (Discussed in Section 4.1.1)

Since *stack* and *heap* memory allocations are possible in both C and C++, detecting overflow issues for both is essential to mitigate vulnerabilities. As listed above, Address Sanitizer is reported to be helpful for this condition.

Described more clearly in Section 5.7, Address Sanitizer suffers from causing considerable impact on performance, and requiring a great amount of virtual memory. The reason for the suffering is due to its working algorithm. As described in the paper [3], Address Sanitizer works by *poisoning* the *red zones* (*i.e.*, memory areas that are set as trap for buggy code or exploits to access) around memory allocations to catch unauthorized access. In other words, the neighborhood of an allocation is padded from both sides with safe areas by Address Sanitizer.

The trade-off between security and resource consumption has to be considered carefully in order to find an optimum point. For example, in certain conditions Address Sanitizer may not catch partial out-of-bound memory access[2].

According to the official docs[3], the average slowdown is reported to be 2x, which makes it not so feasible to use in production. However, the reported slowdown is not up-to-date and the limited experiments conducted during writing this thesis show a more optimistic result.

[2] `https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm#unaligned-accesses`

[3] `https://github.com/google/sanitizers/wiki/AddressSanitizer#introduction`

In addition to having increased memory usage, Address Sanitizer also has the following major limitations[4]:

- Requiring to map more than 16 terabytes of memory on 64-bit platforms. It should be noted that this amount of memory is not reserved, but only mapped.
- No support for static linking.

Address Sanitizer can be enabled with the `-fsanitize=memory` preference on GNU and LLVM toolchains.

4.1.1 Use Cases

Address Sanitizer in its default configuration is able to detect the issues demonstrated in Sections 3.1.1 3.1.2, and 3.1.3.

On top of that, the following situations can also be detected by Address Sanitizer:

Use After Return/Scope

Often times in C, less often in C++, a function/method (*e.g.* factory methods) returns a pointer. Usually in these cases the function allocates memory in the heap and the ownership is transferred to the caller. However, if somehow the address of a stack-allocated variable is returned as a pointer, depending on the calling conventions, by the time the caller accesses the returned address the object would be already destroyed since the stack context of the function would no longer be valid.

This situation has similar consequences as Use-After-Free (Discussed in Section 3.1.3). Since Use-After-Free is a very dangerous security concern, the Use After Return feature detection of Address Sanitizer makes it a feasible tool for security hardening.

[4] `https://clang.llvm.org/docs/AddressSanitizer.html#limitations`

Address Sanitizer does not detect use after return by default, possibly due to this weakness being rare. If needed, it should be configured explicitly to detect use after return weaknesses.

Static Initialization Order Fiasco

Static Initialization Order Fiasco refers to the situation in which the initialization order of static storage duration of variables cannot be determined in a meaningful way. In other words, the specification does not provide ways of deterministically finding out the ordering of initialization.

Within a single *translation unit*, the initialization is deemed to occur from top to bottom[5] therefore making the problem disappear for certain cases. Usually, C++ programs are compiled with multiple translation units, thus static initialization order fiasco is a notorious problem.

Address Sanitizer can be used to prevent static initialization order fiasco to occur. However, it is reported by the official docs to be turned off by default[6].

Address Sanitizer can work in two modes to detect static initialization order fiasco:

Loose order checking. In loose order checking, ASan detects if a statically stored variable depends on another statically stored variable from another translation unit in which its value is not initialized when it is accessed.

Strict order checking. In strict order checking, ASan detects if a statically stored variable depends on another statically stored variable from another translation unit.

[5] `https://en.cppreference.com/w/cpp/language/siof`

[6] `https://github.com/google/sanitizers/wiki/AddressSanitizerInitializationOrderFiasco#initialization-order-checker-in-addresssanitizer`

Depending on the mode, false-positives may be induced. Therefore, whether to select the mode or not requires careful consideration.

Memory Leaks

Although memory leaks are often not attributed to security, detecting leaks can be crucial for programs to not exhaust the system resources.

Properly written programs should not leak memory. Whether or not properly written, garbage collection in some programming languages is purposed to take care of the leaks (although still possible[7]). Since C and C++ do not have a garbage collection mechanism, it is essential to not cause memory leaks by improper use of raw pointers.

Address Sanitizer can be employed to detect memory leaks in C and C++ code where raw pointers are used. If only reference counting smart pointers are used in C++ code, it might be unnecessary to enable the memory leak detection mechanism of Address Sanitizer.

An example memory leak can be seen in Listing 4.1. In the listing, it is shown that in different contexts it may be hard to interpret ownership, or a release operation might just be forgotten by the programmer. The output of Address Sanitizer can be seen in Listing A.1.

4.2 Memory Sanitizer (MSan)

Memory Sanitizer (*abbr.* MSan) is a code sanitizer for C and C++ introduced by Google and designed to detect uninitialized memory reads.

Uninitialized memory read is a commonly encountered code weakness that is often not found in programming languages other than C and C++. The reason

[7] `https://www.usenix.org/legacy/event/usenix08/tech/full_papers/tang/tang_html`

```c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

// A function that trims the input string.
// Caller becomes the owner of the return string.
char *trimString(const char* const inputString) {
  const size_t length = strlen(inputString);
  size_t leftWhiteSpaceCount = 0;
  size_t rightWhiteSpaceCount = 0;

  for (size_t i = 0; i < length; ++i) {
    if (inputString[i] == ' ')
      ++leftWhiteSpaceCount;
    else
      break;
  }

  for (size_t i = length - 1; i >= leftWhiteSpaceCount; --i) {
    if (inputString[i] == ' ')
      ++rightWhiteSpaceCount;
    else
      break;
  }

  const size_t trimmedLength = length -
    (leftWhiteSpaceCount + rightWhiteSpaceCount);
  // + 1 is for null termination.
  char * const trimmedString =
    malloc(sizeof(char) * (trimmedLength + 1));
  if (!trimmedString)
    return NULL;

  strncpy(trimmedString, inputString + leftWhiteSpaceCount,
          trimmedLength);

  trimmedString[trimmedLength] = '\0';

  return trimmedString;
}

void parseAndPrintString(const char* const string) {
  // Trim the forwarded string, not realizing the caller becomes
  // the owner of the allocated string:
  printf("%s", trimString(string));

  // Leak: trimmedString should have been released from memory.
}
```

Listing 4.1: Memory leak example in C.

```cpp
#include <iostream>

struct Rectangle {
  int x, y;
  int width, height;
};

int area(const Rectangle& rectangle) {
  return rectangle.width * rectangle.height;
}

int main() {
  Rectangle rectangle1; // uninitialized members
  Rectangle rectangle2 {}; // initialized members

  int area1 = area(rectangle1); // area1 can be any integer
  int area2 = area(rectangle2); // area2 is 0

  if (area1 == area2)
    std::cout << "Not expected case." << std::endl;
  else
    std::cout << "Expected case." << std::endl;

  return 0;
}
```

Listing 4.2: Utilization of uninitialized memory in C++.

for that is that C and C++ do not initialize variables by default while many other languages do.

This weakness may be tricky for programmers to spot, consider the Listing 4.2. In this particular example the security risks would be limited because function pointers are not involved. However, when function pointers are involved, then remote code execution could happen.

Uninitialized memory read has similar consequences as Use After Free (Discussed in Section 3.1.3). In both cases, the content of the memory location is not predictable and is considered garbage. The output of Memory Sanitizer can be seen in Listing A.2.

Memory Sanitizer, as of January 2024, is not supported by the GCC toolchain.

Memory Sanitizer requires Address Space Layout Randomization (ASLR) support, notably enabled by `-fPIE` in case of GNU Toolchain. ASLR is often used

for security hardening, which achieves randomization of the memory addresses of processes in order to prevent exploitation [15].

Memory Sanitizer can be enabled with the `-fsanitize=memory` setting.

Memory Sanitizer has the following major limitations[8]:

- Requiring to map 64 TB of memory. It should be noted that this amount of memory is not reserved, but only mapped.
- No support for static linking.
- Usage of **2x** more real memory than a native run, and **3x** more with origin tracking feature.

4.3 Thread Sanitizer (TSan)

Thread Sanitizer (*abbr.* TSan) is another code sanitizer introduced by Google, and it is designed to detect data races.

A *thread* can be considered as an entity in which a dedicated portion of code runs in parallel to other threads. This feature makes it feasible to write fast performing programs because certain parts can run in parallel, yielding a faster result for calculations and operations.

However, as described in Section 3.1.3, data races may occur due to parallel code accessing the same part of memory. In this situation, if both threads read at the same time there is no concern. However, if both threads write or if one writes and the other one reads at the same time, this situation becomes dangerous.

Two threads must be synchronized in order to solve a race condition. This can be achieved in multiple ways, but the main preference for many programmers is using *mutexes* [16]. In some cases lock-free updates are an alternative, with the help of atomic read and write operations.

[8] `https://clang.llvm.org/docs/MemorySanitizer.html#limitations`

Since debugging parallel running code is notoriously harder than debugging single-threaded code, a data race might be hard to spot. In that sense, Thread Sanitizer may be employed to detect these data races.

Thread Sanitizer has the following major limitations[9]:

- Having memory overhead of **5x**, in addition to +1Mb per each thread, although memory overhead can be configured with settings.
- Requiring to map unspecified amount of memory. It should be noted that memory is not reserved, but only mapped.
- No support for static linking.
- Usage of **2x** more real memory than a native run, and **3x** more with origin tracking feature.
- Non-position-independent executables are not supported.

A data race that is caught by the default configuration of Thread Sanitizer can be seen in Listing 4.3. The output of Thread Sanitizer can be seen in Listing A.4.

4.4 Undefined Behavior Sanitizer (UBSan)

Undefined Behavior Sanitizer (*abbr.* UBSan) is a code sanitizer that specializes in detecting and tracing undefined behaviors. As explained in Section 3.3, the standard set of both C and C++ programming languages explicitly do not define every behavior possible. As with other cases in the field of Computer Science, this is a trade-off that is deliberately chosen by the standard makers, and it gives room for the toolchains to optimize the program.

[9] `https://clang.llvm.org/docs/ThreadSanitizer.html#limitations`

```c
#include <stdio.h>
#include <pthread.h>
#include <assert.h>

void* incrementInteger(void *intPtr) {
  assert(intPtr);
  ++(*(int*)intPtr);
  return NULL;
}

int main() {
  int variable = 0;
  pthread_t thread;
  pthread_create(&thread, NULL, incrementInteger, &variable);

  // Data race: (both main and `thread` threads write to, and
  //             main thread reads the variable)
  if (variable) ++variable;

  printf("%d\n", variable); // Not certain if variable is 1 or 2.
  pthread_join(thread, NULL);

  return 0;
}
```

Listing 4.3: Data race between threads in C.

Undefined behavior sanitizer, unlike other code sanitizers, is reported to yield relatively less overhead. The compactness of undefined behavior sanitizer makes it feasible to be used in production for increased security.

UBSan works in a modular way. if `-fsanitize=undefined` is used, all checks are activated except `float-divide-by-zero`, `unsigned-integer-overflow`, `implicit-conversion`, `local-bounds`, and `nullability-*`.

4.4.1 Use Cases

It can be assumed that UBSan is able to detect all of the common undefined behaviors. The following list which is not exhaustive, indicates the common undefined behavior occurrences:

-fsanitize=bool Utilization of a boolean which is not *true* or *false*.

```
1 int main() {
2   int a;
3   int b;
4   int c = *(&a + 1); // c is not necessarily b.
5   return 0;
6 }
```

Listing 4.4: Illegal pointer dereference.

-fsanitize=null Detection of null pointer dereference, as explained in Section 3.1.5.

-fsanitize=enum Detection of assigning an *enum* to a value that cannot be represented by that enumeration.

-fsanitize=bounds Detection of utilizing an array outside its bounds, but only when the bounds are statically known. If the bounds are not statically known, Address Sanitizer (Section 4.1) should be used.

-fsanitize=implicit-{signed,unsigned}-integer-truncation Detection of implicit casting of integer types, where information loss occurs. For example, implicit casting of `long int` type to `short` type.

-fsanitize={integer,float}-divide-by-zero Detection of divide-by-zero bugs.

-fsanitize=signed-integer-overflow Detection of integer overflows. For example, `int i = INT_MAX + 1;` can result in the number being represented as negative. In most implementations, Two's Complement[10] methodology is used to represent negative numbers, and overflows occur because the most significant bit (MSB), which indicates the sign of the number, flips.

An example use case regarding undefined pointer reference can be seen in Listing 4.4 and the output of Undefined Behavior Sanitizer can be seen in Listing A.3.

[10] `https://en.wikipedia.org/wiki/Two%27s_complement`

4.4.2 Undefined Behaviors Found in Minetest Project

During experimentation, undefined behaviors were found in *Minetest* project and its dependency game engine *IrrlichtMt*. Fixes were made and provided to the upstream repositories[11][12]. The undefined behaviors found can be listed as such:

Class member variable of type boolean is not initialized before use. The C++ standard [14] states (3.9.1) that "Using a bool value in ways described by this International Standard as undefined, such as by examining the value of an uninitialized automatic object, might cause it to behave as if it is neither true nor false".

Taking address of a variable that belongs to a packed structure. Members of a packed structure should not be addressed, because they might not result in an aligned pointer. In certain architectures this may result in an exception (crash), and in others it may cause performance penalty.

Assigning potentially big double to unsigned integer. The C++ standard [14] states (4.9.1) that "A prvalue of a floating point type can be converted to a prvalue of an integer type. The conversion truncates; that is, the fractional part is discarded. The behavior is undefined if the truncated value cannot be represented in the destination type".

Assigning negative floating point number to unsigned integer. The same standard clause quoted in the previous entry ("Assigning potentially big double to unsigned integer") applies to this as well. Additionally, *Burkhard Stubert* explains in a blog post[13] regarding a real world implication of this issue. It is

[11] `https://github.com/minetest/irrlicht/pull/288`

[12] `https://github.com/minetest/minetest/pull/14365`

[13] `https://embeddeduse.com/2013/08/25/casting-a-negative-float-to-an-unsigned-int`

described that while on Intel the cast yields an expected result, on ARM such operation yields to 0.

GNU's Undefined Behavior sanitizer revealed a compiler bug regarding initialization of *thread_local* variables such that a global *thread_local* variable of type *LogStream* is not initialized upon first use within `main()` for the purposes of logging. The C++ standard (3.7.2) says that [14]:

> *A variable with thread storage duration shall be initialized before its first odr-use (3.2) and, if constructed, shall be destroyed on thread exit.*

```
1 /home/fuzun/minetest/src/log.h:304:21: runtime error: member access within null
    ↪  pointer of type 'struct LogStream'
2   #0 0x5592388a201e in getAvailableWorlds() (/home/fuzun/minetest/bin/
    ↪ minetest+0x1b9701e)
3   #1 0x5592383c9207 in main (/home/fuzun/minetest/bin/minetest+0x16be207)
4   #2 0x7f2fea992d8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.
    ↪ h:58
5   #3 0x7f2fea992e3f in __libc_start_main_impl ../csu/libc-start.c:392
6   #4 0x5592383d57c4 in _start (/home/fuzun/minetest/bin/minetest+0x16ca7c4
    ↪ )
```

Listing 4.5: Stack trace output of Minetest upon undefined behavior occurrence during start-up.

In order to investigate the issue, `UBSAN_OPTIONS=print_stacktrace=1` environment variable was enabled to make Undefined Behavior Sanitizer to print the stack trace upon the occurrence of an undefined behavior. The stack trace, which can be seen in Listing 4.5 showed the usage of printing logs through `infostream` within the function `getAvailableWorlds()` located in `content/subgames.cpp` file, which is a *thread_local* variable of type *LogStream*. Commenting the lines of such usages fixed the issue. It should be noted that this issue did not happen with the LLVM toolchain.

On the other hand, LLVM's Undefined Behavior Sanitizer could not find an undefined behavior that is found by GNU's Undefined Behavior Sanitizer (Listing 4.6),

```
From: Fatih Uzunoglu <fuzun54@outlook.com>
Date: Sun, 11 Feb 2024 20:32:52 +0200
Subject: [PATCH 1/3] Initialize member `floats` in ContentFeatures

---
 src/nodedef.cpp | 2 ++
 1 file changed, 2 insertions(+)

diff --git a/src/nodedef.cpp b/src/nodedef.cpp
index c707511ed..7efac7fc0 100644
--- a/src/nodedef.cpp
+++ b/src/nodedef.cpp
@@ -356,6 +356,8 @@ void ContentFeatures::reset()
  has_on_construct = false;
  has_on_destruct = false;
  has_after_destruct = false;
+ floats = false;
+
  /*
    Actual data
```

Listing 4.6: Patch: Initialize member ‘floats’ in ContentFeatures. (Green indicates addition, red indicates removal.)

and GNU’s Undefined Behavior Sanitizer could not find two undefined behaviors that are found by LLVM’s Undefined Behavior Sanitizer (Listings 4.7, 4.8).

The patches sent to the upstream repositories can be seen in:

Minetest Listings 4.6, 4.7, and 4.8.

Irrlicht Engine Listings 4.9, and 4.10.

```
From: Fatih Uzunoglu <fuzun54@outlook.com>
Date: Sun, 11 Feb 2024 22:58:39 +0200
Subject: [PATCH 2/3] Do not assign big double to u32

---
 src/tool.cpp | 3 +--
 1 file changed, 1 insertion(+), 2 deletions(-)

diff --git a/src/tool.cpp b/src/tool.cpp
index 36ad1c608..3f3c2f7bd 100644
--- a/src/tool.cpp
+++ b/src/tool.cpp
@@ -416,8 +416,7 @@ DigParams getDigParams(const ItemGroupList &groups,
     // The actual number of uses increases
     // exponentially with leveldiff.
     // If the levels are equal, real_uses equals cap.uses.
-    u32 real_uses = cap.uses * pow(3.0, leveldiff);
-    real_uses = MYMIN(real_uses, U16_MAX);
+    const u32 real_uses = std::min<f64>(cap.uses * pow(3.0,leveldiff),U16_MAX);
     result_wear = calculateResultWear(real_uses, initial_wear);
     result_main_group = groupname;
   }
```

Listing 4.7: Patch: Do not assign big double to u32. (Green indicates addition, red indicates removal.)

```
From: Fatih Uzunoglu <fuzun54@outlook.com>
Date: Sun, 11 Feb 2024 23:00:21 +0200
Subject: [PATCH 3/3] Do not assign negative floating point number to unsigned
 integer

---
 src/mapgen/mg_biome.cpp | 6 +++++-
 1 file changed, 5 insertions(+), 1 deletion(-)

diff --git a/src/mapgen/mg_biome.cpp b/src/mapgen/mg_biome.cpp
index 5a4501693..beb457c0f 100644
--- a/src/mapgen/mg_biome.cpp
+++ b/src/mapgen/mg_biome.cpp
@@ -297,7 +297,11 @@ Biome *BiomeGenOriginal::calcBiomeFromNoise(float
    ↪ heat, float humidity, v3s16 po
  // Carefully tune pseudorandom seed variation to avoid single node dither
  // and create larger scale blending patterns similar to horizontal biome
  // blend.
- const u64 seed = pos.Y + (heat + humidity) * 0.9f;
+ // The calculation can be a negative floating point number, which is an
+ // undefined behavior if assigned to unsigned integer. Cast the result
+ // into signed integer before it is casted into unsigned integer to
+ // eliminate the undefined behavior.
+ const u64 seed = static_cast<s64>(pos.Y + (heat + humidity) * 0.9f);
  PcgRandom rng(seed);

  if (biome_closest_blend && dist_min_blend <= dist_min &&
```

Listing 4.8: Patch: Do not assign negative floating point number to unsigned integer. (Green indicates addition, red indicates removal.)

```
From: Fatih Uzunoglu <fuzun54@outlook.com>
Date: Sun, 11 Feb 2024 19:21:06 +0200
Subject: [PATCH 1/2] SDL: set default for ActiveIcon

getActiveIcon() may be called before ActiveIcon
is initialized, leading to undefined behavior.
---
 source/Irrlicht/CIrrDeviceSDL.h | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

diff --git a/source/Irrlicht/CIrrDeviceSDL.h b/source/Irrlicht/CIrrDeviceSDL.h
index ebd8e03..7b5f011 100644
--- a/source/Irrlicht/CIrrDeviceSDL.h
+++ b/source/Irrlicht/CIrrDeviceSDL.h
@@ -260,7 +260,7 @@ namespace irr
         }
       };
       std::vector<std::unique_ptr<SDL_Cursor, CursorDeleter>> Cursors;
-    gui::ECURSOR_ICON ActiveIcon;
+    gui::ECURSOR_ICON ActiveIcon = gui::ECURSOR_ICON::ECI_NORMAL;
   };

  private:
```

Listing 4.9: Patch: SDL: set default for ActiveIcon. (Green indicates addition, red indicates removal.)

```
From: Fatih Uzunoglu <fuzun54@outlook.com>
Date: Sun, 11 Feb 2024 19:28:32 +0200
Subject: [PATCH 2/2] Do not use `core::max_()` with variable that cannot be
 addressed

header struct is packed, and `core::max_()` takes the input as reference.
---
 source/Irrlicht/CImageLoaderTGA.cpp | 4 +++-
 1 file changed, 3 insertions(+), 1 deletion(-)

diff --git a/source/Irrlicht/CImageLoaderTGA.cpp b/source/Irrlicht/
    ↪ CImageLoaderTGA.cpp
index bc8298f..9324f3d 100644
--- a/source/Irrlicht/CImageLoaderTGA.cpp
+++ b/source/Irrlicht/CImageLoaderTGA.cpp
@@ -10,6 +10,7 @@
 #include "CImage.h"
 #include "irrString.h"

+#define MAX(x, y) (((x) > (y)) ? (x) : (y))

 namespace irr
 {
@@ -138,7 +139,8 @@ IImage* CImageLoaderTGA::loadImage(io::IReadFile* file
    ↪ ) const
  if (header.ColorMapType)
  {
    // Create 32 bit palette
-   const irr::u16 paletteSize = core::max_((u16)256, header.ColorMapLength);
+   // `core::max_()` is not used here because it takes its inputs as references.
+   const irr::u16 paletteSize = MAX((u16)256u, header.ColorMapLength);
    palette = new u32[paletteSize];

    if( paletteSize > header.ColorMapLength )
```

Listing 4.10: Patch: Do not use 'core::max_()' with variable that cannot be addressed. (Green indicates addition, red indicates removal.)

5 Experiments and Discussion

In this chapter, results of experiments regarding feasibility of using code sanitizers in production are presented and discussion on code sanitizers and alternative approaches are made.

All experiments reported in this chapter were done with the following toolchains on `x86_64-pc-linux-gnu` architecture.

LLVM `Ubuntu clang version 14.0.0-1ubuntu1.1.`

GNU `gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0`

Also, the hardware configuration used during experimenting is the following:

CPU `AMD Ryzen 7 5800H`

GPU `NVidia RTX 3070 Mobile`

Note that Memory Sanitizer is not included in Sections 5.2.2 and 5.2.3 since it requires instrumented `libc++`, which requires building LLVM `libc++` with Memory Sanitizer and using different standard libraries for different code sanitizers could skew the results.

5.1 Using Code Sanitizers for Vulnerability Mitigation

As demonstrated in the previous sections, code sanitizers provide a flexible opportunity of catching code weaknesses that ultimately cause vulnerabilities. However, they also induce overhead.

The most notable overhead across code sanitizers is reported to be requiring more memory, and requiring to map significant amount of virtual address space. This means that on platforms where *ulimit* is not available for resource limit configuration, it may be not feasible to use code sanitizers at all.

Another limitation that is common to code sanitizers is reported to be that they do not support code fortification[1] (`FORTIFY_SOURCE`), which is a security feature[2] that enables lightweight support for detecting buffer overflows. As described in the *Red Hat* article, `FORTIFY_SOURCE` does not detect all kinds of buffer overflows. Thus code sanitizers can be employed after disabling source fortification since Address Sanitizer can detect buffer overflows more exhaustively.

Undefined Behavior Sanitizer (Section 4.4) is reported to be the lightest among the code sanitizers described in this thesis. If memory issues are not a concern, using only Undefined Behavior Sanitizer may provide enough security.

It should be possible to selectively enable some code sanitizers at run-time. This mechanism makes it possible to minimize the overhead caused by code sanitizers depending on the environment.

[1] `https://github.com/google/sanitizers/issues/247`

[2] `https://www.redhat.com/en/blog/enhance-application-security-fortifysource`

5.2 Using Code Sanitizers in Production

As already mentioned above, code sanitizers cause increased memory usage and decreased performance. Although it is experimentally proven that code sanitizers have much less overhead compared to similar tools, such as Valgrind, their impact on resource use may not meet certain expectations in particular configurations. For example, using code sanitizers in production in the field of game development may not be feasible.

At the same time, an increasing trend of using code sanitizers in production has been observed by *Szabolcs Nagy*. Nagy warned in 2016 that Address Sanitizer is not ready for use cases in production, as can be seen with the following direct quotation[3]:

> *There is an alarming trend that Address Sanitizer and related compiler instrumentations from compiler-rt are used as a hardening solution and run in production. Even though these are debugging and testing tools, there is no clear warning against production use in their documentation*
>
> *...*
>
> *ASan should not be used for hardening in production systems in its current form, so at least the language ("hardening", "protection", "safe") should be fixed.*
>
> *My simple local root exploit is that ASan uses a lot of environment variables without checking for secure execution of setuid binaries:*
>
> ```
> ASAN_OPTIONS='verbosity=2 log_path=foo' ./suid.exe
> ```
>
> *...*

The warning was made in 2016, particularly towards Address Sanitizer; it is not clear if the warning is still relevant in 2024. Still, care must be taken to ensure that

[3] `https://www.openwall.com/lists/oss-security/2016/02/17/9`

these pitfalls are not encountered if code sanitizers are used in production.

For the particular issue mentioned in this message, the mitigation can be achieved by not respecting the user preferences for code sanitizer configuration. That is, environment variables that are picked by the code sanitizers should either be unset before code sanitizers are initialized, or, code sanitizers should be configured to not take environment variables into consideration (either in building of the code sanitizers themselves, or in building the application, whichever is feasible)

From a technical perspective, it appears that enabling Undefined Behavior Sanitizer (Section 4.4) in production code would be more beneficial than not in most circumstances due to its low overhead and few limitations.

5.2.1 Disk Size Overhead

Disk size overhead may appear to be something that can be easily neglected due to the increase of commercially available disk capacities over the years. However, CPU caches are still limited and *spatial locality* can be impacted negatively due to increased binary size of the programs [17]. Therefore, disk size increase caused by code sanitizers should be monitored.

During experimentation, on Linux, it is observed that while LLVM toolchain links code sanitizers statically, GNU toolchain links code sanitizers dynamically by default.

LLVM does not appear to support dynamically linking code sanitizers aside from Address Sanitizer (requires setting `-shared-libasan`).

For disk size overhead, *BoringSSL*[4] is used as a reference project. BoringSSL is Google's alternative for *OpenSSL*, mainly created due to licensing issues experienced with using OpenSSL. BoringSSL's project file (`CMakeLists.txt`) provides accessory functionality to easily enable all of the major code sanitizers. The version of the

[4] `https://github.com/google/boringssl`

Table 5.1: Disk size overhead of major code sanitizers with shared `libssl.so` as reference.

Code Sanitizer	LLVM	GNU
None	3,081 KB	5,769 KB
ASAN	4,512 KB	6,663 KB
MSAN	8,871 KB	N/A
TSAN	3,381 KB	5,985 KB
UBSAN	7,691 KB	9,233 KB

project used is *git* commit `6855f30`.

The following toolchain options are used (`-shared-libasan` is added to link libasan dynamically with LLVM/Clang). It should be considered that with GNU toolchain all sanitizers were dynamically linked, and with LLVM toolchain only Address Sanitizer was linked dynamically. It should be noted that the `CMakeLists.txt` file is patched so that GNU toolchain could be used with sanitizers. Lastly, since Memory Sanitizer is not available on the GNU toolchain, it is only used with LLVM toolchain.

Address Sanitizer `-fsanitize=address -fsanitize-address-use-after-scope -fno-omit-frame-pointer`

Thread Sanitizer `-fsanitize=thread`

Memory Sanitizer `-fsanitize=memory -fsanitize-memory-track-origins -fno-omit-frame-pointer`

Undefined Behavior Sanitizer `-fsanitize=undefined`

Observing the data provided in Table 5.1, the following statements can be made:

1. Memory Sanitizer has the largest disk size overhead on LLVM toolchain with **187.93%** disk size increase.

2. Thread Sanitizer has the smallest disk size overhead on both LLVM and GNU toolchains with an average of **6.74%** disk size increase.

Table 5.2: Static library sizes of code sanitizers for `x86_64-pc-linux-gnu` architecture. Names are converted accordingly for LLVM.

Static Library	GNU	LLVM
libasan.a	2,706 KB	3,218 KB
liblsan.a	982 KB	N/A
libtsan.a	2,397 KB	2,977 KB
libubsan.a	916 KB	1,002 KB
libmsan.a	N/A	2,642 KB

3. Undefined Behavior sanitizer has the largest disk size overhead on the GNU toolchain, and the second largest on the LLVM toolchain with an average of **104.84%** disk size increase. The additional checks added by Undefined Behavior Sanitizer naturally increases the binary size. In the future, this impact may be reduced by hardware support (*e.g.* a single instruction for null check and then pointer dereference).

4. Code Sanitizers have a bigger size impact on the LLVM toolchain than the GNU toolchain. This may be explained by the relatively smaller file size with no code sanitizers (**87.24%** increase).

In Table 5.2 static libraries of code sanitizers are provided. In production, it can be estimated that code sanitizer libraries are statically linked because their presence cannot be assumed in the deployed platform configuration. This means that if sanitizers were statically linked into BoringSSL, the sizes in Table 5.1 would be increased relatively by the sizes specified in Table 5.2.

Interestingly, GNU toolchain is observed to provide separate `libasan.a` and `liblsan.a`. It should be noted that *Leak Sanitizer* (*abbr.* LSan) is a sanitizer[5] that is integrated into Address Sanitizer 4.1. This means that by default configuration (`-fsanitize=address`), it can be expected that both `libasan.a` and `liblsan.a` need to be linked into the program. Unlike GNU toolchain, LLVM toolchain was not observed to provide a separate library for Leak Sanitizer.

[5] `https://github.com/google/sanitizers/wiki/AddressSanitizerLeakSanitizer`

Table 5.3: Executable sizes of Minetest.

Code Sanitizer	GNU	LLVM
None	12131 KB	9234 KB
ASAN	41343 KB	29663 kB
TSAN	18219 KB	13849 KB
MSAN	-	35053 KB
UBSAN	71444 KB	30058 KB

Lastly, disk size overhead data is also gathered for the Minetest project that is used for experimenting with memory usage overhead (Discussed in Section 5.2.2) and performance overhead (Discussed in Section 5.2.3) in the following sections. The overhead can be seen in Table 5.3.

5.2.2 Memory Usage Overhead

In both this section and the next section (Section 5.2.3), *Minetest*[6] is used as reference during benchmarking. The reason of choosing such a project is due to games having general characteristic of high resource utilization. It is estimated that by going this way, real impact of code sanitizers regarding performance degradation and increased memory usage can be observed. The used version of the project is *git* commit `e10d8080b`. The version of the underlying game engine that Minetest uses (*Irrlicht Engine*[7]) is *git* commit `bdcd27a`.

In this section, memory usage overhead of code sanitizers are discussed. It can be said that code sanitizers have different memory usage characteristics and that it is not possible to generalize code sanitizers regarding memory usage. For example, it is found out that while Undefined Behavior sanitizer has a negligible memory impact, Thread Sanitizer increased memory usage by **7x**.

According to the data provided in Table 5.4, it can be said that sanitizers in LLVM toolchain have minimal impact on memory consumption compared to GNU

[6] `https://github.com/minetest/minetest`

[7] `https://github.com/minetest/irrlicht`

Table 5.4: Memory consumption of Minetest.

Code Sanitizer	GNU	LLVM
None	306 MB	301 MB
ASAN	1.1 GB	992 MB
TSAN	2 GB	405 MB
UBSAN	355 MB	318 MB

toolchain. The biggest difference is observed with the Thread Sanitizer, where LLVM's Thread Sanitizer overhead was less than **1.5x** and GNU's Thread Sanitizer overhead was almost **7x**.

Furthermore, the following statements can be made based on the data available in Table 5.4:

1. GNU's Thread Sanitizer consumed **4x more** memory compared to LLVM's Thread Sanitizer.

2. With both GNU and LLVM, Undefined Behavior sanitizer did not have considerable impact on memory usage. Based on this, it can be said that Undefined Behavior sanitizer may be used in production even when memory is relatively scarce.

3. With both GNU and LLVM, Address Sanitizer increases memory consumption by a factor of about **2**.

5.2.3 Performance (CPU-utilization) Overhead

Games need to render certain amount of frames per second in order to appear fluent to the eyes. Unlike most software, where *FPS* (Frames per Second) is not relevant, this requirement puts a huge burden on games to meet the performance criteria. Usually, FPS rates are used as an ultimate benchmark to realize how well a game is performing.

Nowadays, it can be said that most games are GPU-bound rather than CPU-bound [18]. This means that as long as GPU can keep up with the tasks that it has been assigned to, the performance of the CPU has minimal impact on the game's performance. At the same time, this increases the usability potential of code sanitizers in production because the performance is bound to the GPU; increased CPU utilization would have minimal impact on the overall performance of the game. At the same time, online mode, which has become mainstream in games would benefit from using code sanitizers to prevent remote code execution. Nevertheless, it is important to not cause stuttering due to blocking the rendering thread for a time that is more than the period of rendering one frame which corresponds to `1/FPS` seconds.

Minetest, the benchmarked game, is rather heavy on the CPU than the GPU due to having simplistic graphics. This makes it more CPU bound than GPU bound.

According to the data provided in Table 5.5, it can be said that LLVM and GNU have relatively similar performance overhead except for Thread Sanitizer. Undefined Behavior sanitizer did not have a meaningful impact on the performance, which hints that it may be used in environments where performance matters. Thread Sanitizer had the worst impact on performance, which can be used as a hint that it is not suitable to use in production where high performance is needed with threaded code.

Additionally, the following statements can be made based on the data available in Table 5.5:

1. GNU's Thread Sanitizer decreased the frame rate by **6x** while LLVM's Thread Sanitizer decreased the frame rate by **3x**.

2. Both GNU's and LLVM's Undefined Behavior Sanitizers had impact of less than **2%**.

3. GNU's Address Sanitizer performed **6%** worse compared to LLVM's Address

Table 5.5: Average FPS observed in Minetest.

Code Sanitizer	GNU	LLVM
None	165 FPS	165 FPS
ASAN	130 FPS	140 FPS
TSAN	29 FPS	62 FPS
UBSAN	162 FPS	162 FPS

Sanitizer. Although they both had non-negligible frame rate performance impact, the performance decrease was not observed to be as marginal as with the Thread Sanitizer.

5.3 Porting the Code to Secure Languages

Porting the code to safer languages, such as *Rust*, can be considered an alternative to dealing with security problems in unsafe languages.

The study titled "Translating C to safer Rust" done by *Emre* et al. points out the issue of porting existing code to Rust requiring sheer effort, as can be seen with the following direct quote from their research [19]:

> *One obvious objection to porting existing software into Rust is the sheer effort required to rewrite the code in a new language ...*

Further research is needed to determine how many man-hours are needed to port existing C and C++ code, SLOC-wise.

Emre et al. also declare the availability of tools that automate the porting process. For example, *c2rust* attempts at automating porting C code into Rust. They also claim that multiple port automation attempts have been backed by the industry, which shows a bright future for these kinds of tools.

It should be noted that these attempts use Rust in *unsafe* mode, which ultimately reflects that the porting is merely a syntactic conversion rather than benefiting from Rust where it shines: security. The research is limited if Rust in unsafe mode

provides any security benefits over C and C++ programming languages. Based on this, it can be said that these conversion tools do not bring benefits from the security perspective, giving an advantage to using code sanitizers and keeping the code in C or C++ programming languages.

5.4 Static Code Analysis

Although static and dynamic code analysis are orthogonal to each other, they might cover the same code weakness targets. For example, *cppcheck*, a well known static code analyzer[8] for C++ programming language, is able to detect most of the undefined behaviors as long as they are detectable statically.

Clang Static Analyzer[9], which is part of the LLVM project, is another example to static code analyzers.

Even though both static and dynamic code analysis overlap in their targets, they should ideally be employed together. If it is necessary to pick one, dynamic analysis should be used because weaknesses that are not statically detected would appear in code execution after all. Another problem with static code analysis is *false positives*. Static code analyzers may often report false positives, which would require attention to unnecessary parts of the code base.

In that sense, static code analyzers cannot be used as a substitute for using code sanitizers which normally work on the basis of dynamic code analysis, or porting C and C++ code to safer programming languages.

[8] `https://github.com/danmar/cppcheck`

[9] `https://clang-analyzer.llvm.org`

5.5 Utilizing Language Extensions

In some cases, there are notable language extensions which may go under the branch of porting the code, but still, these extensions do not constitute as new programming languages but rather provide mechanisms of achieving specific needs. These needs may be tailored towards performance or security.

5.5.1 Checked C

Checked C is an extension to C programming language, which was introduced by Microsoft in 2018 [20]. In their white paper, *Elliott* et al. provide a review on its alternatives (Safe C dialects) and the need of Checked C's development.

According to the paper, Checked C brings the following security-focused improvements to C programming language:

Checked pointers. Type `_Ptr<T>` makes it possible for the compiler to confirm that the given pointer is valid or not.

Bounded arrays. Type `_Array_ptr<T>` makes it possible to bound arrays in order to prevent buffer overflows (Discussed in Sections 3.1.1, 3.1.2).

Safe interfaces. Checked C brings additional functions to the standard library that take the specialized `_Ptr<T>` or `_Array_ptr<T>` to make the standard library functions more secure.

Checked regions. A clear separation of potentially unsafe and guaranteed safe areas, akin to blocks that are found in different languages (*e.g.* Rust's *unsafe* areas).

According to the benchmark found in the article, porting standard C to Checked C requires changing on average **17.5%** of lines of code; runtime overhead was on average **8.6%** while compile time overhead was on average **24.3%** [20].

It should be noted that Checked C has remained as a specific asset of Microsoft even though it is seemingly denoted as open-source. What is meant by this statement is that the only support of Checked C comes from Microsoft's own Checked C repository[10]. GCC or LLVM toolchains, for example do not support Checked C by default.

5.5.2 SafeCLib

safeclib[11] is noted to be a library for C that implements secure functions defined in the C11 Annex K standard [21]. Annex K in C11 ISO standard is titled as "Bounds-checking interfaces", and is an addendum to the C 2011 standard, which intends to prevent buffer overruns. C11 ISO standard Annex K starts with the following background information (direct quotation from [21]):

> *Traditionally, the C Library has contained many functions that trust the programmer to provide output character arrays big enough to hold the result being produced. Not only do these functions not check that the arrays are big enough, they frequently lack the information needed to perform such checks. While it is possible to write safe, robust, and error-free code using the existing library, the library tends to promote programming styles that lead to mysterious failures if a result is too big for the provided array.*
>
> *...*
>
> *This annex provides alternative library functions that promote safer, more secure programming. The alternative functions verify that output buffers are large enough for the intended result and return a failure indicator if they are not. Data is never written past the end of an array.*

[10] `https://github.com/Microsoft/checkedc-clang/releases`

[11] `https://github.com/rurban/safeclib`

All string results are null terminated.

This annex also addresses another problem that complicates writing robust code: functions that are not reentrant because they return pointers to static objects owned by the function. Such functions can be troublesome since a previously returned result can change if the function is called again, perhaps by another thread.

Safeclib provides additional security to programs that adhere to the C 2011 ISO standards by implementing the Annex K that is often not implemented by C library implementations.

According to safeclib's official documentation, the key motivation of 2011 C specification is to help mitigate security attacks, especially buffer overrun. Based on that, it can be said that making use of safeclib would increase the security of a program written in C. However, existing code would need to be modified to make use of the new functions provided by the library. This itself does not make it possible to make existing programs secure without modifications, unlike using code sanitizers.

On the other hand, porting C or C++ code to a completely different language would require considerably more effort. Therefore, if the decision to satisfy security is to port the code to safer languages, but this option being not feasible due to limited resources, safeclib or the previously discussed Checked C can be a valuable alternative.

5.6 Comparison of Valgrind and Code Sanitizers

Valgrind is a framework and a collection of tools that allows detecting memory and threading issues at run-time. According to the official website[12], it includes the following tools:

[12] `https://valgrind.org`

Table 5.6: Slow-down factors of Valgrind tools: Memcheck, Addrcheck, Nulgrind [22]

Program	t(s)	Memcheck	Addrcheck	Nulgrind
gcc	1.5	35.5	23.7	9.2
gzip	1.8	22.7	17.7	4.7
mesa	2.3	43.1	35.9	5.6

Memcheck. Memory error detector. May be an alternative to Address Sanitizer (Section 4.1) and Memory Sanitizer (Section 4.2).

Addrcheck. Memcheck without uninitialized value read checking, may be an alternative to Address Sanitizer (Section 4.2). Removed in Valgrind 3.1.0.

Nulgrind. Minimal Valgrind tool used to debug Valgrind itself. No other Valgrind tool has less overhead than Nulgrind.

Valgrind currently runs on the most commonly used platforms, including *X86*, *AMD64*, *ARM*, and *ARM64*.

As can be seen in the enumeration above, Valgrind can be used to eliminate security issues, and it can also be used to improve performance with profiling.

The main reason of development of code sanitizers and the obsolescence of Valgrind is the vast performance degradation Valgrind induces. According to the research paper *Seward* et al. published [22], *Memcheck* has such memory cost that the consumption is slightly more than doubled. And the slow-down factor of the same tool reaches **30** in some benchmarks.

According to the Table 5.6, the slow-down factor of Memcheck reaches **23** when benchmarked with *gcc*, the slow-down factor of Addrcheck reaches **15** when benchmarked with *mesa*, and the slow-down factor of Nulgrind reaches **6.1** when benchmarked with *gcc*. gcc is part of the GNU toolchain, gzip is the GNU zip tool, and mesa is a popular graphics stack.

In the same research paper [22], it is asserted that according to a survey done

regarding Memcheck's performance, 9 praised and 32 complained about its performance.

Comparing these slow-down factors with code sanitizers' factors, one can see a clear distinction. For example, Address Sanitizer reportedly has an average slow-down factor of mere 2.

Although Valgrind can be used in debugging sessions, it cannot be utilized in production due to its resource consumption and slow performance. For this reason, it cannot constitute a feasible alternative to porting code into safer languages.

5.7 Weaknesses of Code Sanitizers

Although code sanitizers have been improved a lot since their introduction, they still have considerable weaknesses. In this section, the following weaknesses are discussed.

5.7.1 False-positives and Bugs in Implementation

There have been numerous reports of code sanitizers reporting false-positives. While this is not a concern security-wise, it impacts the usability of the target program.

If the intention is to stop or crash the program running when a weakness is detected by code sanitizer, which is the default behavior of code sanitizers, then having false-positives may cause frequent crashes which would ultimately affect the usability of the target program.

Since false-positives can be considered bugs in code sanitizers, it would be a wise idea to consistently follow the official sanitizer base repository's issues page[13].

UBFuzz, a research project [23] which tries to find bugs in code sanitizer implementations uses fuzzing in order to spot bugs. It can be expected that in the near

[13] `https://github.com/google/sanitizers/issues`

future code sanitizers will behave more and more expectedly due to their popularity, because more collective effort is going to be spent on improving them.

5.7.2 Sanitizer Elision Due to Optimizations

Certain optimizations made by the compiler (or more generally, the toolchain) may cause eliding code sanitizers. This situation negates the utility of code sanitizers. Therefore, it is crucial to not make toolchains optimize the code in a way that would make code sanitizers less effective.

In the research conducted by *Isemann* et al. [24], *LookUB*, a framework that aims finding optimizer transformations leading to code sanitizer elision is introduced. The authors claim that they have found as many as 17 sanitizer-eliding optimizations in LLVM/Clang.

Their work led to discovering 19 previously-undetected bugs in projects including *Linux Containers*, *libmpeg2*, *NTFS-3G*, and *Wine*. All of the listed projects are widely used and respected in the open-source community, which points the success of their research.

The same research indicates that, for the sake of optimization, in certain cases LLVM toolchain removes memory accesses to invalid regions, removes reads from uninitialized memory, reorders functions that do not have observable side effects, and transforms calls to C standard library functions. All of these cases have potential to render code sanitizers ineffective.

The research also indicates that combining code sanitizers may decrease the chances of optimization-caused sanitizer elision.

Table 5.7: Code sanitizer support in major toolchains, as of February 2024.

Code Sanitizer	LLVM	GNU	MSVC
ASAN	✓	✓	✓
MSAN	✓	-	-
TSAN	✓	✓	-
UBSAN	✓	✓	-

5.8 Current Support for Code Sanitizers in Major Toolchains

It has been 12 years since the first code sanitizer was introduced in 2012. During this time, there have been major improvements made to them. Their popularity, which was mostly caused by them having much smaller overhead compared to other tools, made them an important feature to support in toolchains.

Although in this thesis LLVM and GNU toolchains were mostly focused, other popular toolchains such as MSVC also have support for code sanitizers.

As can be seen in Table 5.7, LLVM/Clang has the best support for code sanitizers. MSVC, a popular toolchain provided by *Microsoft* currently supports only Address Sanitizer.

5.9 Reducing the Sanitizer Overhead

Although code sanitizers have relatively small overhead compared to other tools, they still affect the performance and consume resources.

Code sanitizers usually operate orthogonally, which means that they may be combined for better security. Although not every combination is not supported or feasible, using multiple code sanitizers at the same time can maximize the security. The combination of ASan and UBSan is noted to be a popular choice.

Since code sanitizers generally work at run-time, they might provide the flexibility of disabling them according to needs during program start-up. Or, as mentioned

before, programs may be deployed with specific code sanitizers pre-enabled for the target platform.

Consider this example: *Libreoffice*[14], a popular open-source office suite, is used in various different environments. These may include military bases, hospitals, and homes. It may be wise to enable code sanitizers in military bases and hospitals, but not in personal home computers. This way we can effectively balance the trade-off between security and performance. In places where security is more important than performance, enabling code sanitizers in return of a negligible downgrade of performance should be considered. Libreoffice with the code sanitizers enabled would be expected to be more robust against vulnerabilities. Office documents may contain specific exploits that target the programs commonly available in Office suites [25]. With code sanitizers, resilience against the threats that arise from this situation can be achieved.

Most code sanitizers may be configured to disable certain features. Certain features which are considered less useful may be disabled in order to improve performance and lessen resource consumption.

[14] `https://libreoffice.org`

6 Conclusion

In this thesis, code sanitizers were shown to be effective for reducing the security risks of C and C++ programming languages after experimenting them with popular open-source projects and code snippets that are classified as weak code according to Common Weakness Enumeration (CWE™). Based on the discussion and the data, it can be recommended to use code sanitizers more broadly in software development.

Additionally, capabilities of Undefined Behavior Sanitizer were demonstrated based on the undefined behaviors found and fixed in the Minetest project. It can be said that not utilizing code sanitizers in Minetest led undefined behaviors creeping into the project's code base. It can be assumed that human nature tends to omit details, such that if the compiler does not warn, things may go unnoticed. For example, in Minetest there were occurrences of using variables that were not initialized before.

Although code sanitizers come with their own weaknesses, such as increased usage of disk space, increased usage of memory, and increased usage of CPU utilization, they offer a way to mitigate security weaknesses found in C and C++ programming languages with much less overhead than comparable tools. In other words, code sanitizers allow programmers to care less about security so that their efforts can be spent in more meaningful ways than catching undefined behaviors and other memory issues.

Currently, a trend of mitigating security risks by porting code into more secure

Table 6.1: Feasibility of using code sanitizers in production (✓: unlikely to cause issues, ∗: may cause issues, -: likely to cause issues).

Scarce Resource	ASAN	TSAN	UBSAN
Disk	∗	✓	-
Memory	-	∗	✓
CPU	∗	-	✓

languages, such as *Rust*, can be observed. However, porting code comes with challenges. For example, Rust is a relatively new programming language that not many people are familiar with, and it takes considerable amount of time to port already existing code.

A series of experiments have been done and presented, to determine, if the disk size consumption, memory utilization, and CPU utilization increase significantly and whether the improved security compensates the performance downgrade.

In terms of non-resource heavy programs, the answer is more straightforward. Code sanitizers can be used in production, because the performance downgrade is often affordable. However, the decision regarding what to do should be made on a case by case basis. The reason is that for some projects porting code into secure languages might be more feasible, and for other projects using code sanitizers in production might be more feasible.

One particular code sanitizer, Undefined Behavior Sanitizer, is found to cause negligible impact on memory usage and performance. However, it increased the disk usage considerably. In Table 6.1, feasibility of utilizing code sanitizers in production with respect to resource scarcities can be seen. Further research is needed regarding the impact on performance due to reduced spatial locality utilized by the CPU cache mechanisms [17]. It should be noted that LLVM's sanitizers performed better than GNU in all overhead tests.

If the project code base consists of many million lines of code, porting the code into different programming languages might be quite challenging and time-

consuming. On the other hand, if the project is rather small, it may make more sense to port the code to a safer programming language. That way the users can enjoy the relatively higher performance without trade-offs.

Since code sanitizers operate at run-time, there might be the flexibility of simply enabling or disabling the used code sanitizer. This flexibility can be useful in some situations. Security is not necessary in all contexts, for example, if performance decrease is negligible and if there is no internet connection, programs may run with all security mitigations disabled.

Additionally, hardware assisted sanitizers can be utilized to further reduce the sanitizer overhead. As of 2024, `AArch64` has native memory tagging support, which is used by HWASAN, a variant of Address Sanitizer.

References

[1] Stack Overflow, *Stack Overflow Developer Survey 2023 — survey.stackoverflow.co*, [Accessed 02-01-2024]. [Online]. Available: https://survey.stackoverflow.co/2023/#section-most-popular-technologies-programming-scripting-and-markup-languages.

[2] B. Stroustrup, "An overview of C++", in *Proceedings of the 1986 Special Interest Group on Programming Languages (SIGPLAN) Workshop on Object-Oriented Programming*, New York, New York, USA: ACM, 1986, pp. 7–18. DOI: 10.1145/323779.323736.

[3] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker", in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, Boston, Massachusetts, USA: USENIX Association, Jun. 2012, pp. 309–318, ISBN: 978-931971-93-5.

[4] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "Sok: Sanitizing for security", in *2019 IEEE Symposium on Security and Privacy (SP)*, San Francisco, California, USA, 2019, pp. 1275–1295. DOI: 10.1109/SP.2019.00010.

[5] Y. Zhang, C. Pang, G. Portokalidis, N. Triandopoulos, and J. Xu, "Debloating address sanitizer", in *31st USENIX Security Symposium (USENIX Security 22)*, Boston, Massachusetts, USA: USENIX Association, Aug. 2022, pp. 4345–4363, ISBN: 978-1-939133-31-1.

[6] J. Zhang, S. Wang, M. Rigger, P. He, and Z. Su, "SANRAZOR: Reducing redundant sanitizer checks in C/C++ programs", in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, Virtual, USA: USENIX Association, Jul. 2021, pp. 479–494, ISBN: 978-1-939133-22-9.

[7] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: data race detection in practice", in *Proceedings of the Workshop on Binary Instrumentation and Applications*, New York, New York, USA: ACM, 2009, pp. 62–71. DOI: 10.1145/1791194.1791203.

[8] E. Stepanov and K. Serebryany, "MemorySanitizer: fast detector of uninitialized memory use in C++", in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, San Francisco, California, USA, 2015, pp. 46–55. DOI: 10.1109/CGO.2015.7054186.

[9] J. Wagner, V. Kuznetsov, G. Candea, and J. Kinder, "High system-code security with low overhead", in *2015 IEEE Symposium on Security and Privacy*, San Jose, California, USA, 2015, pp. 866–879. DOI: 10.1109/SP.2015.58.

[10] C. Courbet, "NSan: a floating-point numerical sanitizer", in *Proceedings of the 30th ACM Special Interest Group on Programming Languages (SIGPLAN) International Conference on Compiler Construction*, Virtual, Republic of Korea: ACM, 2021, pp. 83–93. DOI: 10.1145/3446804.3446848.

[11] N. Rotem, L. Howes, and D. Goldblatt, *Warrior1: A performance sanitizer for C++*, 2020. arXiv: 2010.09583 [cs.SE].

[12] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyrklevich, and D. Vyukov, *Memory Tagging and how it improves C/C++ memory safety*, 2018. arXiv: 1802.09517 [cs.CR].

[13] ARM, *Armv8.5-A Memory Tagging Extension White Paper — developer.arm.com*, [Accessed 08-02-2024]. [Online]. Available: https://developer.arm.com/documentation/102925/latest/.

[14] ISO, "ISO International Standard ISO/IEC 14882: 2011, Programming Language C++", *Geneva, Switzerland: International Organization for Standardization (ISO)., Tech. Rep*, 2011.

[15] H. Marco-Gisbert and I. Ripoll Ripoll, "Address space layout randomization next generation", *Applied Sciences*, vol. 9, no. 14, 2019. DOI: 10.3390/app9142928.

[16] E. C. Cooper and R. P. Draves, "C threads", Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, USA, Tech. Rep. CMU-CS-88-154, Jun. 1988. DOI: 10.1184/R1/6603980.v1.

[17] S. Kumar and C. Wilkerson, "Exploiting spatial locality in data caches using spatial footprints", *Special Interest Group on Computer Architecture (SIGARCH)*, vol. 26, no. 3, pp. 357–368, Apr. 1998. DOI: 10.1145/279361.279404.

[18] J. A. Shiraef, "An exploratory study of high performance graphics application programming interfaces", M.S. thesis, University of Tennessee at Chattanooga, USA, 2016.

[19] M. Emre, R. Schroeder, K. Dewey, and B. Hardekopf, "Translating C to safer Rust", *Proc. ACM Program. Lang.*, vol. 5, no. Object-Oriented Programming, Systems, Languages & Applications (OOPSLA), Oct. 2021. DOI: 10.1145/3485498.

[20] A. S. Elliott, A. Ruef, M. Hicks, and D. Tarditi, "Checked C: Making C safe by extension", in *2018 IEEE Cybersecurity Development (SecDev)*, IEEE, Cambridge, Massachusetts, USA, 2018, pp. 53–60.

[21] ISO/IEC, *C11 standard (ISO/IEC 9899:2011) - Annex K*, 2011.

[22] J. Seward and N. Nethercote, "Using Valgrind to detect undefined value errors with bit-precision.", in *USENIX Annual Technical Conference, General Track*, Anaheim, California, USA, 2005, pp. 17–30.

[23] S. Li and Z. Su, *UBfuzz: finding bugs in sanitizer implementations*, 2024. arXiv: 2401.04538 [cs.CR].

[24] R. Isemann, C. Giuffrida, H. Bos, E. van der Kouwe, and K. v. Gleissenthall, "Don't look UB: Exposing sanitizer-eliding compiler optimizations", *Proc. ACM Program. Lang.*, vol. 7, no. Programming Language Design and Implementation (PLDI), Jun. 2023. DOI: 10.1145/3591257.

[25] C. Smutz and A. Stavrou, "Preventing exploits in Microsoft Office documents through content randomization", in *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404*, Kyoto, Japan: Springer-Verlag, 2015, pp. 225–246. DOI: 10.1007/978-3-319-26362-5_11.

Appendix A Code Sanitizer Outputs

```
==254==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 10 byte(s) in 1 object(s) allocated from:
    #0 0x5640af36514e in __interceptor_malloc (/home/fuzun/program.out+0
    ↪ xa114e) (BuildId: e840af3ddf1d3a48376b2e7bd58e9bb0f49c7c83)
    #1 0x5640af3a0037 in trimString (/home/fuzun/program.out+0xdc037) (
    ↪ BuildId: e840af3ddf1d3a48376b2e7bd58e9bb0f49c7c83)
    #2 0x5640af3a00e4 in parseAndPrintString (/home/fuzun/program.out+0
    ↪ xdc0e4) (BuildId: e840af3ddf1d3a48376b2e7bd58e9bb0f49c7c83)
    #3 0x5640af3a010f in main (/home/fuzun/program.out+0xdc10f) (BuildId:
    ↪ e840af3ddf1d3a48376b2e7bd58e9bb0f49c7c83)
    #4 0x7f46c0730d8f in __libc_start_call_main csu/../sysdeps/nptl/
    ↪ libc_start_call_main.h:58:16

SUMMARY: AddressSanitizer: 10 byte(s) leaked in 1 allocation(s).
```

Listing A.1: Address Sanitizer output.

```
==276==WARNING: MemorySanitizer: use-of-uninitialized-value
    #0 0x55ebca4a96e5 in main (/home/fuzun/program.out+0xa76e5) (BuildId: 7
    ↪ eb17d1f9d0c89ff79ebfe8f8940a1bfe483bdb0)
    #1 0x7f56f8f6fd8f in __libc_start_call_main csu/../sysdeps/nptl/
    ↪ libc_start_call_main.h:58:16
    #2 0x7f56f8f6fe3f in __libc_start_main csu/../csu/libc-start.c:392:3
    #3 0x55ebca421364 in _start (/home/fuzun/program.out+0x1f364) (BuildId: 7
    ↪ eb17d1f9d0c89ff79ebfe8f8940a1bfe483bdb0)

SUMMARY: MemorySanitizer: use-of-uninitialized-value (/home/fuzun/
    ↪ program.out+0xa76e5) (BuildId: 7
    ↪ eb17d1f9d0c89ff79ebfe8f8940a1bfe483bdb0) in main
```

Listing A.2: Memory Sanitizer output.

```
ubsan.c:5:7: runtime error: load of address 0x7fff3ef4ff14 with insufficient space for
    ↪  an object of type 'int'
0x7fff3ef4ff14: note: pointer points here
  8e 00 00 00 00 00 00 00  00 5d 7c ed 2e 73 97 62  01 00 00 00 00 00 00 00  90 9d
    ↪ c0 df 4a 7f 00 00
            ^

```

Listing A.3: Undefined Behavior Sanitizer output.

```
==================
WARNING: ThreadSanitizer: data race (pid=668)
  Read of size 4 at 0x7ffd746e545c by main thread:
    #0 main <null> (tsan+0x1374)

  Previous write of size 4 at 0x7ffd746e545c by thread T1:
    #0 incrementInteger <null> (tsan+0x12f5)

  Location is stack of main thread.

  Location is global '<null>' at 0x000000000000 ([stack]+0x00000001e45c)

  Thread T1 (tid=670, finished) created by main thread at:
    #0 pthread_create ../../../../src/libsanitizer/tsan/tsan_interceptors_posix.cpp
    ↪ :969 (libtsan.so.0+0x605b8)
    #1 main <null> (tsan+0x1368)

SUMMARY: ThreadSanitizer: data race (/home/fuzun/tsan+0x1374) in main
==================
2
ThreadSanitizer: reported 1 warnings
```

Listing A.4: Thread Sanitizer output.