# AddressSanitizer

## A Fast Address Sanity Checker

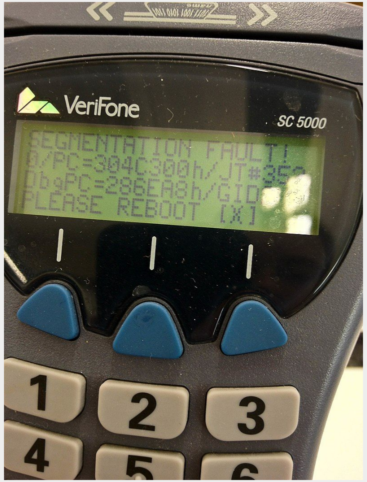
Braden Crimmins, Matthew Ruiz, Alan Yang

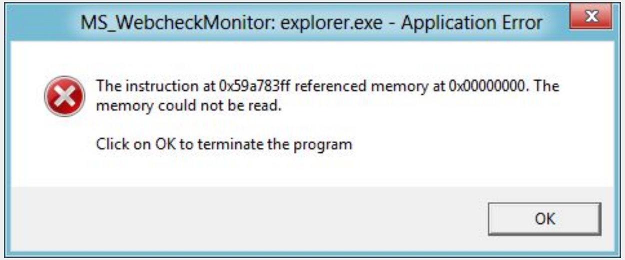
# Motivation

[=] Code bases which are non-trivial in nature are rarely proven to be correct.

[=] Buffer overflows, use-after-free bugs, and other similar errors create unexpected behavior and introduce exploitable security issues.

[=] Detecting and preventing these errors is valuable, especially with little additional programmer effort.



# Specification

**Goal:** Develop a tool that can detect undefined/incorrect behavior.

**[=]** Should not modify observable program behavior

**[=]** Should be (reasonably) efficient

    [-] Costs incurred: time, memory

**[=]** Should be statistically meaningful

    [-] False negatives are OK

    [-] False positives are probably not

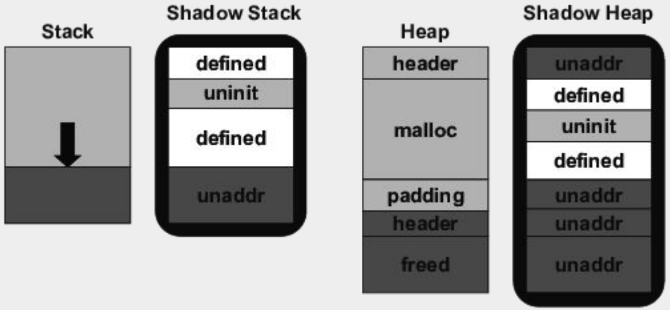
# Techniques Before ASan

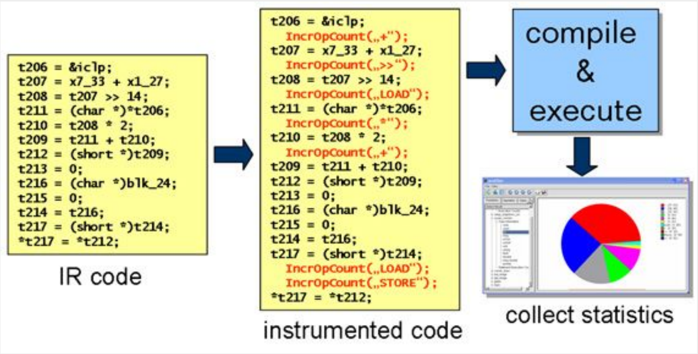
**[=]** Shadow memory: reserving large chunks of memory for metadata

**[=]** Instrumentation: insertion of diagnostic code to track behavior

**[=]** Debug allocators: specialized implementations of `free` and `malloc`

**[=]** Valgrind uses dynamic instrumentation and shadow memory



# Central Tensions

### Coverage vs. Performance

**[=]** Instrumentation - how many checks to add?

- [-] Too many - too slow!
- [-] Runtime vs. compile-time

**[=]** Detecting use-after-free

- [-] Custom heap allocator
- [-] Poison pages
- [-] Magic value redzones

### Memory Trade-Offs

**[=]** Multi-level lookup tables give more flexibility

- [-] Adding indirection is slow

**[=]** How much metadata to track?

- [-] More coverage means more memory cost

# AddressSanitizer

First released by Google in 2012

Inserts instrumentation code which detects bugs at runtime.

Consists of two main components:
**[=]** Instrumentation module
**[=]** Runtime library

**This introduced Google’s “code sanitizer” class of programs.**

Other examples include LeakSanitizer, ThreadSanitizer and MemorySanitizer.

# Shadow Memory

**[=]** Stores information about application data by mapping to a “shadow” address.

**[=]** Tracks information about the base memory location

[-] Has it been allocated?

[-] Has it been initialized?

# Shadow Memory in AddressSanitizer

**Key Intuition:** Most memory is aligned to 8 bytes or more. This allows for compact encoding of memory states.

Each given 8-byte chunk of memory is assigned *one* corresponding byte in shadow memory. This allows a compact representation of the full memory space.

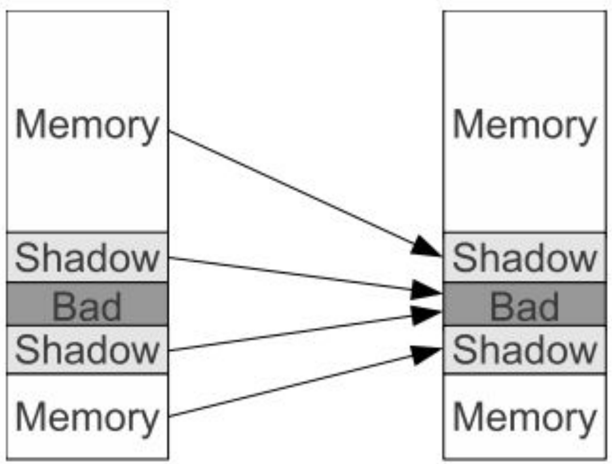



Figure 1: AddressSanitizer memory mapping.

**AddressSanitizer uses scale-offset mapping for the stack and heap.**

# Shadow Memory and Redzoning - Example

```
void foo() {
  char a[10];
  <function body> }
```

AddressSanitizer uses this tool to place canaries or ‘redzones’ between each piece of application data. Any read or write into a redzone is detected and reported as an error.

# Shadow Memory and Redzoning - Example

```
void foo() {
  char a[10];
  <function body> }
```

```
void foo() {
```

**AddressSanitizer uses this tool to place canaries or ‘redzones’ between each piece of application data. Any read or write into a redzone is detected and reported as an error.**

# Shadow Memory and Redzoning - Example

```
void foo() {
  char a[10];
  <function body> }
```

```
void foo() {
  char rz1[32]
  char arr[10];
  char rz2[32-10+32];
```

**AddressSanitizer uses this tool to place canaries or 'redzones' between each piece of application data. Any read or write into a redzone is detected and reported as an error.**

# Shadow Memory and Redzoning - Example

```
void foo() {
  char a[10];
  <function body> }
```

```
void foo() {
  char rz1[32]
  char arr[10];
  char rz2[32-10+32];
  unsigned *shadow =
    (unsigned*)(((long)rz1>>8)+Offset);
```

**AddressSanitizer uses this tool to place canaries or ‘redzones’ between each piece of application data. Any read or write into a redzone is detected and reported as an error.**

# Shadow Memory and Redzoning - Example

```
void foo() {
  char a[10];
  <function body> }
```

```
void foo() {
  char rz1[32]
  char arr[10];
  char rz2[32-10+32];
  unsigned *shadow =
    (unsigned*)(((long)rz1>>8)+Offset);
  // poison the redzones around arr.
  shadow[0] = 0xffffffff;  // rz1
  shadow[1] = 0xffff0200;  // arr and rz2
  shadow[2] = 0xffffffff;  // rz2
```

**AddressSanitizer uses this tool to place canaries or 'redzones' between each piece of application data. Any read or write into a redzone is detected and reported as an error.**

# Shadow Memory and Redzoning - Example

```
void foo() {
  char a[10];
  <function body> }
```

```
void foo() {
  char rz1[32]
  char arr[10];
  char rz2[32-10+32];
  unsigned *shadow =
    (unsigned*)(((long)rz1>>8)+Offset);
  // poison the redzones around arr.
  shadow[0] = 0xffffffff;  // rz1
  shadow[1] = 0xffff0200;  // arr and rz2
  shadow[2] = 0xffffffff;  // rz2
  <function body>
```

**AddressSanitizer uses this tool to place canaries or ‘redzones’ between each piece of application data. Any read or write into a redzone is detected and reported as an error.**

# Shadow Memory and Redzoning - Example

```
void foo() {
  char a[10];
  <function body> }
```

```
void foo() {
  char rz1[32]
  char arr[10];
  char rz2[32-10+32];
  unsigned *shadow =
    (unsigned*)(((long)rz1>>8)+Offset);
  // poison the redzones around arr.
  shadow[0] = 0xffffffff;  // rz1
  shadow[1] = 0xffff0200;  // arr and rz2
  shadow[2] = 0xffffffff;  // rz2
  <function body>
  // un-poison all.
  shadow[0] = shadow[1] = shadow[2] = 0;  }
```

**AddressSanitizer uses this tool to place canaries or ‘redzones’ between each piece of application data. Any read or write into a redzone is detected and reported as an error.**

# Runtime Library

**[=]** Enables runtime updates to the shadow memory by overwriting malloc and free.

**[=]** malloc reserves memory with appropriate redzone and padding. It also ‘poisons’ the redzones.

**[=]** free poisons the memory and quarantines it, so it will not be used again soon.

# Results and Metrics

**[=]** 3.4x average memory usage increase
[-] Valgrind induces a 2.125x increase

**[=]** 3.7x average slowdown
[-] Valgrind has a 20x average slowdown

**[=]** Some false negatives can occur
[-] Unaligned partially out-of-bounds accesses
[-] Larger redzones can catch more bugs

[=] Detects use-after-free and out-of-bounds accesses to heap, stack, and global objects

[=] Discovered over 300 bugs in the Chromium browser at time of publication

[=] Detected several heap based use-after-free bugs in LLVM itself!

# Work Since Initial Publication

Google has produced a sanitizer tool suite

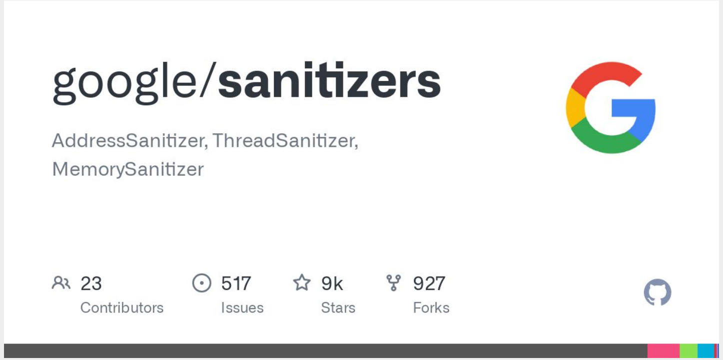
[=] AddressSanitizer

[=] MemorySanitizer

[=] ThreadSanitizer

[=] LeakSanitizer

***Debloating Address Sanitizer (2022)***
*Zhang et al.*
[=] Introduces ASan--
[=] Nearly 30% speedup



# Analysis

## Strengths

**[=]** ASan's use of compile-time instrumentation makes it powerful and (relatively) efficient

**[=]** Covers wide range of bugs

**[=]** Easy-to-use and widely available (LLVM/clang, gcc)

## Limitations

**[=]** Insecure against adversarial inputs

- [-] Canary avoidance

**[=]** Can't find uninitialized reads

**[=]** High performance cost

- [-] Large memory footprint
- [-] Runtime can be improved

# Readings

**[=]** Serebryany et al. AddressSanitizer: A Fast Address Sanity Checker. 2012

**[=]** Zhang et al. Debloating Address Sanitizer. 2022

**[=]** Seward, Nethercote. Using Valgrind to detect undefined value errors with bit-precision. 2005

# AddressSanitizer

## A Fast Address Sanity Checker

Braden Crimmins, Matthew Ruiz, Alan Yang