Address & Thread Sanitizer in GCC 4.8

Dodji Seketeli <dodji@redhat.com>

LRDE Epita - November 2013 - Paris, France
Outline

Address Sanitizer Status

Thread Sanitizer Status
Now let’s talk about …

Address Sanitizer Status

Thread Sanitizer Status
Address Sanitizer is a memory error detector.
A brief recap

- Address Sanitizer is a memory error detector.
  - Instruments each memory access.
A brief recap

- Address Sanitizer is a memory error detector.
  - Instruments each memory access.
  - Checks if it’s OK to access memory at that address.
A brief recap

- Address Sanitizer is a memory error detector.
  - Instruments each memory access.
  - Checks if it’s OK to access memory at that address.
  - Emits an error when accessing a non-valid address.
A brief recap

- Address Sanitizer is a memory error detector.
  - Instruments each memory access.
  - Checks if it’s OK to access memory at that address.
  - Emits an error when accessing a non-valid address.
  - libsanitizer runtime library replaces malloc/free functions.
Address Sanitizer is a memory error detector.

- Instruments each memory access.
- Checks if it’s OK to access memory at that address.
- Emits an error when accessing a non-valid address.
- libsanitizer runtime library replaces malloc/free functions.
- Addresses of freed memory are marked as being non-valid aka “poisoned”.
Suppose we have this code:

```c
*address = ...; // or: ... = *address;
```
Suppose we have this code:

*address = ...; // or: ... = *address;

Asan instruments it as:

if (is_poisoned (address))
{
    report_error (address, access_size, /*is_write=*/true);
}
*address = ...; // or: ... = *address;
Application address space is divided in two parts:
Application address space is divided in two parts:
- One part really for the application. Let's call it part A.
Application address space is divided in two parts:

- One part really for the application. Let's call it part A.
- One part to store metadata about validity of bytes in part A. That's the Shadow memory.

Each 8 bytes of application memory has metadata encoded in 1 byte in shadow memory. Each byte of shadow memory can take 9 different values:

- 0: All bytes in the corresponding 8-bytes region are accessible.
- -1: All bytes in the corresponding 8-bytes region are non-accessible (aka poisoned).
- k: The first k bytes of the 8-bytes region are poisoned.
Application address space is divided in two parts:
- One part really for the application. Let's call it part A.
- One part to store metadata about validity of bytes in part A. That's the Shadow memory.

Each 8 bytes of application memory has metadata encoded in 1 byte in shadow memory.
Application address space is divided in two parts:
- One part really for the application. Let's call it part A.
- One part to store metadata about validity of bytes in part A. That's the Shadow memory.

Each 8 bytes of application memory has metadata encoded in 1 byte in shadow memory.

Each byte of shadow memory can take 9 different values (surprise, heh):
Application address space is divided in two parts:
- One part really for the application. Let's call it part A.
- One part to store metadata about validity of bytes in part A. That's the Shadow memory.

Each 8 bytes of application memory has metadata encoded in 1 byte in shadow memory.

Each byte of shadow memory can take 9 different values (surprise, heh):
- 0: All bytes in the corresponding 8-bytes region are accessible.
Application address space is divided in two parts:
- One part really for the application. Let's call it part A.
- One part to store metadata about validity of bytes in part A. That's the Shadow memory.

Each 8 bytes of application memory has metadata encoded in 1 byte in shadow memory.

Each byte of shadow memory can take 9 different values (surprise, heh):
- 0: All bytes in the corresponding 8-bytes region are accessible.
- -1: All bytes in the corresponding 8-bytes region are non-accessible (aka poisoned).
Instrumentation principles: memory address validity

- Application address space is divided in two parts:
  - One part really for the application. Let's call it part A.
  - One part to store metadata about validity of bytes in part A. That's the Shadow memory.

- Each 8 bytes of application memory has metadata encoded in 1 byte in shadow memory.

- Each byte of shadow memory can take 9 different values (surprise, heh):
  - 0: All bytes in the corresponding 8-bytes region are accessible.
  - -1: All bytes in the corresponding 8-bytes region are non-accessible (aka poisoned).
  - k: The first k bytes of the 8-bytes region are poisoned.
So the “is_poisoned” function now becomes:

```c
bool
is_poisoned (char *address, size_t access_size, bool is_write_access)
{
/* Get the address of the shadow memory. */
char *shadow_address = mem_to_shadow (address);

/* And now check if the shadow value says we are accessing 
a poisoned memory slot ... */
char shadow_value = *shadow_address;
if (shadow_value)
{
    if (is_access_to_poisoned_memory (shadow_value, address, access_size))
        report_error (address, access_size, is_write_access);
}
}
```

And the “is_access_to_poisoned_memory” function is:

```c
bool
is_access_to_poisoned_memory (char shadow_value char *address, char access_size)
{
    last_accessed_byte = (address & 7) + access_size - 1;
    return last_accessed_byte >= shadow_value;
}
```
So the “is_poisoned” function now becomes:

```c
bool is_poisoned (char *address, size_t access_size, bool is_write_access)
{
    /* Get the address of the shadow memory. */
    char *shadow_address = mem_to_shadow (address);

    /* And now check if the shadow value says we are accessing
     a poisoned memory slot ... */
    char shadow_value = *shadow_address;
    if (shadow_value)
    {
        if (is_access_to_poisoned_memory (shadow_value, address, access_size))
            report_error (address, access_size, is_write_access);
    }
}
```

And the “is_access_to_poisoned” function is:

```c
bool is_access_to_poisoned_memory (char shadow_value char *address, char access_size)
{
    last_accessed_byte = (address & 7) + access_size - 1;
    return last_accessed_byte >= shadow_value;
}
```
Instrumenting global & stack variables

▶ To catch use out-of-bounds on global and stack variables.
Instrumenting global & stack variables

- To catch use out-of-bounds on global and stack variables.
- Global variables

  - Insert a red zone (poisoned memory region) between two global variables.
  - A constructor function tells the asan runtime about each global variable and about the red zones.

Stack Variables

  - Insert a red zone
  - At the top of the stack
  - Between each variable slot
  - At the bottom of the stack that contains metadata for the runtime: function name, number of variables, offset of the variables slot and their length.
Instrumenting global & stack variables

- To catch use out-of-bounds on global and stack variables.
- Global variables
  - Insert a red zone (poisoned memory region) between two global variables.
- Stack Variables
  - Insert a red zone at the top of the stack between each variable slot.
  - At the bottom of the stack that contains metadata for the runtime: function name, number of variables, offset of the variables slot and their length.
Instrumenting global & stack variables

▶ To catch use out-of-bounds on global and stack variables.
▶ Global variables
  ▶ Insert a red zone (poisoned memory region) between two global variables.
  ▶ A constructor function tells the asan runtime about each global variable and about the red zones.
To catch use out-of-bounds on global and stack variables.

Global variables
- Insert a red zone (poisoned memory region) between two global variables.
- A constructor function tells the asan runtime about each global variable and about the red zones.

Stack Variables
To catch use out-of-bounds on global and stack variables.

Global variables
- Insert a red zone (poisoned memory region) between two global variables.
- A constructor function tells the asan runtime about each global variable and about the red zones.

Stack Variables
- Insert a red zone
To catch use out-of-bounds on global and stack variables.

Global variables

- Insert a red zone (poisoned memory region) between two global variables.
- A constructor function tells the asan runtime about each global variable and about the red zones.

Stack Variables

- Insert a red zone
  - At the top of the stack
Instrumenting global & stack variables

- To catch use out-of-bounds on global and stack variables.

Global variables
- Insert a red zone (poisoned memory region) between two global variables.
- A constructor function tells the asan runtime about each global variable and about the red zones.

Stack Variables
- Insert a red zone
  - At the top of the stack
  - Between each variable slot
Instrumenting global & stack variables

- To catch use out-of-bounds on global and stack variables.
- **Global variables**
  - Insert a red zone (poisoned memory region) between two global variables.
  - A constructor function tells the asan runtime about each global variable and about the red zones.
- **Stack Variables**
  - Insert a red zone
    - At the top of the stack
    - Between each variable slot
    - At the bottom of the stack that contains metadata for the runtime: function name, number of variables, offset of the variables slot and their length.
Instrumentation Patterns at GIMPLE level

- load/store through pointers

- For example:
  ```c
  int n = strlen(str);
  /* For this, we instrument access to str[0] and str[n]. */
  
  variable = *the_pointer;
  /* some stuff that don't touch the_pointer .... */
  variable = *the_pointer; /* We shouldn't instrument this access to the_pointer, right? */
  ```

- Avoid instrumenting "adjacent" memory accesses to the same addresses in the same basic block.

Dodji Seketeli <dodji@redhat.com> Address & Thread Sanitizer in GCC 4.8
Instrumentation Patterns at GIMPLE level

- load/store through pointers
- builtin memory access function calls

For example:

```c
int n = strlen(str);
/* For this, we instrument access to str[0] and str[n]. */

variable = *the_pointer;
/* some stuff that don't touch the_pointer .... */
variable = *the_pointer; /* We shouldn't instrument this access to the_pointer, right? */
```
Instrumentation Patterns at GIMPLE level

- load/store through pointers
- builtin memory access function calls
  - Basically instrument access to the memory region of the arguments

For example:

```c
int n = strlen(str);
/* For this, we instrument access to str[0] and str[n]. */
```

Avoid instrumenting "adjacent" memory accesses to the same addresses in the same basic block.

```c
variable = *the_pointer;
/* some stuff that don't touch the_pointer .... */
variable = *the_pointer; /* We shouldn't instrument this access to the_pointer, right? */
```
Instrumentation Patterns at GIMPLE level

- load/store through pointers
- builtin memory access function calls
  - Basically instrument access to the memory region of the arguments
  - For example:

    ```
    int n = strlen (str);
    
    /*
     * For this, we instrument access to str[0] and str[n].
     */
    ```

- Avoid instrumenting "adjacent" memory accesses to the same addresses in the same basic block.
load/store through pointers

- builtin memory access function calls
  - Basically instrument access to the memory region of the arguments
  - For example:
    ```c
    int n = strlen (str);
    /*
     * For this, we instrument access to str[0] and str[n].
     */
    ```

- Avoid instrumenting “adjacent” memory accesses to the same addresses in the same basic block.

  ```c
  variable = *the_pointer;
  /* some stuff that don’t touch the_pointer .... */
  variable = *the_pointer; /* We shouldn’t instrument this access to the_pointer, right? */
  ```
- Improve performance

Dodji Seketeli <dodji@redhat.com> Address & Thread Sanitizer in GCC 4.8

10/16
- Improve performance
  - Introduce a builtin like __builtin_asan_mem_test

- Teach relevant parts of the compiler to not optimize that builtin out
- Teach the vectorizer about that builtin to make it work in vectorisation contexts
- Introduce a new pass that would generalize the redundant instrumentation removal.
- Keep up with the new features in asan@llvm

- Detect use of address of variables that escape a scope:
  ```c
  some_class *ptr = 0;
  {
      some_class belongs_to_a_scope;
      ptr = &belongs_to_a_scope;
  }
  do_something_with (ptr); // <-- we catch this in asan@gcc
  ```
  More generally, keep track of what happens in asan

- Pro-actively propose killing features
Improving performance:
- Introduce a built-in like `__builtin_asan_mem_test`
- Teach relevant parts of the compiler to not optimize that built-in out
- Teach the vectorizer about that built-in to make it work in vectorisation contexts
TODO

- Improve performance
  - Introduce a builtin like __builtin_asan_mem_test
  - Teach relevant parts of the compiler to not optimize that builtin out
  - Teach the vectorizer about that builtin to make it work in vectorisation contexts
  - Introduce a new pass that would generalize the redundant instrumentation removal.

- Keep up with the new features in asan@llvm

- Detect use of address of variables that escape a scope:
  ```
  // DO NOT OPTIMIZE
  some_class *ptr = 0;
  {
    some_class belongs_to_a_scope;
    ptr = &belongs_to_a_scope;
  }
  do_something_with (ptr); // <-- we catch this in asan@gcc
  ```

- More generally, keep track of what happens in asan

- Pro-actively propose killing features

Dodji Seketeli <dodji@redhat.com> Address & Thread Sanitizer in GCC 4.8
➤ Improve performance
  ➤ Introduce a builtin like __builtin_asan_mem_test
  ➤ Teach relevant parts of the compiler to not optimize that builtin out
  ➤ Teach the vectorizer about that builtin to make it work in vectorisation contexts
  ➤ Introduce a new pass that would generalize the redundant instrumentation removal.
➤ Keep up with the new features in asan@llvm
- Improve performance
  - Introduce a builtin like __builtin_asan_mem_test
  - Teach relevant parts of the compiler to not optimize that builtin out
  - Teach the vectorizer about that builtin to make it work in vectorisation contexts
  - Introduce a new pass that would generalize the redundant instrumentation removal.
- Keep up with the new features in asan@llvm
  - Detect use of address of variables that escape a scope:
    ```c
    some_class *ptr = 0;
    {
      some_class belongs_to_a_scope;
      ptr = &belongs_to_a_scope;
    }
    do_something_with (ptr); // <-- we catch this in asan@gcc
    ```
- Improve performance
  - Introduce a builtin like `__builtin_asan_mem_test`
  - Teach relevant parts of the compiler to not optimize that builtin out
  - Teach the vectorizer about that builtin to make it work in vectorisation contexts
  - Introduce a new pass that would generalize the redundant instrumentation removal.
- Keep up with the new features in asan@llvm
  - Detect use of address of variables that escape a scope:
    ```
    some_class *ptr = 0;
    {
      some_class belongs_to_a_scope;
      ptr = &belongs_to_a_scope;
    }
    do_something_with (ptr); // <-- we catch this in asan@gcc
    ```
  - More generally, keep track of what happens in asan

Dodji Seketeli <dodji@redhat.com> Address & Thread Sanitizer in GCC 4.8
- Improve performance
  - Introduce a builtin like `__builtin_asan_mem_test`
  - Teach relevant parts of the compiler to not optimize that builtin out
  - Teach the vectorizer about that builtin to make it work in vectorisation contexts
  - Introduce a new pass that would generalize the redundant instrumentation removal.
- Keep up with the new features in asan@llvm
  - Detect use of address of variables that escape a scope:
    ```c
    some_class *ptr = 0;
    {
      some_class belongs_to_a_scope;
      ptr = &belongs_to_a_scope;
    }
    do_something_with (ptr); // <-- we catch this in asan@gcc
    ```
  - More generally, keep track of what happens in asan
  - Pro-actively propose killing features

Dodji Seketeli <dodji@redhat.com> Address & Thread Sanitizer in GCC 4.8
Now let’s talk about ...

Address Sanitizer Status

Thread Sanitizer Status
thread-sanitizer is a data race detector for C/C++ programs.
thread-sanitizer is a data race detector for C/C++ programs.

But what’s a data race? (please don’t fall asleep)
thread-sanitizer is a data race detector for C/C++ programs.

But what’s a data race? (please don’t fall asleep)

- An “Event”: a memory access from a given thread.
thread-sanitizer is a data race detector for C/C++ programs.

But what’s a data race? (please don’t fall asleep)

- An “Event”: a memory access from a given thread.
- A “Happens-before” partial order relation, defined on a set of events.
A brief recap

- thread-sanitizer is a data race detector for C/C++ programs.
- But what’s a data race? (please don’t fall asleep)
  - An “Event”: a memory access from a given thread.
  - A “Happens-before” partial order relation, defined on a set of events.
    - A Happens-Before B means “A is always observed before B”.

Dodji Seketeli <dodji@redhat.com> Address & Thread Sanitizer in GCC 4.8
thread-sanitizer is a data race detector for C/C++ programs.

But what’s a data race? (please don’t fall asleep)

- An “Event”: a memory access from a given thread.
- A “Happens-before” partial order relation, defined on a set of events.
  - A Happens-Before B means “A is always observed before B”.
- So we have a data race on a memory location L if:
A brief recap

• thread-sanitizer is a data race detector for C/C++ programs.
• But what’s a data race? (please don’t fall asleep)
  • An “Event”: a memory access from a given thread.
  • A “Happens-before” partial order relation, defined on a set of events.
    • A Happens-Before B means “A is always observed before B”.
  • So we have a data race on a memory location L if:
    • If there are two events A and B on L that are not ordered
thread-sanitizer is a data race detector for C/C++ programs.

But what’s a data race? (please don’t fall asleep)

- An “Event”: a memory access from a given thread.
- A “Happens-before” partial order relation, defined on a set of events.
  - A Happens-Before B means “A is always observed before B”.
- So we have a data race on a memory location L if:
  - If there are two events A and B on L that are not ordered
  - and there is no common lock held on their memory location
thread-sanitizer is a data race detector for C/C++ programs.

But what’s a data race? (please don’t fall asleep)

- An “Event”: a memory access from a given thread.
- A “Happens-before” partial order relation, defined on a set of events.
  - A Happens-Before B means “A is always observed before B”.
- So we have a data race on a memory location L if:
  - If there are two events A and B on L that are not ordered
  - and there is no common lock held on their memory location
  - and either A or B is a write event.
Instrumentation principles

- Instruments each memory access by prepending it with a libsanitizer (runtime) function like \_tsan\_read4(addr);
Instrumentation principles

- Instruments each memory access by prepending it with a libsanitizer (runtime) function like  `__tsan_read4(addr)`;
- Then the runtime does the magic of figuring out if two accesses to the same address from different threads represents a data race.
Instrumentation patterns

- Instrument \_sync\* and \_atomic\* built-in functions.
Instrumentation patterns

- Instrument \_\_sync* and \_\_atomic* built-in functions.
- Instrument classic memory accesses.
- Don’t instrument local variables which address don’t escape.
Instrumentation patterns

- Instrument __sync* and __atomic* built-in functions.
- Instrument classic memory accesses.
- Don’t instrument local variables which address don’t escape.
- Don’t instrument reads on global constants including vtables.
Don't instrument redundant accesses
- Don’t instrument redundant accesses
  - A read that comes right before a write to the same location.
- Don’t instrument redundant accesses
  - A read that comes right before a write to the same location.
- Introduce a builtin for the accesses, like for asan and do the same things.
Don’t instrument redundant accesses
  ▶ A read that comes right before a write to the same location.
▶ Introduce a builtin for the accesses, like for asan and do the same things.
▶ Monitor tsan@llvm
So ...

▶ Questions?
So ...

▶ Questions?

▶ Thank You!