Enable MSan & TSan on Go for Arm platform

Fangming.Fang@arm.com
Agenda

● MSan
  ▪ Msan overview
  ▪ Shadow memory
  ▪ Shadow propagation
  ▪ Instrumentation in Go
  ▪ Demo

● TSan
  ▪ TSan overview
  ▪ Algorithm
  ▪ Instrumentation in Go
  ▪ Demo
MemorySanitizer (Msan) overview

Use of uninitialized memory (UUM) is such an issue that is hard to reproduce and debug. So MemorySanitizer, MSan for short, is introduced to address that issue.

- MSan is a dynamic tool that detects uses of uninitialized memory in C. The tool is based on compile time instrumentation and relies on bitprecise shadow memory at run-time. Shadow propagation technique is used to avoid false positive reports on copying of uninitialized memory.

- MSan is part of LLVM trunk and implemented as an LLVM optimization pass.
### Shadow memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>0x7fffffff</td>
</tr>
<tr>
<td>Shadow</td>
<td>0x60000000</td>
</tr>
<tr>
<td>Protected</td>
<td>0x40000000</td>
</tr>
<tr>
<td></td>
<td>0x20000000</td>
</tr>
<tr>
<td></td>
<td>0x00000000</td>
</tr>
</tbody>
</table>

1 application bit = 1 shadow bit
- 1 = poisoned (uninitialized)
- 0 = clean (initialized)

Shadow = Addr & ShadowMask
- The ShadowMask constant is platform-specific
Shadow propagation

Basic shadow propagation rules

- $A = \text{load } P$: check $P'$, $A' = \text{load } (P \& \text{ShadowMask})$
- $\text{store } P, A$: check $P'$, $\text{store } (P \& \text{ShadowMask}), A'$
- $A = \text{const}$: $A' = 0$
- $A = \text{undef}$: $A' = 0.x.f.f$
- $A = B \& C$: $A' = (B' \& C') | (B' \& C') | (B' \& C')$
- $A = B | C$: $A' = (B' \& C') | (\sim B \& C') | (B' \& \sim C')$
- $A = B \text{ xor } C$: $A' = B' \mid C'$
- $A = B \ll C$: $A' = (\text{sign-extend}(C' \neq 0)) | (B' \ll C')$

Approximate propagation

- $A = B + C$ $\rightarrow$ $A' = B' \mid C'$

Exact propagation logic is way too complex. Bitwise OR is common propagation logic.

- Never makes a value "less poisoned".
- Never makes a poisoned value clean.
CGo

MSan does not apply to pure Go application where all memory is initialized. But Go can call C code by the way called CGo. People can use CGo to pass memory back and forth between C and Go.

False positive will happen if C creates some uninitialized memory, passes that memory to Go, Go fills it in and returns to C.
MSan framework in Go

Go application

Instrumented app

memory access API

Go API

domsanread msanwrite msanmalloc msanfree

CGo API

__msan_read_go __msan_write_go __msan_malloc_go __msan_free_go

MSan runtime lib

__msan_check_mem_is_initialized __msan_unpoison __msan_poison
Sigaction

sigaction

callCgoSigaction

_cgo_sigaction

MOVD _cgo_sigaction(SB), R3
BL R3

Intercepted by msan lib

sigaction

Mmap

mmap

callCgoMmap

_cgo_mmap

MOVD _cgo_mmap(SB), R9
BL R9

Intercepted by msan lib

mmap
Demo

// C code
void f(int32_t *p, int n) {
    int32_t *volatile q = (int32_t *)malloc(sizeof(int32_t) * n);
    memcpy(p, q, n * sizeof(*p));
    free(q);
}
void g(int32_t *p, int n) {
    if (p[4] != 1) {   // take code branch depending on the fifth element uninitialized
        abort();
    }
}

// Go code
func main() {
    a := make([]int32, 10)
    C.f((*C.int32_t)(unsafe.Pointer(&a[0])), C.int(len(a)))
    a[3] = 1   // Initialize the fourth element
    C.g((*C.int32_t)(unsafe.Pointer(&a[0])), C.int(len(a)))
}

CC=clang go test -msan -- c msan_fail.go
ThreadSanitizer (TSan)

Data race is that two concurrent accesses to a shared location, at least one of them for writing. It is kind of threading bugs and hard to reproduce and debug. So ThreadSanitizer, TSan for short, is developed to address this issue.

TSan is a dynamic detector of data race and a subproject of LLVM project. It observes following events generated by the running program.

- Memory access events: READ / WRITE
- Synchronization events:
  - Lock events: WrLock, RdLock, WrUnlock, RdUnlock
  - Happens-before events: SIGNAL, WAIT
Algorithm – Hybrid state machine

**Global state:** the information about the synchronization events that have been observed so far (lock sets, happens-before arcs).

**Per-ID state:** the information about each memory location of the running program. Per-ID state consists of two segment sets: the writer segment set $SSWr$ and the reader segment set $SSRd$. $SSWr$ of a given ID is a set of segments where the writes to this ID appeared. $SSRd$ is a set of all segments where the reads from the given ID appeared.
HANDLE-READ-OR-WRITE-EVENT(IsWrite, Tid, ID)
1 ▷ Handle event READ Tid(ID) or WRITE Tid(ID)
2 \( (SS_{WR}, SS_{RD}) \leftarrow \text{GET-PER-ID-STATE}(ID) \)
3 \( \text{Seg} \leftarrow \text{GET-CURRENT-SEGMENT}(Tid) \)
4 if IsWrite
5 ▷ Write event: update \( SS_{WR} \) and \( SS_{RD} \)
6 \( SS_{RD} \leftarrow \{ s : s \in SS_{RD} \land s \not\in \text{Seg} \} \)
7 \( SS_{WR} \leftarrow \{ s : s \in SS_{WR} \land s \not\in \text{Seg} \} \cup \{ \text{Seg} \} \)
8 else ▷ Read event: update \( SS_{RD} \)
9 \( SS_{RD} \leftarrow \{ s : s \in SS_{RD} \land s \not\in \text{Seg} \} \cup \{ \text{Seg} \} \)
10 \text{SET-PER-ID-STATE}(ID, SS_{WR}, SS_{RD})
11 if Is-Race(\( SS_{WR}, SS_{RD} \))
12 ▷ Report a data race on ID
13 \text{REPORT-RACE}(IsWrite, Tid, Seg, ID)

IS-RACE(\( SS_{WR}, SS_{RD} \))
1 ▷ Check if we have a race.
2 \( N_w \leftarrow \text{SEGMENT-SET-SIZE}(SS_{WR}) \)
3 \text{for} i \leftarrow 1 \text{ to } N_w
4  \text{do} \ W_1 \leftarrow SS_{WR}[i]
5 \( LS_1 \leftarrow \text{GET-WRITER-LOCK-SET}(W_1) \)
6  ▷ Check all write-write pairs.
7 \text{for} j \leftarrow i + 1 \text{ to } N_w
8  \text{do} \ W_2 \leftarrow SS_{WR}[j]
9 \( LS_2 \leftarrow \text{GET-WRITER-LOCK-SET}(W_2) \)
10 \text{ASSERT}(W_1 \not\subseteq W_2 \land W_2 \not\subseteq W_1)
11 if \( LS_1 \cap LS_2 = \emptyset \)
12  then return true
13 \text{▷ Check all write-read pairs.}
14 \text{for} R \in SS_{RD}
15  \text{do} \ LS_R \leftarrow \text{GET-READER-LOCK-SET}(R)
16 if \( W_1 \not\subseteq R \) and \( LS_1 \cap LS_R = \emptyset \)
17  then return true
18 return false
TSan runtime library

TSan, as a runtime library, is implemented in compiler-rt which is a part of LLVM project.

TSan requires specific memory layout to implement TSan algorithm. And the memory layout is also arch-related. In order to support TSan on arm, we have to add memory mapping for arm in Tsan library.

TSan runtime library provides many APIs to record memory access events and synchronization events.
TSAN API - Memory access

void __tsan_read(ThreadState *thr, void *addr, void *pc)
void __tsan_write(ThreadState *thr, void *addr, void *pc)
void __tsan_read_pc(ThreadState *thr, void *addr, void *callpc, void *pc)
void __tsan_write_pc(ThreadState *thr, void *addr, void *callpc, void *pc)
void __tsan_read_range(ThreadState *thr, void *addr, uintptr size, void *pc)
void __tsan_write_range(ThreadState *thr, void *addr, uintptr size, void *pc)
TSAN API - Atomic operation

void __tsan_go_atomic32_load(ThreadState *thr, uptr cpc, uptr pc, u8 *a)
void __tsan_go_atomic64_load(ThreadState *thr, uptr cpc, uptr pc, u8 *a)
void __tsan_go_atomic32_store(ThreadState *thr, uptr cpc, uptr pc, u8 *a)
void __tsan_go_atomic64_store(ThreadState *thr, uptr cpc, uptr pc, u8 *a)
void __tsan_go_atomic32_exchange(ThreadState *thr, uptr cpc, uptr pc, u8 *a)
void __tsan_go_atomic64_exchange(ThreadState *thr, uptr cpc, uptr pc, u8 *a)
void __tsan_go_atomic32_fetch_add(ThreadState *thr, uptr cpc, uptr pc, u8 *a)
void __tsan_go_atomic64_fetch_add(ThreadState *thr, uptr cpc, uptr pc, u8 *a)
void __tsan_go_atomic32_compare_exchange(ThreadState *thr, uptr cpc, uptr pc, u8 *a)
void __tsan_go_atomic64_compare_exchange(ThreadState *thr, uptr cpc, uptr pc, u8 *a)
TSan framework in Go

Instrumented app

Go application

Synchronization event

Memory access event

Go runtime

raceread  racewrite  racereadrange  ....

TSan API

TSan runtime lib

__tsan_write  __tsan_read  __tsan_read_pc  ...
func main() {
    done := make(chan bool)
    x := 0
    startRacer(&x, done)
    store(&x, 43) // write x
    <-done
}

func store(x *int, v int) {
    *x = v
}

func startRacer(x *int, done chan bool) {
    go racer(x, done)
}

// start a go routine

func racer(x *int, done chan bool) {
    time.Sleep(10*time.Millisecond)
    store(x, 42) // write x
    done <- true
}

// Instrument application with TSan
Thank you

Join Linaro to accelerate deployment of your Arm-based solutions through collaboration

contactus@linaro.org