Lection 07

Dynamic analysis: binary translation vs. compiler instrumentation

Static vs. dynamic analysis

- We've covered various static analysis techniques for various goals
- Dynamic analysis can help with some situations when static cannot
  - Produce (almost) no false positives
  - Produce a failing input data, ready to debug
  - Cons: expensive or very expensive
  - Cons: will find an error only on the executed path

Dynamic analysis tools

- Basic idea: execute a program on user specified data with some "control" over its behavior
  - This should be good enough to detect "interesting" situations when they occur
  - Then issue a warning and (maybe) terminate a program
- Various approaches to execute "control"
  - Dynamic binary translation
  - Static instrumentation (compiler-level or binary)

Binary translation

- A process of transforming a binary program to some other program
  - Dynamic: do that at runtime
  - Terminology: host/guest program/OS
- Difficulties:
  - Code discovery
  - Self-modifying code
  - Jump to computed address

Incremental translation

- Lazily compile guest binary code fragments as we go
  - A fragment is called a translation block (usually a basic block or an extended bb)
  - Compiled blocks are cached
  - Cached blocks are chained for speed
  - Need to be able to break chains for interrupt processing

Valgrind

- Dynamic binary translation framework + tools for popular defects
- Framework features
  - Kernel interface virtualisation (API level)
  - Debuginfo reading
  - Error management
  - Code JITting and management
  - a GDB server
  - **Tools API** (instrumentation)

VEX compiler

- Extended bb compiler
  - Based on a simple IR (SSA-like)
  - Machine code --> IR --> Instrumented IR --> machine code (insn selection, regalloc)
  - Starting at specified insn, up to next branch
  - Each insn individually translated
  - Optimised over the whole block
  - Follows uncond branches and calls to known destinations (avg block size ~ 10 guest insns)

VEX compiler - II

- IR: simple single-assignment language for straight-line code
  - Loads, stores, assignment to IR temporaries, arithmetic
  - GET and PUT to model register access
  - Side exits (conditional)
- Guest state: holds register values
  - GET and PUT reference offsets in it
  - Dedicate a host register to point at it

Memcheck tool

- Find common memory related errors
  - Observes all memory accesses AND all malloc/free calls
  - Verifies each access is allowable
  - Verifies that undefinedness will not cause observable behaviour
  - As a side effect, checks for memory leaks

char* p = malloc(10); ... p[10] ...	error: out of bounds read
char* p = malloc(10); free(p); ... p[5] ...	error: reading freed
char p[10]; ... if (p[5] == 'x') {..}	error: branch on undefined
char* p = malloc(10); p = NULL;	error: lost block

Memcheck tool - II

❑IR modification example

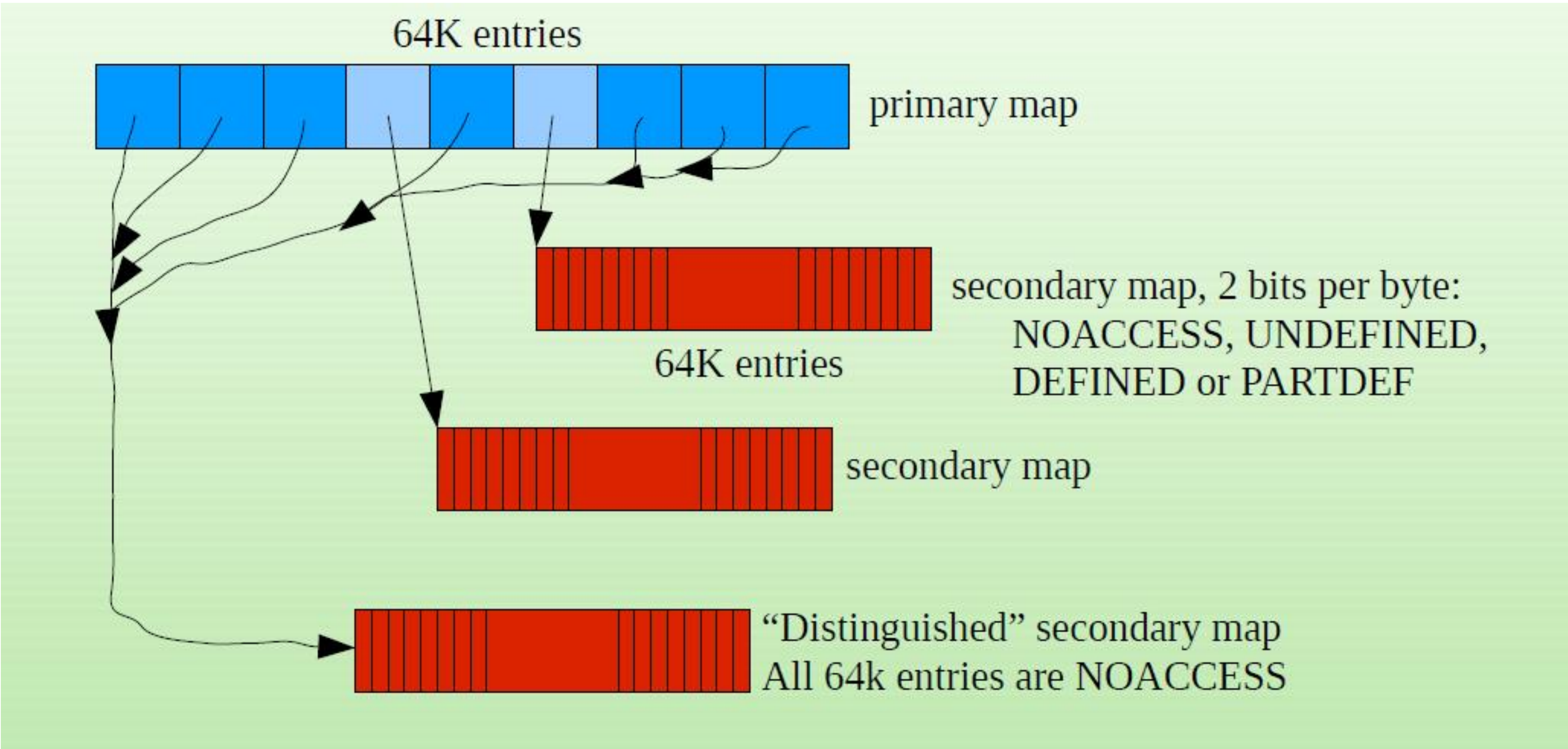
```
Original code         In IR               Instrumentation IR
subq %rax, %rdi       tL = GET(328)       qL = GET(1328)
                      tR = GET(416)       qR = GET(1416)
                      tRes = Sub64(tL, tR) qRes =
                                          Left64(UifU64(qL, qR))
                      PUT(416) = tRes     PUT(1416) = qR
jz 0x1234             ExitIf CmpEQ64(tL, tR) CallIf
                       0x1234             (CmpNEZ64(qRes))
                                            report_error()
movq (%rcx), %rdx     tA = GET(360)       qA = GET(1360)
                      tD = LOAD64le(tA)   CallIf (CmpNEZ64(qA))
                                            report_error()
                                          qD = Call
                                          helper_LOAD64le(tA)
                      PUT(368) = tD       PUT(1368) = qD
```

tXX: 64 bit IR temps holding original values.
qXX: 64 bit IR temps holding definedness bits.
0 = defined, 1 = undefined.

Memcheck architecture

- Shadow registers and memory
  - Initially – 1 bit/byte for addressability, 8 bits/byte for tracking undefinedness
- Compression optimization
  - Bit-precise tracking is crucial for quality but needed rarely for bitwise ops
  - Introduce 2 bits/byte scheme: NOACCESS, UNDEFINED, DEFINED or PARTDEF
  - Full 8 bits mapping only for PARTDEF (rare), slower (AVL tree) but common case is much faster
- Multithreading: serialize threads

Memcheck mapping



Compiler instrumentation

- If we have source code, we can insert all required instrumentation statically
  - Easier - have the compiler power behind you
  - Faster - no dynamic translation penalty
  - Faster - because the compiler can then throw away or optimize instrumentation (if done on IR)
- Cons: sometimes we haven’t that code :)
- For both binary translation and compiler way, need to (re)compile with -g

Google Sanitizers

- Most known attempt to build tools around compiler instrumentation
- A family of tools: address, memory, thread, kernel, … sanitizer
- Supported in LLVM and then GCC (LLVM is main version, GCC lags behind a bit)
- Faster than Valgrind (2x-5x vs 10x-30x)

Address Sanitizer

- Find heap/stack/global out of bounds accesses
- Find use after free
- Idea: compiler instrumentation + shadow memory and runtime library
- Instrumentation:

```
if (IsPoisoned(address)) {
  ReportError(address, kAccessSize, kIsWrite);
}
*address = ...;
```

Address Sanitizer - II

- Shadow memory
  - 8 bytes to 1 byte
  - All 8 bytes in qword are unpoisoned (i.e. addressible). The shadow value is 0.
  - All 8 bytes in qword are poisoned (i.e. not addressible). The shadow value is negative.
  - First k bytes are unpoisoned, the rest 8-k are poisoned. The shadow value is k. This is guaranteed by the fact that malloc returns 8-byte aligned chunks of memory.

Address Sanitizer - III

ISPRAS

- Then the instrumentation looks like:

```
byte *shadow_address = MemToShadow(address);
byte shadow_value = *shadow_address;
if (shadow_value) {
  if (SlowPathCheck(shadow_value, address,
                     kAccessSize)) {
     ReportError(address, kAccessSize, kIsWrite);
  }
}
// Check the cases where we access first k bytes
// of the qword and these k bytes are unpoisoned.
bool SlowPathCheck(shadow_value, address,
                  kAccessSize) {
  last_accessed_byte = (address & 7) +
                        kAccessSize - 1;
  return (last_accessed_byte >= shadow_value);
}
```

Address Sanitizer - IV

ISPRAS

- Shadow memory mapping
  - 64-bit: Shadow = (Mem >> 3) + 0x7fff8000;
  - 32-bit: Shadow = (Mem >> 3) + 0x20000000;
- ReportError function
  - Now a call, was an insn making SIGILL (ud2a)

```
shr $0x3,%rax                   # shift by 3
mov $0x100000000000,%rcx
or %rax,%rcx                    # add offset
cmpb $0x0,(%rcx)                # load shadow
je 1f <foo+0x1f>
mov %rdi,%rax                   # failing address in %rax
Ud2a                            # generate SIGILL
movq $0x1234,(%rdi)             # original store
```

Stack buffer overflow

- Redzones that are poisoned/unpoisoned on function entry/exit

```
void foo() {
  char redzone1[32]; // 32-byte aligned
  char a[8]; // 32-byte aligned
  char redzone2[24];
  char redzone3[32]; // 32-byte aligned
  int *shadow_base = MemToShadow(redzone1);
  shadow_base[0] = 0xffffffff; // poison redzone1
  shadow_base[1] = 0xffffff00; // poison redzone2, unpoison 'a‘
  shadow_base[2] = 0xffffffff; // poison redzone3

  <... function code ...>

  shadow_base[0] = shadow_base[1] = shadow_base[2] = 0;
  // unpoison all return;
}
```

Run-time library

- Malloc/free wrappers
  - malloc allocates memory with redzones and poisons redzones (in shadow)
  - free poisons deallocating regions and places it on quarantine
- Strlen etc. interceptors
- Error printing, stack traces, ...

Thread Sanitizer

- Instrumentation: memory accesses
- Shadow memory: 8 byte -> N 8-byte words
  - Each shadow word has thread id, clock, read/write bit, access size, address offset
  - Each shadow word represents an access
- State machine: updates shadow words on memory access
  - If cannot order two memory accesses to same region (different threads, no locking), report a race

Memory Sanitizer

ISPRAS

- Detects uninitialized memory reads (not supported by Address Sanitizer)
- Propagating shadow data
  - Can copy uninitialized data (or too many warnings)
  - Can process it, too
  - Propagate through expressions, calls
  - Report only on branches or side-effects (calls)
- Track origins of uninitialized data
  - Similar to valgrind --track-origins=yes
  - Secondary shadow memory
  - Propagating it, too

Memory Sanitizer - II

- Need to track all memory stores or false positives
  - May happen in standard libraries, syscalls, asm, JIT compilation
- Possible solutions
  - Instrument "everything" (at least std libs)
  - Add dynamic tool like valgrind / DynamoRIO

References

- https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm
- https://github.com/google/sanitizers/wiki/MemorySanitizer
- https://gcc.gnu.org/wiki/cauldron2012?action=AttachFile&do=get&target=kcc.pdf
- https://fosdem.org/2017/schedule/event/valgrind_memcheck/attachments/slides/1841/export/events/attachments/valgrind_memcheck/slides/1841/valgrind_memcheck.pdf
- https://fosdem.org/2017/schedule/event/valgrind_vex_future/attachments/slides/1842/export/events/attachments/valgrind_vex_future/slides/1842/valgrind_vex_future.pdf
- http://valgrind.org/docs/valgrind2007.pdf
- http://valgrind.org/docs/shadow-memory2007.pdf