Valgrind

From Magic to Science

Shachar Raindel

The Technion

Haifux, 2010

Outline

Outline

Outline

Valgrind

"Valgrind is an award-winning instrumentation framework for building dynamic analysis tools" - Valgrind's front page

- Mostly known for Memcheck, which pinpoint many common problems in C/C++ code
- Extremely useful tool for the novice C/C++ programmer
- Also useful for experienced ones
- Similar to Purify, Bounds-Checker, CodeGuard and Insure++
- Supports X86/Linux, AMD64/Linux, PPC32/Linux, PPC64/Linux and X86/Darwin (Mac OS X)
- ARM/Linux and MIPS/Linux ports are in progress, some versions for *BSD
- Supports wine - test your windows code with Valgrind!
- Available for X86/Linux since ~2003, actively developed

Sample (Bad) Code

- We will use the following code for demonstration purposes:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #define SIZE 100
4  int main() {
5      int i, sum = 0;
6      int *a = malloc(SIZE);
7      for( i=0; i < SIZE; ++i ) sum += a[i];
8      a[26] = 1;
9      a = NULL;
10     if(sum > 0) printf("Hi!\n");
11     return 0;
12 }
```

- Contains many bugs. Compiles without warnings or errors.

Live Demo

- We are going to:
  - compile the code

    gcc -Wall -ansi -pedantic -g -o sample sample.c
  - run the bare code

    ./sample
  - run the program with valgrind

    valgrind --leak-check=full ./sample
- Results analysis will follow shortly after

Invalid Reads

Example

```
==8990== Invalid read of size 4
==8990== at 0x804844A: main (sample.c:7)
==8990== Address 0x417e08c is 0 bytes after a block of size 100
alloc'd
==8990== at 0x4024C1C: malloc (vg_replace_malloc.c:195)
==8990== by 0x8048430: main (sample.c:6)
```

- We read past the end of the allocated array
- Trying to read from area which we are not allowed to access
- Could result in a SEGFAULT and surely doesn't do what we want
- Valgrind provides enough details to find the problem.

Invalid Writes

Example

==8990== Invalid write of size 4
==8990== at 0x8048463: main (sample.c:8)
==8990== Address 0x417e090 is 4 bytes after a block of size 100 alloc'd
==8990== at 0x4024C1C: malloc (vg_replace_malloc.c:195)
==8990== by 0x8048430: main (sample.c:6)

- Similar to invalid read
- Details provided by valgrind:
  - Location of fault (addresses, line number if debug-information present)
  - Stack-trace to fault (you can get more using $--num-callers=30$)
  - Relevant blocks details and allocation/de-allocation stack-trace

Memory Leaks

- At the end of the run, Valgrind does "Garbage Collection"
- Unreferenced memory in C/C++ $\Rightarrow$ memory leak

Example

==8990== 100 bytes in 1 blocks are definitely lost in loss record 1 of 1
==8990== at 0x4024C1C: malloc (vg_replace_malloc.c:195)
==8990== by 0x8048430: main (sample.c:6)

- Valgrind provides stack-trace for the allocation point
- 3 kinds:
  - Definitely lost (no pointers to allocation)
  - Probably lost (pointers only to the middle of the allocation)
  - Still reachable (block hasn't been free'd before exit, but pointers to it still exists)

Use of Uninitialized Value

Example

==8990== Conditional jump or move depends on uninitialised value(s)
==8990== at 0x8048476: main (sample.c:10)

- Valgrind checks and make sure the program flow is deterministic
- Usage of values which haven't been initialized in conditions is reported
- Also if they are passed as parameters for syscalls
- Valgrind will detail location in which the uninitialized data was used
- To get trace to the source of it, add "*--track-origins=yes*" to the command-line

Outline

Danger, Will Robinson

- `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0166`
- Most of the time Valgrind's errors describe latent flaws in the program
- However, sometimes Valgrind is wrong
- Cryptography related code requires special attention
- A debian developer commented out code that valgrind didn't like
- Resulting in a latent bug
- With massive security implications
- All because valgrind claimed a value is used uninitialized
- Was intentionally used so, to collect more entropy

Problems that Valgrind doesn't Detect

- Buffer overflows which ends up accessing valid memory:

Example

```
char *p = malloc(1024); /* block 1 */
char *q = malloc(1024); /* block 2 */
p += 1200; /* "p" now points into block 2 */
*p = 'a';  /* invalid write - undetected */
```

- Accesses to stack variables and global variables are not checked
- Business logic/algorithmic problems are not detected
- Checks only code that was executed
- Doesn't work well with statically-linked code

Outline

Other Tools in Valgrind

- We talked about Memcheck – the default tool in Valgrind
- But Valgrind can do much more than that
- Contains many tools, some stable, some experimental
- You can even write your own tools in few days of work
- Valgrind ships with the following tools:

Memory Error Checkers

– Ptrcheck – Memcheck

Thread Error Detectors

– Helgrind – DRD

Profilers

– Cachegrind – Callgrind
– Massif

Sample Tools and others

– Lackey – None
– BBV

Ptrcheck

- Activate by adding *--tool=exp-ptrcheck*
- Similar in goals to Memcheck
- Still experimental
- Uses very different approach than Memcheck
- Can detect failures Memcheck doesn't detect (like the code we ▸ saw)
- Has got false positives (different than the ones Memcheck gets)
- Slower than Memcheck
- Doesn't check for memory leakage and validity of accesses
- Use alongside Memcheck for better coverage

Profiling Tools

- Cachegrind
  - Traces the code memory accesses and jump patterns
  - Simulates a 2-level cache and branch predictor
  - Provide details about cache misses and their source
  - Can help optimizing performance critical code
- Callgrind
  - Extends Cachegrind
  - Propagates the costs along the call-tree
  - Has a KDE front-end – KCacheGrind
- Massif
  - Memory allocations profiler
  - Keeps stack-trace of every memory allocation/deallocation
  - Print memory usage status in peak times and upon specific intervals

Thread Error Detectors

- Functionality similar to Intel's ThreadChecker
- Detects a variety of threading related problems:
  - Threading API misuse
  - Lock order problems (potential dead-locks)
  - Data-Races
- Two similar implementations in Valgrind
  - Helgrind
    - The Internet claims it produces many false positives
    - Supposedly catches more errors too
  - DRD

Outline

Client Requests

- Valgrind provides communication channel for tested programs
- Good for unit-test harnesses
- Also good if you are doing weird stuff in your code
  - A special memory allocator, such as object-pool
    - Use VALGRIND_CREATE_MEMPOOL to mark a memory area as an object-pool
    - Use VALGRIND_MEMPOOL_ALLOC to mark an object as allocated
    - Use VALGRIND_MEMPOOL_FREE to mark an object as free
  - Self-Modifying code, i.e. JIT compiler
    - Use VALGRIND_DISCARD_TRANSLATIONS to report about areas in which code has been changed
- Useful also for hunting bugs
  - For example, check for memory leaks, using VALGRIND_DO_LEAK_CHECK

Suppression Files

- Valgrind tends to be very noisy
- Most of the times it is indicating bugs that should be fixed
  - But not always the one we want to fix right now
- Sometimes it is correct code, which Valgrind failed to understand
  - Mostly in sophisticated/extremely optimized library code
  - Also possible when having unusual interactions with the kernel
- Valgrind includes a mechanism to silent a specific error
  - Works with all tools that report errors
  - Simple file format, see documentation for details
  - Valgrind includes suppression for many common libs

Outline

VEX - Binary Translation

- We want to inspect all memory accesses in the code
- Straight forward solution - CPU emulation
- But this is really slow
- Treat the program binary as "source-code"
- Allow the tools to modify the code being compiled
- VEX's front-end translates X86 opcodes into Intermediate Representation code
- VEX's back-end translates IR code back to X86 code
- Tools can manipulate the IR code in the middle

IR code

- VEX translates each guest opcode into a block of IR code
- IR code is similar to assembly for a RISC-style machine
- IR code assumes machine with infinite number of variables
- The "guest state" is stored in a special memory area

IR translation of "addl %eax, %ebx"

```
—— IMark(0x24F275, 7) ——
t3 = GET:I32(0) # get %eax, a 32-bit integer
t2 = GET:I32(12) # get %ebx, a 32-bit integer
t1 = Add32(t3,t2) # addl
PUT(0) = t1 # put %eax
```

IR code - Properties

- IR code is fully typed:
  - All variables and results of calculations have type
  - No implicit type conversion
  - VEX performs sanity checks on these types all of the time
- IR code is in a Single Static Assignment form
  - Each variable is assigned only once
  - Simplifies the instrumentation of the code
  - Also simplifies the optimization of the code when running it
- IR code is presented to the tools in semi-parsed form
  - Easy to manipulate lists of instructions
  - Instructions are presented in a convinient data-structure
  - Useful functions for manipulating IR code (add/remove instructions, etc.)

JIT, VEX and IR

- The $X86 \rightarrow IR \rightarrow X86$ translation is done in a Just In Time manner
- Every basic-block is translated upon its first execution

Definition

Basic-Block - a linear sequence of code, with one entry point, one exit point and no jump instructions contained within it.

- The translation is done so that Valgrind's dispatcher regains control after each basic-block
- Caching of translated code blocks improves execution speed

Bootstrap Code

- Small, shared launcher - launches the relevant tool
- Each tool's binary contains a complete copy of the core code
- The interesting stuff in bootstrap:
  - Reading the debug information for the target program
  - Initialize VEX - Valgrind's binary-translation mechanism
  - Call tool-specific init code
  - Load the target program
  - Setup the environment for the target program run
  - Initialize Valgrind's thread-scheduler
    - The scheduler makes sure that only one thread runs at a time
    - Still replicates the entire thread structure to the OS
    - Also handles signals
  - Kick-start the "guest" application main thread, using VEX and Valgrind's dispatcher

Function Call Redirection

- Valgrind implements a redirection mechanism
- This mechanism can "hijack" various function calls
- Done in the binary translation level
- Some examples where this ability is useful:
  - Memory allocation functions
  - Various sys-calls (write/read files, etc.)
  - Loading dynamic-link library
  - Replace some optimized functions with debug versions
- The guest code can request that as well
  - Redirect requests are indicated by a specially mangled function name
  - Special "magic-sequance" to call the original function
  - Nice C macros make it developer friendly
  - See documentation for details

Client Requests - Behind the Scenes

- The VEX compiler recognizes few magic-sequances
- The magic-sequances are a no-op when run on normal CPU
- In X86, the magic-sequance is “roll 3, %edi ; roll 13, %edi; roll 29, %edi ; roll 19, %edi;”
- Followed by “xchgl %reg, %reg”, where reg indicates the kind of the magic-sequance
- Similar style request for the other platforms
- Used to trigger Valgrind’s Client Request mechanism
- On X86, client request param is passed on EAX, result is returned on EDX
- Also used for calling functions without the redirects
- And for getting the original address of a redirected call

Outline

Memcheck Basics, A-bits

- Memcheck tracks what memory the guest application may access
- Two levels of allowed access:
  - Write to / read from the memory
  - Use the value in memory for anything serious
- The first one is tracked with "A" bits (Access)
  - One bit per byte of memory
  - Set to 0 if guest is not allowed to access (i.e. free'd block)
  - Set to 1 upon memory allocation
  - Report an error if guest code touched memory with A == 0

V-bits

- The second access level is tracked with "V" bits (Valid)
  - One bit per bit of memory
  - Set to 1 if the original memory bit haven't been defined yet
  - Set to zero once the memory bit is set
  - State is transitive

Example

If c is not defined, evaluating $a = b + c$ will get a to be undefined too

  - Reports an error when conditional jumps and syscalls are given undefined values

Leak Check

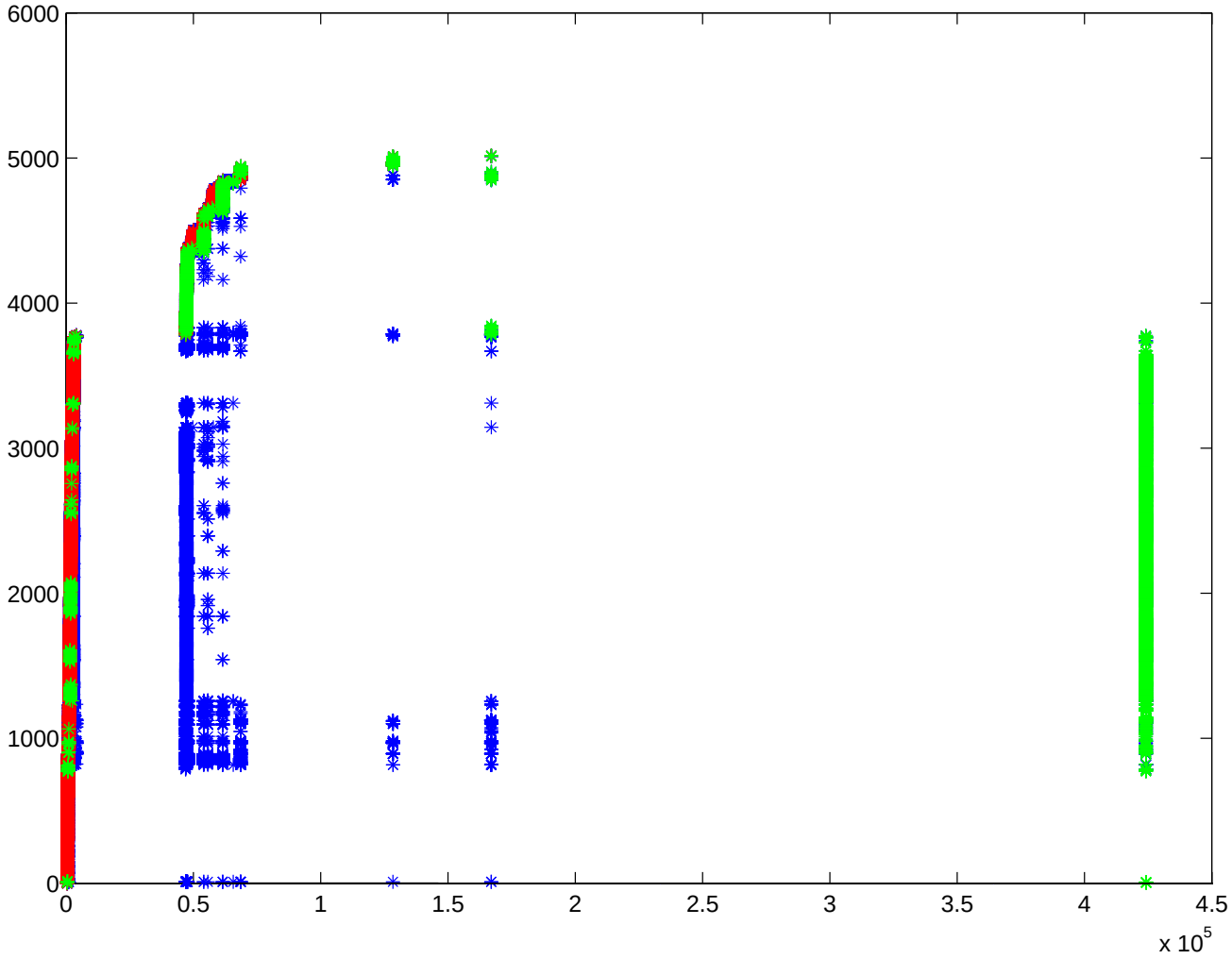
- Memcheck does leak check in the end of a run of a program
- It is possible to trigger it through a client request as well
- Very similar in concept to garbage–collection
- Chases pointers from globals and stack variables
- Only considers pointers which are defined (V-bit = 0)
- If there are no pointers to a block, it is “definitely lost”
- If there are still valid pointers to a block, it is “still reachable”
- If there is a pointer to somewhere in the middle of a block, it is “possibly lost”
- If the only pointers to a block are from definitely lost blocks, it is “indirectly lost”
- Valgrind keeps the stack-trace of the allocation point for each block

Outline

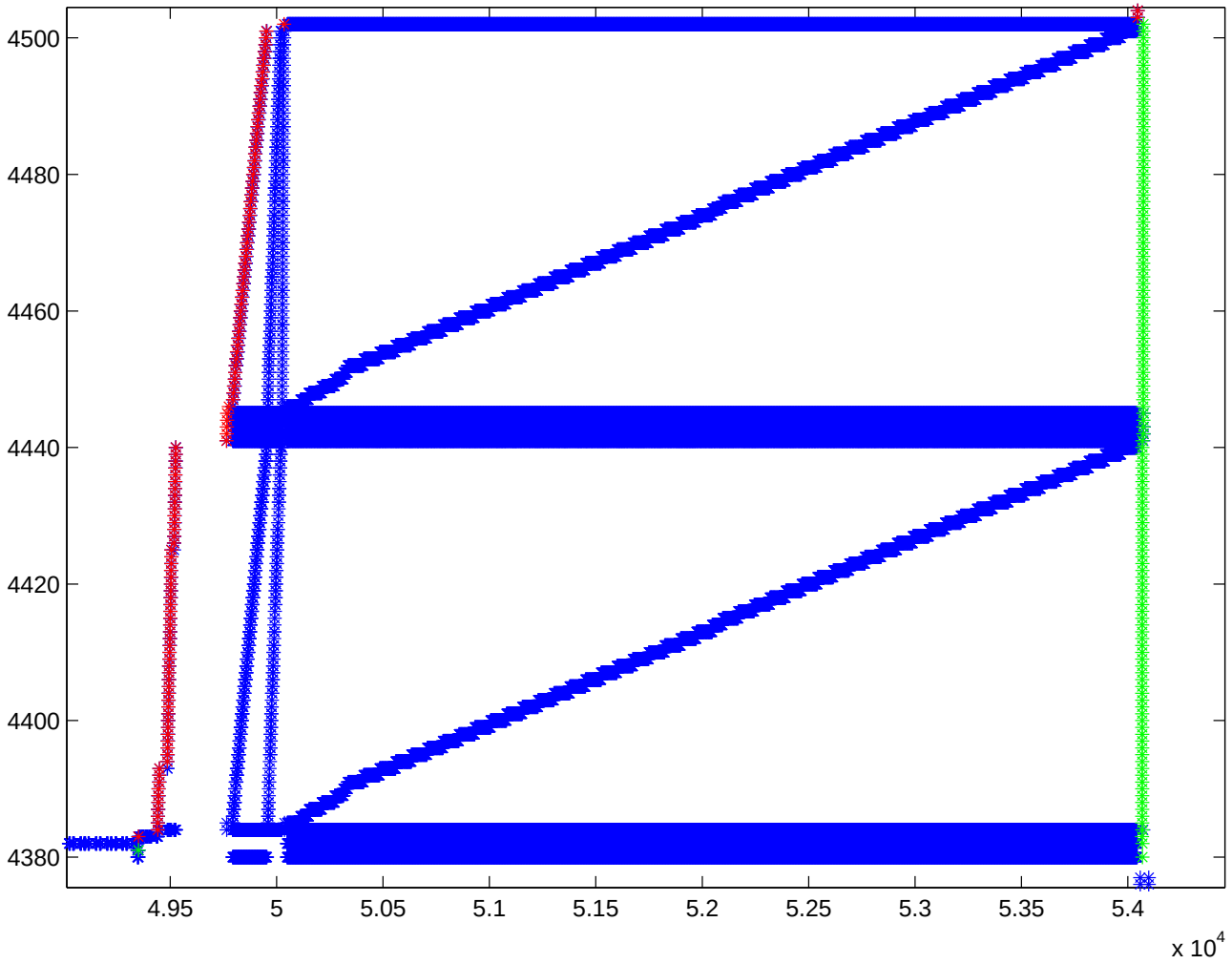
My Research (So Far)

- In my research, I'm looking into memory allocation schemes
- My claim is that with proper hinting from the application these can be improved significantly
- I use valgrind to measure the current situation
- Attempting to quantify possible improvement
- A new tool, which I call mtrace
- Practically merges parts of lackey and of massif
- Traces all allocations and memory accesses
- Post-processing scripts for gathering statistics and data
- Nice plots of memory access patterns in Matlab (next slides)

Memory Access Patern of Xvnc



Memory Access Patern of Xvnc (Detail)



Questions?

?