Compilers Laboratory

DCC
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

ICEx Instituto de Ciências Exatas

UFMG

Valgrind: A Developer's Intro

Andrei Rimsa Álvares
andrei@decom.cefetmg.br

Summary

- Introduction
- Valgrind on Eclipse
- Creating a tool
- Understanding Valgrind
- Building the tool: sectionprof
- Conclusion

Compilers Laboratory

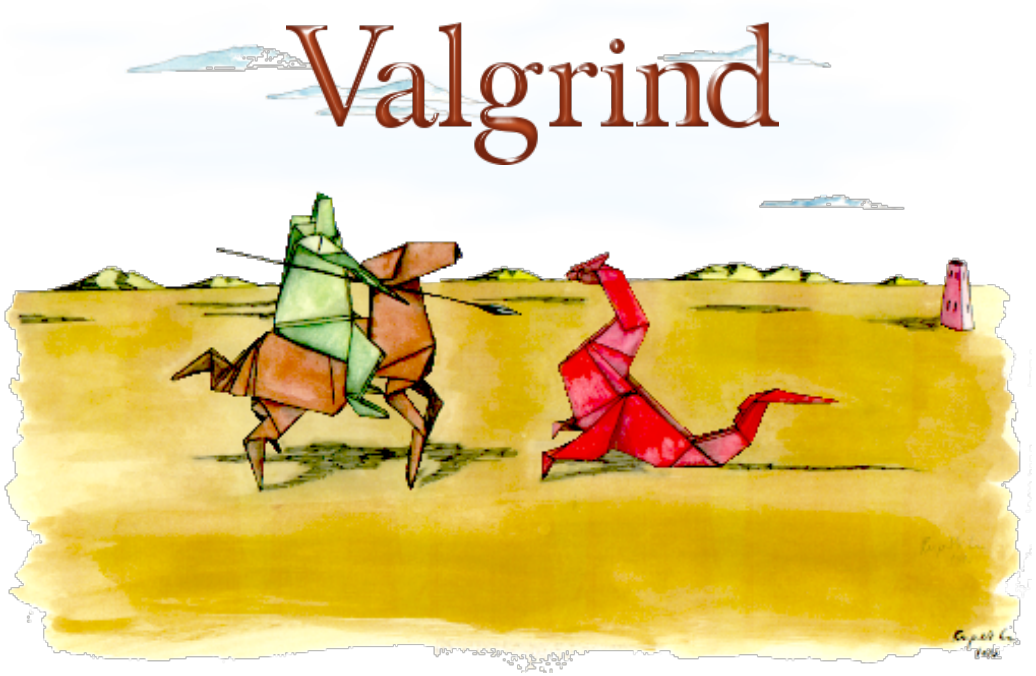
DEPARTMENT OF COMPUTER SCIENCE
FEDERAL UNIVERSITY OF MINAS GERAIS, BRAZIL

Introduction

LAC

Introdution

- Before we dive into Valgrind, we need to address some questions...



http://www.valgrind.org

How do you say Valgrind?

Introdution

- From the Valgrind's FAQ

The "Val" as in the word "value".

The "grind" is pronounced with a short 'i' ie. "grinned" (rhymes with "tinned") rather than "grined" (rhymes with "find").

Why the name Valgrind?

♠: http://valgrind.org/docs/manual/faq.html#faq.pronounce

Introdution

- From the Wikipedia about Vahalla♥

"Valhalla" (1896) by Max Brückner

"the holy doors of the ancient gate Valgrind stand before Valhalla"

What is Valgrind?

♥: https://en.wikipedia.org/wiki/Valhalla

Introdution

- Valgrind is framework for...

Dynamic
Binary
Instrumentation

Dynamic behavior (execution)
instead of static structure (code sections)

Binary code (low level) as
opposed to source code (high level)

Program execution manipulation
by adding, modifying or
removing instructions

Are there
others DBIs?

Introdution

- Some (famous) alternatives to Valgrind are...

https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

http://www.dynamorio.org

What are DBI's used for?

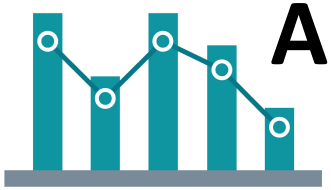
Introdution

- DBI's are used for...

Dynamic
Binary
Analysis

Same as before

We can analyze the behavior of a program, instrumenting it during runtime



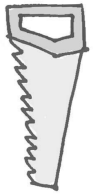
Warning: we do not transform the executable, only the execution

What are some analysis examples?

Introdution

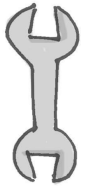
And how does valgrind works?

- Each DBI has its own analysis (tools), specifically for valgrind:



Memcheck: detects memory-management problems, where all reads and writes of memory are checked, and calls to malloc/new/free/delete are intercepted

Cachegrind: a cache profiler that performs detailed simulation of the L1, D1 and L2 caches

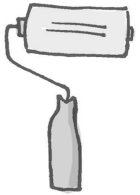


Callgrind: an extension to cachegrind that produces extra information about callgraphs

Helgrind: a thread debugger which finds data races in multithreaded programs

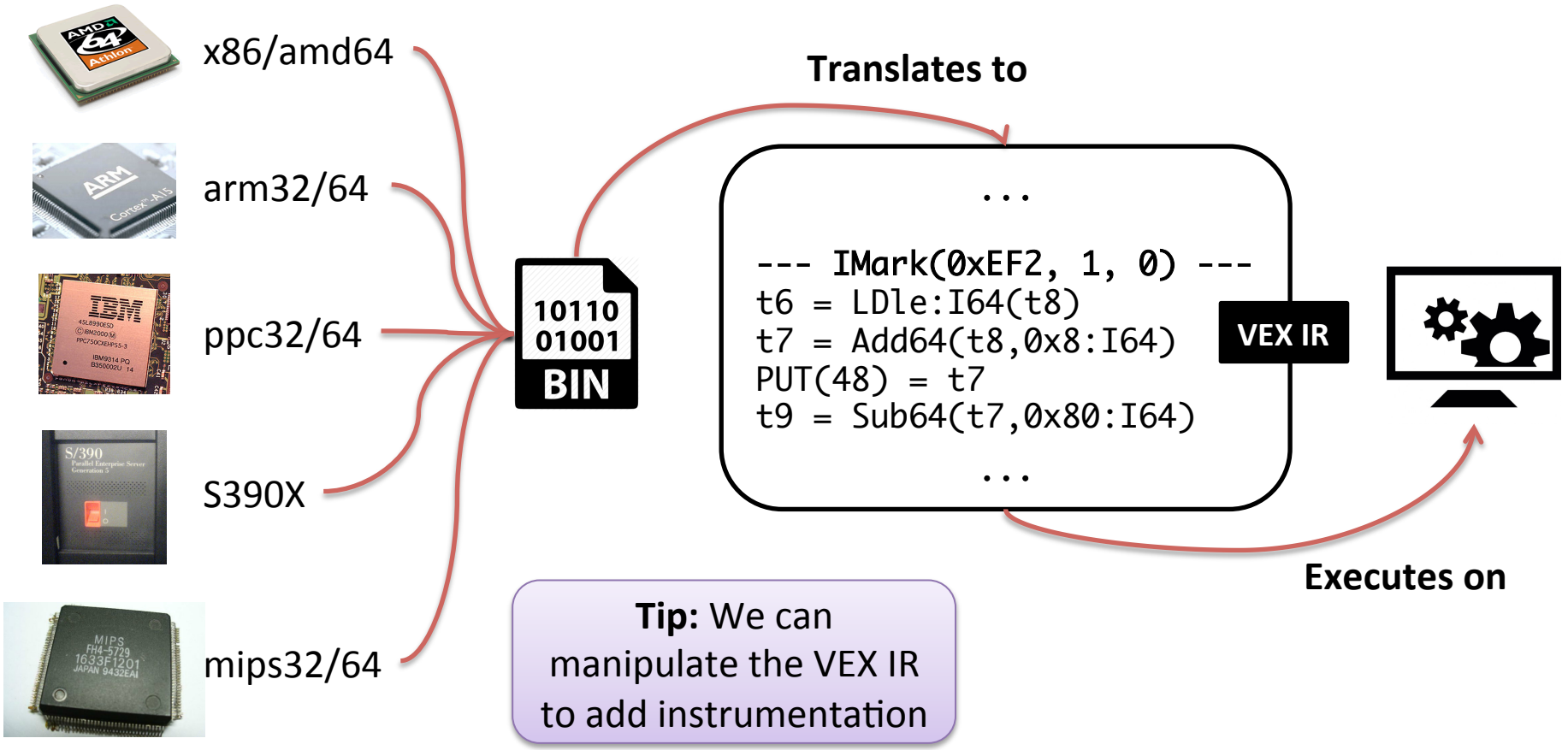
Massif: a heap profiler that performs detailed heap profiling by taking regular snapshots of a program's heap

DRD: a tool for detecting errors in multithreaded C and C++ programs



Introdution

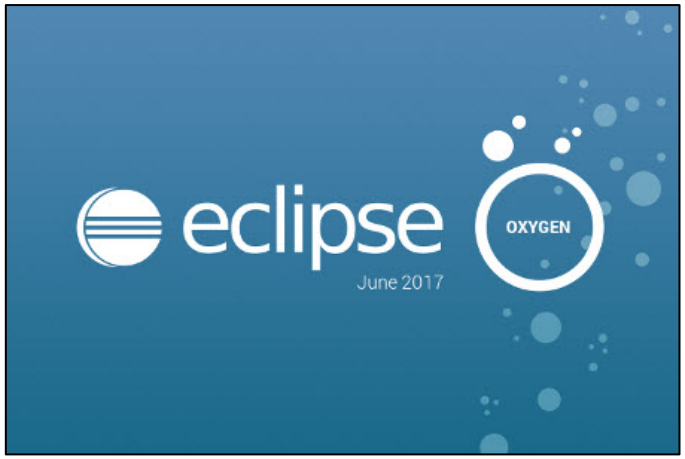
- Valgrind is a program emulator/simulator that allows instructions manipulation/injection



Compilers Laboratory

DEPARTMENT OF COMPUTER SCIENCE
FEDERAL UNIVERSITY OF MINAS GERAIS, BRAZIL

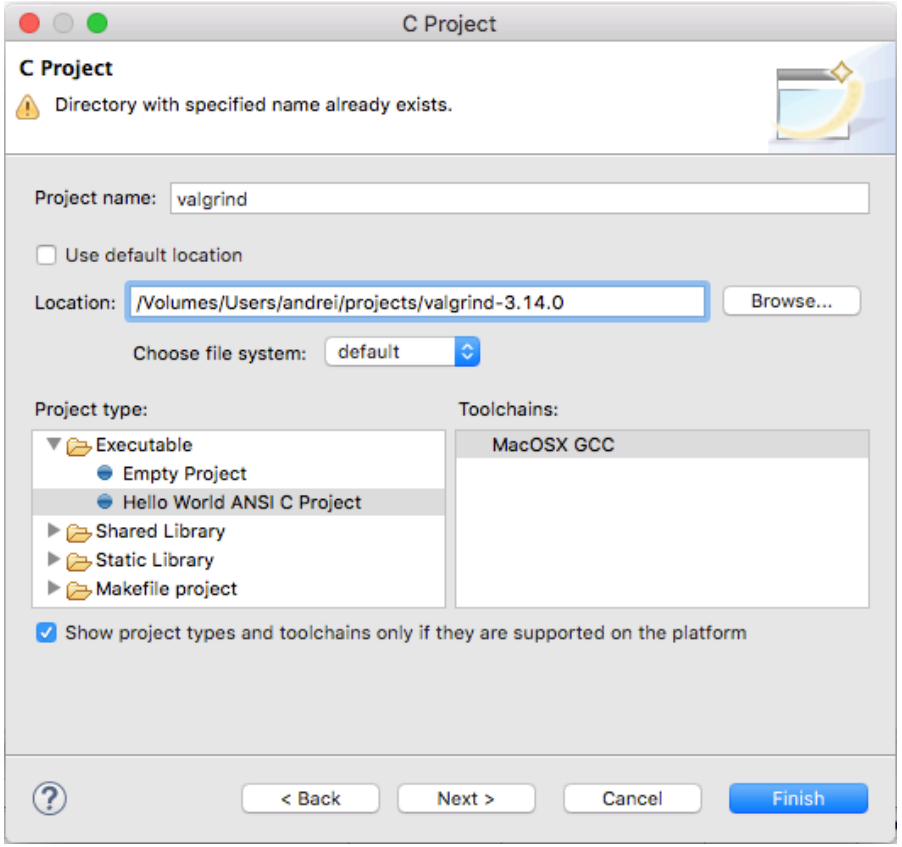
Valgrind on Eclipse



LAC

Eclipse

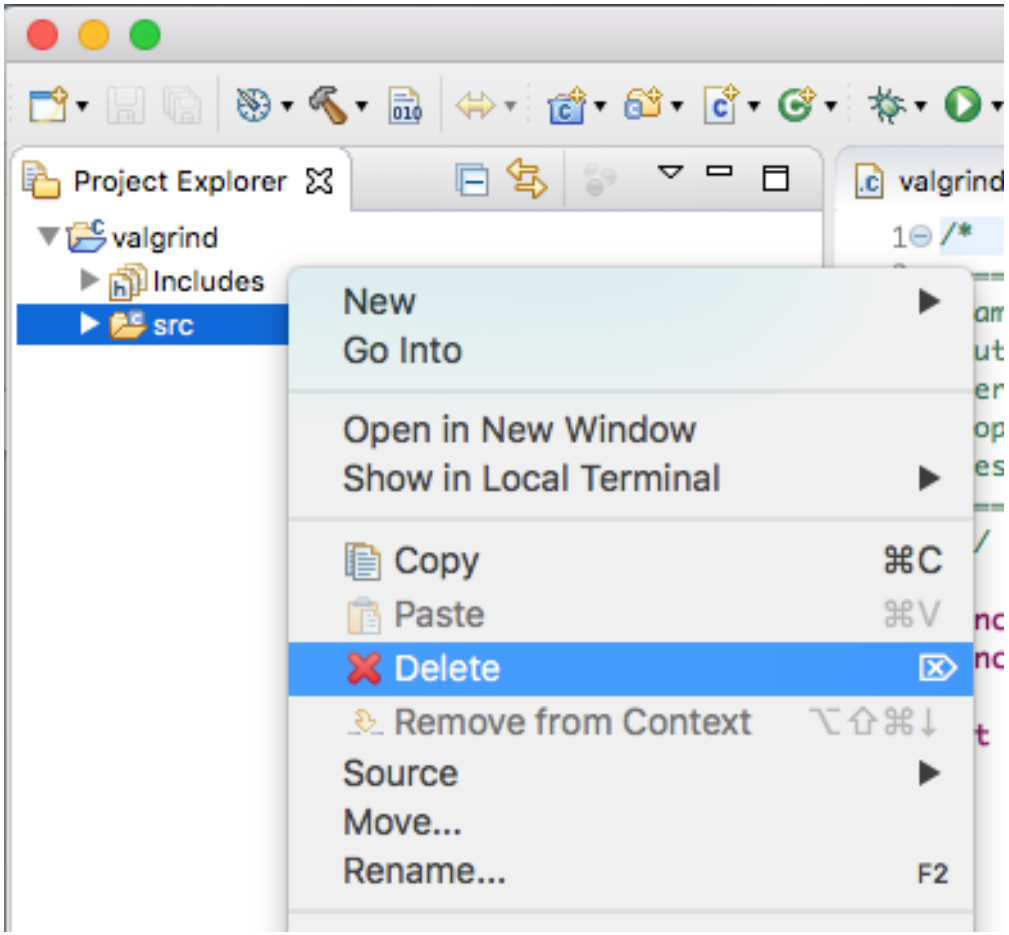
- Create a new project
  - File > New > Project
  - C/C++ > C Project
    - Project Name: valgrind
    - Location: **~/projects/valgrind-3.14.0**
    - Project type: Hello World ANSI C Project
    - Tool chain: (default)
    - Click Finish



Warning: select valgrind-3.14.0 as path name for the next stages

Eclipse

- Remove src directory
  - Right click "src" directory
  - Choose delete



Tip: this ensures correct C includes path configuration

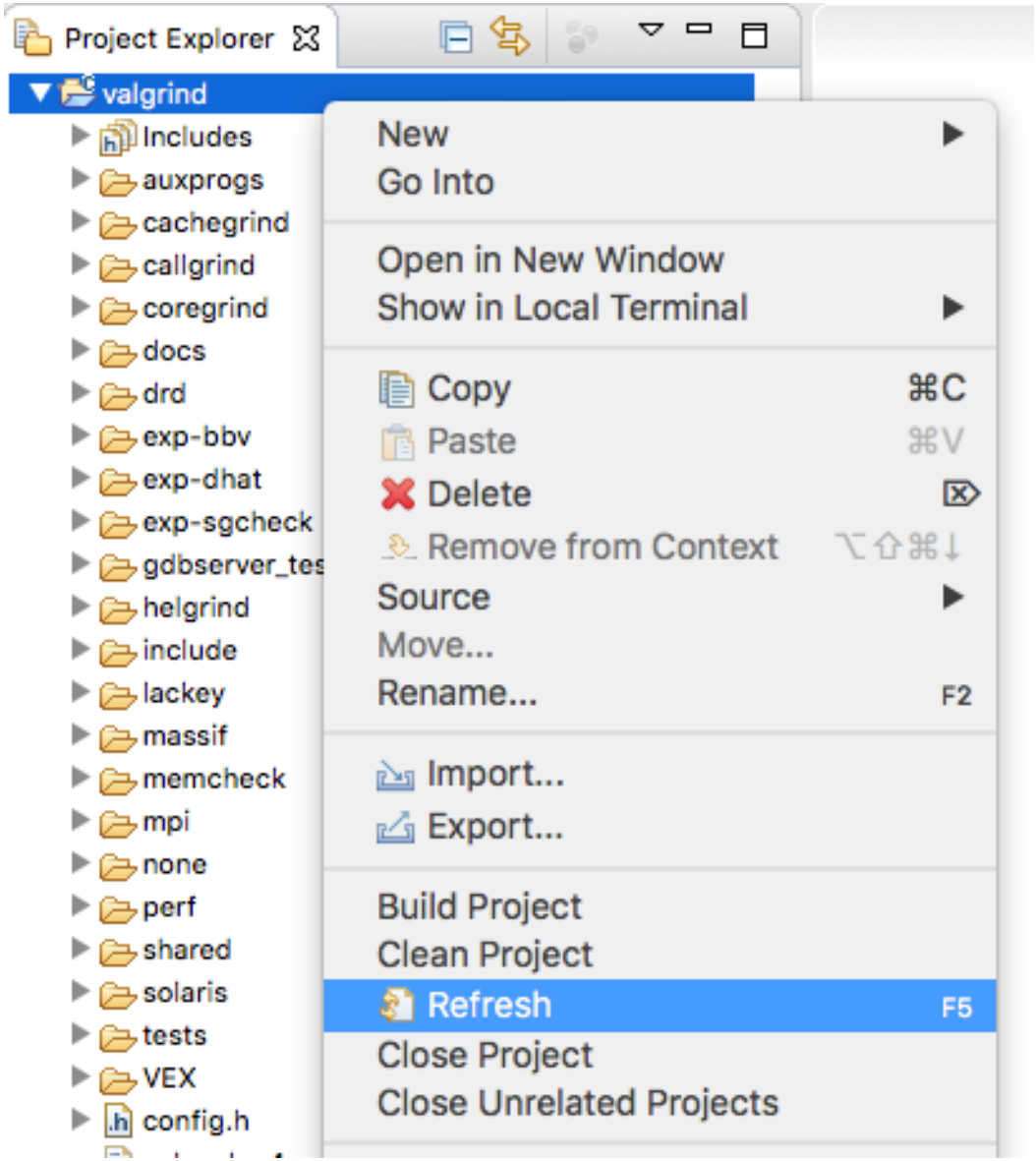
Eclipse

- Download and unpack valgrind in our directory

```
$ cd ~/projects
$ wget http://www.valgrind.org/downloads/valgrind-3.14.0.tar.bz2
$ tar –jxvf valgrind-3.14.0.tar.bz2
$ rm valgrind-3.14.tar.bz2
```

Eclipse

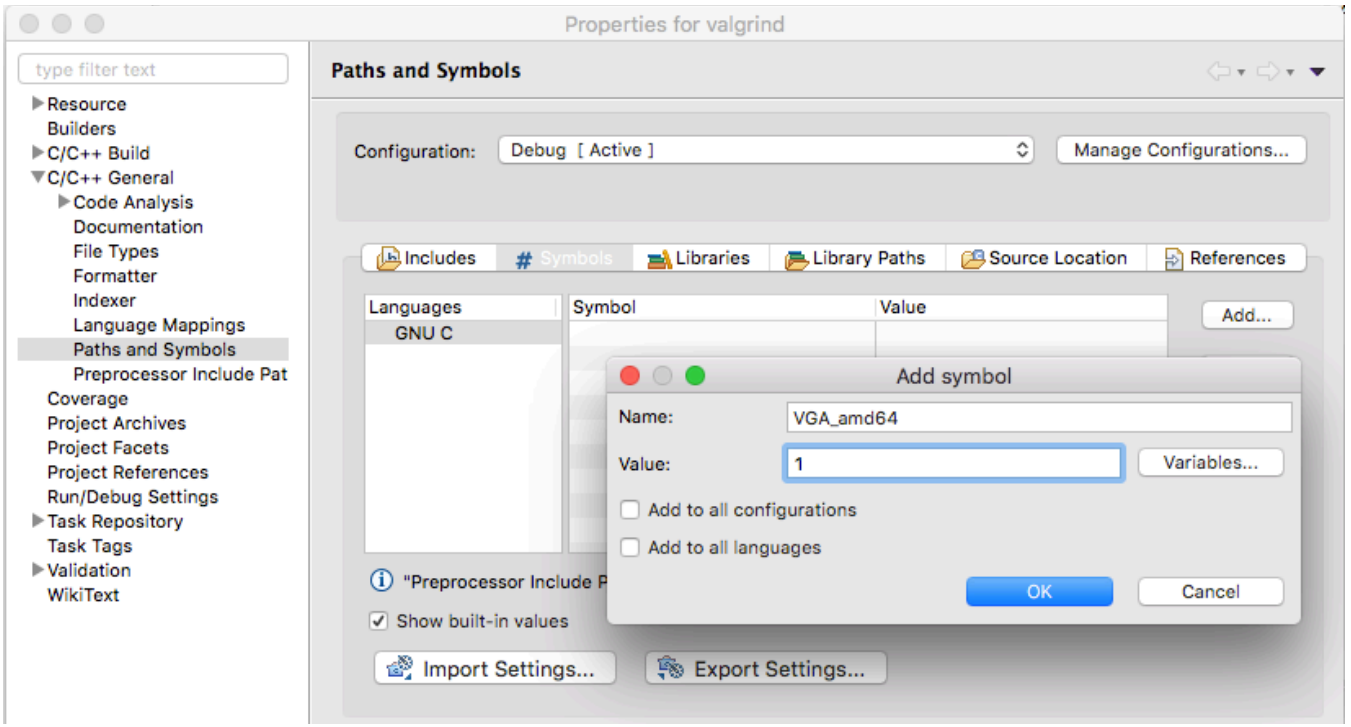
- Refresh the project directory to see the whole structure



Eclipse

Tip: rebuild index and open/ close project if necessary

- Configure project variable
  - Right click on project and select "Properties"
    - C/C++ General > Paths and Symbols and select “# Symbols” tab
    - Click “Add” and select name “VGA_amd64” and value “1”



Creating a tool

The Tool

- Let's create the structure of a tool, based on **nulgrind**♣

Info: We shall name it **sectionprof**, more details in following sections

```
$ cd ~/projects/valgrind-3.14.0
$ mkdir sectionprof
$ cd sectionprof
$ sed -e "s/none/sectionprof/g" -e "s/nl_main/main/g" \
      -e "s/nl-/sp-/g" -e "s/NONE_/SECTIONPROF_/g" \
      ../none/Makefile.am > Makefile.am
$ sed -e "s/nl_/sp_/g" ../none/nl_main.c > ./main.c
$ mkdir docs
$ cp ../none/docs/nl-manual.xml ./docs/sp-manual.xml
$ cp -R ../lackey/tests .
```

Tip: another good candidate is **lackley**

Warning: we should update tool information on main.c and docs/sp-manual.xml

♣: http://valgrind.org/docs/manual/nl-manual.html

The Tool

- Add our tool to valgrind's compiling infrastructure: `Makefile.am` and `configure.ac` in project's root directory

Makefile.am

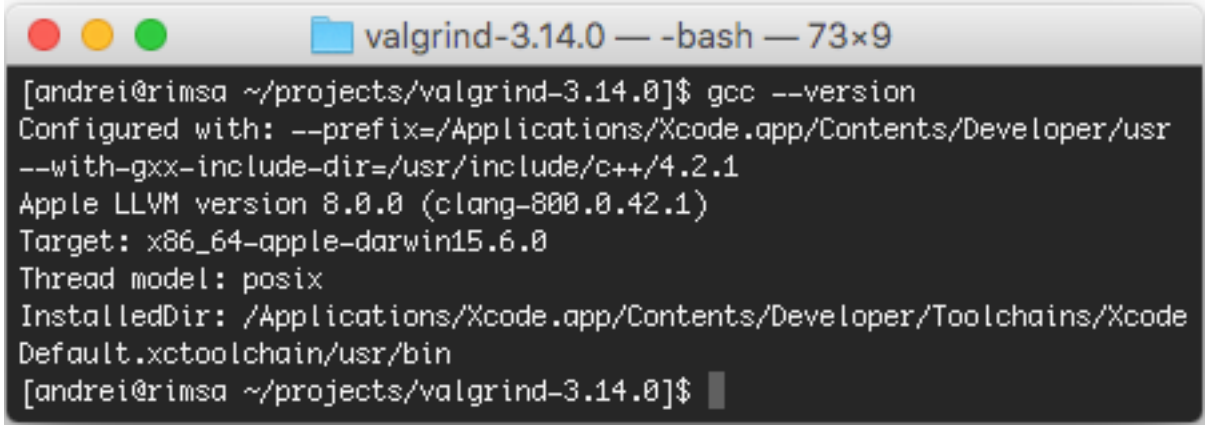
```
AUTOMAKE_OPTIONS = foreign 1.10 dist-bzip2

include $(top_srcdir)/Makefile.all.am

TOOLS =         memcheck \
                cachegrind \
                callgrind \
                massif \
                lackey \
                none \
                sectionprof \
                helgrind \
                drd

EXP_TOOLS =     exp-sgcheck \
                exp-bbv \
                exp-dhat
```

configure.ac — Edited

```
none/tests/linux/Makefile
none/tests/darwin/Makefile
none/tests/solaris/Makefile
none/tests/amd64-linux/Makefile
none/tests/x86-linux/Makefile
none/tests/amd64-darwin/Makefile
none/tests/x86-darwin/Makefile
none/tests/amd64-solaris/Makefile
none/tests/x86-solaris/Makefile
sectionprof/Makefile
sectionprof/tests/Makefile
exp-sgcheck/Makefile
exp-sgcheck/tests/Makefile
drd/Makefile
drd/scripts/download-and-build-splash2
drd/tests/Makefile
exp-bbv/Makefile
exp-bbv/tests/Makefile
exp-bbv/tests/x86/Makefile
exp-bbv/tests/x86-linux/Makefile
exp-bbv/tests/amd64-linux/Makefile
exp-bbv/tests/ppc32-linux/Makefile
```

The Tool

- Apply the following patch if using clang




```
$ wget -O inline.patch https://pastebin.com/raw/WWvGmwQS
$ patch -p1 < inline.patch
$ rm inline.patch
```

The Tool

- Generate new recipes for configuring and compiling the project

```
$ ./autogen.sh
```

- Configure the project selection where the files will be installed
  - PS: since we don't have root access to the machine, we will create our own target path: ~/projects/fs

```
$ ./configure --prefix=${HOME}/projects/inst
```

Tip: check configure's output for sectionprof

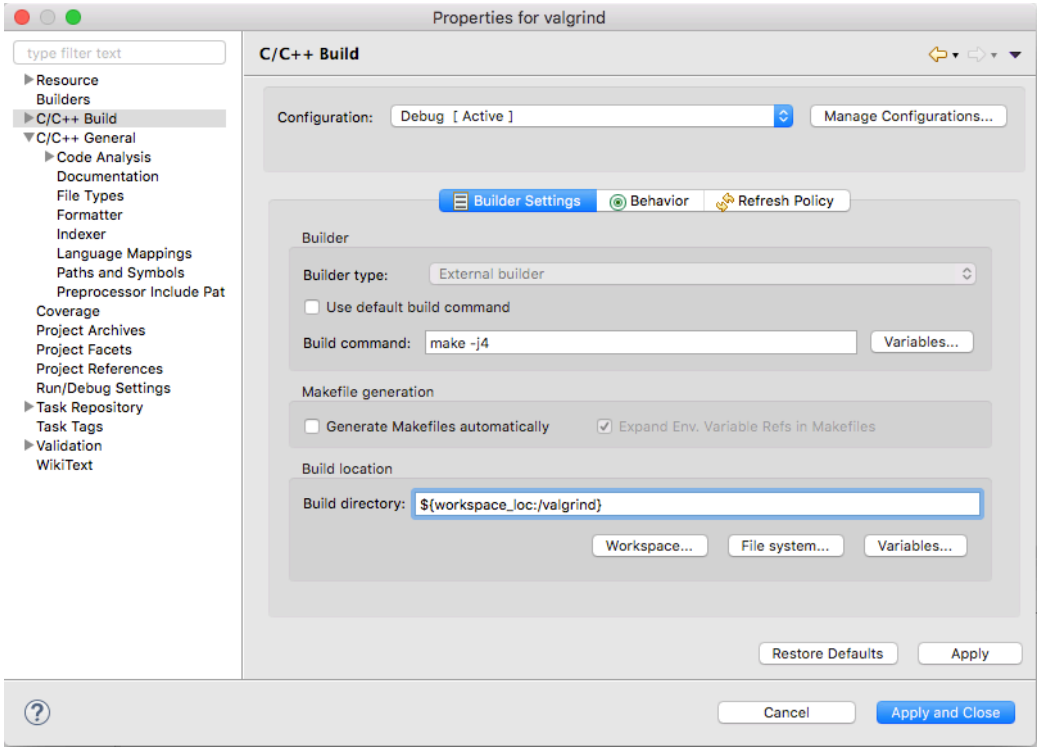
- Fnally, configure the PATH so we can find the valgrind executable binary after compilation

```
$ export PATH=${HOME}/projects/inst/bin:$PATH
```

Warning: reconfigure path if executed in another terminal

The Tool

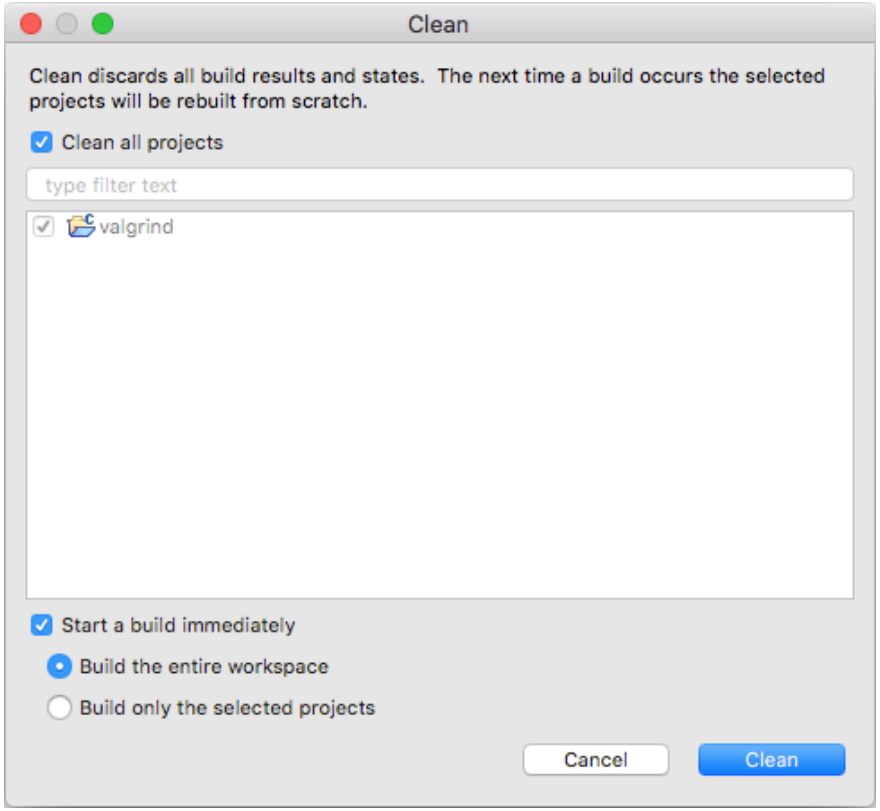
- Configure build options
  - Right-click in the project and select properties
  - In "C/C++ Build"



- Unmark "Use default build command" and use "make -j4"
- Disable "Generate Makefiles automatically"
- Remove "/Debug" from "Build directory"

The Tool

- Build the project
  - Select "Project" menu and uncheck "Build Automatically"
  - Select "Project" menu and choose "clean"

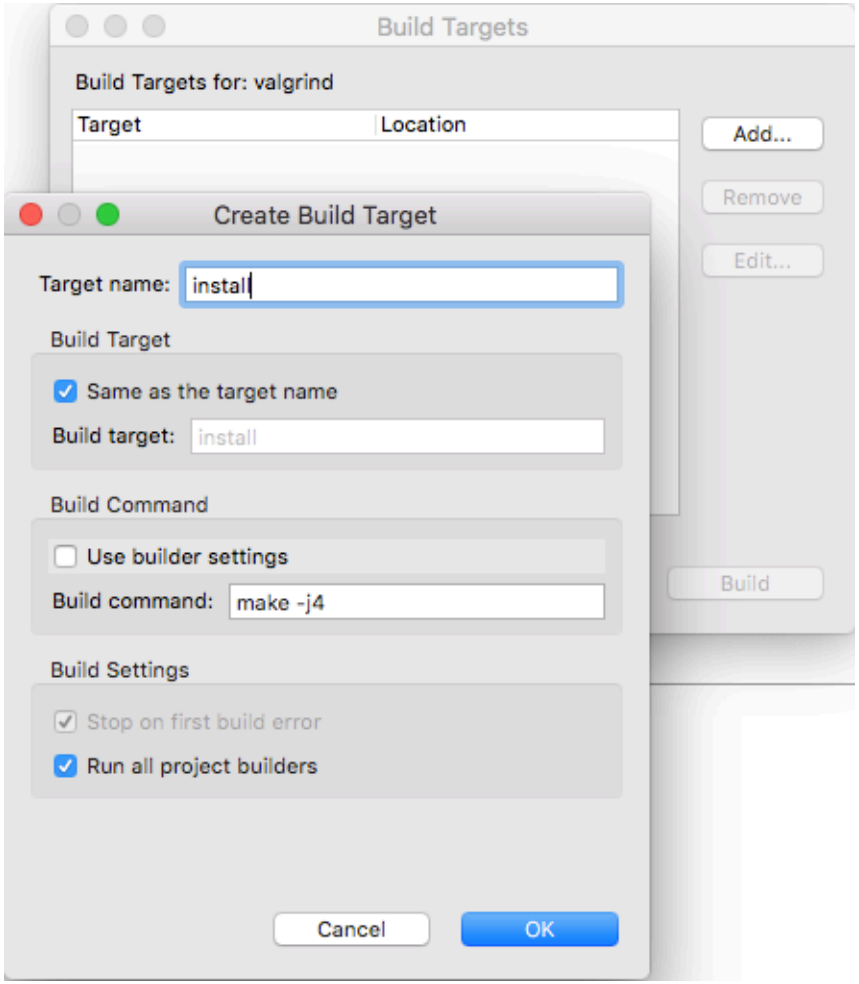


- Ensure that "Start a build immediately" is checked and click the "Clean" button to build the whole project

Tip: we will use control+b (PC) or cmd+b (MAC) for further builds

The Tool

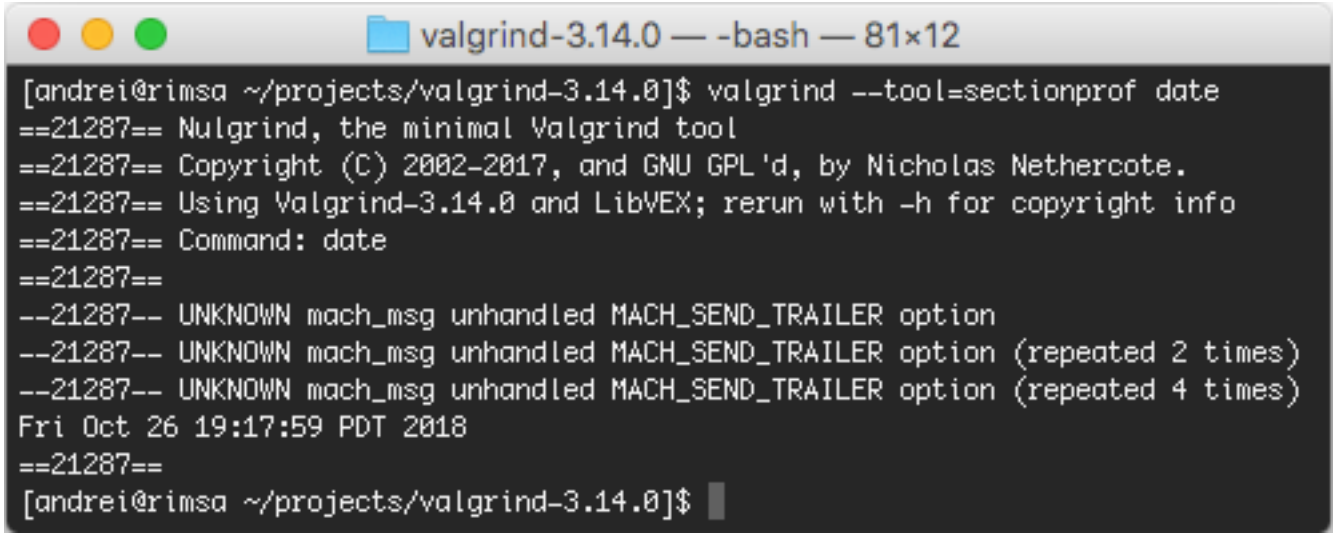
- Configure installing options
  - Project (menu) > Build Targets > Build
  - Select "Add"
    - Target name: "install"
    - Uncheck: "Use builder settings"
    - Build command: "make -j4 install"
  - Select installed target and click "Build"



Tip: we will use F9 for further installs

The Tool

- Test our tool: `valgrind –tool=sectionprof date`



Compilers Laboratory

DEPARTMENT OF COMPUTER SCIENCE
FEDERAL UNIVERSITY OF MINAS GERAIS, BRAZIL

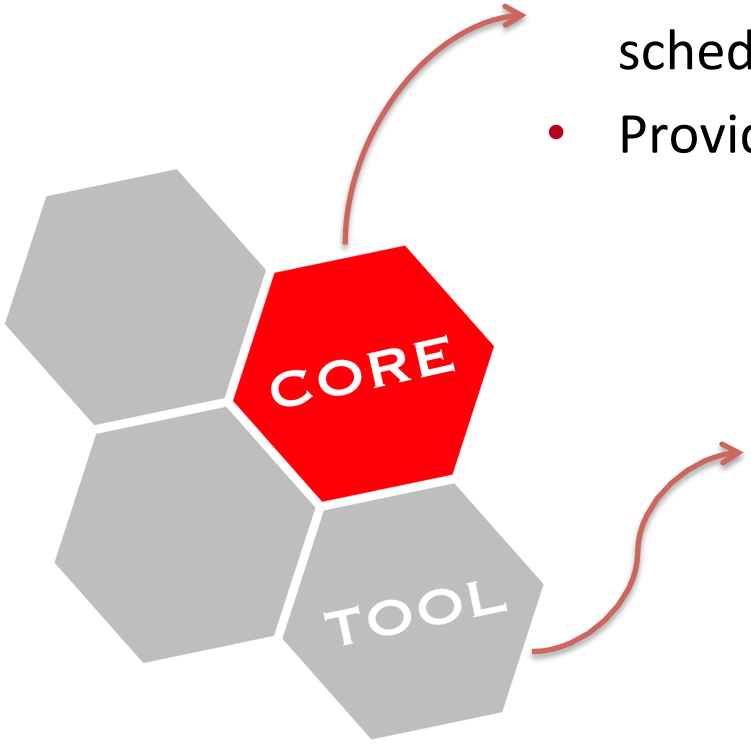
Understanding Valgrind

LAC

Valgrind

- The Valgrind architecture is divided in two parts
  - Low-level infrastructure to support instrumentation
  - JIT compiler, memory manager, signal handler, scheduler (for pthreads)
  - Provides useful services

Info: the core is responsible for all the heavy work



  - Responsible for the instrumentation
  - Requests services from the core
  - Is notified of events from the core

The Tool

- A tool must define, at lease, these for functions
  - `pre_clo_init()`: useful for most of the initialisations (e.g.: creating data structures)
  - `post_clo_init()`: useful for perform initialisations after the processing of command line options
  - `instrumentation()`: allows the instrumentation of the VEX IR (RISC-like intermediate language) – useful to perform callbacks when interesting things happen
  - `fini()`: useful for formatting final results (e.g.: summary of collected informations) and/or writing log files

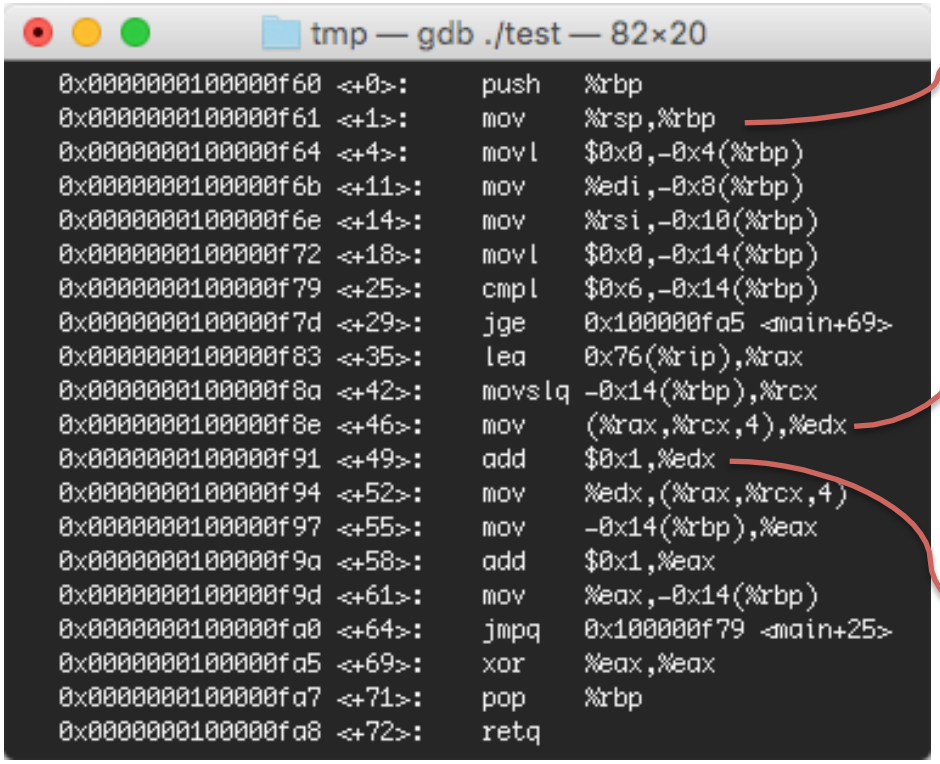
Info: use `VG_DETERMINE_INTERFACE_VERSION` to set `pre_clo_init` and `VG_(basic_tool_funcs)` for the other remaining functions

Tip: the function names can vary

The VEX IR

Tip: you can use ppIRSB() to print a super block

- Assembly instructions (for the various supported architectures) are converted to a 3-address like VEX instructions




```
--- IMark(0xF61, 3, 0) ---
PUT(56) = t10
PUT(184) = 0x100000F64:I64
```

```
--- IMark(0xF8E, 3, 0) ---
t22 = Shl64(t18,0x2:I8)
t20 = Add64(0x100001000:I64,t22)
t25 = LDle:I32(t20)
t62 = 32Uto64(t25)
t24 = t62
```

```
--- IMark(0xF91, 3, 0) ---
t63 = 64to32(t24)
t26 = t63
t3 = Add32(t26,0x1:I32)
t64 = 32Uto64(t3)
t30 = t64
PUT(32) = t30
PUT(184) = 0x100000F94:I64
```

Info: Available in `VEX/pub/libvex_ir.h`

VEX Instruction

Tip: you can use `ppIRStmt()` to print a statement

- A VEX instruction (IR statement) can be one of the folllowing:

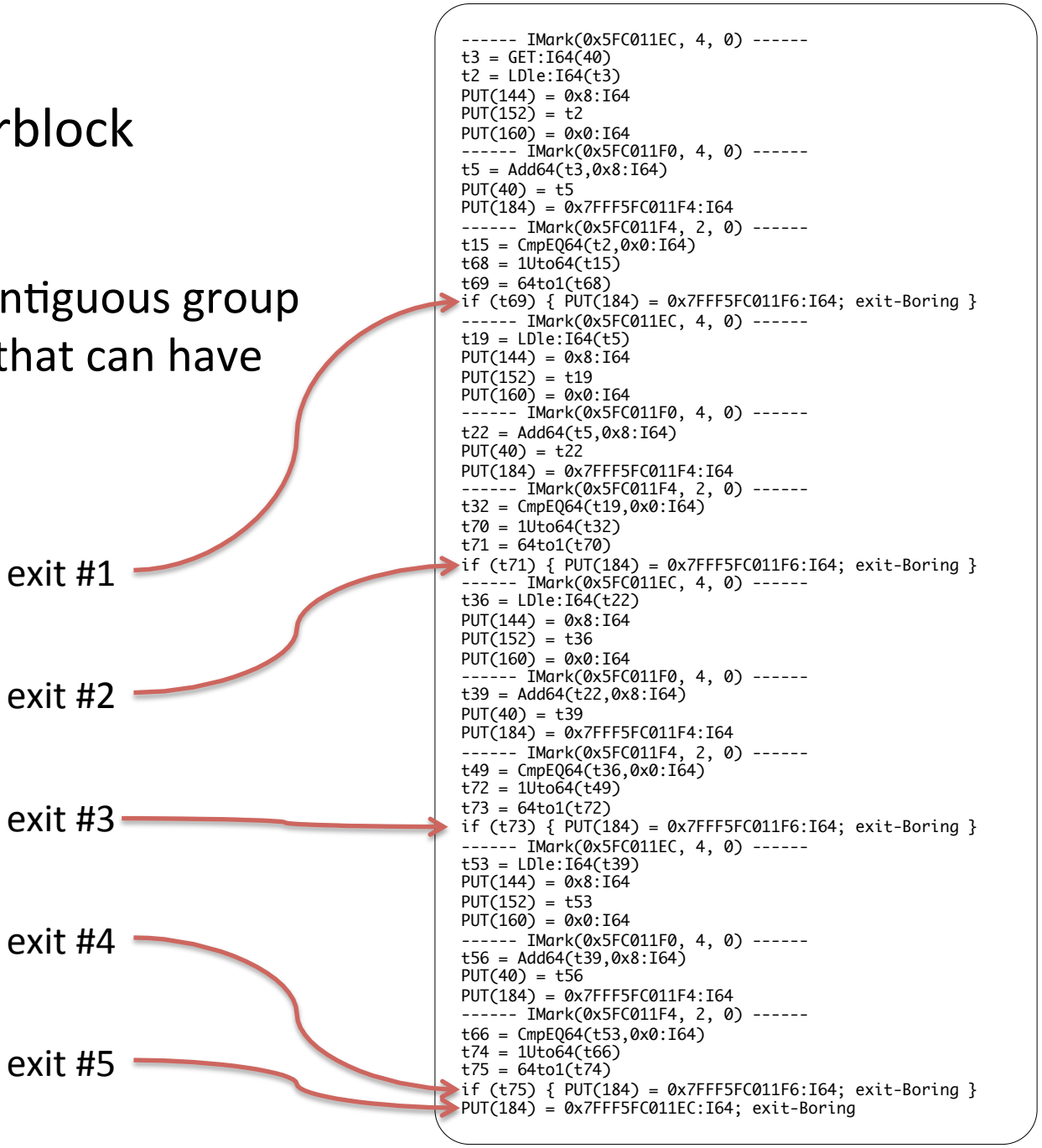
`IRStmt`

- `NoOp:` A no-operation instruction
- `IMark:` Mark the start of the statements that represents a machine instruction (**Meta**)
- `AbiHint:` An ABI hint, which reflect the plataform's ABI (**Meta**)
- `Put:` Write a guest register at a fixed offset
- `PutI:` Write a guest register at a non-fixed offset
- `WrTmp:` Assign a value to a temporary (virtual register)
- `Store:` Write a value to memory (not conditionally)
- `StoreG:` Guarded store (conditionally write to memory)
- `LoadG:` Guarded load (conditionaly read from memory)
- `CAS:` Atomic compare and swap operation
- `LLSC:` Either load-linked or store conditional
- `Dirty:` Call (possibly conditionally) a C function (callback)
- `MBE:` Memory bus event (fence or acquire/release lock)
- `Exit:` Condionally exit from the middle of a super block

VEX Superblock

- A superblock is a contiguous group of VEX instructions that can have multiple exit points

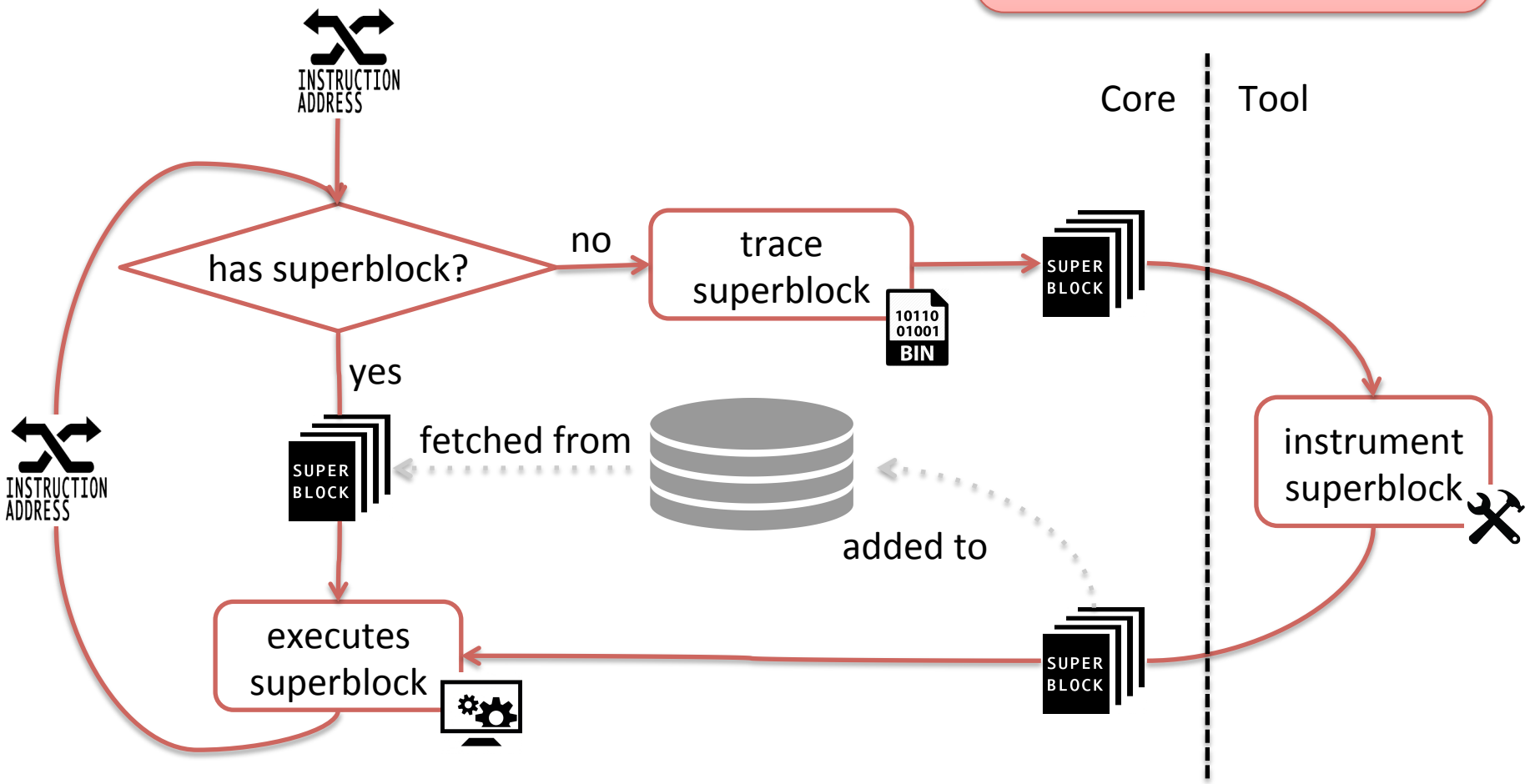
Warning: a compiler's basic block has only one exit point



Valgrind instrumentation

- Valgrind execution workflow

Warning: a super block can be retranslated as well, but let's ignore it for now



Building the tool: sectionprof

sectionprof

- Let's build a tool that performs some profiling within a code section

- The tool must support the following operations
  1) Select a code section to be profiled (start and end addresses) – it can be profiled multiple times during a single execution
  2) Count number of instructions **existing** and **executed** instructions according to the following categories:
     - Arithmetic and logic operations (add, sub, and, or, ...)
     - Memory operations (load/store)
  3) Max amount of dynamically allocated memory

Part I

- Our section will be comprised within an **start** and **end** addresses
- Our tool will receive this addresses via command-line parameters:
  - `--section-start=0xaddr`
  - `--section-end=0xaddr`
- Some command line parsing rules
  - Start and end addresses are not mandatory
  - End address can only exit if start address is set
  - Start and end address cannot be identical

Part I

- Add a global structure to hold both addresses and a function to set the default values

```
struct {
    Addr section_start, section_end;
} sp_clo;

static
void sp_set_defaults() {
    sp_clo.section_start = 0;
    sp_clo.section_end = 0;
}
```

- Add a call to sp_set_defaults() in sp_pre_clo_init()

```
static
void sp_pre_clo_init(void) {
  // ...

  sp_set_defaults();
}
```

Part I

- Request a service from the core to parse command line options

```
static
void sp_pre_clo_init(void) {
    // ...
    VG_(needs_command_line_options)(sp_process_cmd_line_option,
             sp_print_usage, sp_print_debug_usage);
    // ...
}
```

Tip: copy these 3 functions from lackley for a better starting point

  - Add some useful includes

```
#include "pub_tool_libcassert.h"
#include "pub_tool_libcbase.h"
#include "pub_tool_libcprint.h"
#include "pub_tool_machine.h"
#include "pub_tool_options.h"
```

Part I

- Implement these 3 functions

How to show the command line options?

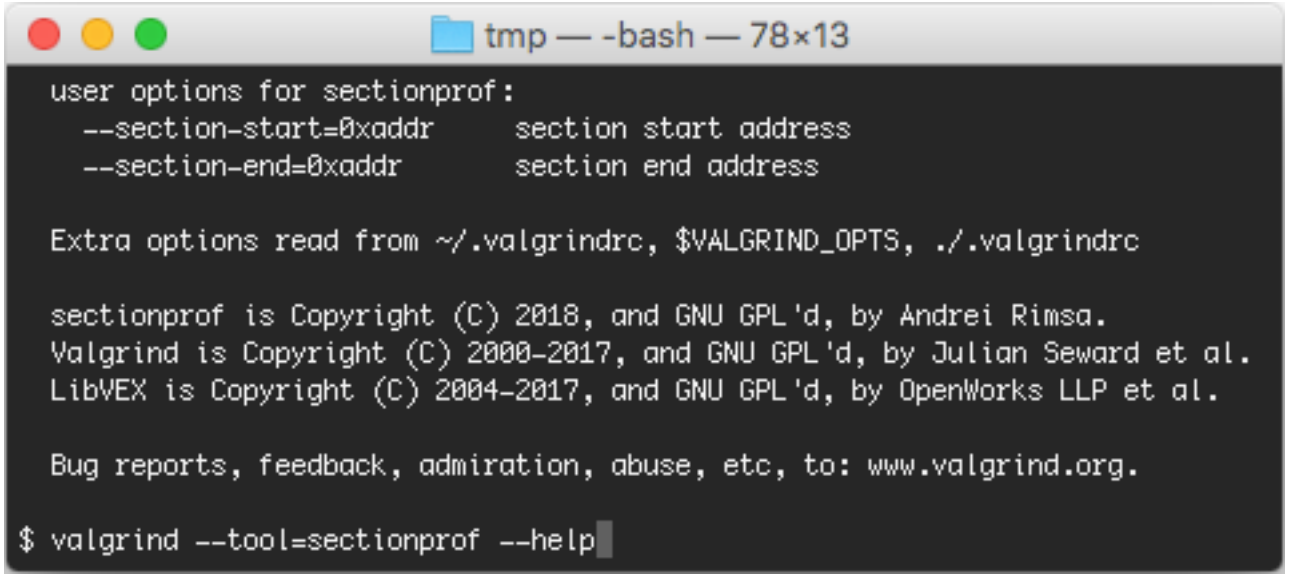
```
static
Bool sp_process_cmd_line_option(const HChar* arg) {
     if VG_BHEX_CLO(arg, "--section-start", sp_clo.section_start, 0, -1) {}
     else if VG_BHEX_CLO(arg, "--section-end", sp_clo.section_end, 0, -1) {}
     else
          return False;

     return True;
}
```

```
     static
     void sp_print_usage(void) {
          VG_(printf)(
               "    --section-start=0xaddr    section start address\n"
               "    --section-end=0xaddr      section end address\n"
          );
     }
```

```
static
void sp_print_debug_usage(void) {
     VG_(printf)(
          "    (none)\n"
     );
}
```

Part I

- Use `--help` with the tool to show the usage



How to ensure section address restrictions?

Part I

- Add the restriction in `sp_post_clo_init()` because it is called after processing command line options
  - First approach

Which one is better?

```
static
void sp_post_clo_init(void) {
    tl_assert(sp_clo.section_start || !sp_clo.section_end);
    tl_assert(!sp_clo.section_start || sp_clo.section_start != sp_clo.section_end);
}
```

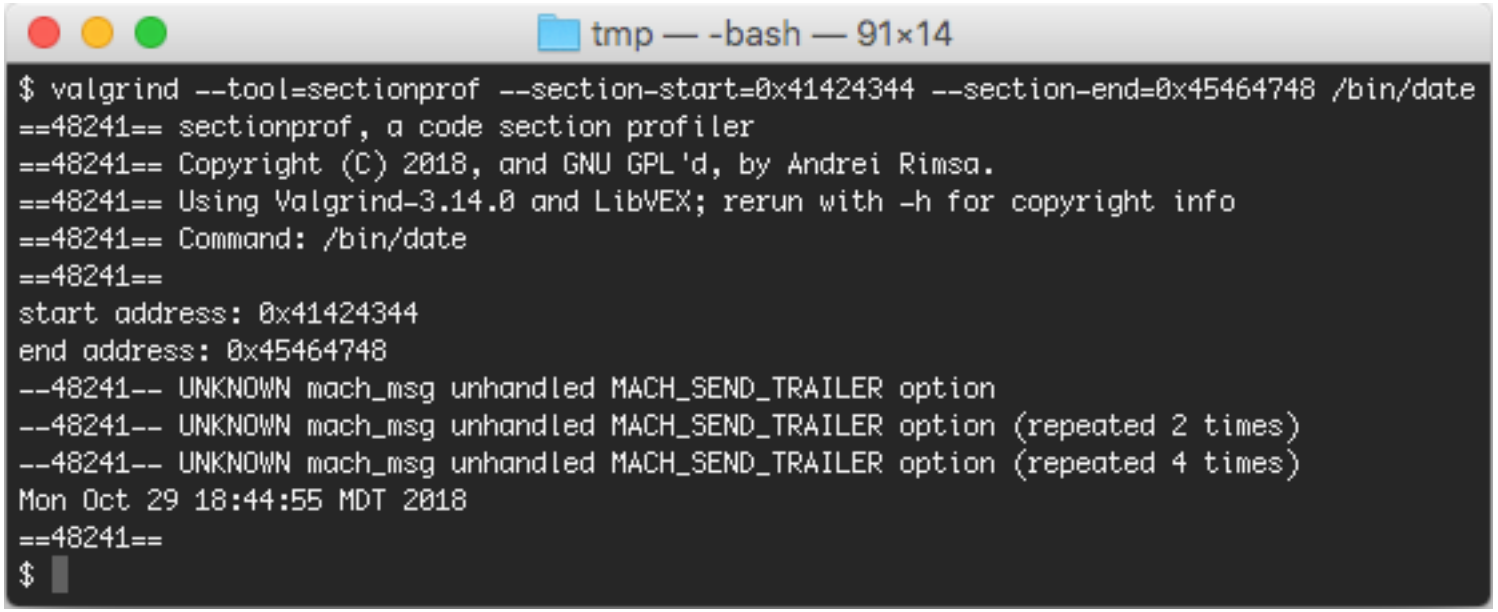
  - Second approach

Tip: print the addresses to test

```
static
void sp_post_clo_init(void) {
    if (sp_clo.section_end && !sp_clo.section_start)
        VG_(tool_panic)("no start address for end address");
    else if (sp_clo.section_start && sp_clo.section_start == sp_clo.section_end)
        VG_(tool_panic)("start and end address must be different");
}
```

Part I

- Let's test it: `valgrind --tool=sectionprof --section-start=0x41424344 --section-end=0x45464748 /bin/date`



Part II

- So far, our instrumentation only returns the same superblock received by the core

```
static
IRSB* sp_instrument(VgCallbackClosure* closure,
                    IRSB* bb,
                    const VexGuestLayout* layout,
                    const VexGuestExtents* vge,
                    const VexArchInfo* archinfo_host,
                    IRType gWordTy, IRType hWordTy) {
    return bb;
}
```

Warning: we can't change the superblock, but we can create our own

Tip: see how lackley tool does the instrumentation

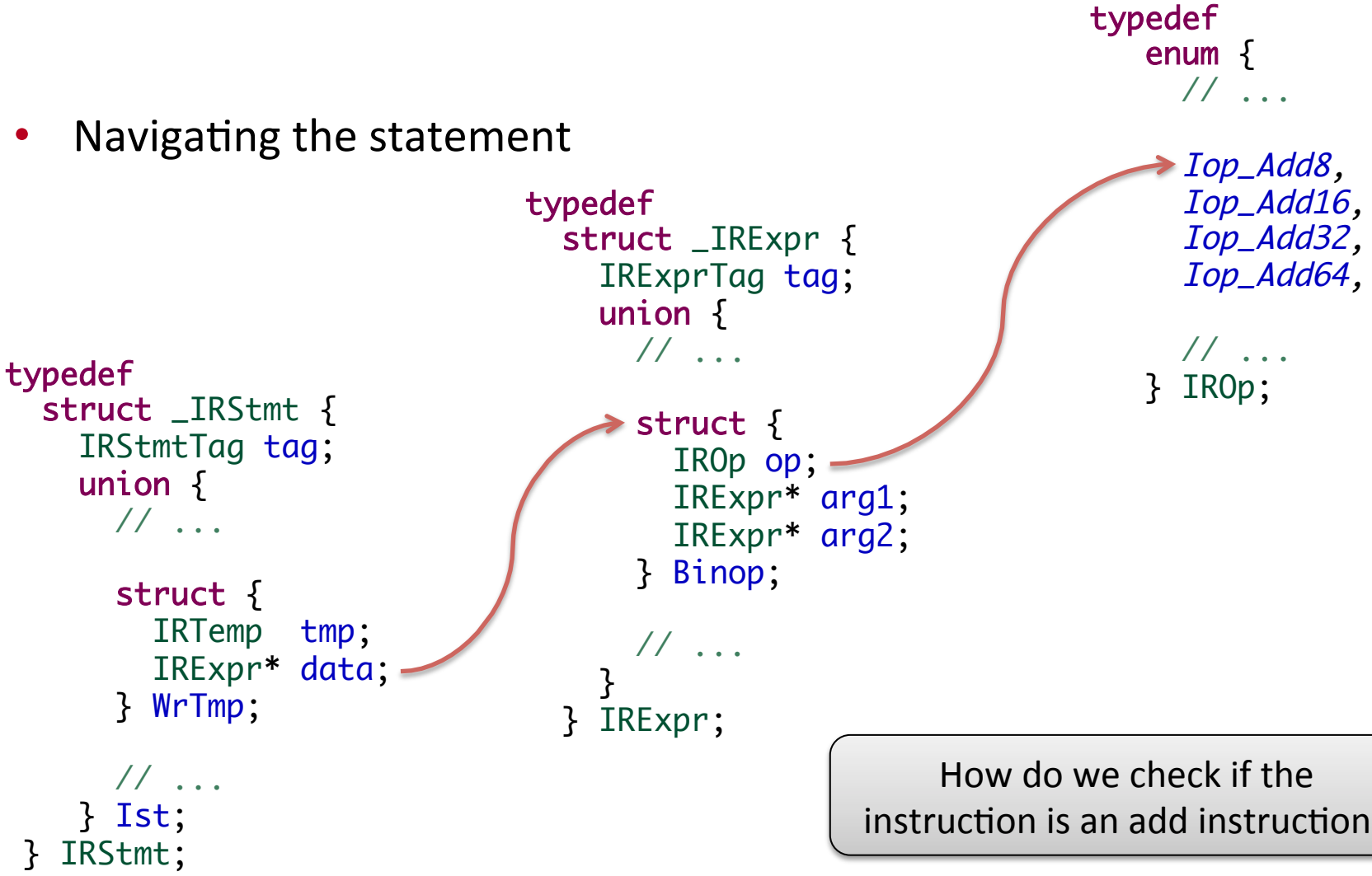
```
static
IRSB* sp_instrument(VgCallbackClosure* closure, IRSB* sbIn,
         const VexGuestLayout* layout, const VexGuestExtents* vge,
         const VexArchInfo* archinfo_host, IRType gWordTy, IRType hWordTy) {
     Int i;
     IRSB* sbOut;

     // We don't currently support this case.
     if (gWordTy != hWordTy)
         VG_(tool_panic)("host/guest word size mismatch");

     // Set up SB
     sbOut = deepCopyIRSBExceptStmts(sbIn);

     // Copy verbatim any IR preamble preceding the first IMark
     i = 0;
     while (i < sbIn->stmts_used && sbIn->stmts[i]->tag != Ist_IMark) {
         addStmtToIRSB(sbOut, sbIn->stmts[i]);
         i++;
     }

     // Copy instructions to new superblock
     for (/*use current i*/; i < sbIn->stmts_used; i++) {
         IRStmt* st = sbIn->stmts[i];
         if (!st || st->tag == Ist_NoOp) continue;

         addStmtToIRSB(sbOut, st);
     }

     return sbOut;
}
```

Info: notice that we renamed bb for sbIn

We need to count add instruction, but where are they?

Part II

- Navigating the statement



Part II

- Let's do it for BinOp, since it will be more useful for the others stats
  - Inside instructions for and before *addStmtToIRSB*, add a switch statement for WrTmp instructions with BinOp expressions

```
switch (st->tag) {
  case Ist_WrTmp:
    switch (st->Ist.WrTmp.data->tag) {
      case Iex_Binop:
        handleOp(sbOut, st->Ist.WrTmp.data->Iex.Binop.op);
        break;
      default:
        break;
    }

    break;
  default:
    break;
}
```

Should we account for unary operations?

Part II

- There are also Qop (Quaternary operation), Triop (Ternary operations) and Unop (Unary operation)

```
case Ist_WrTmp:
     switch (st->Ist.WrTmp.data->tag) {
          case Iex_Qop:
               handleOp(sbOut, st->Ist.WrTmp.data->Iex.Qop.details->op);
               break;
          case Iex_Triop:
               handleOp(sbOut, st->Ist.WrTmp.data->Iex.Triop.details->op);
               break;
          case Iex_Binop:
               handleOp(sbOut, st->Ist.WrTmp.data->Iex.Binop.op);
               break;
          case Iex_Unop:
               handleOp(sbOut, st->Ist.WrTmp.data->Iex.Unop.op);
               break;
          default:
               break;
     }

     break;
```

Part II

- Before implementing handleOp let's create the structure to hold the statistics

```
typedef enum {
    AddOp, SubOp, MulOp, DivOp, ModOp,
    NotOp, AndOp, OrOp, XorOp, ShiftOp,
    LoadOp, StoreOp,
    OthersOp
} OpKind;

#define N_OPERATIONS (OthersOp + 1)
```

```
typedef struct {
    ULong exist;
    ULong executed;
} op_stats;

struct {
    op_stats operations[N_OPERATIONS];
    op_stats total;
} sp_stats;
```

Am I missing any basic operation?

- And initialize the stats with zeros at sp_set_defaults

```
static
void sp_set_defaults() {
    // ...
    VG_(memset)(&sp_stats, 0, sizeof(sp_stats));
}
```

Info: replace every glibc function by its counterpart using the VG_() macro

Part II

```
static
void handleOp(IRSB* sbOut, IROp op) {
     OpKind kind;
     switch (op) {
          case Iop_Add8:
          case Iop_Add16:
          case Iop_Add32:
          case Iop_Add64:
          case Iop_AddF32:
          case Iop_AddF64:
          case Iop_AddF64r32:
          case Iop_AddF128:
          case Iop_AddD64:
          case Iop_AddD128:
               kind = AddOp;
               break;
          default:
               kind = OthersOp;
               break;
     }

     sp_stats.operations[kind].exist++;
     sp_stats.total.exist++;
}
```

- Let's first count the number of add operations

And how do we count the number of times it was executed?

Part II

- Now, we need to add an instruction to the superblock that will perform a callback to our code
  - Add the callback function

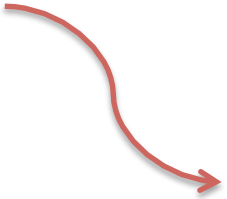
```
static VG_REGPARM(1)
void trace_operation(OpKind kind) {
    sp_stats.operations[kind].executed++;
    sp_stats.total.executed++;
}
```

  - And add the callback instruction to the superblock in handleOp

```
static
void handleOp(IRSB* sbOut, IROp op) {
    IRExpr** argv;
    IRDirty* di;

    // ...

    argv = mkIRExprVec_1(mkIRExpr_HWord(kind));
    di   = unsafeIRDirty_0_N(/*regparms*/1, "trace_operation",
                VG_(fnptr_to_fnentry)(trace_operation), argv);
    addStmtToIRSB(sbOut, IRStmt_Dirty(di));
}
```

Part II

```
IRSB {
   t0:I64   t1:I64   t2:I64   t3:I64   t4:I64   t5:I64
   t6:I8    t7:I64   t8:I64   t9:I64   t10:I64
   ------ IMark(0x1D354C, 3, 0) ------
   t0 = GET:I64(72)
   t6 = GET:I8(64)
   STle(t0) = t6
   ------ IMark(0x1D354F, 5, 0) ------
   PUT(16) = 0x1:I64
   PUT(184) = 0x1001D3554:I64
   ------ IMark(0x1D3554, 1, 0) ------
   t2 = GET:I64(48)
   t1 = LDle:I64(t2)
   t8 = Add64(t2,0x8:I64)
   PUT(48) = t8
   PUT(56) = t1
   PUT(184) = 0x1001D3555:I64
   ------ IMark(0x1D3555, 1, 0) ------
   t4 = LDle:I64(t8)
   t5 = Add64(t8,0x8:I64)
   PUT(48) = t5
   t9 = Sub64(t5,0x80:I64)
   ====== AbiHint(t9, 128, t4) ======
   PUT(184) = t4; exit-Return
}
```



```
IRSB {
   t0:I64   t1:I64   t2:I64   t3:I64   t4:I64   t5:I64
   t6:I8    t7:I64   t8:I64   t9:I64   t10:I64

   ------ IMark(0x1D354C, 3, 0) ------
   t0 = GET:I64(72)
   t6 = GET:I8(64)
   STle(t0) = t6
   ------ IMark(0x1D354F, 5, 0) ------
   PUT(16) = 0x1:I64
   PUT(184) = 0x1001D3554:I64
   ------ IMark(0x1D3554, 1, 0) ------
   t2 = GET:I64(48)
   t1 = LDle:I64(t2)
   DIRTY 1:I1 ::: trace_operation[rp=1]{0x2580015c0}(0x0:I64)
   t8 = Add64(t2,0x8:I64)
   PUT(48) = t8
   PUT(56) = t1
   PUT(184) = 0x1001D3555:I64
   ------ IMark(0x1D3555, 1, 0) ------
   t4 = LDle:I64(t8)
   DIRTY 1:I1 ::: trace_operation[rp=1]{0x2580015c0}(0x0:I64)
   t5 = Add64(t8,0x8:I64)
   PUT(48) = t5
   DIRTY 1:I1 ::: trace_operation[rp=1]{0x2580015c0}(0xA:I64)
   t9 = Sub64(t5,0x80:I64)
   ====== AbiHint(t9, 128, t4) ======
   PUT(184) = t4; exit-Return
}
```

Part II

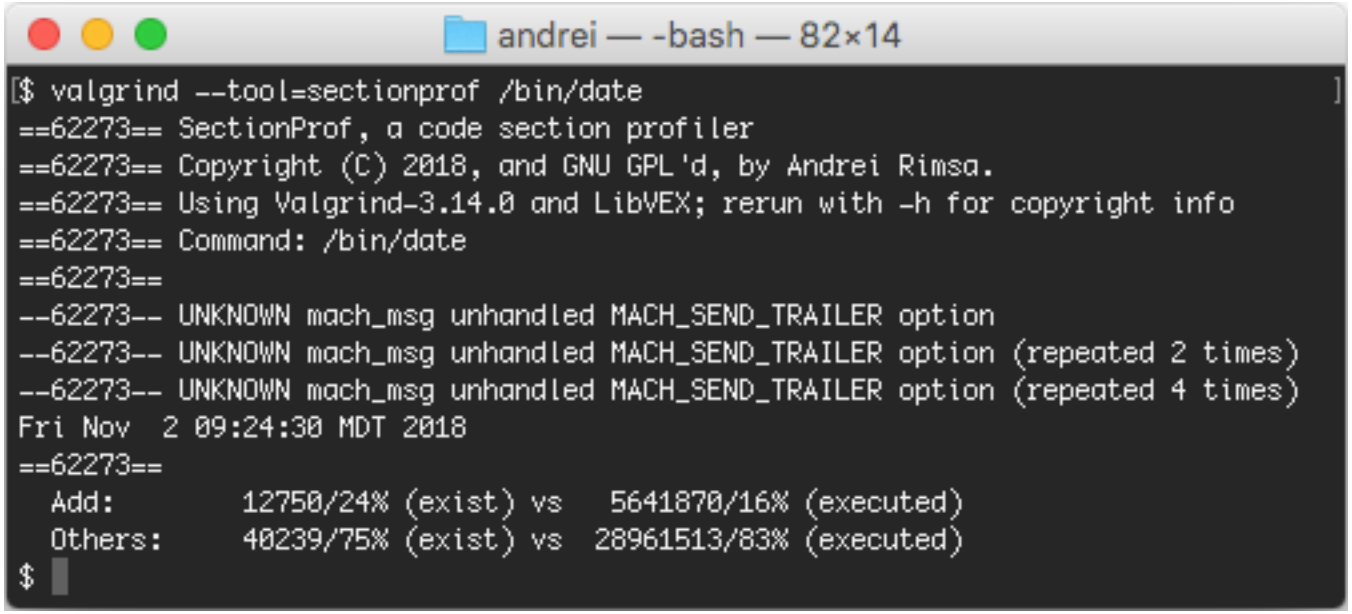
- Show the stats in the end

```
static
void sp_fini(Int exitcode) {
  VG_(printf)("  Add:    %9llu/%02d%% (exist) vs %9llu/%02d%% (executed)\n",
    sp_stats.operations[AddOp].exist,
    (Int) (sp_stats.operations[AddOp].exist * 100 / sp_stats.total.exist),
    sp_stats.operations[AddOp].executed,
    (Int) (sp_stats.operations[AddOp].executed * 100 / sp_stats.total.executed));
  VG_(printf)("  Others: %9llu/%02d%% (exist) vs %9llu/%02d%% (executed)\n",
    sp_stats.operations[OthersOp].exist,
    (Int) (sp_stats.operations[OthersOp].exist * 100 / sp_stats.total.exist),
    sp_stats.operations[OthersOp].executed,
    (Int) (sp_stats.operations[OthersOp].executed * 100 / sp_stats.total.executed));
}
```

Can you take a guess how many add instructions are executed compared to the others?

Part II

- Final stats for `/bin/date`



How do we account just for a code section now?

Part II

- We need markers for instrumentation vs execution time
  - Add them to sp_stats structure

```
struct {
    struct {
        Bool existing;
        Bool executing;
    } enabled;

    // ...
} sp_stats;
```

  - And initialize their values in

Info: enable instrumentation for the whole program if section is not defined

```
static
void sp_post_clo_init(void) {
    // ...

    sp_stats.enabled.existing =
            sp_stats.enabled.executing =
                    (sp_clo.section_start == 0 && sp_clo.section_end == 0);
}
```

Part II

- Use instruction mark (IMark) to check for section start/end
  - Add two callback functions to enable/disable execution

```
static VG_REGPARM(0)
void enable_execution(void) {
    sp_stats.enabled.executing = True;
}
```

```
static VG_REGPARM(0)
void disable_execution(void) {
    sp_stats.enabled.executing = False;
}
```

  - Add IMark case for an instruction and check if addresses match

```
case Ist_IMark:
    if (sp_clo.section_start && st->Ist.IMark.addr == sp_clo.section_start) {
        sp_stats.enabled.existing = True;
        addStmtToIRSB(sbOut, IRStmt_Dirty(unsafeIRDirty_0_N(0, "enable_execution",
                      VG_(fnptr_to_fnentry)(enable_execution), mkIRExprVec_0())));
    } else if (sp_clo.section_end && st->Ist.IMark.addr == sp_clo.section_end) {
        sp_stats.enabled.existing = False;
        addStmtToIRSB(sbOut, IRStmt_Dirty(unsafeIRDirty_0_N(0, "disable_execution",
                      VG_(fnptr_to_fnentry)(disable_execution), mkIRExprVec_0())));
    }

    break;
```

Part II

- Now let's account for the section
  - At instrumentation time: wrap the exist++ with the marker test

```
static
void handleOp(IRSB* sbOut, IROp op) {
    // ...

    if (sp_stats.enabled.existing) {
        sp_stats.operations[kind].exist++;
        sp_stats.total.exist++;
    }

    // ...
}
```

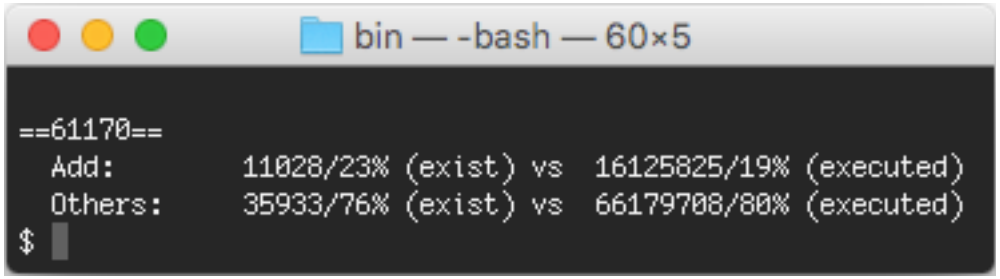
  - At execution time: wrap the executed++ with the marker test

```
static VG_REGPARM(1)
void trace_operation(OpKind kind) {
    if (sp_stats.enabled.executing) {
        sp_stats.operations[kind].executed++;
        sp_stats.total.executed++;
    }
}
```

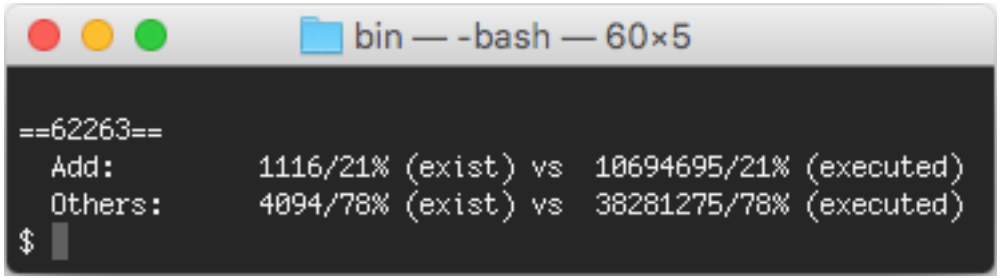
Part II

Info: `is.S.x` is from NPB benchmark

- Finally, test it
  - For the whole program: `valgrind --tool=sectionprof ./is.S.x`



  - Inside main only: `valgrind --tool=sectionprof --section-start=0x100001180 --section-end=0x10000166d ./is.S.x`



Part II

- We won't implement load/store, but some tips on what to instrument
  - Instruction: `stmt->Ist.WrTmp`
    - Load expression: `expr->Iex.Load`
  - Instruction: `stmt->Ist.LoadG`
  - Instruction: `stmt->Ist.LoadG`
  - Instruction: `stmt->Ist.Store`
  - Instruction: `stmt->Ist.StoreG`
  - Instruction: `stmt->Ist.Dirty`
  - Instruction: `stmt->Ist.CAS`
  - Instruction: `stmt->Ist.LLSC`

Tip: check lackley for reference, try to understand their event system for instrumentation

Part III

- In order to track memory management functions, we will need to hook them via shared libraries

Which functions we need to hook?

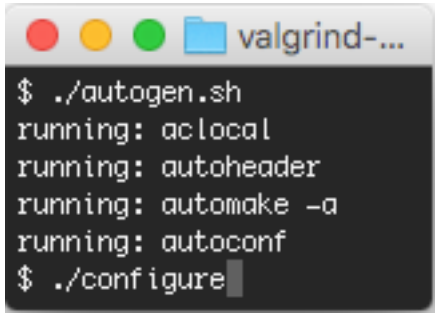
```
#----------------------------------------------------------------------
# vgpreload_sectionprof-<platform>.so
#----------------------------------------------------------------------

noinst_PROGRAMS += vgpreload_sectionprof-@VGCONF_ARCH_PRI@-@VGCONF_OS@.so
if VGCONF_HAVE_PLATFORM_SEC
noinst_PROGRAMS += vgpreload_sectionprof-@VGCONF_ARCH_SEC@-@VGCONF_OS@.so
endif

if VGCONF_OS_IS_DARWIN
noinst_DSYMS = $(noinst_PROGRAMS)
endif

vgpreload_sectionprof_@VGCONF_ARCH_PRI@_@VGCONF_OS@_so_SOURCES     =
vgpreload_sectionprof_@VGCONF_ARCH_PRI@_@VGCONF_OS@_so_CPPFLAGS    = \
        $(AM_CPPFLAGS_@VGCONF_PLATFORM_PRI_CAPS@)
vgpreload_sectionprof_@VGCONF_ARCH_PRI@_@VGCONF_OS@_so_CFLAGS      = \
        $(AM_CFLAGS_PSO_@VGCONF_PLATFORM_PRI_CAPS@)
vgpreload_sectionprof_@VGCONF_ARCH_PRI@_@VGCONF_OS@_so_DEPENDENCIES = \
        $(LIBREPLACEMALLOC_@VGCONF_PLATFORM_PRI_CAPS@)
vgpreload_sectionprof_@VGCONF_ARCH_PRI@_@VGCONF_OS@_so_LDFLAGS     = \
        $(PRELOAD_LDFLAGS_@VGCONF_PLATFORM_PRI_CAPS@) \
        $(LIBREPLACEMALLOC_LDFLAGS_@VGCONF_PLATFORM_PRI_CAPS@)

if VGCONF_HAVE_PLATFORM_SEC
vgpreload_sectionprof_@VGCONF_ARCH_SEC@_@VGCONF_OS@_so_SOURCES     =
vgpreload_sectionprof_@VGCONF_ARCH_SEC@_@VGCONF_OS@_so_CPPFLAGS    = \
        $(AM_CPPFLAGS_@VGCONF_PLATFORM_SEC_CAPS@)
vgpreload_sectionprof_@VGCONF_ARCH_SEC@_@VGCONF_OS@_so_CFLAGS      = \
        $(AM_CFLAGS_PSO_@VGCONF_PLATFORM_SEC_CAPS@)
vgpreload_sectionprof_@VGCONF_ARCH_SEC@_@VGCONF_OS@_so_DEPENDENCIES = \
        $(LIBREPLACEMALLOC_@VGCONF_PLATFORM_SEC_CAPS@)
vgpreload_sectionprof_@VGCONF_ARCH_SEC@_@VGCONF_OS@_so_LDFLAGS     = \
        $(PRELOAD_LDFLAGS_@VGCONF_PLATFORM_SEC_CAPS@) \
        $(LIBREPLACEMALLOC_LDFLAGS_@VGCONF_PLATFORM_SEC_CAPS@)
endif
```

Info: update the Makefile.am

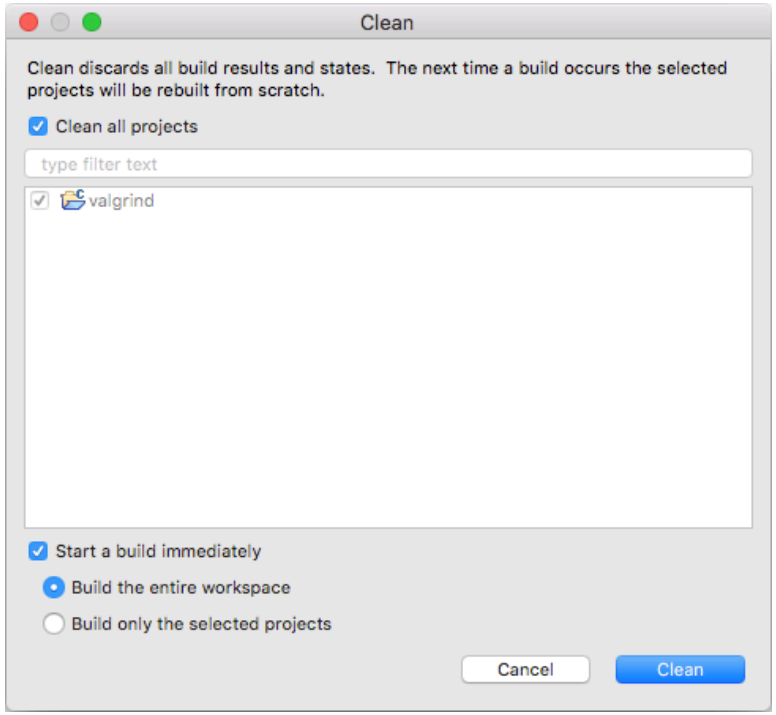
Tip: get it from massif's Makefile.am

Part III

- Rebuild the whole project from the start, ./autogen.sh and configure



- Then, clean the project in eclipse and mark it to rebuild



Part III

- We will track the evolution of dynamic memory footprint by mantining the amount of memory allocated so far (current) and the maximum value of allocated memory (max)

```
struct {
    // ...

    struct {
        Int current;
        Int max;
    } mem;

    // ...
} sp_stats;
```

And how do we track it?

Part III

- Request the core to replace memory management calls

```
static
void sp_pre_clo_init(void) {
    // ...

    VG_(needs_malloc_replacement)(
             sp_malloc,
             sp___builtin_new,
             sp___builtin_vec_new,
             sp_memalign,
             sp_calloc,
             sp_free,
             sp___builtin_delete,
             sp___builtin_vec_delete,
             sp_realloc,
             sp_malloc_usable_size,
             0);

   // ...
}
```

Tip: check `massif` tool for reference

Part III

- Before replacing the calls, let's create our own allocator

```
static
void* alloc_block(SizeT req_szB, SizeT req_alignB, Bool is_zeroed) {
     SizeT actual_szB;
     void* p;

     if (((SSizeT) req_szB) < 0)
          return 0;

     // Allocate and zero if necessary.
     p = VG_(cli_malloc)(req_alignB, req_szB);
     if (!p)
          return 0;

     if (is_zeroed)
          VG_(memset)(p, 0, req_szB);

     actual_szB = VG_(cli_malloc_usable_size)(p);
     tl_assert(actual_szB >= req_szB);

     return p;
}
```

What about a deallocator?

Part III

- ... and our own deallocator

```
static
void dealloc_block(void* p) {
    VG_(cli_free)(p);
}
```

What about a reallocator?

Part III

- ... and our own reallocator

```
static
void* realloc_block(void* p_old, SizeT new_req_szB) {
    SizeT old_req_szB;
    void* p_new;

    if (!p_old)
        return alloc_block(new_req_szB, VG_(clo_alignment), /*is_zeroed*/False);

    p_new = alloc_block(new_req_szB, VG_(clo_alignment), /*is_zeroed*/False);
    if (p_new) {
        old_req_szB = VG_(cli_malloc_usable_size)(p_old);
        VG_(memcpy)(p_new, p_old,
                 (new_req_szB <= old_req_szB ? new_req_szB : old_req_szB));

        dealloc_block(p_old);
    }

    return p_new;
}
```

Part III

- Add a include file and a macro

```
#include "pub_tool_replacemalloc.h"
#define SP_UNUSED(arg) (void)arg;
```

- Now implement the replacement methods

```
static void* sp_malloc(ThreadId tid, SizeT szB) {
     SP_UNUSED(tid);
     return alloc_block(szB, VG_(clo_alignment), /*is_zeroed*/False);
}

static void* sp___builtin_new(ThreadId tid, SizeT szB) {
     SP_UNUSED(tid);
     return alloc_block(szB, VG_(clo_alignment), /*is_zeroed*/False);
}

static void* sp___builtin_vec_new(ThreadId tid, SizeT szB) {
     SP_UNUSED(tid);
     return alloc_block(szB, VG_(clo_alignment), /*is_zeroed*/False);
}

static void* sp_calloc(ThreadId tid, SizeT m, SizeT szB) {
     SP_UNUSED(tid);
     return alloc_block(m*szB, VG_(clo_alignment), /*is_zeroed*/True);
}
```

```
static void* sp_memalign(ThreadId tid, SizeT alignB, SizeT szB) {
    SP_UNUSED(tid);
    return alloc_block(szB, alignB, False);
}

static void sp_free(ThreadId tid, void* p) {
    SP_UNUSED(tid);
    dealloc_block(p);
}

static void sp___builtin_delete(ThreadId tid, void* p) {
    SP_UNUSED(tid);
    dealloc_block(p);
}

static void sp___builtin_vec_delete(ThreadId tid, void* p) {
    SP_UNUSED(tid);
    dealloc_block(p);
}

static void* sp_realloc(ThreadId tid, void* p_old, SizeT new_szB) {
    SP_UNUSED(tid);
    return realloc_block(p_old, new_szB);
}

static SizeT sp_malloc_usable_size(ThreadId tid, void* p) {
    SP_UNUSED(tid);
    return VG_(cli_malloc_usable_size)(p);
}
```

How to perform the instrument now?

Part III

- For memory allocation, update current and check max

```
static
void* alloc_block(SizeT req_szB, SizeT req_alignB, Bool is_zeroed) {
    // ...

    if (sp_stats.enabled.executing) {
        sp_stats.mem.current += req_szB;
        if (sp_stats.mem.current > sp_stats.mem.max)
            sp_stats.mem.max = sp_stats.mem.current;
    }

    // ...
}
```

Part III

- For memory deallocation, update current and ensure no negative value

```
static
void dealloc_block(void* p) {
    if (sp_stats.enabled.executing) {
        sp_stats.mem.current -= VG_(cli_malloc_usable_size)(p);
        if (sp_stats.mem.current < 0)
            sp_stats.mem.current = 0;
    }

    // ...
}
```

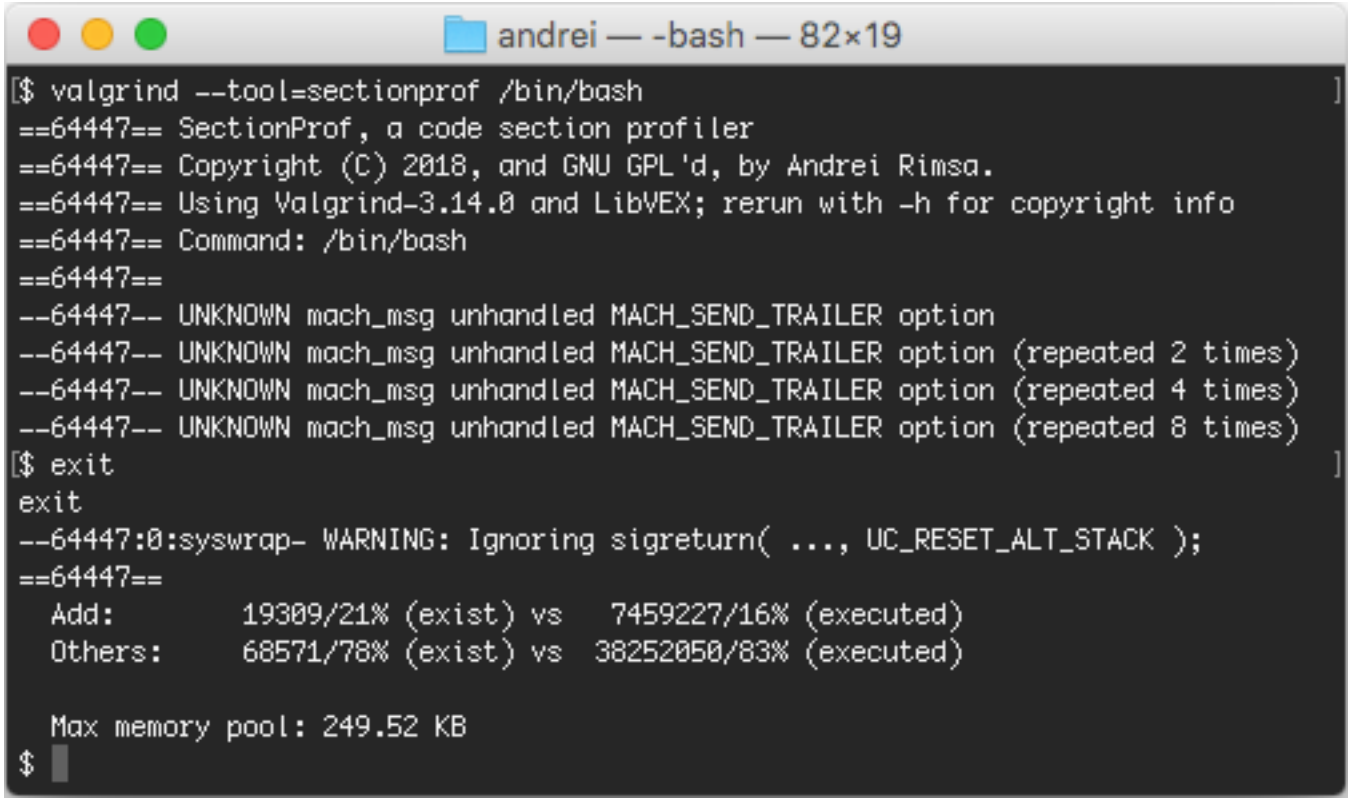
Part III

- Just print the max memory value in the end

```
void sp_fini(Int exitcode) {
    // ...

    if (sp_stats.mem.max < 1024)
        VG_(printf)("\n  Max memory pool: %d B\n", sp_stats.mem.max);
    else if (sp_stats.mem.max < (1024 * 1024))
        VG_(printf)("\n  Max memory pool: %.2f KB\n",
                ((float) sp_stats.mem.max) / 1024);
    else
        VG_(printf)("\n  Max memory pool: %.2f MB\n",
                ((float) sp_stats.mem.max) / (1024 * 1024));
}
```

Part III

- Test it: `valgrind --tool=sectionprof /bin/bash`



Part III

andrei — -bash — 81×30

```
$ valgrind --tool=sectionprof /bin/bash
==77387== SectionProf, a code section profiler
==77387== Copyright (C) 2018, and GNU GPL'd, by Andrei Rimsa.
==77387== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
==77387== Command: /bin/bash
==77387==
--77387-- UNKNOWN mach_msg unhandled MACH_SEND_TRAILER option
--77387-- UNKNOWN mach_msg unhandled MACH_SEND_TRAILER option (repeated 2 times)
--77387-- UNKNOWN mach_msg unhandled MACH_SEND_TRAILER option (repeated 4 times)
--77387-- UNKNOWN mach_msg unhandled MACH_SEND_TRAILER option (repeated 8 times)
$ exit
exit
--77387:0:syswrap- WARNING: Ignoring sigreturn( ..., UC_RESET_ALT_STACK );
==77387==
  Add:        19330/16% (exist) vs   7459097/14% (executed)
  Sub:        16801/13% (exist) vs   2505182/04% (executed)
  Mul:           59/00% (exist) vs      5503/00% (executed)
  Div:            0/00% (exist) vs         0/00% (executed)
  Mod:           18/00% (exist) vs      1438/00% (executed)
  Not:           51/00% (exist) vs      5638/00% (executed)
  And:         2099/01% (exist) vs   2674243/05% (executed)
  Or:           515/00% (exist) vs    183623/00% (executed)
  Xor:           81/00% (exist) vs     28816/00% (executed)
  Shift:       1616/01% (exist) vs   1237240/02% (executed)
  Load:       15983/13% (exist) vs   5388088/10% (executed)
  Store:      16654/13% (exist) vs   1927887/03% (executed)
  Others:     47457/39% (exist) vs  31608166/59% (executed)

  Max memory pool: 249.44 KB
$
```

Info: for the complete version:
`https://pastebin.com/raw/DCTTKG3w`

Questions?