Methods of binary analysis and valgrind tool

Raghunandan Palakodety

s6rapala@uni-bonn.de

January 4, 2015

Abstract

Writing complex programs in low-level languages, e.g C/C++, can be an arduous task when safeguards are minimal and hence code quality can take a toll. Tools and frameworks can be conducive in detecting subtle errors which often go unnoticed by the programmer. Some existing frameworks provided are used, to build tools to perform interesting program and malware analysis. In particular, Dynamic Binary Instrumentation framework, provides libraries to developers to build tools to analyze the program executables at run-time. Such tools are known as dynamic binary analysis tools. These dynamic binary analysis tools, rely on using shadow memory to *shadow*, in software, every byte of memory used by a program with another value that says something about it. Critical errors such as bad memory accesses, data races, and uses of uninitialised data are detected as the program runs on a synthetic cpu. Often such tools built generates analysis code, which essentially is a RISC like instructions inserted at various places in the intermediate representation of machine code. It it then stored in a translation cache and later executed via *dispatcher*. One such tool, `Memcheck`, built using valgrind framework operates on definedness of values down to the level of bits. Tools which track definedness at byte or word granularities give out false positives, that is, falsely flag errors resulting from correct uses of partially defined bytes. Whereas Memcheck's definedness checking at the level of individual bits accounts for low false positive and false negative rates.

1 Introduction

This paper elucidates various techniques for *instrumenting* a program at run-time, at machine-code level complementing to the very definition of dynamic binary analysis. Binary analysis can be conducive in environments where library code is often not made available. In the case of source analysis, programs are analyzed at source code level. Section 2 in this document starts off with a motivating example for the need of dynamic binary analysis tools. The definition and comparison of analysis approaches are presented in section 3. Section 4 explains two ways in which machine code (x86 instructions) is translated into intermediate representations which are subsequently instrumented by the tool. Later, section 5, gives details on *shadow values* and *shadow memory* which helps a tool to remember the history of memory location and/or the value residing. Section 5.1 describes the most basic form of Memcheck's shadow memory implementation to illustrate the data structures used.

2 Memory management Issues

Huge code bases involve numerous memory allocations and freeings. Adversely, scope does exists for intrinsic bugs such as memory leaks which go undetected until all the memory is run out and subsequent calls to *malloc* or *new* (i.e in C/C++ respectively) fail. Since garbage collection is not provided in C/C++, substantial development time might be spent handling freeing the memory. Pointer usage, in a case where the pointer tries to access a location past the end of the array, compiles just fine. But during runtime, segmentation fault occurs for the code snippet below.

```
class someClass{
  public:
    int someArray[10];
    int m_value1, m_value2;
    someClass(); //default constructor
    someClass(int, int, int* );
    void printValues();
    ~someClass(){}
};
```

```
someClass::someClass(int value1, int value2, int *array){
   m_value1 = value1;
   m_value2 = value2;
   for(int i = 0; i < 10; i++ )
      someArray[i] = array[i];
}
int main(){
  int a[10] = {0,1,2,3,4,5,6,7,8,9};
  someClass *someClassObj = new someClass(1,2,a);
  cout<<"Accessing array beyond the declared:
      "<<someClassObj->someArray[200000]<<endl;
}
```

In systems programming using C/C++, uninitialized variables in stack frame can take values of previous variables which occupied those addresses. Uninitialized variables are even more likely to appear in classes, where member declarations are generally separated from the constructor implementation. Ultimately, member variables do not hold a predictable value when they are just declared as shown in the code snippet below,

```
int main(){
  someClass someClassObj;
  cout<<"Accessing uninitialised member
  variables: "<<someClassObj.m_value1<<"\t"<<someClassObj.m_value2;
}
```

Additionally, programs with such uninitialized variables may even pass software tests. Un-initialized variables make the program non-determinsitic and cause bugs often difficult to reproduce by testers.

Improper usage of memory deteriorate performance of huge projects. One such categories of bugs include, freeing memory more than once as shown in the code snippet below. Freeing twice, often happens while writing library code, where a programmer might free twice and may manifest as a vulnerability (e.g denial-of-service attack) in programs that are linked with such libraries [1].

```
int main(){
  int a[10] ={ 0,1,2,3,4,5,6,7,8,9};
  someClass *someClassObj = new someClass(1,2,a);
  variables:
      "<<someClassObj->m_value1<<"\t"<<someClassObj->m_value2;
  delete someClassObj;
  delete someClassObj;
}
```

Also the appropriate operators must be taken into consideration, for instance, *free* corresponds to *malloc* and *delete* complements to *new* operator.

The above mentioned issues are handled by a tool, `Memcheck` which checks for memory related errors during run-time. This tool is grouped under the category dynamic binary analysis tools, where the executable is profiled to give more details on errors.

2.1 Brief description of the different possible memory leaks

The valgrind documentation [8] includes following list of nine memory leak types. Any possible kind of memory leak should fall into one of these categories.

```
     Pointer chain        AAA Category BBB Category
     -------------        ------------ ------------
(1)  RRR ------------> BBB              DR
(2)  RRR ---> AAA ---> BBB DR           IR
(3)  RRR             BBB                DL
(4)  RRR      AAA ---> BBB DL           IL
(5)  RRR ------?-----> BBB              (y)DR, (n)DL
(6)  RRR ---> AAA -?-> BBB DR           (y)IR, (n)DL
(7)  RRR -?-> AAA ---> BBB (y)DR, (n)DL (y)IR, (n)IL
(8)  RRR -?-> AAA -?-> BBB (y)DR, (n)DL (y,y)IR, (n,y)IL, (_,n)DL
(9)  RRR      AAA -?-> BBB DL           (y)IL, (n)DL

  Pointer chain legend:
  - RRR: a root set node or DR block
```

```
- AAA, BBB: heap blocks
- --->: a start-pointer
- -?->: an interior-pointer

Category legend:
- DR: Directly reachable
- IR: Indirectly reachable
- DL: Directly lost
- IL: Indirectly lost
- (y)XY: it's XY if the interior-pointer is a real pointer
- (n)XY: it's XY if the interior-pointer is not a real pointer
- (_)XY: it's XY in either case
```

Although internally, VALGRIND distinguishes nine different types of memory leaks, the generated output report will only include 4 main categories:

Still Reachable: Covers cases 1 and 2 (for the BBB blocks)

Directly Lost: Covers case 3 (for the BBB blocks)

Indirectly Lost: Covers cases 4 and 9 (for the BBB blocks)

Possibly Lost: Covers cases 5, 6, 7 and 8 (for the BBB blocks)

Directly and Indirectly Lost leaks are also referred as **Definitely Lost** leaks. The output from memcheck tool shows the **Definitely Lost** category, for the code snippet which will be shown during the presentation.

```
==25156==
==25156== LEAK SUMMARY:
==25156==    definitely lost: 48 bytes in 1 blocks
==25156==    indirectly lost: 0 bytes in 0 blocks
==25156==      possibly lost: 0 bytes in 0 blocks
==25156==    still reachable: 0 bytes in 0 blocks
==25156==         suppressed: 0 bytes in 0 blocks
==25156==
==25156== For counts of detected and suppressed errors, rerun with:
    -v
==25156== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0
    from 0)
```

From the above categories, the suggested order to fix leaks [9] is the following one:

1. **Directly Lost** leaks (reported as definitely lost in the output report).
2. **Indirectly Lost** leaks.
3. **Possibly Lost** leaks (as they may really be 'definitely lost').
4. **Still Reachable** leaks (if really needed to fix them)

However, there are limitations with `Memcheck` in the case of reporting bounds errors while using static arrays or stack allocated data, and thus vulnerable to *stack smashing error* in the case, where more than ten characters are given as input.

```
void func()
{
    char array[10];
    gets(array);
}
int main(int argc, char **argv)
{
    func();
}
```

Memcheck cannot handle such errors as shown in figure 1. However, an experimental tool `exp-sgcheck` finds stack and global overruns.

In order to develop new *heavyweight* tools such as `Memcheck`, the tool writer or developer is required to understand a technique known as *shadowing*. This technique is explained in section 5.1 and it is one of the quintessential requirements for building dynamic-analysis tools.

3 Introduction to analysis approaches

To have a better understanding of analyzing approaches, types of analysis are discussed. Source analysis is performed on programming language constructs and hence this type of analysis is programming language dependent. Whereas, binary analysis works at the level of machine code, when stored as object code (pre-linking) or executable code (post-linking) and hence it is language-independent but architecture-dependent. Many software defects that cause memory and threading errors can be detected both dynamically and statically. Depending on *when* analysis occurs, program analyses can be categorized into static analysis and dynamic analysis.

```
=21258==    by 0x80485A0: func() (stackSmashing.cpp:10)
=21258==    by 0x20312030: ???
=21258==  Address 0x20312031 is not stack'd, malloc'd or (recently) free'd
=21258==
=21258==
=21258== Process terminating with default action of signal 11 (SIGSEGV)
=21258==  Access not within mapped region at address 0x20312031
=21258==    at 0x4324B19: ??? (in /lib/i386-linux-gnu/libgcc_s.so.1)
=21258==    by 0x43256E0: _Unwind_Backtrace (in /lib/i386-linux-gnu/libgcc_s.so.1)
=21258==    by 0x423DF16: backtrace (backtrace.c:127)
=21258==    by 0x41A3FDF: __libc_message (libc_fatal.c:180)
=21258==    by 0x423DCE4: __fortify_fail (fortify_fail.c:32)
=21258==    by 0x423DC99: __stack_chk_fail (stack_chk_fail.c:29)
=21258==    by 0x80485A0: func() (stackSmashing.cpp:10)
=21258==    by 0x20312030: ???
=21258==  If you believe this happened as a result of a stack
=21258==  overflow in your program's main thread (unlikely but
=21258==  possible), you can try to increase the size of the
=21258==  main thread stack using the --main-stacksize= flag.
=21258==  The main thread stack size used in this run was 8388608.
=21258==
=21258== HEAP SUMMARY:
=21258==     in use at exit: 0 bytes in 0 blocks
=21258==   total heap usage: 1 allocs, 1 frees, 28 bytes allocated
=21258==
=21258== All heap blocks were freed -- no leaks are possible
=21258==
=21258== For counts of detected and suppressed errors, rerun with: -v
=21258== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
egmentation fault (core dumped)
```

Figure 1: Memcheck misses stack related errors

Static analysis involves analyzing program's source code or machine code without running it. Examples of static analysis include checking for correctness such as type checking, and few static analysis tools can find ways for optimization and identifying bugs. Tools performing static analysis only need to read a program in order to analyze it.

Dynamic analysis is evaluation of an application during run-time. Profilers and execution visualizers are examples of dynamic analysis tools. Tools performing dynamic analysis, *instrument* the client program with analysis code. The analysis code may be inserted entirely inline and may also include external routines called from the inline analysis code. The analysis code runs as part of program's normal execution, but does extra work such as measuring performance and identifying bugs.

Dynamic and Static analyses discussed are complementary because no single approach can find every error. But, the primary advantage of dynamic analysis is, it reveals subtle vulnerabilities and defects which are difficult to discover using static analysis. This topic deals with dynamic binary analysis, i.e analysis of machine code produced at run-time, and dynamic binary

instrumentation i.e the technique of *instrumenting* machine code with analysis code at run-time.

Lastly, Dynamic Binary Instrumentation (DBI) frameworks support two ways to represent code and allow instrumentation. These are discussed in section 4.

3.1 Distinction between Dynamic binary analysis and Dynamic Binary Instrumentation

Dynamic binary analysis or DBA is a particular kind of program analysis. DBA is defined as analyzing the behaviour of *client program* or *executable* at **run-time** through the injection of instrumentation code [10]. This instrumentation code executes as part of normal instruction stream after being injected. This procedure allows to gain insight into the behaviour and state of an application at various points in execution.

Dynamic Binary Instrumentation or DBI is a particular instrumentation technique adopted for DBA. Dynamic binary analysis requires client program or executable to be instrumented with analysis code. The analysis code can be injected by a program grafted onto the client process, or by an external process. If the client uses dynamically-linked code the analysis code must be added after the dynamic linker has done its job. This entire process also known as Dynamic Binary Instrumentation occurs at run-time.

4 Instrumentation techniques

Considering `IA-32` instruction set architecture, instruction set's length and compatibility of representing instructions for `x86` architecture in other forms should be known prior to instrumentation. Firstly, `x86` instructions with varying lengths (1 and 15 bytes in length) and branches (using `JLE`, `JGE`) are transformed into an *Intermediate Representation*(IR), and then neccessary code is added to this IR to guide instrumentation. There are two ways of instrumentation

- Disassemble and resynthesise
- Copy and Annotate[1]

[1] In depth details are not discussed in this paper.

DBI frameworks such as DynamoRIO and PIN use *Copy and Annotate* to represent code and instrumentation. In this technique, machine code instructions are copied except for necessary control flow changes shown in the figure 2. Preserving original application instructions is important for du-

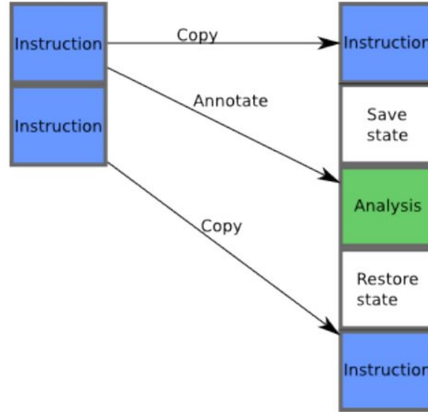

Figure 2: Verbatim copy program, generate annotations for analysis

plicating native behaviour, especially for tools that would like to study the instruction makeup of application. Such instructions are annotated via data structures (e.g. in DynamoRIO) or an instruction querying API (e.g. Intel's Dynamic Binary Instrumentation framework, PIN).

These annotations and APIs are conducive to guide instrumentation. The added analysis code to the original code is monitored by the tools to avoid distortion of code. Whereas, Valgrind resorts to low-level instruction representation. Its code caching systems translate IA-32 to RISC-like internal representations [5].

4.1 Disassemble and Resynthesize

In this approach, machine code is disassembled and converted into an Intermediate Representation (IR), in which each instruction becomes one or more IR operations. This IR is then instrumented (by adding more IR in some cases) and then converted back to machine code. So, original client code instructions are discarded and final code is generated purely from IR. This particular technique is used by Valgrind.

The above mentioned details are shown in the figure 3, in which object code is tranformed or translated into `IR` in `Disassembly`. Optimization procedures are performed on IR in `Optimization I` for easier instrumentation. `Instrumentation` adds analysis code by the tool resulting in Instrumented IR. Instrumented IR is optimized in `Optimization II` resulting in Optimized Instrumented IR. Conversion of the previous IR into machine instructions takes place in `Assembly`, stored in cache and later executed.

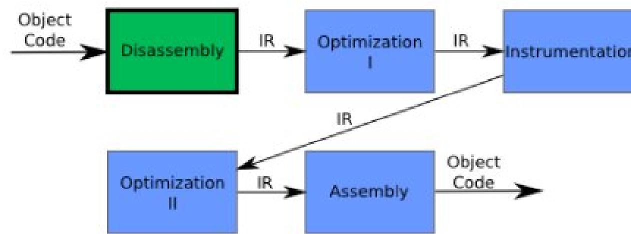

Figure 3: Object code → Code cache

4.1.1 Translating single code block

Valgrind uses disassemble and resynthesise as described in previous section. Valgrind's IR blocks are superblocks, which are also attributed as *single-entry-multiple-exits* code blocks, in which, each IR block contains a list of statements, which are operations with side-effects such as register writes, memory stores, and assignment to temporaries (`t0` to `t84`) shown in figure 4.

Valgrind itself runs on the machine's real or *host* CPU, and runs a *client* program or executable under analysis on a simulated or *guest* CPU. In general, executable requires access to CPU's registers. We refer to the registers in the host CPU as *host registers* and those of simulated CPU as *guest registers*. Due to dynamic binary recompilation process, a guest register's value may reside in one of the host's registers, or it may be spilled to memory for a variety of reasons. Shadow registers are shadows of guest registers.

Valgrind provides a block of memory per client thread called as *ThreadState*. Each one contains space for all thread's guest and shadow registers. Each ThreadState contains space for all thread's guest and shadow registers and is used to hold them at various times, in particular between each code

block (which will be defined shortly). These statements also contain expressions without any side effects such as constants, register reads, memory loads, and arithmetic operations.

As an example, a store statement contains one expression for the store address and another for the store value. The IR has constructs similar to that of RISC, such as load/store, where each operation does perform only on temporaries, as shown in figure 4 The IR is architecture-independent. To

```
raghu@raghu-Aspire-3830TG: ~/workspace/Debug    raghu@raghu-Aspire-3830TG: ~/workspace/Debug    raghu@raghu-Aspire-3830TG: ~/workspace/Debug
==== SB 0 (evchecks 0) [tid 0] 0x40011d0 UNKNOWN_FUNCTION /lib/i386-linux-gnu/ld-2.15.so+0x11d0

------------------------ After instrumentation ------------------------

IRSB {
   t0:I32   t1:I32   t2:I32   t3:I32   t4:I32   t5:I32   t6:I32   t7:I32
   t8:I32   t9:I32   t10:I32   t11:I32   t12:I32   t13:I32   t14:I32   t15:I32
   t16:I32   t17:I32   t18:I32   t19:I32   t20:I32   t21:I32   t22:I32   t23:I32
   t24:I32   t25:I32   t26:I32   t27:I32   t28:I32   t29:I32   t30:I32   t31:I32
   t32:I32   t33:I32   t34:I32   t35:I32   t36:I32   t37:I1   t38:I32   t39:I32
   t40:I32   t41:I32   t42:I32   t43:I1   t44:I32   t45:I32   t46:I32   t47:I32
   t48:I32   t49:I1   t50:I32   t51:I32   t52:I32   t53:I32   t54:I32   t55:I1
   t56:I32   t57:I32   t58:I32   t59:I32   t60:I32   t61:I1   t62:I32   t63:I32
   t64:I32   t65:I32   t66:I32   t67:I32   t68:I32   t69:I1   t70:I32   t71:I1
   t72:I32   t73:I32   t74:I32   t75:I1   t76:I32   t77:I32   t78:I32   t79:I32
   t80:I32   t81:I32   t82:I1   t83:I32   t84:I32

   ------ IMark(0x40011D0, 2, 0) ------
   PUT(68) = 0x40011D0:I32
   DIRTY 1:I1 RdFX-gst(24,4) MoFX-gst(68,4) ::: VG_(helperc_CallDebugger)[rp=1]{0x3809c640}(0x40011D0:I32)
   t33 = GET:I32(360)
   t16 = GET:I32(24)
   PUT(344) = t33
   PUT(8) = t16
   PUT(68) = 0x40011D2:I32
   ------ IMark(0x40011D2, 5, 0) ------
   PUT(68) = 0x40011D2:I32
   DIRTY 1:I1 RdFX-gst(24,4) MoFX-gst(68,4) ::: VG_(helperc_CallDebugger)[rp=1]{0x3809c640}(0x40011D2:I32)
   t34 = Or32(t33,0x0:I32)
   t35 = Left32(t34)
   t36 = t35
   t17 = Sub32(t16,0x4:I32)
   PUT(360) = t36
   DIRTY 1:I1 RdFX-gst(24,4) ::: track_new_mem_stack_4[rp=1]{0x38023ac0}(t17)
   PUT(24) = t17
   t37 = CmpNEZ32(t36)
   DIRTY t37 RdFX-gst(24,4) RdFX-gst(68,4) ::: MC_(helperc_value_check4_fail_no_o){0x38026880}()
   t38 = 0x0:I32
   DIRTY 1:I1 RdFX-gst(24,4) RdFX-gst(68,4) ::: MC_(helperc_STOREV32le)[rp=2]{0x38026020}(t17,0x0:I32)
```

Figure 4: Disassembly: machine code to *treeIR*

create a translation of a single code block, Valgrind follows until one of the following conditions is met.

1. An instruction limit is reached. approximtely 50.
2. A conditional branch is hit.
3. A branch to an unknown target is hit.
4. More than three unconditional branches to known targets have been hit.

Valgrind's core is split into two: coregrind and VEX. There are eight translation phases which are performed by the Valgrind's VEX which is responsible for dynamic code translation and for calling tools' hooks for IR instrumentation. While instrumentation is performed by the tool. Coregrind is responsible for dispatching, scheduling, block cache management, symbol name demangling. These phases are explained not in great detail but it is rather tried to expound some of them as follows,

1. *Disassembly: machine code → tree IR.* The disassembler converts machine-dependent code into (unoptimised) tree IR [13]. Each instruction is disassembled into one or more statements. These statements update the guest registers residing in valgrind memory: guest registers are pulled out from the `ThreadState` into temporaries, operated on and then written back.

   From the figure 4, statements starting with `IMark` indicate no-ops just to indicate where an instruction started, holding arguments where an instruction started, its address and length in bytes. These are used by the profiling tools to check the instruction boundaries. `GET:I32` fetches a 32-bit integer guest register from the `ThreadState` which takes offsets as arguments (for eg: `GET:I32(360)` or `GET:I32(28)`) 360, 28 for guest registers, `%edx` and `%ebp`.

   To sum up details in this phase, machine-independent code is translated into VEX's machine independent IR. The IR is based on single-static-assignment (SSA) form, and has some RISC-like features. Most instructions get disassembled into several IR opcodes.

2. *Optimisation 1: treeIR → flatIR.* This optimisation phase flattens the IR i.e, redundant get and put operations are eliminated. Since, the figure 4 shows post instrumentation code, this intermediate optimisation phase involves complex expression tree flattened into one or more assignments to temporaries.

3. *Instrumentation: flatIR → flatIR.* The code block is then passed to the tool, which can transform it arbitrarily [11], which is conducive for instrumentation by a shadow value tool. Shadow operations are larger than their original counterpart operations as shown in figure 4, where `DIRTY t44 RdFX-gst(24,4) RdFX-gst(68,4)`,$t46 = CmpNEZ32(0x0 : I32)$ in which `DIRTY` and `RdFX` annotations indi-

cate that some registers are read from the `ThreadState` by the function and the address of the called function being `0x38026830`.

To sum up details in this phase, VEX calls the Valgrind tool's hooks to instrument the code.

4. *Optimisation 2: flatIR → flatIR.* Constant folding, deadcode removal, copy propagation, common sub-expression elimination performed in this phase [11].

5. *Tree building: flatIR → treeIR.* A tree builder converts flatIR to treeIR in preparation for instruction selection in the next phase. The resulting code may perform loads in a different order to the original code.

6. *Instruction selection: treeIR → instruction list.* The instruction selector converts the tree IR into a list of instructions which use virtual registers. The instruction selector uses a simple top-down tree matching algorithm.

7. *Register allocation: instruction list → instruction list.* The linear scan register allocator, replaces virtual registers with host registers. The register allocator can remove many register-to-register moves. This can be observed while debugging the core using the option –trace-flags=<8-bit wide bitmask> along with –trace-notbelow=10 where bitmask value takes 00000010.

   In short, register allocation allocates real host registers to virtual registers, using a linear scan algorithm. This can create additional instructions for register spills and reloads (especially in register-constrained architectures like x86.)

8. *Assembly: instruction list → machine code.* The final assembly phase encodes the selected instructions and writes them to a block of memory.

At the end of each block, VEX saves all guest registers to memory, so that the translation of each block is independent of the others.

4.1.2 Block Management

Blocks are produced by VEX, but are cached and executed by coregrind. Each block of code is actually a superblock or IRSB shown in figure 4 (single-entry and multiple-exit) [6]. Translated blocks are stored by coregrind in a big translation table (that has a little less than 420,000 useful entries), so it gets rarely full. The table is partitioned in eight sectors. When it gets 80% full, all the blocks in sector are flushed. The sectors are managed in FIFO order. A comprehensive study of block cache management can be found in [7].

4.1.3 Block execution

Blocks are executed through coregrind's dispatcher, a small hand-written assembly loop (12 instructions on Ubuntu linux platform). Each block returns to the dispatcher loop at the end of its execution and there is no block chaining. The dispatcher lookups for blocks in a cache with 2^{15} entries. The cache has hit-rate approximately 98% as stated by the author in the paper [12]. When the cache lookup fails, the dispatcher fallsback to a slower routine written in C to lookup the translated block in the translation table, or to translate the block if it is not in the table.

From the concepts of pre-emptive multitasking, where processor time is divided among the threads that need it, the valgrind's dispatcher is also responsible for checking if the thread's timeslice ended. When the timeslice ends, the dispatcher relinquishes the control back to the scheduler.

4.1.4 Understanding Valgrind's Core

Valgrind is compiled into a linux shared object, `valgrind.so`. The valgrind shell script adds `valgrind.so` to the `LD_PRELOAD` list of extra libraries to be loaded with any dynamically linked library.

`valgrind.so` is linked with the `-z initfirst` flag, which requests that its initialization code is run before that of any other object in the executable image. Upon achieving this, valgrind gains control and the real CPU becomes trapped in `valgrind.so`. The synthetic CPU provided by valgrind does, however return from this initialization function. So the normal startup actions, orchestrated by `ld.so` (dynamic-linker), continues as usual, except on the synthetic CPU and not on the real CPU. Eventually, `main` is run and

returns, and then the finalization code of the shared objects is run, in inverse order to which they were initialized. Later, `valgrind.so`'s own finalization code is called. It spots this event, shuts down the simulated CPU, prints any error summaries and/or does the leak detection, and returns the initialization code on the real CPU. At this point, valgrind has lost the control over the program and the program finally `exit()`s back to the kernel.

The normal activity, once the valgrind has started up, is as follows. Valgrind never runs any part of the program (also referred to as client) directly. Instead it uses function `VG_(translate)` to translate basic blocks (BBs, straight-line sequences of code) into instrumented translations and those are run instead. The translations are stored in a Translation cache (TC), `vg_tc`. The translation table (TT), `vg_tt` supplies the original-to-translation code address mapping. Auxillary array `VG_(tt_fast)` is used as a direct-map cache (shown in figure 5) for fast lookups in TT. This auxillary array achieves a hit rate of around 98% [12].

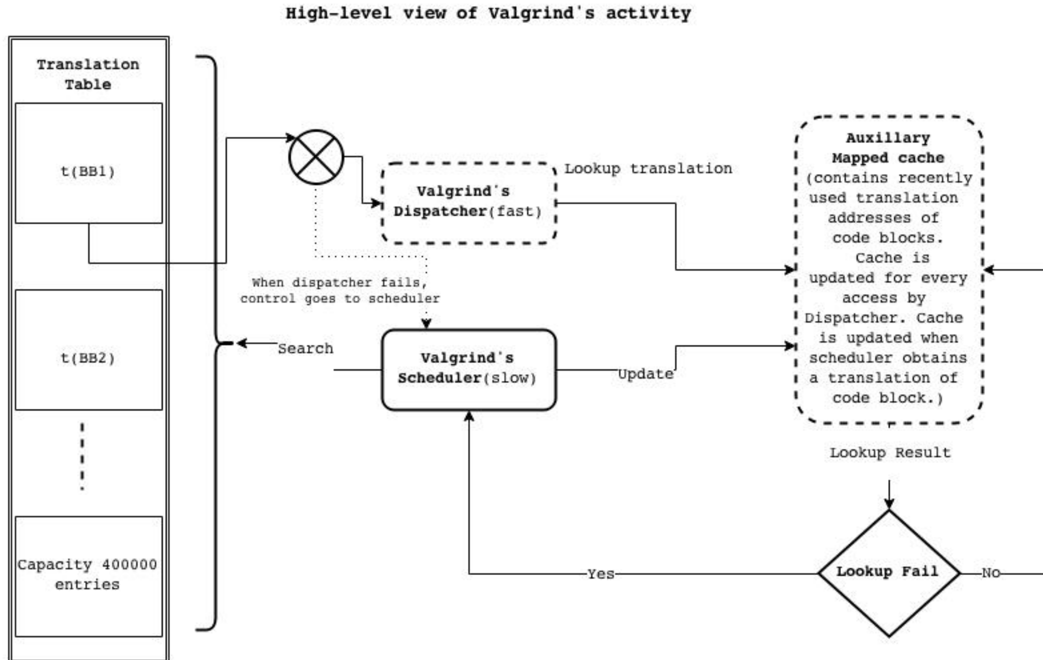

Figure 5: Auxillary mapped cache updated for every access of translation block. Also updated with the address of new translation when created.

Function VG_(disp_run_translations) in dispatch-x86-linux.S (for x86/Linux) is the heart of JIT dispatcher. Once a translated code address has been found, it is executed simply by an x86 call to the translation. At the end of the translation, the next original code address is loaded into %eax, and the translation then does a ret, taking it back to the dispatch loop. The address requested in %eax is looked up first in VG_(tt_fast) by the dispatcher, and if not found, a helper function written in C, VG_(search_transtab) is called. In case, if the translation is not available (i.e if VG_(search_transtab) returns false), control goes back to the top-level C dispatcher VG_(toploop) (not shown in the figure), which arranges for VG_(translate) to make a new translation.

So far, overview about the VG_(translate) and VG_(disp_run_translations) is discussed in this paper. Some details are further illustrated in the manual[8], which is not updated. coregrind/m_scheduler/scheduler.c contains more accurate details of valgrind's dispatcher and scheduler.

4.1.5 Conceptual view of client

As mentioned earlier in ealier sections, Valgrind uses dynamic binary compilation and caching. Valgrind grafts itself into the client process at start-up and then recompiles client's code, one basic block (a basic block ends upon any control transfer instruction) at a time, in a just in time, execution-driven fashion. The compilation process involves disassembling the machine code into an intermediate representation (IR) which is instrumented with a skin or tool plug-in and then converted back into x86 code [10]. The result is called translation and stored in a code cache to be rerun as necessary. The core of valgrind spends most of its execution time making, finding, and running translations.

Figure 6 gives a conceptual view of normal program or client execution. The client can directly interact and access the user-level parts of the machine (e.g general purpose registers). The client can access the system-level parts of the machine through operating system calls or system calls.

Whereas figure 7 shows how this changes when a program is run under the control of a valgrind tool. The client and tool are part of the same process, but the tool mediates everything the client does, having complete control over the client.

Since valgrind is execution driven, executable code, dynamically linked libraries, dynamically generated code are handled well. System calls such

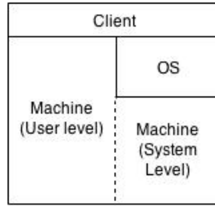

Figure 6: Conceptual view of client under normal execution

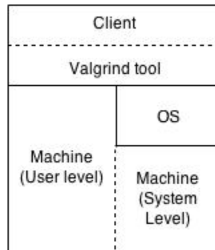

Figure 7: Conceptual view of client when run on VM

as `ioctl`, `write`, `recv` are not under valgrind's control and are performed on real CPU, but indirectly observed. The core of valgrind takes following steps in the case of a system call emanating from the client.

1. Save valgrind's stack pointer.
2. Copy the simulated registers, except the program counter, into the real registers.
3. Do the system call.
4. Copy the simulated registers back out to memory, except the program counter.
5. Restore valgrind's stack pointer.

From practical observations and to conclude this section, the `D&R` approach may require more implementation effort as it is very complex. Also, generating a good code at the end requires more development effort. Valgrind's Just In Time (JIT) compiler uses a lot of conventional compiler [11] technology and Valgrind's Disassemble and resynthesize JIT compiler consumes considerable time to compile the code slowly due to the reason of dynamic recompilation.

4.2 Copy and Anotate

Copy and Annotate can be understood by taking an example into consideration. One such frameworks that adhere to this category is DynamoRIO. DynamoRIO copies application code into its code cache in units of *basic blocks*(as shown in figure 8), which are sequences of instructions ending with a single control transfer instruction (conditional JLE, JGE, JE etc. or unconditional JMP). Similarly, DynamoRIO considers each entry point to begin a new basic block and follows until it reaches a control transfer instruction. DynamoRIO executes the application code by transferring the control to corresponding basic block in code cache [3] as shown in figure 9. A block that directly targets another block already resident in the cache is linked to that block to avoid the cost of returning to the DynamoRIO dispatcher. After the execution of each block, the application's machine state is saved and control is returned to DynamoRIO's *context-switch* to copy the next basic block.

```
original:add %eax, %ecx
         cmp $4, %eax
         jle 0x40106f
```

Figure 8: An example basic block consisting of three IA-32 instructions: an add, a compare, and a conditional direct branch

Frequently executed sequences of basic blocks are combined into *traces* [2], which are placed in a separate code cache. DynamoRIO makes these traces available via its interface for convenient access to hot application code streams.

Figure 10 shows a basic block inside DynamoRIO's code cache. Each exit from a basic block has its own stub. When the basic block is not linked to another block, control goes to the stub, which records a pointer to a stub specific data structure which helps DynamoRIO to determine which exit was taken.

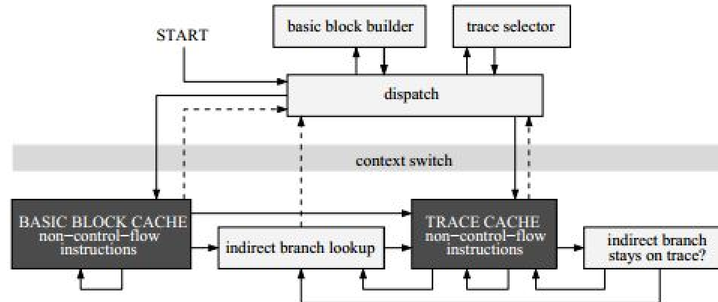

Figure 9: DynamoRIO system infrastructure: Dark shading indicates application code. Note that the context switch is simply between the code cache and DynamoRIO

```
fragment7:  add %eax, %ecx
            cmp $4, %eax
            jle stub0
            jmp stub1
    stub0:  mov %eax, eax-slot
            mov &dstub0, %eax
            jmp context_switch
    stub1:  mov %eax, eax-slot
            mov &dstub1, %eax
            jmp context_switch
```

Figure 10: An example basic block containing multiple exits

5 Requirements of dynamic binary analysis tools

Before explaining the requirements, we consider the previously introduced tool, `Memcheck`, built using Valgrind. Memcheck is memory error detector which was primarily designed for use with C/C++ programs. When an executable is run under Memcheck, this tool instruments almost every operation and issues messages about detected memory errors. Memcheck maintains three kinds [11] of data about the running executable.

1. `A` Bits. Every memory byte is shadowed with a single *A* bit or addressability bit, which indicates if the executable or client can legitimately access it. A value 0 indicates unaddressable byte, a 1 represents an addressable byte. These bits are updated as memory is allocated and

freed. Additionally, checked upon every memory access. With these addressability bits [11], Memcheck can detect use of unaddressable memory such as heap buffer overflows and wild reads/writes.

2. `V` Bits. Every register and memory byte is shadowed with eight validity bits or V bits, which indicate if the value bits are defined. Every value writing operation is shadowed with another operation that updates the respective shadow values. Memcheck uses the V bits to detect uses of undefined values with bit level granularity.

3. `Heap blocks`. Repeated frees of heap blocks are handled by recording the location of every live heap block in a hash table. With this information, Memcheck handles such errors.

5.1 Requirements: Shadow value tools and shadow computation

Dynamic binary analysis tools use shadow memory, that is, those tools that track or shadow every byte of memory used by the executable, with a shadow memory value, that describes a value within the memory location. In other words it is also known as *metadata*. This metadata describes whether accesses are from trusted source or it may describe the number of accesses to the memory location itself. Such tools are called *shadow memory tools* [11].

The analysis code while instrumentation, added by the tool such as Memcheck, updates the shadow memory in response to memory accesses. This shadow memory is useful and lets a tool to track or store the history of every memory location along with the value present in it. For instance, Memcheck, remembers or tracks the allocation and de-allocation operations that have affected memory location and detect accesses of un-addressable memory locations. Using Undefined or uninitialized values are also detected.

From the above description, it is intuitive that large amounts of extra state must be maintained, that is, one shadow byte per byte of original memory. All operations that affect large regions of memory, such as allocations and de-allocations either on heap, stack or through system calls such as `mmap` are instrumented in order to keep the shadow memory up-to-date. Such requirements increases the total amount of code that is run, increase a program's memory footprint and slow down memory accesses. Shadow memory tools thus slow down the programs by a factor of 10-100.

6 Conclusion

Valgrind's use of `D&R` can make simple tools more difficult to write than in `C&A` frameworks. A tool that traces or tracks memory accesses would be about 30 lines of code [11] using Pin framework and 100 in valgrind. `Dr.Memory`, a memory checker built using DynamoRIO is twice as fast as `Memcheck` on average and upto four times faster on individual benchmarks [4].

The ongoing task of porting these tools for other architectures x86/Mac OS X and x86/FreeBSD are available as experimental tools. The challenge of porting these tools to a new architecture requires writing new code for JIT compiler, such as an instruction encoder and decoder. Whereas another challenge of porting these tools to a new OS requires new code for handling details such as signals, address space manager and new system call wrappers to be written.

References

[1] Wan-Teh Chang Adam Langley. Openssl vulnerabilities, common vulnerabilities and exposures.

[2] Derek Bruening. *Efficient, transparent, and comprehensive runtime code manipulation, 2004.* PhD thesis, MIT, The address of the publisher, 9 2004. An optional note.

[3] Derek Bruening and Saman Amarasinghe. Maintaining consistency and bounding capacity of software code caches. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pages 74–85, Washington, DC, USA, 2005. IEEE Computer Society.

[4] Derek Bruening and Qin Zhao. Practical memory checking with dr. memory. In *The International Symposium on Code Generation and Optimization*, Chamonix, France, Apr 2011.

[5] Derek Lane Bruening. *Efficient, transparent, and comprehensive runtime code manipulation.* PhD thesis, Massachusetts Institute of Technology, 2004.

[6] Filipe Cabecinhas, Nuno Lopes, R Crisotomo, and Luıs Veiga. Optimizing binary code produced by valgrind. Technical report, Citeseer, 2008.

[7] Kim Hazelwood Cettei. *Code cache management in dynamic optimization systems*. PhD thesis, Harvard University Cambridge, Massachusetts, 2004.

[8] Nicholas Nethercote Julian Seward. *Valgrind Documentation*. Valgrind Developers.

[9] Aleksander Morgado. Understanding valgrind memory leak reports.

[10] Nicholas Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, Computer Laboratory, University of Cambridge, United Kingdom, November 2004.

[11] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 65–74, New York, NY, USA, 2007. ACM.

[12] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan Notices*, volume 42, pages 89–100. ACM, 2007.

[13] Aaron Pohle. Aufgabenstellung für die diplomarbeit: L4/valgrind: Dynamic binary analysis on l4. Diplomarbeit, Technische Universität Dresden.