Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation

Nicholas Nethercote & Julian Seward

Jon-Pierre Antoine & Trevor Farrelly
Overview

- Dynamic binary instrumentation (DBI)
- Dynamic binary analysis (DBA)
- *Shadow Values*
- Valgrind
Overview

While Valgrind tends to be slower than other DBI frameworks, it allows for more complex, robust DBA tools to be built faster and easier.
Motivation

- Performance vs. Capabilities of DBI frameworks
- Shadow values in Valgrind
  - Impact on various Valgrind tools
Previous Work

- Performance-based research on DBI frameworks
  - Dynamo: A Transparent Dynamic Optimization System
  - Efficient, transparent, and comprehensive runtime code manipulation
  - Code Cache Management in Dynamic Optimization Systems

- Capabilities of the tools built on DBI’s?
  - Less studied.
Shadow Values
Shadow Values

- **Byte-level metadata**
  - Every register
  - Every value in memory

- **Shadow Value Tools**
  - **Memcheck**
  - **Hobbes**
  - **Redux**

```
$ valgrind ./a.out
==4895== Memcheck, a memory error detector
==4895== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4895== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==4895== Command: ./a.out
==4895==
==4895== HEAP SUMMARY:
==4895==     in use at exit: 0 bytes in 0 blocks
==4895==     total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==4895==
==4895== All heap blocks were freed -- no leaks are possible
==4895==
==4895== For counts of detected and suppressed errors, rerun with: -v
==4895== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```
Requirements of Shadow Values

- Shadow State $S$
  - Mirrors program state $S$
- R1-2: Shadow Registers and Memory
- R3-4: Instrument read/write instructions and syscalls
- R5-8: Instrument allocations and deallocations
- R9: Extra Output

“a DBI framework that supports shadow values well will also support most conceivable DBA tools” - § 2, page 3
Valgrind - Deep Dive
Valgrind - Core and Plug-Ins

- ~$ valgrind ./a.out
  - Memcheck, \textit{not} Valgrind!
  - ~$ valgrind --tool=memcheck ./a.out

- Valgrind Core
  - Interface for tools - JIT compiler, memory management, etc.

- Tool plug-in
  - Instrumentation details - How/What to process in shadow values
Valgrind - **Execution**

- Runs client code on a simulated “guest” CPU
  - Each client thread gets a ThreadState \( S' \)

1. Disassembles client code
   - RISC-like *Intermediate Representation* (IR)

2. Injects plug-in instrumentation

3. Reassembles instrumented IR into machine code

*All in a JIT, execution-driven fashion!*
Valgrind - Code Representation

- uses disassemble-and-resynthesise
  - requires more up-front design and implementation effort
  - More verifiable
  - permits binary translation from one platform to another
- Final code is generated purely from IR (discussed next)
Valgrind - Intermediate Representation

- Platform-Agnostic
- RISC-like: ~200 integer, FP and SIMD opcodes
  - Arch-specific instructions are emulated in C functions
- Operates on code blocks, broken on:
  - a. ~50 instructions
  - b. a conditional branch
  - c. a branch to an unknown target
  - d. >3 unconditional branches
Valgrind - **Client Translation**

8-step process to translate client into fully instrumented program:

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Disassembly</td>
<td>Machine Code</td>
<td>Tree IR</td>
</tr>
<tr>
<td>2</td>
<td>Optimization 1</td>
<td>Tree IR</td>
<td>Flat IR</td>
</tr>
<tr>
<td>3</td>
<td>Instrumentation</td>
<td>Flat IR</td>
<td>Flat IR</td>
</tr>
<tr>
<td>4</td>
<td>Optimization 2</td>
<td>Flat IR</td>
<td>Flat IR</td>
</tr>
<tr>
<td>5</td>
<td>Tree Building</td>
<td>Flat IR</td>
<td>Tree IR</td>
</tr>
<tr>
<td>6</td>
<td>Instruction Selection</td>
<td>Tree IR</td>
<td>Instruction List</td>
</tr>
<tr>
<td>7</td>
<td>Register Allocation</td>
<td>Instruction List</td>
<td>Instruction List</td>
</tr>
<tr>
<td>8</td>
<td>Assembly</td>
<td>Instruction List</td>
<td>Machine Code</td>
</tr>
</tbody>
</table>
Valgrind - Client Translation

- Control Flows from one translation to the next via:
  - Dispatcher (fast)
  - Scheduler (slow)

- Dispatcher
  - Translation searched for using probable hash index

- Scheduler
  - Full translation table is searched

- Valgrind does not perform chaining
Valgrind - Other Features

- Client Requests
  - Allows the client program to communicate with Valgrind

- Eventing
  - The IR doesn’t cover all potential state changes
  - Send a notification when one such state change happens

- Function Replacement
  - Replace a function in the client code with one from the tool
  - Can call old function from within replacement
Valgrind - Challenging Client Behavior

● System Calls
  ○ Cannot trace into the kernel

● Threads
  ○ Load/Store instructions could be non-atomic

● Signals
  ○ Client hands control to kernel, may lead to total loss of control

● Self-Modifying Code
  ○ Code blocks are hashed every time they are executed (expensive!)
Bringing it All Together
Valgrind - Shadow Value Support

- **R1:** Shadow Registers?
  
  Yes! - Many features exist to support shadow registers.

- **R2:** Shadow memory?
  
  Not in the core. - Too specific to a given tool’s function to abstract anything away.
Valgrind - Shadow Value Support

- **R3**: Instrument read/write instructions?
  
  Yes! - All R/W ops are visible in the IR.

- **R4**: Instrument read/write syscalls?
  
  Yes! - Via the Eventing system.

  *Ditto for R5, R6, and R7*

- **R8**: Instrument heap allocs and deallocs
  
  Yes! - But not via the eventing system, for historical reasons.
Valgrind - Shadow Value Support

● **R9**: Extra Output

Yes! - Prints to stderr and supports file redirection.

“a DBI framework that supports shadow values well will also support most conceivable DBA tools”

Therefore, Valgrind supports a superset of DBA tools supported by other DBI frameworks.
Evaluation
Evaluation - Performance

- **Lightweight**
  - Mean slow down of 4.3x in no-instrumentation case (*nulgrind*)

- **Heavyweight**
  - *memcheck* mean slow down by factor of 22.2x
  - *taintcheck* ran 37x slower
  - *annelid* 35.2x

- Other DBAs are faster but are less robust
Evaluation - Usability

- Greater difficulty to write simple tools
  - 30 lines of Pin memory trace codes = 100 lines in Valgrind

- Relatively easy to write heavyweight tools
  - Provides powerful instrumentation capabilities
  - Allows them to be robust
  - Reasonable performance
Evaluation - **Limitations**

- Forced to serialize multithreaded processes
  - No set order to thread execution
- Limited architecture and OS
  - runs on x86/Linux, AMD64/Linux, and PPC\{32,64\}/\{Linux,AIX\}
Critique
Critique - Threading

- why not have each thread control its own shadow that corresponds to its host operations?
  - Since each invocation of valgrind has a copy of the core (in addition to the tool)
- How is the implemented solution any different from fine grained locking approach?
Critique - Potential Future Work

- Deeper investigation into shadow values
  - There may exist a novel use case that allows for interesting program analysis.
  - Valgrind provides the best platform for doing so.
Critique - Potential Future Work

- Valgrind’s IR is nearly a full-fledged ISA
  - Could there be potential in treating it as such?
  - Add-in hardware for shadow value analysis?
  - Valgrind-based CPU? *(maybe not)*

```plaintext
1. put %eip
2. get %eax
3. get %ebx
4. addl
5. put eflags val1
6. put eflags val2
7. put eflags val3
8. put eflags val4
9. put %eax
```
Thanks!

Questions?
Sources