Implement Your Own Dynamic Binary Analysis Using the Valgrind Framework

John van Schie

Center for Software Technology, Universiteit Utrecht
http://www.cs.uu.nl/groups/ST

January 17, 2008

Outline

1. Introduction
   - "The Sales Pitch"
2. The Valgrind Framework
   - Dynamic Binary Instrumentation
   - Architecture
   - Framework Features
3. Creating a Valgrind Tool
   - Set Up a Valgrind Tool
   - Branchcount Example

The Valgrind Project

- Main authors and developers are Julian Seward and Nicholas Nethercote, University of Cambridge.
- Available under GPLv2.
- Supported for x86/x86-64/ppc32/ppc64 running Linux.
- Available at `http://www.valgrind.org`.

Valgrind Examples - Memcheck

- A bit precise memory checker.
  - Tracks undefined values.
  - Traces memory leaks.
  - Warns for reads/writes that are usually errors.

Valgrind Examples - Memcheck

- A bit precise memory checker.
  - Tracks undefined values.
  - Traces memory leaks.
  - Warns for reads/writes that are usually errors.

C source

```
#include <stdbool.h>

int foo( bool b ) {
  if( b )
    return 0;
  else
   return 1;
}

int main( ) {
  bool b;
  return foo( b );
}
```

Valgrind Examples - Memcheck

- A bit precise memory checker.
  - Tracks undefined values.
  - Traces memory leaks.
  - Warns for reads/writes that are usually errors.

C source

```
#include <stdbool.h>

int foo( bool b ) {
  if( b )
    return 0;
  else
   return 1;
}

int main( ) {
  bool b;
  return foo( b );
}
```

Valgrind Examples - Memcheck

- A bit precise memory checker.
  - Tracks undefined values.
  - Traces memory leaks.
  - Warns for reads/writes that are usually errors.

C source

```
#include <stdbool.h>

int foo( bool b ) {
  if( b )
    return 0;
  else
   return 1;
}

int main( ) {
  bool b;
  return foo( b );
}
```

Memcheck output

```
==16410== Conditional jump or move
          depends on uninitialised value(s)
==16410==    at 0x400454: foo
             (memcheck-example.c:4)
==16410==    by 0x40047B: main
             (memcheck-example.c:12)
```

Valgrind Examples - Cachegrind

- A cache profiler.
- Points out cache unfriendly code by annotations.

Valgrind Examples - Cachegrind

- A cache profiler.
- Points out cache unfriendly code by annotations.

C++ source

```
void scalarMul( int scalar, vector< vector<int> > matrix
                , size_t nRows, size_t nCols ) {

  for(int j=0; j < nCols; ++j )
    for(int i=0; i < nRows; ++i )
      matrix[i][j] *= scalar;
}
```

Valgrind Examples - Cachegrind

- A cache profiler.
- Points out cache unfriendly code by annotations.

C++ source

Cachegrind annotated code (100.000 rows, 100 cols, 2.3s)

```
      D1mr       D2mr
         .          .  void scalarMul( int scalar
         .          .    , vector< vector<int> > matrix
         0          0    , size_t nRows, size_t nCols ) {
         .          .
         0          0    for(int j=0; j < nCols; ++j )
         0          0      for(int i=0; i < nRows; ++i )
 9,999,999  9,963,671          matrix[i][j] *= scalar;
         0          0  }
```

Valgrind Examples - Cachegrind

- A cache profiler.
- Points out cache unfriendly code by annotations.

Alternative C++ Source

```
void scalarMul( int scalar, vector< vector<int> > matrix
                , size_t nRows, size_t nCols ) {

  for(int i=0; i < nRows; ++i )
    for(int j=0; j < nCols; ++j )
      matrix[i][j] *= scalar;
}
```

Valgrind Examples - Cachegrind

- A cache profiler.
- Points out cache unfriendly code by annotations.

Alternative C++ Source

Cachegrind annotated code (100.000 rows, 100 cols, 1.7)

```
    D1mr     D2mr
      .        .  void scalarMul( int scalar
      .        .     , vector< vector<int> > matrix
      0        0     , size_t nRows, size_t nCols ) {
      .        .
      0        0    for(int i=0; i < nRows; ++i )
      0        0      for(int j=0; j < nCols; ++j )
 650,001  649,999         matrix[i][j] *= scalar;
      0        0  }
```

Valgrind Examples - Other

- Callgrind - Feature rich profiling tool.
- Massif - Measures heap usage.
- Helgrind - Detects possible insufficient locking.

Commonality of the Tools

- **Dynamic**: Run-time analysis.
- **Binary**: Operates on the binary level.
- **Instrumentation**: *Normal* behavior of the program unmodified.

The Valgrind Framework

Valgrind is a framework for building dynamic binary instrumentation tools. Tool developers only need to focus on the analysis.

Why Dynamic Binary Instrumentation Matters

	Pro's	Cons
Dynamic	Precise Dynamic properties	Run-time cost Execution path analysis
Binary	Language independent Code coverage No source pollution	Platform dependent Traceableness

Why Dynamic Binary Instrumentation Matters

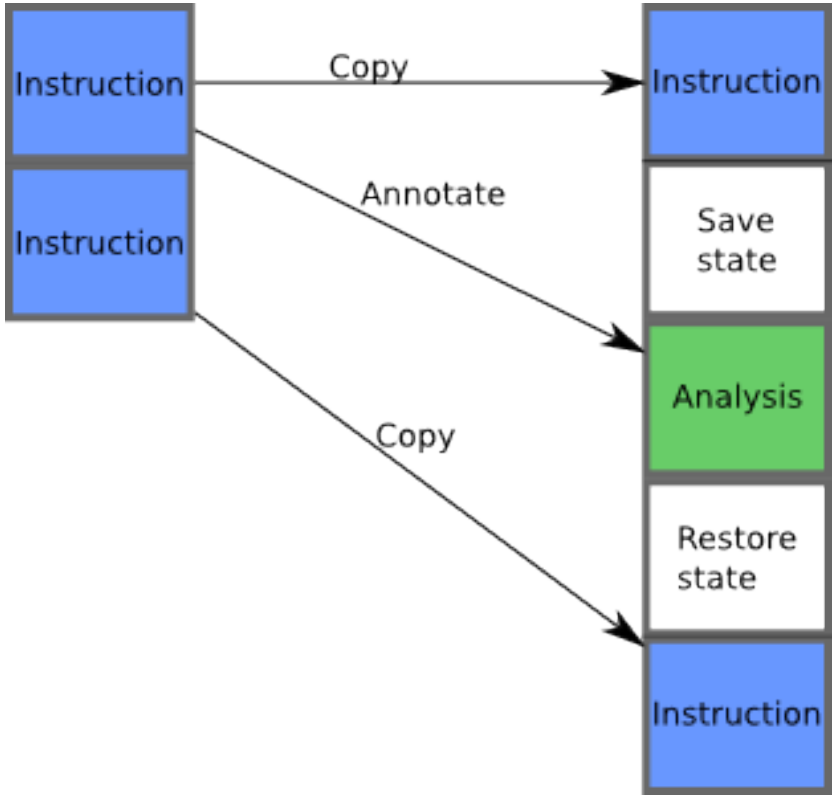
	Pro's	Cons
Dynamic	Precise Dynamic properties	Run-time cost Execution path analysis
Binary	Language independent Code coverage No source pollution	Platform dependent Traceableness

Dynamic analysis complements static analysis, just as binary analysis complements source analysis.

Binary Analysis Approaches

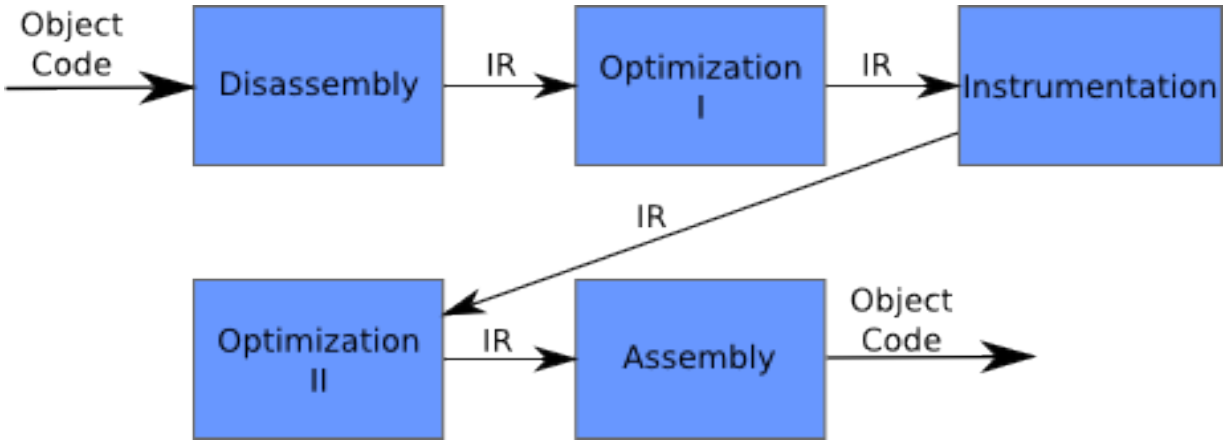
- Binary/Assembly difficult to analyze.
- Example: x86[-64]
  - Many opcodes with inconsistent behavior.
  - Complex addressing.
  - Implicit side-effects.
- Two options:
  - Copy and annotate
  - Disassemble and resynthesize

Copy and Annotate



- Verbatim copy program, generate annotations for analysis.
  - Efficient code stays efficient.
  - Analysis code unoptimized.

Disassemble And Resynthesize



- Transform program to IR, analyze and transform back to executable code.
- Valgrind based on disassemble and resynthesize.
- Similar to Java's JVM.
  - Compiles to a virtual RISC CPU. All client code runs on this CPU.
  - Conventional compiler optimizations.
- Disassemble and resynthesize efficient for heavyweight instrumentation.

Unboxing Example - Introduction

Goal of the Example

This example demonstrates the process the disassemble and resynthesize of a plain C function by Valgrind. Memcheck performs the instrumentation.

Code convention

- **Virtual registers** in upper case (EBP, EAX, ESP).
- **Physical registers** in lower case (%ebp, %eax, %esp).
- **Temporary registers** are in the form t*n*.

Unboxing Example - Introduction

- In the Essential Haskell Compiler[1] integers are unboxed.
- Pointer alignment at 32 or 64 bits, last 2 or 3 bits always zero.
- Left-shift and set the LSB to distinguish unboxed value.

[1] Atze Dijkstra. EHC Web. `http://www.cs.uu.nl/wiki/Ehc/WebHome`, 2004.

Unboxing Example - Introduction

- In the Essential Haskell Compiler[1] integers are unboxed.
- Pointer alignment at 32 or 64 bits, last 2 or 3 bits always zero.
- Left-shift and set the LSB to distinguish unboxed value.

C code to box an integer

```
int box(int ubxInt)
{
  return (ubxInt << 1) | 0x01;
}
```

[1] Atze Dijkstra. EHC Web. http://www.cs.uu.nl/wiki/Ehc/WebHome, 2004.

Unboxing Example - Introduction

- In the Essential Haskell Compiler[1] integers are unboxed.
- Pointer alignment at 32 or 64 bits, last 2 or 3 bits always zero.
- Left-shift and set the LSB to distinguish unboxed value.

C code to box an integer

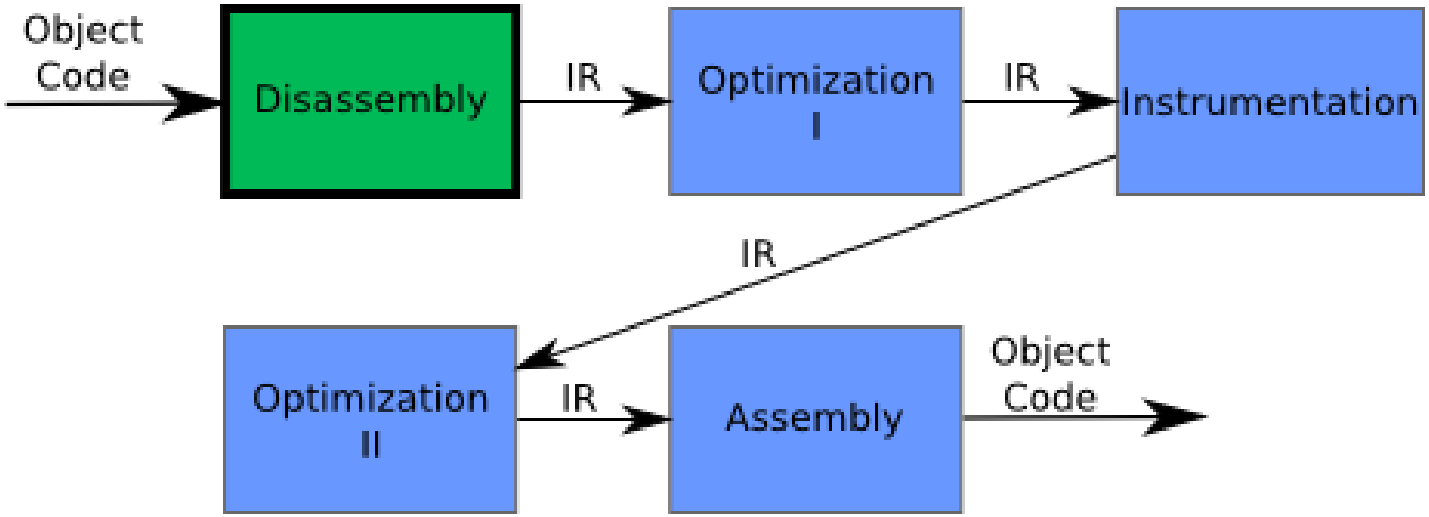
```
int box(int ubxInt)
{
  return (ubxInt << 1) | 0x01;
}
```

ASM output (GCC -O0)

```
...
mov     0x8(%ebp),%eax
add     %eax,%eax
or      $0x1,%eax
...
```

[1] Atze Dijkstra. EHC Web. http://www.cs.uu.nl/wiki/Ehc/WebHome, 2004.

Unboxing Example - Disassembly



Purpose:

- Introduces the virtual RISC CPU.
- Makes side-effects explicit.

Unboxing Example - Disassem

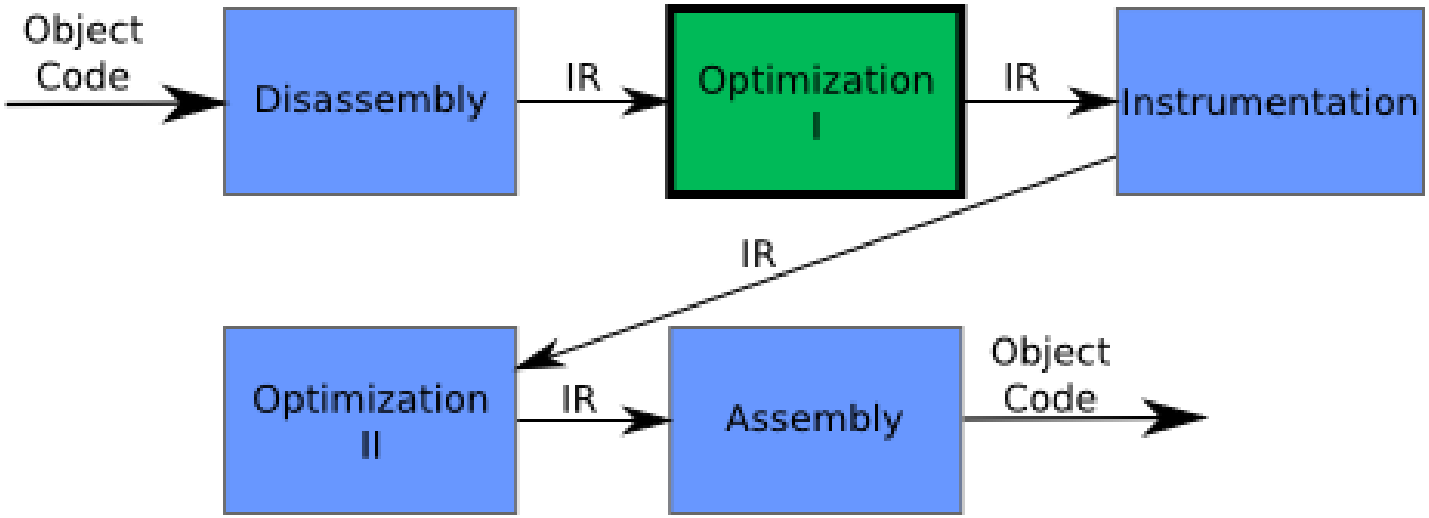
Object code

```
...
mov     0x8(%ebp),%eax
add     %eax,%eax
or      $0x1,%eax
...
```

IR

```
------ IMark(0x8048347, 3) ------
PUT(EIP) = 0x8048347:I32
t19 = Add32(GET:I32(EBP),0x8:I32)
PUT(EAX) = LDle:I32(t19)
------ IMark(0x804834A, 2) ------
PUT(EIP) = 0x804834A:I32
t22 = GET:I32(EAX)
t21 = GET:I32(EAX)
t20 = Add32(t22,t21)
PUT(CC_OP) = 0x3:I32
PUT(CC_DEP1) = t22
PUT(CC_DEP2) = t21
PUT(CC_NDEP) = 0x0:I32
PUT(EAX) = t20
------ IMark(0x804834C, 3) ------
PUT(EIP) = 0x804834C:I32
t25 = GET:I32(EAX)
t24 = 0x1:I32
t23 = Or32(t25,t24)
PUT(CC_OP) = 0xF:I32
PUT(CC_DEP1) = t23
PUT(CC_DEP2) = 0x0:I32
PUT(CC_NDEP) = 0x0:I32
PUT(EAX) = t23
```

Unboxing Example - Optimization 1



- Purpose: Optimize IR for easier instrumentation.
- Conventional compiler optimizations.
  - Dead code elimination.
  - Copy propagation.
  - Constant folding.
  - Common sub-expression elimination.

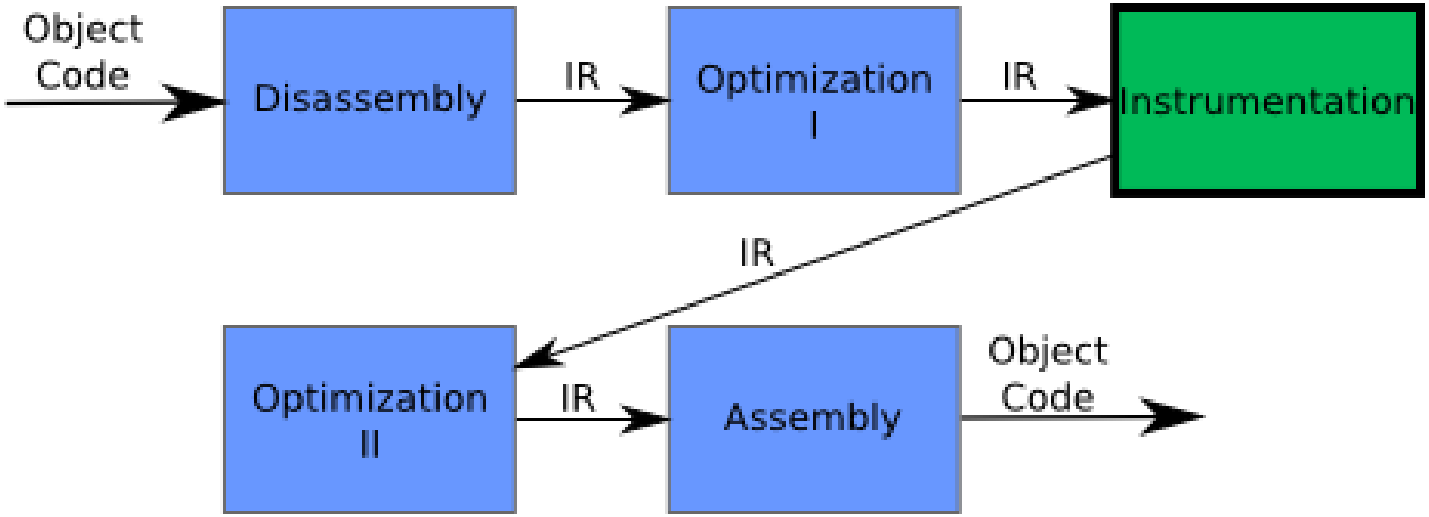
IR

```
------ IMark(0x8048347, 3) ------
PUT(EIP) = 0x8048347:I32
t19 = Add32(GET:I32(EBP),0x8:I32)
PUT(EAX) = LDle:I32(t19)
------ IMark(0x804834A, 2) ------
PUT(EIP) = 0x804834A:I32
t22 = GET:I32(EAX)
t21 = GET:I32(EAX)
t20 = Add32(t22,t21)
PUT(CC_OP) = 0x3:I32
PUT(CC_DEP1) = t22
PUT(CC_DEP2) = t21
PUT(CC_NDEP) = 0x0:I32
PUT(EAX) = t20
------ IMark(0x804834C, 3) ------
PUT(EIP) = 0x804834C:I32
t25 = GET:I32(EAX)
t24 = 0x1:I32
t23 = Or32(t25,t24)
PUT(CC_OP) = 0xF:I32
PUT(CC_DEP1) = t23
PUT(CC_DEP2) = 0x0:I32
PUT(CC_NDEP) = 0x0:I32
PUT(EAX) = t23
```

Optimized IR

```
------ IMark(0x8048347, 3) ------
PUT(EIP) = 0x8048347:I32
t48 = Add32(t45,0x8:I32)
t50 = LDle:I32(t48)
------ IMark(0x804834A, 2) ------

t20 = Shl32(t50,0x1:I8)

------ IMark(0x804834C, 3) ------

t23 = Or32(t20,0x1:I32)
PUT(CC_OP) = 0xF:I32
PUT(CC_DEP1) = t23
PUT(CC_DEP2) = 0x0:I32
PUT(CC_NDEP) = 0x0:I32
PUT(EAX) = t23
```

Unboxing Example - Instrumentation



- Purpose: Add analysis to the executable.
- Phase not performed by framework.

Unboxing Example - Instru

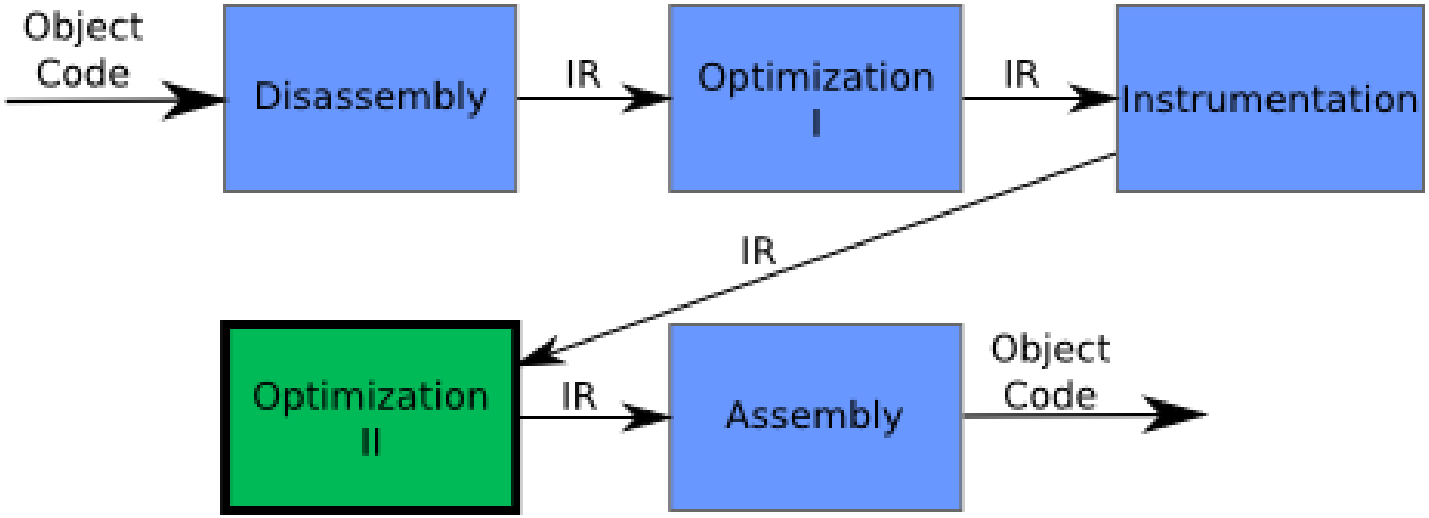
Optimized IR

```
------ IMark(0x8048347, 3) ------
PUT(EIP) = 0x8048347:I32
t48 = Add32(t45,0x8:I32)
t50 = LDle:I32(t48)
------ IMark(0x804834A, 2) ------
t20 = Shl32(t50,0x1:I8)
------ IMark(0x804834C, 3) ------
t23 = Or32(t20,0x1:I32)
PUT(CC_OP) = 0xF:I32
PUT(CC_DEP1) = t23
PUT(CC_DEP2) = 0x0:I32
PUT(CC_NDEP) = 0x0:I32
PUT(EAX) = t23
```

Instrumented IR (Memcheck)

```
------ IMark(0x8048347, 3) ------
PUT(EIP) = 0x8048347:I32
t108 = Left32(t106)
t48 = Add32(t45,0x8:I32)
t109 = CmpNEZ32(t108)
t112 = DIRTY 1:I1 RdFX-gst(16,4)...
t50 = LDle:I32(t48)
------ IMark(0x804834A, 2) ------
t113 = CmpNEZ8(0x0:I8)
t114 = 1Sto32(t113)
t115 = Shl32(t112,0x1:I8)
t116 = Or32(t115,t114)
t118 = Or32(t115,t114)
t20 = Shl32(t50,0x1:I8)
------ IMark(0x804834C, 3) ------
t119 = Not32(0x1:I32)
t121 = Not32(t20)
t122 = Or32(t121,t118)
t123 = And32(t122,t119)
t127 = And32(t118,t123)
t23 = Or32(t20,0x1:I32)
PUT(CC_OP) = 0xF:I32
PUT(CC_DEP1) = t23
PUT(CC_DEP2) = 0x0:I32
PUT(CC_NDEP) = 0x0:I32
PUT(EAX) = t23
```

Unboxing Example - Optimization 2



- Purpose: Optimize the instrumentation code.

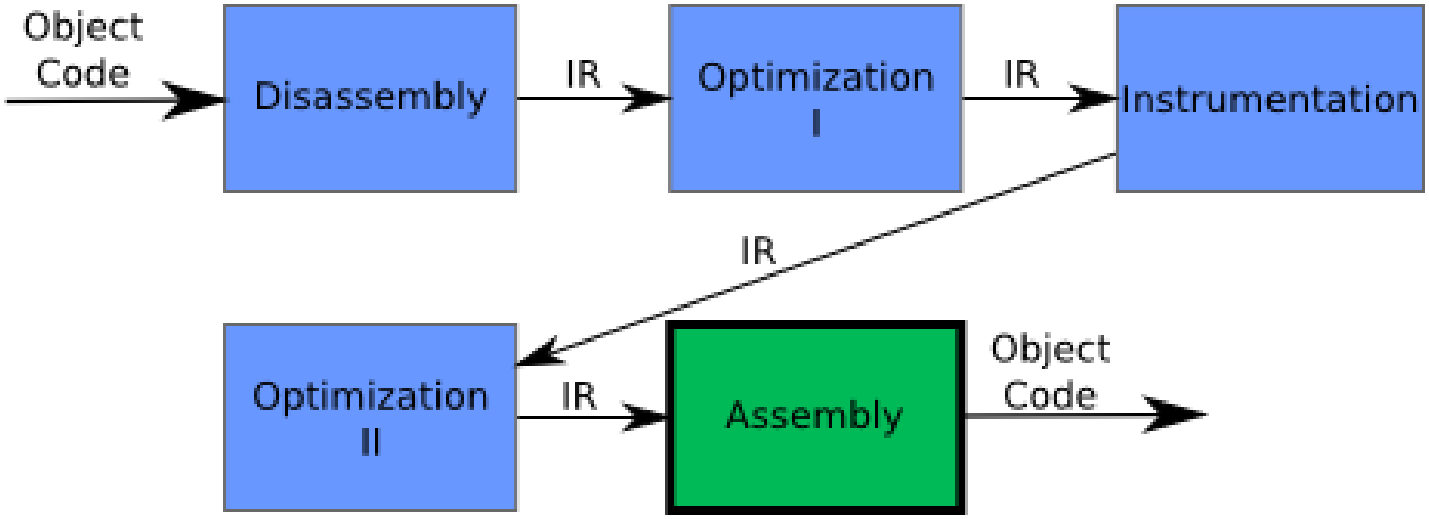
Instrumented IR (Memcheck)

```
------ IMark(0x8048347, 3) ------
PUT(EIP) = 0x8048347:I32
t108 = Left32(t106)
t48 = Add32(t45,0x8:I32)
t109 = CmpNEZ32(t108)
t112 = DIRTY 1:I1 RdFX-gst(16,4)...
t50 = LDle:I32(t48)
------ IMark(0x804834A, 2) ------
t113 = CmpNEZ8(0x0:I8)
t114 = 1Sto32(t113)
t115 = Shl32(t112,0x1:I8)
t116 = Or32(t115,t114)
t118 = Or32(t115,t114)
t20 = Shl32(t50,0x1:I8)
------ IMark(0x804834C, 3) ------
t119 = Not32(0x1:I32)
t121 = Not32(t20)
t122 = Or32(t121,t118)
t123 = And32(t122,t119)
t127 = And32(t118,t123)
t23 = Or32(t20,0x1:I32)
PUT(CC_OP) = 0xF:I32
PUT(CC_DEP1) = t23
PUT(CC_DEP2) = 0x0:I32
PUT(CC_NDEP) = 0x0:I32
PUT(EAX) = t23
```

Optimized instrumented IR

```
------ IMark(0x8048347, 3) ------
PUT(EIP) = 0x8048347:I32

t48 = Add32(t45,0x8:I32)

t112 = DIRTY 1:I1 RdFX-gst(16,4) ...
t50 = LDle:I32(t48)
------ IMark(0x804834A, 2) ------



t112 = Shl32(t111,0x1:I8)



t20 = Shl32(t50,0x1:I8)
------ IMark(0x804834C, 3) ------

t121 = Not32(t20)
t122 = Or32(t121,t115)
t123 = And32(t122,0xFFFFFFFE:I32)
t125 = And32(t115,t123)
t23 = Or32(t20,0x1:I32)
PUT(CC_OP) = 0xF:I32
PUT(CC_DEP1) = t23
PUT(CC_DEP2) = 0x0:I32
PUT(CC_NDEP) = 0x0:I32
PUT(EAX) = t23
```

Unboxing Example - Assembly



- Purpose: Convert IR back to machine instructions.
- Select proper instructions.
- Allocate registers.
- Object code stored and executed.

Unboxing Example - Assembly

Optimized instrumented IR

```
------ IMark(0x8048347, 3) ------
PUT(EIP) = 0x8048347:I32
t48 = Add32(t45,0x8:I32)
t112 = DIRTY 1:I1 RdFX-gst(16,4) ...
t50 = LDle:I32(t48)
------ IMark(0x804834A, 2) ------
t112 = Shl32(t111,0x1:I8)
t20 = Shl32(t50,0x1:I8)


------ IMark(0x804834C, 3) ------
t121 = Not32(t20)
t122 = Or32(t121,t115)
t123 = And32(t122,0xFFFFFFFE:I32)
t125 = And32(t115,t123)
t23 = Or32(t20,0x1:I32)
PUT(CC_OP) = 0xF:I32
PUT(CC_DEP1) = t23
PUT(CC_DEP2) = 0x0:I32
PUT(CC_NDEP) = 0x0:I32
PUT(EAX) = t23
```

Object code

```
movl $0x8048347,0x3C(%ebp)
leal 0x8(%ebx),%esi
movl %esi,%eax
call 0x38006BB0
movl %eax,%edi

shll $1,%edi
movl (%esi),%edx
shll $1,%edx
movl %edx,%esi

notl %esi
orl  %edi,%esi
andl $0xFFFFFFFE,%esi
andl %esi,%edi
orl  $0x1,%edx
movl $0xF,0x20(%ebp)
movl %edx,0x24(%ebp)
movl $0x0,0x28(%ebp)
movl $0x0,0x2C(%ebp)
movl %edx,(%ebp)
```

Framework Features

Function overriding

- Functions can be overridden by a tool.
- Allows action on specific methods.
- Function can call the replaced function (wrapping).
- Internal memory management depends on function overriding.

Events

- Various call-back hooks are available.
- Control over memory allocations and stack usage.
- Essential for system calls support.
- Examples:
  - pre_mem_read, pre_mem_write, post_mem_write
  - new_mem_mmap, die_mem_munmap, new_mem_remap
  - new_mem_stack, die_mem_stack

Client requests

- Executable code can pass information to a Valgrind Tool.
- Request translates to no-op when native executed.
- Examples:
  - Set memory valid for memcheck.
  - Annotate code as self modifying.
  - Start/Stop instrumenting.

Creating a Valgrind Tool

5 Simple Steps

1. Download Valgrind source.
2. Copy the *none* directory.
3. Come up with a cool name and abbreviation.
4. Write 4 C functions.
5. `./configure && make && make install`

4 Simple Functions (And 1 Macro)

C Interface to Valgrind

```
// Macro to register init function
VG_DETERMINE_INTERFACE_VERSION(xyz_pre_clo_init)

// Function to register other 3 functions
static void xyz_pre_clo_init(void);

// Initialize
static void xyz_post_clo_init(void);

// Instrumentation function
static IRSB* xyz_instrument(VgCallbackClosure* closure,
                            IRSB* bbIn,
                            VexGuestLayout* layout,
                            VexGuestExtents* vge,
                            IRType gWordTy, IRType hWordTy);

// Clean up / output report
static void xyz_fini(Int exitcode);
```

Example - Branchcount Instrument Snipplet

Instrumentation code

```
IRSB* bbOut = deepCopyIRSB(bbIn);
switch( bbOut->jumpkind ) {

  case Ijk_Boring: {
    if( !exprIsConst( bbOut->next ) ) {

      // it is a indirect jump
      //
      di = unsafeIRDirty_0_N( 0, "countJumpIndir",
                              VG_(fnptr_to_fnentry)
                                ( &countJumpIndir ),
                              mkIRExprVec_0() );
      addStmtToIRSB( bbOut, IRStmt_Dirty(di) );
    }
    break;
  }
...
```

Example - Branchcount Instrument Snipplet

Instrumentation code

Example output

```
==14425== Jumps direct
==14425==   counted:          500,000,000 ( 14%)
==14425== Jumps indirect
==14425==   counted:          500,000,000 ( 14%)
==14425== Calls direct
==14425==   counted:          500,000,000 ( 14%)
==14425== Calls indirect
==14425==   counted:          500,000,000 ( 14%)
==14425== Returns
==14425==   counted:          1,000,000,000 ( 28%)
==14425== Conditionals
==14425==   counted:          500,000,000 ( 14%)
==14425== Other
==14425==   counted:          0 (  0%)
```

Conclusion

- Valgrind is a framework for dynamic binary instrumentation.
- 5 tools are distributed with the framework:
  - Memcheck.
  - Cachegrind.
  - Callgrind.
  - Massif.
  - Helgrind.
- Valgrind is based on disassemble and resynthesize.
- Developers can just focus on the analysis, very little boiler plate code is required.

Developer Resources

- Source code available at `http://www.valgrind.org/downloads/current.html`.
- Available on Linux for x86/x86-64/ppc32/ppc64.
- Source heavily documented:
  - Virtual registers: `VEX/pub/libvex_guest_<arch>.h`.
  - Options for optimizations: `VEX/pub/libvex.h`.
  - IRSB Struct: `VEX/pub/libvex_ir.h`.
- Questions? Try the valgrind-developer mailing list.

Happy Instrumenting!