Tools and Implementation

Xiangyu Zhang
Outline

- Dynamic analysis tools
- Binary Decision Diagram
- Tools for undeterministic executions
- Static analysis tools
Dynamic Analysis Tools

- Introduction
  - Static instrumentation vs. dynamic instrumentation

- How to implement a dynamic information flow
What Is Instrumentation

Max = 0;
for (p = head; p; p = p->next)
{
    printf("In loop\n");
    if (p->value > max)
    {
        printf("True branch\n");
        max = p->value;
    }
}

What Can Instrumentation Do?

- **Profiler for compiler optimization:**
  - Basic-block count
  - Value profile

- **Micro architectural study:**
  - Instrument branches to simulate branch predictors
  - Generate traces

- **Bug checking:**
  - Find references to uninitialized, unallocated address

- **Software tools that use instrumentation:**
  - Valgrind, Pin, Purify, ATOM, EEL, Diablo, …
Binary Instrumentation Is Dominant

- Libraries are a big pain for source code level instrumentation
  - Proprietary libraries: communication (MPI, PVM), linear algebra (NGA), database query (SQL libraries).

- Easily handle multi-lingual programs
  - Source code level instrumentation is heavily language dependent.
    - More complicated semantics

- Turning off compiler optimizations can maintain a almost perfect mapping from instructions to source code lines

- Worms and viruses are rarely provided with source code

- We will be talking about binary instrumentation only
  - Static
  - Dynamic
Static Instrumentation (Diablo)

*Reading Files*
- *.c*
- *.cpp*
- *.S*

*Compiling Files*
- gcc
- g++
- asm

*Intermediate Files*
- *.o*
- *.a*

*Linking/Generating Files*
- ld

*DIABLO Output Files*
- a.out
- b.out

*DIABLO Flow*
- Read Object Format
- Disassemble Bundles
- Construct ICFG
- Analyses/Optimizations
- Serialize ICFG
- Assemble Bundles
- Write Object Format
Static Instrumentation Characteristics

- Perform the instrumentation before the code is run
  - New binary = original binary + instrumentation
  - Raise binary to IR, transform IR, transfer back to binary

- All libraries are usually statically linked
  - The size of the binary is big

- Program representations are usually built from the binary
  - CFG
  - Call graph
  - PDG is hard to build from binary
    - Points-to analysis on binary is almost impossible
    - Simple DFA is possible
    - CDG is almost precise (PDG=CDG+DDG)
Dynamic Instrumentation - Valgrind

- Developed by Julian Seward at/around Cambridge University, UK
  - Google-O'Reilly Open Source Award for "Best Toolmaker" 2006
  - A merit (bronze) Open Source Award 2004

- Open source
  - works on x86, AMD64, PPC code

- Easy to execute, e.g.:
  - valgrind --tool=memcheck ls

- It becomes very popular
  - One of the two most popular dynamic instrumentation tools
    - Pin and Valgrind
  - Very good usability, extendibility, robust
    - 25MLOC
  - Mozilla, MIT, CMU-security, Me, and many other places

- Overhead is the problem
  - 5-10X slowdown without any instrumentation
Valgrind Infrastructure

Binary Code

Dispatcher

BB Decoder

BB Compiler

Trampoline

New BB

New pc

Input

Tool 1

Tool 2

......

Tool n

Instrumenter

Runtime

pc

BB

state
1: do {
2:     i = i + 1;
3:     s1;
4: } while (i < 2)
5: s2;
Valgrind Infrastructure

1: do {
2:     i=i+1;
3:     s1;
4: } while (i<2)
5: s2;

INPUT:

1: do {
2:     i=i+1;
3:     s1;
4: } while (i<2)

OUTPUT:
1: do {
2:     i = i + 1;
3:     s1;
4: } while (i < 2)
5: s2;

Valgrind Infrastructure

1: do {
2:     i = i + 1;
3:     s1;
4: } while (i < 2)
5: s2;

OUTPUT:

1: do {
   print("1")
2:     i = i + 1;
3:     s1;
4: } while (i < 2)
1: do {
2:     i=i+1;
3:     s1;
4: } while (i<2)
5: s2;

INPUT:
1: do {
2:     print("1")
3:     i=i+1;
4:     s1;
5: } while (i<2)

OUTPUT: 1 1
Valgrind Infrastructure

1: do {
2:     i=i+1;
3:     s1;
4: } while (i<2)
5: s2;

Output:
1: do {
2:     print("1")
3:     i=i+1;
4:     s1;
5:     s2;
} while (i<2)
Valgrind Infrastructure

Binary Code

Dispatcher

BB Decoder

BB Compiler

Trampoline

Input

VALGRIND CORE

Tool 1

Tool 2

Tool n

Runtime

Instrumenter

1: do {
2:     i=i+1;
3:     s1;
4: } while (i<2)
5: s2;

1: do {
   print("1")
   i=i+1;
   s1;
} while (i<2)

11

5: print ("5");
s2;

OUTPUT: 1 1

CS590F Software Reliability
1: do {
2:     i = i + 1;
3:     s1;
4: } while (i < 2)
5: s2;

OUTPUT:
1: do {
     print("1")
     i = i + 1;
     s1;
 } while (i < 2)
5: print("5");
 s2;
Dynamic Instrumentation Characteristics

- A trampoline is required.
- Does not require recompiling or relinking
  - Saves time: compile and link times are significant in real systems.
  - Can instrument without linking (relinking is not always possible).
- Dynamically turn on/off, change instrumentation
  - From t1-t2, I want to execute F’, t3-t4, I want F”
    - Can be done by invalidating the mapping in the dispatcher.
- Can instrument running programs (such as Web or database servers)
  - Production systems.
- Can instrument self-mutating code.
  - Obfuscation can be easily get around.
Dynamic Instrumentation Characteristics

- Overhead is high
  - Dispatching, indexing;
  - Dynamic instrumentation

- Usually does not provide program representations at run time
  - Hard to acquire
  - Unacceptable runtime overhead
  - Simple representations such as BB are provided
  - GET AROUND: combine with static tools
    - Diablo + valgrind
Case Study: Implement A Dynamic Information Flow System in Valgrind
Information Flow System

- IFS is important
  - Confidentiality at runtime = IFS
  - Tainted analysis = IFS
  - Memory reference errors detection = IFS
  - Data lineage system = IFS
  - Dynamic slicing is partly an IFS

- Essence of an IFS
  - A runtime abstract interpretation engine
    - Driven by the executed program path

- Implementation on Valgrind is surprisingly easy
  - Will see
Language and Abstract Model

- **Our binary (RISC)**
  - ADD r1 / #Imm, r2
  - LOAD [r1 / #Imm], r2
  - STORE r1, [r2 / #Imm]
  - MOV r1 / #Imm, r2
  - CALL r1
  - SYS_READ r1, r2
    - r1 is the starting address of the buffer, r2 is the size

- **Abstract state**
  - One bit, the security bit (tainted bit)
  - Prevent call at tainted value.
Implement A New Tool In Valgrind

- Use a template
  - The tool lackey is good candidate
  - Two parts to fill in
    - Instrumenter
    - Runtime

- Instrumenter
  - Initialization
  - Instrumentation
  - Finalization
  - System calls interception

- Runtime
  - Transfer functions
  - Memory management for abstract state
How to Store Abstract State

- Shadow memory
  - We need a mapping
    - Addr → Abstract State
    - Register → Abstract State

```c
typedef struct {
    UChar abits[65536];
} SecMap;

static SecMap* primary_map[65536];
static SecMap default_map;
```
typedef
struct {
    UChar abits[65536];
} SecMap;

static SecMap* primary_map[65536];
static SecMap default_map;

static void init_shadow_memory ( void )
{
    for (i = 0; i < 65536; i++)
        default_map.abits[i] = 0;
    for (i = 0; i < 65536; i++)
        primary_map[i] = &default_map;
}
typedef
   struct {
      UChar abits[65536];
   } SecMap;

static SecMap* primary_map[65536];
static SecMap default_map;

static void init_shadow_memory ( void )
{
   for (i = 0; i < 65536; i++)
      default_map.abits[i] = 0;
   for (i = 0; i < 65536; i++)
      primary_map[i] = &default_map;
}
static SecMap* alloc_secondary_map ()
{
   map =VG_(shadow_alloc)(sizeof(SecMap));
   for (i = 0; i < 65536; i++)
      map->abits[i] = 0;
   return map;
}
typedef
struct {
    UChar abits[65536];
} SecMap;

static SecMap* primary_map[65536];
static SecMap default_map;

static void init_shadow_memory ( void )
{
    for (i = 0; i < 65536; i++)
        default_map.abits[i] = 0;
    for (i = 0; i < 65536; i++)
        primary_map[i] = &default_map;
}

static SecMap* alloc_secondary_map ()
{
    map =VG_(shadow_alloc)(sizeof(SecMap));
    for (i = 0; i < 65536; i++)
        map->abits[i] = 0;
    return map;
}

void Accessible (addr)
{
    if (primary_map[(addr) >> 16] == default_map)
        primary_map[(addr) >> 16] = alloc_secondary_map(caller);
}
void SK_(pre_clo_init)(void)
{
    VG_(details_name) ("CS590F IFS");
    ...
    init_shadow_memory();
    ...
    VG_(needs_shadow_memory) ();
    VG_(needs_shadow_regs) ();
    ...
    VG_(register_noncompact_helper)((Addr) & RT_load);
    VG_(register_noncompact_helper)((Addr) & …);
    ...
}
Finalization

- EMPTY

```c
void SK_(fini)(Int exitcode)
{
}
```
UCodeBlock* SK_(instrument)(UCodeBlock* cb_in, …) {

    …
    UCodeBlock cb = VG_(setup_UCodeBlock)(…);
    …
    for (i = 0; i < VG_(get_num_instrs)(cb_in); i++) {
        u = VG_(get_instr)(cb_in, i);
        switch (u->opcode) {
            case LD:
                …
            case ST:
                …
            case MOV:
                …
            case ADD:
                …
            case CALL:
                …
        }
    }

    return cb;
}
Instrumentation & Runtime - LOAD

LD [r1], r2

SHADOW(r2) = SM(r1) | SHADOW(r1)

```
switch (u->opcode) {
    case LD:
        VG_(ccall_RR_R) (cb, (Addr) RT_load, u->r1, SHADOW(u->r1), SHADOW(U->r2))
    }
```

```
UChar RT_load (Addr r1, UChar sr1)
{
    UChar s_bit=primary_map[a >> 16][a && 0xffff];
    return (s_bit | sr1);
}
```
Instrumentation & Runtime - STORE

ST r1, [r2]

SM(r2)=SHADOW(r1) | SHADOW (r2)

switch (u->opcode) {
    case ST:
        VG_(ccall_RRR_0) (cb, (Addr) RT_store, u->r2, SHADOW (u->r1), SHADOW(u->r2);
    }

void RT_store (Addr a, UChar sr1, UChar sr2)
{
    UChar s_bit= sr1 | sr2;
    Accessible(a);
    primary_map[a >> 16][a && 0xffff]=s_bit;
}
Instrumentation & Runtime - MOV

```cpp
switch (u->opcode) {
    case MOV:
        ulnstr2(cb, MOV..., SHADOW(u->r1), ..., SHADOW(u->r2))
} 
```

MOV r1, r2

SHADOW(r2) = SHADOW (r1)
Instrumentation & Runtime - ADD

ADD r1, r2

SHADOW(r2) =  SHADOW (r1) | SHADOW (r2)

switch (u->opcode) {
  case ST:
    VG_(ccall_RR_R) (cb, (Addr) RT_add, SHADOW(u->r1), SHADOW (u->r2), SHADOW(u->r2);
  }

UChar RT_add (UChar sr1, UChar sr2)
{
  return sr1 | sr2;
}
switch (u->opcode) {
    case ST:
        VG_(ccall_R_0) (cb, (Addr) RT_call, SHADOW(u->r1));
    }

if (SHADOW(r1))  printf ("Please call CS590F")

UChar RT_call (UChar sr1)
{
    if (sr1) VG_(printf) ("Please call CS590F\n");
}
void * SK_(pre_syscall) (... UInt syscallno...) {
    ... 
    if (syscallno==SYSCALL_READ) {
        get_syscall_params (... , &r1, &r2,...);
        for (i=0;i<r2;i++) {
            a = &r1[i];
            Accessible(a);
            primary_map[a >> 16][a && 0xffff]=1;
        }
    }
    ...
}
Done!

- Let us run it through a buffer overflow exploit

```c
void (* F) ();
char A[2];
...
read(B, 256);
i=2;
A[i]=B[i];
...
(*F) ();
```
void (*F)();
char A[2];
...read(B, 256);
...
i=2;
...
A[i]=B[i];
...
(*F)();

Virtual Space

SM (r1[0-r2])=1

SM(&i)=SHADOW(r1)

Shadow Space
void (* F) ();
char A[2];
...
read(B, 256);
...

SHADOW(r2)=SM(r2) | SHADOW (r2)
r2=&B[2];
...
A[i]=B[i];
...

(*F)();

SM (r3)=SHADOW(r2) | SHADOW (r3)
r3=&A[2]

Virtual Space

Shadow Space

SHADOW(r2)=SM(r2) | SHADOW (r2)
r2=&B[2];
void (* F) ();
char A[2];
...
read(B, 256);
...
i=2;
...
A[i]=B[i];
...

(*F) ();

if (SHADOW(r1)) printf ("Call ...");
What Is Not Covered

- Information flow through control dependence
  - Valgrind is not able to handle
  - Valgrind + diablo

```c
p=getpassword( );
...
if (p=="zhang") {
    send (m);
}
```
Outline

- Dynamic analysis tools
- Binary Decision Diagram
- Tools for undeterministic executions
- Static analysis tools
Why BDD?

- It is an efficient representation for boolean functions
  - What can be represented by boolean functions?
    - Sets, relations, …
  - What is program analysis about (both static and dynamic)
    - Manipulating sets

- Existing applications
  - In PA
    - Points-to analysis
    - Dynamic slicing
    - Data lineage
    - Test prioritization (??)
  - Others
    - Circuit optimization
Points-to Analysis Using BDD

X:  a = new O();
Y:  b = new O();
Z:  c = new O();
a = b;
b = a;
c = b;

Points-to set:
{ (a,X) (b,Y) (c,Z) (a,Y) (b,X) (c,X) (c,Y) }

Unification based flow-insensitive analysis
BDD representation

- A BDD is a compact representation of a boolean function
- The points-to relations can be encoded into a boolean function

\[
\begin{align*}
  a &\rightarrow 00 & X &\rightarrow 00 \\
  b &\rightarrow 01 & Y &\rightarrow 01 \\
  c &\rightarrow 10 & Z &\rightarrow 10 \\
\end{align*}
\]

Domains: V H
\[
v_1v_0h_1h_0
\]
\[
(a,Y) \rightarrow 00\ 01
\]
BDD representation

\[
\begin{align*}
V & \quad H \\
\text{(a, X)} & \quad 00 00 \\
\text{(a, Y)} & \quad 00 01 \\
\text{(b, X)} & \quad 01 00 \\
\text{(b, Y)} & \quad 01 01 \\
\text{(c, X)} & \quad 10 00 \\
\text{(c, Y)} & \quad 10 01 \\
\text{(c, Z)} & \quad 10 10
\end{align*}
\]
BDD Representation

\[ v_1 v_0 h_1 h_0 \]

\begin{align*}
  a/X & \rightarrow 00 \\
  b/Y & \rightarrow 01 \\
  c/Z & \rightarrow 10 \\
\end{align*}
BDD Representation

\[ V \quad H \]
\[ v_1 v_0 h_1 h_0 \]

\begin{align*}
(a, X) & \rightarrow 00 00 \\
(a, Y) & \rightarrow 00 01 \\
(b, X) & \rightarrow 01 00 \\
(b, Y) & \rightarrow 01 01 \\
(c, X) & \rightarrow 10 00 \\
(c, Y) & \rightarrow 10 01 \\
(c, Z) & \rightarrow 10 10
\end{align*}
BDD Representation

\[
V \quad H
\]

\[
\begin{align*}
(a, X) & \rightarrow 00 00 \\
(a, Y) & \rightarrow 00 01 \\
(b, X) & \rightarrow 01 00 \\
(b, Y) & \rightarrow 01 01 \\
(c, X) & \rightarrow 10 00 \\
(c, Y) & \rightarrow 10 01 \\
(c, Z) & \rightarrow 10 10
\end{align*}
\]
BDD Representation

\begin{align*}
\text{V} & \quad \text{H} \\
v_1v_0h_1h_0 & \\
(a,X) & \quad 00 \quad 00 \\
(a,Y) & \quad 00 \quad 01 \\
(b,X) & \quad 01 \quad 00 \\
(b,Y) & \quad 01 \quad 01 \\
(c,X) & \quad 10 \quad 00 \\
(c,Y) & \quad 10 \quad 01 \\
(c,Z) & \quad 10 \quad 10 
\end{align*}
Final Reduced BDD

\[ a/X \rightarrow 00 \]
\[ b/Y \rightarrow 01 \]
\[ c/Z \rightarrow 10 \]

\[
\begin{align*}
V & \quad H \\
v_1v_0h_1h_0 & \quad (a,X) \quad 00 \quad 00 \\
& \quad (a,Y) \quad 00 \quad 01 \\
& \quad (b,X) \quad 01 \quad 00 \\
& \quad (b,Y) \quad 01 \quad 01 \\
& \quad (c,X) \quad 10 \quad 00 \\
& \quad (c,Y) \quad 10 \quad 01 \\
& \quad (c,Z) \quad 10 \quad 10
\end{align*}
\]
BDD Operations

- Set operations
  - Union, intersection, ...

- Relational product

\[
\{(a, c) \mid \exists b . (a, b) \in X \land (b, c) \in Y\}\}
\]

- Cost of the operations is proportional to the number of nodes, not the elements in the set (relation)
Mapping Points-to Transfer Functions to BDD Operations

X: \( a = \text{new } 0(); \)
Y: \( b = \text{new } 0(); \)
Z: \( c = \text{new } 0(); \)

\( b = a; \)

Relational product rule

\[
\{(a, c) \mid \exists b. (a, b) \in X \land (b, c) \in Y\}\}
\]

\( (a, X) \quad (b, Y) \quad (c, Z) \)

\( (b, a) \quad (b, X) \)
What is Data Lineage
  • Given a value during the execution, the lineage of the value is the set of input that contributes to computation of the value.

BDD is the perfect choice for lineage sets
  • \( Z=X+Y \) \( \Rightarrow \) \( L(Z) = L(X) \cup L(Y) \)

BDD in dynamic slicing

BDD in ...

Tool
  • BuDDy
Outline

- Dynamic analysis tools
- Binary Decision Diagram
- Tools for undeterministic executions
- Static analysis tools
Jockey

- Execution record/replay tool (OPEN SOURCE)
  - X86 binaries
  - Used as a user-space library
  - Handle multi-threading programs
  - Checkpointing

- How it works
  - Use code pattern matching to identify all the system calls and replace them
  - Record phase
  - Replay phase
Simics-A Simulator

- full system simulation technology (NOT FULLY OPEN SOURCE)
  - the software cannot detect the difference between real production hardware and Simics' virtual environment.
  - Have the full control over the entire execution context
    - Application code
    - OS code
    - Driver code
  - Fast

- Widely used in multi-core related research
Outline

- Dynamic analysis tools
- Binary Decision Diagram
- Tools for undeterministic executions
- Static analysis tools
Static Analysis Tool

- Previously
  - SUIF
  - TRIMARAN

- Currently
  - CodeSurfer
  - CIL