A Compiler-level Intermediate Representation based Binary Analysis and Rewriting System

Kapil Anand
Matthew Smithson
Khaled Elwazeer
Aparna Kotha
Jim Gruen
Nathan Giles
Rajeev Barua
Binary Framework

High-level language program (C, C++,…….)

Compiler

Binary executable program

Binary executable program

Binary Framework

Improved binary executable program

Analysis
Applications

• Analyses of vulnerable code

• Analyses of untrusted code

• End-user security enforcements
  – Custom security policies

• Platform specific optimizations
  – Memory hierarchy
  – Multimedia instructions
Advantages of binary frameworks

• Absence of source code
  – IP-protected and legacy binaries
  – Hand-coded assembly

• What-You-See-Is-Not-What-You-Execute
  – Balakrishnan and Reps, 2007
Current Scenario

• Mainly employed as instrumentation tools

• Significantly lag behind compilers in capability
  – No complex analyses

• Known limitation
  – Executables have limited semantics as compared to source-code
Limitations of existing frameworks

• Dynamic
  – Difficult to perform complex analyses

• Capability
  – Define their own intermediate representation
  – Significantly constrained

• Functionality
  – Unable to obtain a functional representation

• Practicality
  – Use meta-data information
  – Not available in commercial binaries
Compiler-level IR

- Properties
  - Abstract Stack: Unconstrained representation
  - Symbols: Richer representation
Physical stack v/s Abstract stack

Abstract Stack

```c
foo( )
{
    bar(10);
}

bar (int a)
{
    int d,e;
    d = a;
    e = d;
}
```

Physical Stack

```c
foo:
    sub %esp, %esp, 4
    mov 10, (%esp)
    call bar

bar:
    sub %esp, %esp, 8
    mov 12(%esp), 4(%esp)
    mov 4(%esp), (%esp)
```

Disables compiler transformations
Examples of disallowed transformations

- Security Enforcements
  - Stack canary for buffer overflow protection
Compiler-level IR

• Properties
  – Abstract Stack: Unconstrained representation
  – Symbols: Richer representation
Examples of new formulation

- Lack of symbols ⇒ New formulations for standard compiler analyses

<table>
<thead>
<tr>
<th></th>
<th>Binary code without symbols</th>
<th>High level code with symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>eax = add ebx, ecx</code></td>
<td><code>eax = add ebx, ecx</code></td>
</tr>
<tr>
<td>2</td>
<td><code>store eax, [mem]</code></td>
<td><code>mov eax, sym</code></td>
</tr>
<tr>
<td>3</td>
<td><code>....</code></td>
<td><code>....</code></td>
</tr>
<tr>
<td>4</td>
<td><code>ecx = add [mem], ecx</code></td>
<td><code>ecx = add eax, ecx</code></td>
</tr>
</tbody>
</table>

Load-Store Forwarding

Copy Propagation
Symbolic Execution

• Several challenges
  – Symbolic memory addresses (user input)

• Source-level symbolic execution: Constraint solvers
  – KLEE

• Executables
  – Flat-memory
  – Existing approaches ignore symbolic accesses (BitBlaze)
  – New memory models (S2E, Mayhem)

• Our approach
  – Segmented memory representation
  – Only consider the constraints related to current segments
  – Improved efficiency of constraint solvers
SecondWrite Flow

EXISTING LLVM FLOW


OUR ADDITIONS

Original Input Binary → Binary FrontEnd → LLVM IR Optimizations → Richer LLVM → Binary Layout Modifications

Format Library

ISA XML

- Stack abstraction
- Symbol promotion
- Binary Aware Optimizations

Symbolic Execution → LLVM CodeGen → Binary

Vulnerabilities
Assumptions

- Compilation Model (Reps, 2004)
- No Self Modifying/Obfuscated Code
Obtaining compiler IR

- Convert physical stack to abstract stack
- Convert memory accesses to symbolic accesses
Stack deconstruction

Procedure Call

Caller Callee

ESP-relative accesses

Caller

Procedure Call

Callee

Physical stack frame Independent abstract frames per procedure
Argument discovery

• Existing binary analysis based on direct stack offsets
  – Stack base at entry points: Zero Base
  – Positive stack offsets: Arguments

• Guaranteeing discovery of all arguments impossible
  – Indirect memory argument references
  – Undiscovered inter-procedural stack accesses

• Existing frameworks
  – Preserve the monolithic unmodified stack
  – Circumvent the problem
Example

C code

```c
foo(int a, int b) {
    int *p, *q;
    p = &a;
    *q = ...;
    ... = b;
}
```

Assembly code

```
1 lea 20(%esp), 8(%esp)
2 store ..., (%edx)
3 load 8(%esp),%ecx
4 load 4(%ecx)
```

- Compiler
  - q and p do not alias
  - q: edx
  - &b = &a + 4
- Binary
  - Aliasing of q and p not determinable
  - Argument “a”: Recognizable
  - Argument “b”: Statically unrecognizable
Solution

• Combined static-dynamic solution
  – Apply existing methods for recognizing arguments
  – Employ runtime checks for remaining scenarios
Run-time check

Run-TimeCheck:
if (unknown - CurStack) < Size)
   No translation
else
   Translate to parent-stack-arg

Procedure Call (parent-stack-arg)

Unknown Access

Run-time check

Translated Access

Callee

CurStack

Size
Obtaining compiler IR

- Convert physical stack to abstract stack
- Convert memory accesses to symbolic accesses
Recognition of memory locations

• IDAPro
  – Only recognizes directly accessed stack locations
    • mov %eax, 10(%esp)

• Divine
  – Analyzes indirect memory accesses
  – Recognizes arrays and variables

• Identified variables cannot be promoted to symbols
  – Aliasing indirect memory accesses
Identified Variables

<table>
<thead>
<tr>
<th>C code</th>
<th>Assembly code</th>
</tr>
</thead>
</table>
| main() {
  int A[10], i, x;
  x = read-from-file();
  for (i = 0; i < x; i++) {
    A[i] = 10
  }
} | main:
1  main:
2  %ebx = read_from_file
3  mov  %ebx, 40(%esp)  //Initializing x
4  mov  0, %eax    //Initializing i
5  
6  Loop:
7  mov $10, (%esp,%eax,4)  //Reference A[i]
8  add $1, %eax  //Increment i
9  .... |

- x : esp + 40
## Aliasing Accesses

### C code

```c
main() {
    int A[10], i, x;
    x = read-from-file();
    for (i = 0; i < x; i++) {
        A[i] = 10
    }
}
```

### Assembly code

```
1  main:
2   %ebx = read_from_file
3   mov %ebx, 40(%esp)  // Initializing x
4   mov 0, %eax        // Initializing i
5   ...
6  Loop:
7   mov $10, (%esp,%eax,4)  // Reference A[i]
8   add $1, %eax         // Increment i
9   ...
```

- **ValueSet (x) = TOP (Unknown)**
- **ValueSet (%eax) = ValueSet(i) = \{0, x\} = TOP**

- **Instruction 7:**
  
  \[
  \text{ValueSet (address) = ValueSet (%esp + 4*%eax)} = %esp + \text{TOP}
  \]
Unsafe promotion of identified variables

<table>
<thead>
<tr>
<th>C code</th>
<th>Assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>main() {</td>
<td>1  main:</td>
</tr>
<tr>
<td>int A[10], i, x;</td>
<td>%ebx = read_from_file</td>
</tr>
<tr>
<td>x = read-from-file();</td>
<td>mov  %ebx, mem_40</td>
</tr>
<tr>
<td>for (i = 0; i &lt; x; i++) {</td>
<td>//Initializing x</td>
</tr>
<tr>
<td>A[i] = 10</td>
<td>mov  0, %eax</td>
</tr>
<tr>
<td>}</td>
<td>//Initializing i</td>
</tr>
<tr>
<td>}</td>
<td>6  Loop:</td>
</tr>
<tr>
<td></td>
<td>7  mov $10, (%esp,%eax,4)</td>
</tr>
<tr>
<td></td>
<td>//Reference A[i]</td>
</tr>
<tr>
<td></td>
<td>8  add $1, %eax</td>
</tr>
<tr>
<td></td>
<td>//Increment i</td>
</tr>
<tr>
<td></td>
<td>9  ....</td>
</tr>
</tbody>
</table>

- Symbol promotion at Instruction 3
- Safe symbol promotion:
  - Handle aliasing accesses
  - Maintain all dataflow edges

UNSAFE
# Aliasing accesses

<table>
<thead>
<tr>
<th>1. store eax, ebx[esi]</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. load 8[esp], edx</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>3. store ecx, 8[esp]</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>4. load 8[esp], edi</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>5. load ebx[esi], edx</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1. store eax, ebx[esi]</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. <strong>mov</strong> sym, edx</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>3. <strong>mov</strong> ecx, sym</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>4. <strong>mov</strong> sym, edi</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>store</strong> sym, 8[esp]</td>
</tr>
<tr>
<td>5. load ebx[esi], edx</td>
</tr>
</tbody>
</table>
Suboptimal Promotion

- Promoting load after Instruction 4
- Promoting store before Instruction 3 and 6
- Three symbol promotion require three promoting memory accesses
Partition-based symbol promotion

• Problem:
  – Whole program based solution
  – More fine grained solution space is required

• Solution:
  – Divide the program into a set of non-overlapping promotional lifetimes
  – Symbol promotion decision independently for each such lifetime

• Property:
  – Promote a lifetime only if provably beneficial
SecondWrite

• **X86 ISA** support
  – Two compilers (gcc and Visual Studio)
  – Two OS (Linux and Windows)
  – Three languages (C,C++,F)
  – Spec and OMP Benchmark
  – Real programs: Apache server, Linux Coreutils, lynx browser, tar and zip utilities
Scalability

![Scalability Graph](image-url)
Abstract Stack

• Run-time check
  – Required in only one program (gcc)
  – Less than 1% of procedures
Original symbolic accesses
Symbolic Execution

- KLEE for binaries
  - Original KLEE found bugs in coreutils source-code
  - Identified same bugs in coreutils binaries

<table>
<thead>
<tr>
<th>Program</th>
<th>No Promotion</th>
<th>With Promotion</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time(s)</td>
<td>STP Time(s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>htget</td>
<td>300</td>
<td>186</td>
</tr>
<tr>
<td>cut</td>
<td>300</td>
<td>252</td>
</tr>
<tr>
<td>split</td>
<td>300</td>
<td>225</td>
</tr>
</tbody>
</table>

3X improvement
Correct Functionality

- 1.0 = Runtime of input binary
- Average Improvement:
  - 40% (Unoptimized – O0)
  - 7% (Optimized - O3)
Runtime Impact of symbol promotion

- Average Improvement increases from 1% to 7%
Summary

• Translation of executables to compiler-IR in a static rewriting framework with no metadata.

• Allows application of source-level research to executables

• Limitations but highly valuable
Backup
Things remaining

• AVG different colour
• Speaking
  – Motivation of stack and symbols
  – Simplify description of IDA and Divine
  – Focus on functionality more
  – Say some names in limitations
  – Assumptions – tone down
  – LLVM, KLEE – stress more
  – Remind ppl about four characteristics
## Current Scenario

<table>
<thead>
<tr>
<th>Property</th>
<th>Functional</th>
<th>Capability</th>
<th>Practical</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATOM (Link time)</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>PLTO (Link time)</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Spike (Link time)</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>UQBT</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>IDA Pro / Hex Rays</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Jakstab</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>BAP (TIE)</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>CodeSurfer/X86</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td><strong>SecondWrite</strong></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>