Stack Memory Abstraction and Symbolic Analysis Framework for Executables

Kapil Anand
Khaled Elwazeer
Aparna Kotha
Matthew Smithson
Rajeev Barua
Angelos Keromytis
Binary Framework

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

Compiler

Binary executable program

Binary executable program

Improved binary executable program

Binary Framework

Analysis
Applications

• Analyses of vulnerable code

• Malware Analysis

• End-user security enforcements
  – Custom security policies

• Platform specific optimizations
  – Memory hierarchy
  – Multimedia instructions
Existing Binary Tools

• Significantly lag behind compilers in capability
  – No complex analyses
Goals

• **Static analysis**
  – Limitations of dynamic frameworks
    • No time to do complex analyses
    • Limited code coverage

• **Functionality**
  – Ability to rewrite code correctly and obtain functional representations

• **High-level Intermediate Representation (IR)**
  – No extra constraints in comparison to source-code

• **Practicality**
  – No use of meta-data information

• **Scalability**
  – Applicable to real-world programs
Assumptions

• Compilation memory model (Reps, 2004)

• No self modifying/dynamically generated code

• No obfuscated code

• Disassembly assumptions
  – No calculated addresses (WCRE, 2013)
EXISTING LLVM COMPILER

C
C++
Fortran
...

LLVM front end

LLVM IR

LLVM IR Analysis

Optimized LLVM IR

x86 backend

Output binary

Output C code

Our New Code

Richer LLVM IR

Binary reader & Disassembler

Input binary

LLVM IR

Symbolic Analysis Framework

Redundancy Elimination

Dependence Analysis

Security Vulnerabilities

Alias Analysis

Binary Analysis Tools

x86 ISA
XML
Challenges

• Underlying analyses are not capable to reach source-level analyses.

• Presence of executable artifacts limits representation and functionality.
Analysis in Executables

• **Source-level frameworks**
  – Capability: Precision of underlying analyses
  – Only analyze instructions involving program variables.

• **Existing binary frameworks**
  – Ignore memory models

• **Example: Symbolic Analysis**
  • Represents the values of program variables as symbolic expressions
  • Applications: Value numbering, dependence
**Limitations of source level analysis:**

**Symbolic Analysis**

```
int main(){
    int a,b,d;
    ....
    b=a+2;
    ......
    d=b+10;
}
```

**Symbolic Relations:**

```
b=a+2
d=a+12
```

<table>
<thead>
<tr>
<th>Allocations:</th>
<th>No Memory abstraction</th>
<th>With Memory abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: -4(%ebp)</td>
<td>x1</td>
<td>x1</td>
</tr>
<tr>
<td>b: -8(%ebp)</td>
<td></td>
<td>x1+2</td>
</tr>
<tr>
<td>d: -16(%ebp)</td>
<td></td>
<td>x1+2</td>
</tr>
</tbody>
</table>

```
main:
1 mov $esp,$ebp
2 sub 24,$esp //Local Allocation
3 mov -4(%ebp), %eax //Load a
4 add $2, %eax    //Compute a+2
5 mov %eax, -8(%ebp) //Store b
...
6 mov -8(%ebp), %eax //Load b
7 add $10, %eax    //Compute b+10
8 mov %eax, -16(%ebp) //Store d
```

```
Symbolic Relations:
b=a+2
d=a+12
```
Memory Abstraction

• **Stack Memory abstraction in executables**
  – Associating memory reference to a set of locations on stack.

• **Need variable like entities in executables**
  • *A-loc abstraction*: (Balakrishnan and Reps, 2004)
  • Offset and size of the a-loc
Limitations to a-locs

StackDiff: Stack modification due to each instruction
• Not statically determinable in all scenarios
Executable artifacts

- StackDiff Indeterminable: *Unknown Calls*
  - A direct call to an external procedure with unknown prototype.
  - An indirect call with unresolved targets.
  - An indirect call and its targets have different *StackDiff*. 
Existing solutions

- **Existing Solutions**
  - CodeSurfer/X86
    - No a-locs (variables) in such scenarios
    - Forfeits **Capability**
  - IDAPro:
    - Unsafe assumptions
    - Forfeits **Functionality**

- **Our approach: Hybrid mechanism for functionality and capability**
  - Static solution: **Capability**
  - Dynamic solution: **Functionality**
  - Formulate symbolic constraints based on control flow graph
Constraints Formulation

• Constraints Generation
  – $StackDiff$: Unknown stack modifications
  – $StackDepth$: Stack height at each instruction
  – Standard stack modification instructions are analyzed to derive an expression of $StackDepth$ in terms of $StackDiff$

• Boundary Conditions
  – $StackDepth$ from different paths at join points in CFG are same
    • Compiler accesses have to be deterministic
  – $StackDepth$ at return instruction is zero
    • Return oriented programming
Constraints Solution

• Possible solutions
  – All $StackDiff$ are available: Static solution
  – Some $StackDiff$ are available
  – None of the $StackDiff$ are available

• Dynamic solution
  – Declare $StackDiff$ as a return variable.
  – Dynamic adjustment of Stack Pointer in callee procedure
Example of static solution

Symbolic Equations:
\[ x_1 + 24 = x_2 + 24 \]

Solution:
\[ x_1 = 0 \]
\[ x_2 = 0 \]
Example of dynamic solution

StackDiff

\[ x_1 + x_2 \]

\[ x_1 + x_2 - 2 \]

Symbolic Equations:
\[ x_1 + x_2 = 2 \]

Static solution not possible

```
sub 24, %esp
mov $10, 8(%esp)
call *%eax
call *%ebx
sub 2,%esp
add 24,%esp
ret
```

```
sub 24, %esp
mov $10, 8(%esp)
y1=call *%eax
sub y1,%esp
y2=call *%ebx
sub y2,%esp
sub 2, %esp
add 24, %esp
ret
```
Symbolic Value Analysis

• SVA: A flow-sensitive context-insensitive interprocedural analysis

• Computes symbolic maps
  – Vars and a-locs

• Symbolic Grammar
  
  Sym := Sym+T | T
  T := T*F | F
  F := l | n
  l := [IR Variables]
  n := [Int]

• Symbolic Value Set: Finite set of symbolic expressions defined by the symbolic grammar

• A transfer instruction for each instruction in the IR
### Symbolic Value Analysis

```c
int main(){
    int a, b, d;
    //...
    b = a + 2;
    //...
    d = b + 10;
}
```

#### Symbolic Relations:
- \( b = a + 2 \)
- \( d = a + 12 \)

<table>
<thead>
<tr>
<th>Allocations:</th>
<th>No Memory Abstraction</th>
<th>With Memory Abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a ): -4(%ebp)</td>
<td>x1</td>
<td>x1</td>
</tr>
<tr>
<td>( b ): -8(%ebp)</td>
<td>x1 + 2</td>
<td>x1 + 2</td>
</tr>
<tr>
<td>( d ): -16(%ebp)</td>
<td>x3</td>
<td>x1 + 2</td>
</tr>
</tbody>
</table>

#### Code Analysis:

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Description</th>
<th>No Memory Abstraction</th>
<th>With Memory Abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mov $esp,$ebp</td>
<td>main:</td>
<td>x1</td>
<td>x1</td>
</tr>
<tr>
<td>2</td>
<td>sub 24,$esp</td>
<td>//Local Allocation</td>
<td>x1 + 2</td>
<td>x1 + 2</td>
</tr>
<tr>
<td>3</td>
<td>mov -4(%ebp), %eax</td>
<td>//Load a</td>
<td>x3</td>
<td>x1 + 2</td>
</tr>
<tr>
<td>4</td>
<td>add $2, %eax</td>
<td>//Compute a+2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>mov %eax, -8(%ebp)</td>
<td>//Store b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>mov -8(%ebp), %eax</td>
<td>//Load b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>add $10, %eax</td>
<td>//Compute b+10</td>
<td>x3+10</td>
<td>x1+12</td>
</tr>
<tr>
<td>8</td>
<td>mov %eax, -16(%ebp)</td>
<td>//Store d</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
• **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

• **Symbolic Framework**
  – Analysis Time
    • Within 1 minute for most SPEC benchmarks
  – Storage requirements within 500 MB
Hybrid solution

% of procedures containing unknown calls

Benchmarks

Static
Dynamic
Enhanced Memory Abstraction

% of new a-locs as compared to base a-locs

Benchmarks

- bzip2
- sjeng
- omnetpp
- soplex
- h264
- cactus
- deal
- povray
- perlbench
- gcc
- xalan
- GMEAN-C
- GMEAN-C++
Applications

• **Value numbering**
  – Redundant instructions in optimized executables
  – Simplifies the IR, speeding up subsequent binary analysis

• **Dependence Analysis**
  – Parallelizing compilers employ symbolic analysis in data dependence tests
  – Extension of data dependence tests to executables

• **Security Analysis**
  – Extension of SVA to detect information flow vulnerabilities
  – Format string and directory traversal
Value Numbering

Normalized # Equivalent computations

Without memory-based symbolic analysis
With memory-based symbolic analysis

Benchmarks:
bwaves, lbm, equake, mcf, art, wupw, libquant, leslie3d, namd, astar, bzip2, milc, sjeng, sphinx, zeusmp, omnetpp, hmmer, soplex, h264, cactus, gromacs, deal, calculix, povray, gobmk, perl, gcc, apache, GeoMean
## Security

<table>
<thead>
<tr>
<th>Programs</th>
<th>LOC</th>
<th>Known/New</th>
<th>Type of vulnerability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mingetty</td>
<td>500</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Csplit</td>
<td>1,060</td>
<td>NEW</td>
<td>Format String</td>
</tr>
<tr>
<td>muh</td>
<td>2857</td>
<td>KNOWN</td>
<td>Format String</td>
</tr>
<tr>
<td>pfingerd</td>
<td>4689</td>
<td>KNOWN</td>
<td>Format String</td>
</tr>
<tr>
<td>gzip</td>
<td>5830</td>
<td>KNOWN</td>
<td>Directory Traversal</td>
</tr>
<tr>
<td>ipupdate</td>
<td>6,335</td>
<td>KNOWN</td>
<td>Format String</td>
</tr>
<tr>
<td>gif2png</td>
<td>9354</td>
<td>KNOWN</td>
<td>Directory Traversal</td>
</tr>
<tr>
<td>wu-ftpdf</td>
<td>17576</td>
<td>KNOWN</td>
<td>Format String</td>
</tr>
<tr>
<td>tar</td>
<td>20518</td>
<td>KNOWN</td>
<td>Directory Traversal</td>
</tr>
<tr>
<td>KeePassX</td>
<td>26089</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>yafc</td>
<td>32,241</td>
<td>NEW</td>
<td>Directory Traversal</td>
</tr>
<tr>
<td>tnftp</td>
<td>34,762</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>gftp</td>
<td>42,390</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>irc2</td>
<td>44,837</td>
<td>NEW</td>
<td>Directory Traversal</td>
</tr>
<tr>
<td>wget</td>
<td>46,611</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>sudo</td>
<td>53,144</td>
<td>KNOWN</td>
<td>Format String</td>
</tr>
<tr>
<td>openssh</td>
<td>73335</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>ayttm</td>
<td>80,013</td>
<td>NEW</td>
<td>Format String</td>
</tr>
<tr>
<td>curl</td>
<td>122,248</td>
<td>NEW</td>
<td>Directory Traversal</td>
</tr>
<tr>
<td>bitchx</td>
<td>133,728</td>
<td>NEW</td>
<td>Format String</td>
</tr>
<tr>
<td>lynx</td>
<td>135876</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>apache</td>
<td>232778</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>mySQL</td>
<td>1.7 million</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

- Six New Vulnerabilities
- Eight Existing Vulnerabilities
- False Positive:
  - 79%
  - Source-level tools: 84%
Summary

• Improved memory abstraction model for executables

• Symbolic analysis framework for executables

• Employed the above framework for several applications
Advantages of binary frameworks

• **Absence of source code**
  – COTS executables and legacy binaries
  – Hand-coded assembly

• **Source code analysis not reliable**
    • Balakrishnan and Reps, 2007
Applications

• Value numbering
  – Redundant instructions in optimized executables
  – Simplifies the IR, speeding up subsequent binary analysis

• Dependence Analysis
  – Parallelizing compilers employ symbolic analysis in data dependence tests
  – Extension of data dependence tests to executables

• Security Analysis
  – Extension of SVA to detect information flow vulnerabilities
  – Format string and directory traversal
Security

<table>
<thead>
<tr>
<th>Programs</th>
<th>LOC</th>
<th>Known/New</th>
<th>Type of vulnerability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mingetty</td>
<td>500</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Csplit</td>
<td>1,060</td>
<td>NEW</td>
<td>Format String</td>
</tr>
<tr>
<td>muh</td>
<td>2857</td>
<td>KNOWN</td>
<td>Format String</td>
</tr>
<tr>
<td>pfingerd</td>
<td>4689</td>
<td>KNOWN</td>
<td>Format String</td>
</tr>
<tr>
<td>gzip</td>
<td>5830</td>
<td>KNOWN</td>
<td>Directory Traversal</td>
</tr>
<tr>
<td>ipupdate</td>
<td>6,335</td>
<td>KNOWN</td>
<td>Format String</td>
</tr>
<tr>
<td>gif2png</td>
<td>9354</td>
<td>KNOWN</td>
<td>Directory Traversal</td>
</tr>
<tr>
<td>wu-ftp</td>
<td>17576</td>
<td>KNOWN</td>
<td>Format String</td>
</tr>
<tr>
<td>tar</td>
<td>20518</td>
<td>KNOWN</td>
<td>Directory Traversal</td>
</tr>
<tr>
<td>KeePassX</td>
<td>26089</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>yafc</td>
<td>32,241</td>
<td>NEW</td>
<td>Directory Traversal</td>
</tr>
<tr>
<td>tnftp</td>
<td>34,762</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>gftp</td>
<td>42,390</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>irc2</td>
<td>44,837</td>
<td>NEW</td>
<td>Directory Traversal</td>
</tr>
<tr>
<td>wget</td>
<td>46,611</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>sudo</td>
<td>53,144</td>
<td>KNOWN</td>
<td>Format String</td>
</tr>
<tr>
<td>openssh</td>
<td>73335</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>ayttm</td>
<td>80,013</td>
<td>NEW</td>
<td>Format String</td>
</tr>
<tr>
<td>curl</td>
<td>122,248</td>
<td>NEW</td>
<td>Directory Traversal</td>
</tr>
<tr>
<td>bitchx</td>
<td>133,728</td>
<td>NEW</td>
<td>Format String</td>
</tr>
<tr>
<td>lynx</td>
<td>135876</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>apache</td>
<td>232778</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>mySQL</td>
<td>1.7 million</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

- Six New Vulnerabilities
- Eight Existing Vulnerabilities
- False Positive:
  - 79%
  - Source-level tools: 84%
## Existing Tools

<table>
<thead>
<tr>
<th>Property</th>
<th>Functional</th>
<th>High IR</th>
<th>Works without Metadata</th>
<th>Scalable</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATOM (Link time)</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>PLTO (Link time)</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Spike (Link time)</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>UQBT</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>IDA Pro / Hex Rays</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Jakstab</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>BAP (TIE)</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>CodeSurfer/X86</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>SecondWrite</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Disassembly (WCRE’2013)

- **Speculative disassembly**
  - Apply recursive disassembly
  - Speculatively assume unknown portions as code and disassemble
  - Retain unknown portion as data in IR to maintain the correctness
  - Indirect instruction: Translate to new location in IR

- **Binary characterization**
  - Limits the beginning points in “unknown portions”
  - Assumption: “An address is not computed”
  - Scan text and data segment to identify all possible entry points in code segment
## Transfer Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Operation</th>
<th>Transfer Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Assignment</td>
<td>$R_1 := R_2$</td>
<td>$SR = (SR - SR(R_1)) \cup {(R_1, SR(R_2))}$</td>
</tr>
<tr>
<td>2. Arithmetic</td>
<td>$R_3 := R_2 \text{OP} R_1$</td>
<td></td>
</tr>
</tbody>
</table>
  
  - if $\text{OP} = +$
    \[ \text{tmp} = \nabla (SR(R_2) \oplus SR(R_1)) \]
  
  - if $\text{OP} = \ast$
    \[ \text{tmp} = \nabla (SR(R_2) \otimes SR(R_1)) \]
  
  else  //Create a new symbolic expression
    \[ \text{tmp} = R_3 \]
    \[ SR = (SR - SR(R_3)) \cup \{(R_3, \text{tmp})\} \]
| 3. Load       | $R_1 := \ast(R_2)$         | 
  
  \[ \{F, P\} = \ast(\text{Mem}_e(R_2), s) \]
  
  if $|P| = 0$
    \[ \text{tmp} = \nabla (\bigcup_{v \in F} SM_e(v)) \]
  
  else
    \[ \text{tmp} = T \]
    \[ SR = (SR - SR(R_1)) \cup \{(R_1, \text{tmp})\} \]
| 4. Store      | $\ast(R_2) := R_1$         | 
  
  \[ \{F, P\} = \ast(\text{Mem}_e(R_2), s) \]
  
  if $|F| = 1$ & $|P| = 0$ & Func is not recursive & $F$ has no heap a-locs  //Strong Update
    \[ SM'_e = \{(SM_e - SM_e(v)) \cup \{(v, SR(R_1))\} | v \in F\} \]
  
  else  //Weak Update
    \[ SM'_e = \{(SM_e - SM_e(y) | y \in \{F \cup P\}) \cup \{(v, \nabla(SR(R_1) \cup SM_e(v))) | v \in F\} \cup \{(p, T) | p \in P\} \}
| 5. SSA Phi    | $R_{n+1} \phi(R_1, R_2, ..., R_n)$ | $SR = (SR - SR(R_{n+1})) \cup \{R_1, \nabla (\bigcup_{i \in (1, n)} SR(R_i))\}$ |
Unknown Symbolic Values: $X_I$, where $X_I = \text{StackDiff}$ of procedure call $I$

Initial/Helper Variables:
- $\text{Targ}(T)$: Set of procedures targeted by call target address $T$
- $\text{StackDiff}(f)$: StackDiff of procedure $f$
- $Y\_\text{SET}(F) = \bigcup_{f \in F} \text{StackDiff}(f)$
- $\text{BeginP} = \text{Entry point of procedure } P$; $\text{PredBB} = \text{Predecessors of basic block } BB$
- $\text{BeginBB}, \text{EndBB} = \text{Entry point, terminator of basic block } BB$
- $S_I = \text{Stack height after instruction } I$
- $S_{BB} = \text{Stack height at beginning of basic block } BB$
- $\text{PrevI} = \text{the previous instruction to } I$ ($I \neq \text{BeginBB}$)
- $S_{I'} = \text{if } (I \neq \text{BeginBB}) \text{ then } S_{\text{PrevI}} \text{ else } S_{BB}$
- $R$: A register, $\text{Size}(R)$: Size of register $R$, $N$: A constant

Initial Conditions: $S_{\text{BeginP}} = 0$

Data flow rules:

For every instruction $I$:

- $I = \text{push } R \Rightarrow S_I = S_{I'}, + \text{size}(R)$
- $I = \text{pop } R \Rightarrow S_I = S_{I'}, - \text{size}(R)$
- $I = \text{add esp, } N \Rightarrow S_I = S_{I'}, - N$
- $I = \text{sub esp, } N \Rightarrow S_I = S_{I'}, + N$
- $I = \text{jmp } L \Rightarrow S_{\text{BeginL}} = S_I$
- $I = \text{call } Y \Rightarrow$
  - if ($Y\_\text{SET}(\text{Targ}(Y))$ contains a single constant $C$)
    $$S_I = S_{I'}, + C$$
  - else
    $$S_I = S_{I'}, + X_I$$
- default (if not an invalidation condition) $\Rightarrow S_I = S_{I'}$

Boundary Conditions:
1. $\forall BB: \forall \text{Pred} \in \text{PredBB}, S_{\text{BeginBB}} = S_{\text{EndPred}}$
2. $I = \text{ret}: \text{Constraint } S_{I'} = 0$

Invalidation Conditions:
1. $I = \text{esp} \leftarrow ... /* \text{Any assignment except in data-flow rules}*/$
2. $I$ accesses return address
Symbolic Abstraction

- Data Objects:
  - Variables
  - A-locs

- Symbolic grammar

  \[ \text{Sym} := \text{Sym} + \text{T} | \text{T} \\
  \text{T} := \text{T} * \text{F} | \text{F} \\
  \text{F} := \text{l} | \text{n} \]

  \[ \text{l} := \text{[IR Variables]} \]

  \[ \text{n} := \text{[Int]} \]

- Symbolic Value Set: Finite set of symbolic expressions defined by the symbolic grammar