BDA: Practical Dependence Analysis for Binary Executables by Unbiased Whole-Program Path Sampling and Per-Path Abstract Interpretation

Zhuo Zhang, Wei You, Guanhong Tao, Guannan Wei, Yonghwí Kwon, and Xiangyu Zhang
Intro: Binary Program Dependence Analysis

- Determine data dependence between instructions in binary executables
Intro: Binary Program Dependence Analysis

• Determine data dependence between instructions in binary executables

```
1 ... 
2 a = 0;       1 ... 
2 mov [rax], 0x0
3 ... 
4 int b = a;   3 ... 
4 mov rbx, [rcx]
```
Intro: Binary Program Dependence Analysis

- Determine data dependence between instructions in binary executables
Intro: Binary Program Dependence Analysis

• Determine data dependence between instructions in binary executables

```
1 ...  
2 a = 0;  
3 ...  
4 int b = a;  
1 ...  
2 mov [rax], 0x0  
3 ...  
4 mov rbx, [rcx]
```
Intro: Binary Program Dependence Analysis

• Determine data dependence between instructions in binary executables

```c
1 ... 1 ...
2 a = 0; 2 mov [rax], 0x0
3 ... 3 ...
4 int b = a; 4 mov rbx, [rcx]
```
Intro: Binary Program Dependence Analysis

• Determine data dependence between instructions in binary executables

• Have many applications
  • Precise call graph construction
  • Malware analysis to expose hidden behaviors
  • Binary rewriting
  • ......
Intro: Binary Program Dependence Analysis

- Determine data dependence between instructions in binary executables
- Have many applications
- A key challenge is to identify if multiple memory read/write instructions access the same memory location

```
1 ...                      1 ...
2 a = 0;                   ?
3 ...                      2 mov [rax], 0x0  
4 int b = a;               3 ...                      ?
5 mov rbx, [rcx]           4 mov rbx, [rcx]
```
Intro: Binary Program Dependence Analysis

• Determine data dependence between instructions in binary executables

• Have many applications

• A key challenge is to identify if multiple memory read/write instructions access the same memory location

```assembly
1 ... ??
2 mov [rax], 0x0
3 ... ??
4 mov rbx, [rcx]
```
Intro: Existing Works

• The state-of-the-art: **Value Set Analysis (VSA)**
  • Integrated into a variety of binary analysis frameworks (Angr, BAP)
  • Compute a set of possible values for each operand of an instruction
Intro: Existing Works

• The state-of-the-art: Value Set Analysis (VSA)
  • Integrated into a variety of binary analysis frameworks (Angr, BAP)
  • Compute a set of possible values for each operand of an instruction
  • Use a *strided interval* to denote the set of values
Intro: Existing Works

• The state-of-the-art: **Value Set Analysis (VSA)**
  • Integrated into a variety of binary analysis frameworks (Angr, BAP)
  • Compute a set of possible values for each operand of an instruction
  • Use a *strided interval* to denote the set of values

\[ s[lb, ub] \]

lower bound

\[ s[lb, ub] \rightarrow \text{upper bound} \]

representing \{lb, lb+s, lb+2s, \ldots, ub\}

\[ 2[0, 8] \]

e.g., representing \{0, 2, 4, 6, 8\}
Intro: Existing Works

- The state-of-the-art: **Value Set Analysis (VSA)**
  - Integrated into a variety of binary analysis frameworks (Angr, BAP)
  - Compute a set of possible values for each operand of an instruction
  - Use a *strided interval* to denote the set of values
  - Have difficulty scaling to complex programs

```c
1 int *p;
2 if (...) 2 ...
3     p = ...; 3 mov rax, rbx
4 else 4 ...
5     p = ...; 5 mov rax, [rcx]
6     *p = 0; 6 mov [rax], 0
```
Intro: Existing Works

- The state-of-the-art: **Value Set Analysis (VSA)**
  - Integrated into a variety of binary analysis frameworks (Angr, BAP)
  - Compute a set of possible values for each operand of an instruction
  - Use a *strided interval* to denote the set of values
  - Have difficulty scaling to complex programs

```c
int *p;
if (...) {
    p = ...;
    mov rax, rbx
}
else {
    p = ...;
    mov rax, [rcx]
}
*p = 0;
mov [rax], 0
```
Intro: Existing Works

• The state-of-the-art: **Value Set Analysis (VSA)**
  - Integrated into a variety of binary analysis frameworks (Angr, BAP)
  - Compute a set of possible values for each operand of an instruction
  - Use a *strided interval* to denote the set of values
  - Have difficulty scaling to complex programs

```c
int *p;
if (...) {
    p = ...;
} else {
    p = ...;
}
*p = 0;
```

```asm
mov rax, rbx
mov rax, [rcx]
mov [rax], 0
```

2[0, 2] representing \{0, 2\}

100[0, 100] representing \{0, 100\}
Intro: Existing Works

• The state-of-the-art: **Value Set Analysis (VSA)**
  • Integrated into a variety of binary analysis frameworks (Angr, BAP)
  • Compute a set of possible values for each operand of an instruction
  • Use a *strided interval* to denote the set of values
  • Have difficulty scaling to complex programs

```c
1 int *p;
2 if (...) 2 ...
3 p = ...; 3 mov rax, rbx
4 else 4 ...
5 p = ...; 5 mov rax, [rcx]
6 *p = 0; 6 mov [rax], 0
```

2[0, 2] representing \{0, 2\}

100[0, 100] representing \{0, 100\}

2[0, 100] representing \{0, 2, 4, ..., 100\}
Intro: Existing Works

• The state-of-the-art: **Value Set Analysis (VSA)**

<table>
<thead>
<tr>
<th></th>
<th>ANGR-VSA</th>
<th>BAP-VSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.gzip</td>
<td>X</td>
<td>TIMEOUT (12 hours)</td>
</tr>
<tr>
<td>175.vpr</td>
<td>X</td>
<td>TIMEOUT (12 hours)</td>
</tr>
<tr>
<td>176.gcc</td>
<td>X</td>
<td>TIMEOUT (12 hours)</td>
</tr>
<tr>
<td>181.mcf</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>186.crafty</td>
<td>X</td>
<td>TIMEOUT (12 hours)</td>
</tr>
<tr>
<td>197.parser</td>
<td>X</td>
<td>TIMEOUT (12 hours)</td>
</tr>
<tr>
<td>252.eon</td>
<td>X</td>
<td>TIMEOUT (12 hours)</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>X</td>
<td>TIMEOUT (12 hours)</td>
</tr>
<tr>
<td>254.gap</td>
<td>X</td>
<td>TIMEOUT (12 hours)</td>
</tr>
<tr>
<td>255.vortex</td>
<td>X</td>
<td>TIMEOUT (12 hours)</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>X</td>
<td>TIMEOUT (12 hours)</td>
</tr>
<tr>
<td>300.twolf</td>
<td>X</td>
<td>TIMEOUT (12 hours)</td>
</tr>
</tbody>
</table>
Intro: Existing Works

• The state-of-the-art: Value Set Analysis (VSA)

• More efficient but conservative technique: ALTO
Observation

• Observation 1: probabilistic guarantees are sufficient for many practical applications
Observation

has a very low likelihood of missing any true positive

• Observation 1: **probabilistic guarantees** are sufficient for many practical applications
  • E.g., indirect control-flow transfer targets can help construct precise call graphs
Observation

has a very low likelihood of missing any true positive

• Observation 1: probabilistic guarantees are sufficient for many practical applications
  • E.g., indirect control-flow transfer targets can help construct precise call graphs

Strict Soundness
  → Never miss any true positives
  → Produce a large number of bogus call edges

Probability Guarantees
  → Discover most of the true edges
  → Have a low chance of missing some true positive edges
Observation

• Observation 1: probabilistic guarantees are sufficient for many practical applications

• Observation 2: a dependence relation can be disclosed by many whole-program paths
Observation

• Observation 1: probabilistic guarantees are sufficient for many practical applications

• Observation 2: a dependence relation can be disclosed by many whole-program paths
  • For a program with n statements
    • The number of dependences: \( O(n^2) \)
    • The number of paths is \( O(2^n) \)
Observation

• Observation 1: probabilistic guarantees are sufficient for many practical applications

• Observation 2: a dependence relation can be disclosed by many whole-program paths

→ BDA: a sampling-based abstract interpretation technique for dependence analysis
Observation

• Observation 1: probabilistic guarantees are sufficient for many practical applications

• Observation 2: a dependence relation can be disclosed by many whole-program paths

→ BDA: a sampling-based abstract interpretation technique for dependence analysis

We will use source code examples to explain our idea. But BDA operates on stripped binary executable.
Naïve Sampling Algorithm?

To toss a fair coin at each predicate

1. `void foo(char *buf){`
2. `scanf(“%s”, buf);`
3. `if (!check1(buf))`
4. `return;`
5. `if (!check2(buf))`
6. `return;`
7. `if (!check3(buf))`
8. `return;`
9. `if (!check4(buf))`
10. `return;`
11. `if (!check5(buf))`
12. `return;`
13. `if (!check6(buf))`
14. `printf(“%s”, buf);`
15. `printf(“%s”, buf);
16.}`
Naïve Sampling Algorithm?

To toss a fair coin at each predicate

1. **void** foo(char *buf){
2.     **scanf**("%s", buf);
3.     **if** (!check1(buf))
4.         return;
5.     **if** (!check2(buf))
6.         return;
7.     **if** (!check3(buf))
8.         return;
9.     **if** (!check4(buf))
10.        return;
11.    **if** (!check5(buf))
12.        return;
13.   **if** (!check6(buf))
14.        **printf**("%s", buf);
15.  **printf**("%s", buf);
16.}
Naïve Sampling Algorithm?

To toss a fair coin at each predicate

1. void foo(char *buf){
2. scanf("%s", buf);
3. if (!check1(buf))
4. return;
5. if (!check2(buf))
6. return;
7. if (!check3(buf))
8. return;
9. if (!check4(buf))
10. return;
11. if (!check5(buf))
12. return;
13. if (!check6(buf))
14. return;
15. printf("%s", buf);
16.}
Naïve Sampling Algorithm?

To toss a fair coin at each predicate

```c
1. void foo(char *buf){
2.     scanf("%s", buf);
3.     if (!check1(buf))
4.         return;
5.     if (!check2(buf))
6.         return;
7.     if (!check3(buf))
8.         return;
9.     if (!check4(buf))
10.        return;
11.    if (!check5(buf))
12.        return;
13.   if (!check6(buf))
14.        return;
15.    printf("%s", buf);
16.}
```
Naïve Sampling Algorithm?

To toss a fair coin at each predicate

1. `void foo(char *buf){`
2. `scanf("%s", buf);`
3. `if (!check1(buf))`
4. `return;`
5. `if (!check2(buf))`
6. `return;`
7. `if (!check3(buf))`
8. `return;`
9. `if (!check4(buf))`
10. `return;`
11. `if (!check5(buf))`
12. `return;`
13. `if (!check6(buf))`
14. `printf("%s", buf);`
15. `printf("%s", buf);`
16. `}`
Naïve Sampling Algorithm? **NO**

To toss a fair coin at each predicate

P(**Red Path**): \( \frac{1}{64} \)
*Find Dependence*

P(**Blue Path**): \( \frac{1}{2} \)
*Cannot Find Dependence*
Naïve Sampling Algorithm? \textit{NO}

To toss a fair coin at each predicate

7 paths in total

\begin{itemize}
\item \textbf{P(\textit{Red Path})}: \textit{1/7}
  \textit{Find Dependence}
\item \textbf{P(\textit{Blue Path})}: \textit{1/7}
  \textit{Cannot Find Dependence}
\end{itemize}
Workflow of BDA

• Phase 1: Path Sampling
  Sample whole-program paths under a *uniform distribution*

• Phase 2: Per-path Abstract Interpretation
  Compute the possible values for individual instructions, *following the given sample path*

• Phase 3: Posterior Analysis
  Mitigate the possible *incomplete path coverage* during sampling
Phase 1: Path Sampling

• Input: Binary executable and its inter-procedural control flow graph

• Output: A number of whole-program path samples
1. int main()
2. {
3.     int a;
4.     if (rand())
5.         gee(&a);
6.     else foo(&a);
7. }
8. void foo(int *a)
9. {
10.    gee(a);
11.    if (rand())
12.        *a+=1;
13. }
14. void gee(int *a)
15. {
16.     if (rand())
17.         *a=0;
18.     else *a=2;
19. }

Phase 1: Inter-procedural Control Flow Graph

Entry

BB_1

BB_4

BB_5

BB_6

BB_8

BB_10

BB_11

BB_12

BB_14

BB_16

BB_17

BB_18

Exit
Phase 1: Inter-procedural Control Flow Graph

1. `int main(){
2.     int a;
3.     if (rand())
4.         gee(&a);
5.     else foo(&a) ;
6. }
7.
8. void foo(int *a){
9.     gee(a) ;
10.    if (rand())
11.     *a+=1;
12. }
13.
14. void gee(int *a){
15.    if (rand())
16.     *a=0;
17.    else *a=2;
18. }

Diagram of Control Flow Graph:

Entry: BB_1

BB_1

BB_4

BB_5

BB_8

BB_14

BB_16

BB_17

BB_18

BB_6

Exit: BB_10

BB_11

BB_12
Phase 1: Inter-procedural Control Flow Graph

1. `int main(){`
2. `    int a;`
3. `    if (rand())`
4. `        gee(&a);`
5. `    else foo(&a);`
6. `}`
7. 
8. `void foo(int *a){`
9. `    gee(a);`
10. `    if (rand())`
11. `        *a+=1;`
12. `}`
13. 
14. `void gee(int *a){`
15. `    if (rand())`
16. `        *a=0;`
17. `    else *a=2;`
18. `}`
Phase 1: Inter-procedural Control Flow Graph

1. `int main(){
2.     int a;
3.     if (rand())
4.         gee(&a);
5.     else foo(&a);
6. }
7. void foo(int *a){
8.     gee(a);
9.     if (rand())
10.    *a+=1;
11. }
12. }
13. void gee(int *a){
14.    if (rand())
15.       *a=0;
16.     else *a=2;
17. }
18. }

Intra-procedural Control Flow

Entry

BB_1

BB_4

BB_5

BB_14

BB_16

BB_17

BB_18

BB_6

BB_8

BB_10

BB_11

BB_12

Exit
Phase 1: Inter-procedural Control Flow Graph

1. int main(){
2.    int a;
3.    if (rand())
4.        gee(&a);
5.    else foo(&a);
6. }

7.

8. void foo(int *a){
9.    gee(a);
10.   if (rand())
11.      *a+=1;
12. }

13.

14. void gee(int *a){
15.    if (rand())
16.      *a=0;
17.    else *a=2;
18. }

Entry

BB_1

BB_4

BB_5

BB_8

BB_14

BB_16

BB_17

BB_18

BB_6

BB_10

BB_11

BB_12

Exit
Phase 1: Inter-procedural Control Flow Graph

```
1. int main(){
2.    int a;
3.    if (rand())
4.        gee(&a);
5.    else foo(&a) ;
6.  }
7. }
8. void foo(int *a){
9.    gee(a) ;
10.   if (rand())
11.      *a+=1;
12. }
13. }
14. void gee(int *a){
15.    if (rand())
16.      *a=0;
17.    else *a=2;
18. }
```
Phase 1: Path Counting

Toss a *biased* coin at predicates to sample each path uniformly.
Phase 1: Path Counting

• $m$ inter-procedural paths from BB_4 to BB_6
Phase 1: Path Counting

- $m$ inter-procedural paths from BB_4 to BB_6
- $n$ inter-procedural paths from BB_5 to BB_6
Phase 1: Path Counting

- $m$ inter-procedural paths from BB_4 to BB_6
- $n$ inter-procedural paths from BB_5 to BB_6
- $n/(m+n)$ probability to take BB_5 from BB_1
Phase 1: Path Counting

Compute the weight for each basic block, which denotes the number of inter-procedural paths from the block to the exit of its enclosing function.
Phase 1: Path Counting

The path counting is performed in reverse topological order.

1. Sort the call graph (gee $\rightarrow$ foo $\rightarrow$ main)

2. Sort the nodes inside each function
Phase 1: Path Counting

The path counting is performed in reverse topological order

1. Sort the call graph (gee → foo → main)

2. Sort the nodes inside each function
Phase 1: Path Counting

The path counting is performed in reverse topological order.

1. Sort the call graph (gee → foo → main)

2. Sort the nodes inside each function

There are two paths inside gee()
Phase 1: Path Counting

\[ W[BB_{12}] = 1 \]

Return node’s weight is 1
Phase 1: Path Counting

\[
W[BB_{10}] = W[BB_{12}] + W[BB_{11}]
\]

\[
= 1 + 1 = 2
\]
Phase 1: Path Counting

Callsite node’s weight is the product of the **callee weight** and the **continuation weight**

\[
W[BB_{\_8}] = W[BB_{\_14}] \times W[BB_{\_10}] = 2 \times 2 = 4
\]
Phase 1: Path Counting

- 2 inter-procedural paths from BB_4 to BB_6
- 4 inter-procedural paths from BB_5 to BB_6
- 2/3 probability to take BB_5 from BB_1
Phase 1: Path Sampling

Path 1:

- \text{BB}_1
- \text{BB}_4
- \text{BB}_{14}
- \text{BB}_{16}
- \text{BB}_{18}
- \text{BB}_6

\[ \frac{1}{3} \quad \frac{1}{2} \quad \frac{1}{3} \quad \frac{1}{2} \quad \frac{1}{2} \quad \frac{1}{2} \]

\[ P_1 = \frac{1}{6} \quad P_2 = \frac{1}{6} \]

Path 2:

- \text{BB}_1
- \text{BB}_5
- \text{BB}_8
- \text{BB}_{14}
- \text{BB}_{17}
- \text{BB}_{18}

\[ \frac{2}{3} \quad \frac{1}{3} \quad \frac{2}{3} \]

\[ \text{BB}_6 \]

\[ \text{BB}_8 \]

\[ \text{BB}_{10} \quad \text{BB}_{12} \]

\[ \text{BB}_6 \]
Phase 1: Practical Challenges

• The weight of each block is extremely large
  • The number of whole-program paths: $O(2^n)$
Phase 1: Practical Challenges

- The weight of each block is extremely large
  - The number of whole-program paths: $O(2^n)$
  - How to handle biased distribution (e.g., $1 : 10^{1000}$)

A simple random number generator will introduce substantial error on such a biased odds.
Phase 1: Practical Challenges

• The weight of each block is extremely large
  • The number of whole-program paths: $O(2^n)$
  • How to handle biased distribution (e.g., $1 : 10^{1000}$)
  • We develop a novel algorithm to simulate the biased distribution
Phase 1: Practical Challenges

• The weight of each block is extremely large
• Loops and recursion [*Bounded unrolling*]
• Multi-exit [*Two different kinds of weights*]
Phase 2: Per-Path Abstract Interpretation

• Follow the given sampled path
Phase 2: Per-Path Abstract Interpretation

• *Follow the given sampled path*

• Use singleton value, instead of strided interval
Phase 2: Per-Path Abstract Interpretation

• Follow the given sampled path
• Use singleton value, instead of strided interval
• Is to-some-extent similar to concrete execution
Phase 2: Per-Path Abstract Interpretation

• *Follow the given sampled path*
• Use singleton value, instead of strided interval
• Is to-some-extent similar to concrete execution
• Compute the possible values for individual instructions, which will be used to collect the *definition and use information* about memory
Phase 2: Per-Path Abstract Interpretation

\&v \rightarrow 0xDEADBEEF

A. if (rand())
B. v = 0;
C. v = 1;
D. if (rand())
E. output(v);
F. output(-v);
G. return;
Phase 2: Per-Path Abstract Interpretation

&v \rightarrow 0xDEADBEEF

DEF(0xDEADBEEF) = \{B\}

A. if (rand())

B. v = 0;

C. v = 1;

D. if (rand())

E. output(v);

F. output(-v);

G. return;
Phase 2: Per-Path Abstract Interpretation

\& v \rightarrow 0xDEADBEEF

\text{DEF}(0xDEADBEEF) = \{B\}

\text{USE}(0xDEADBEEF) = \{E\}
Phase 2: Per-Path Abstract Interpretation

&v \rightarrow \text{0xDEADBEEF}

\text{DEF(0xDEADBEEF)} = \{B\}

\text{USE(0xDEADBEEF)} = \{E\}

A. \text{if (rand())}
B. v = 0;
C. v = 1;
D. \text{if (rand())}
E. \text{output(v);}
F. \text{output(-v);}
G. return;
Phase 3: Posterior Analysis

- Cannot sample all the whole-program paths
- Miss some dependence belonging to uncovered paths
Phase 3: Posterior Analysis

- Cannot sample all the whole-program paths
- Miss some dependence belonging to uncovered paths
Phase 3: Posterior Analysis

- Cannot sample all the whole-program paths
- Miss some dependence belonging to uncovered paths

B. $v = 0$;
C. $v = 1$;

E. output($v$);
F. output($-v$);
Phase 3: Posterior Analysis

- Cannot sample all the whole-program paths
- Miss some dependence belonging to uncovered paths

```
C. v = 1;
E. output(v);
G. return;
D. if (rand())
A. if (rand())
F. output(-v);
```
Phase 3: Posterior Analysis

• Merge per-path memory write information at each control-flow joint point

\&v \to 0xDEADBEEF

A. if (rand())
B. v = 0;
C. v = 1;
D. if (rand())
E. output(v);
F. output(-v);
G. return;
Phase 3: Posterior Analysis

• Merge per-path memory write information at each control-flow joint point

\&v \rightarrow 0xDEADBEEF

\text{DEF}(0xDEADBEEF) = \{B\}

\text{DEF}(0xDEADBEEF) = \{C\}
Phase 3: Posterior Analysis

- Merge per-path memory write information at each control-flow joint point

&v \rightarrow 0xDEADBEEF

DEF(0xDEADBEEF) = \{B, C\}
Phase 3: Posterior Analysis

- Cross-check the memory read information to detect dependence

$\text{DEF}(0x\text{DEADBEEF}) = \{B, C\}$

$\text{USE}(0x\text{DEADBEEF}) = \{E\}$

$E. \text{output}(v);$
Probabilistic Guarantees

- Assume $m$ out of total $n$ paths disclose a dependence, and let $k = m/n$
- For one sample, the probability $p_d$ of observing a given dependency $d$ is:
  \[
  \left( \frac{2^{63}}{2^{63} + 1} \right)^{2L} \cdot k \leq p_d = \tilde{p} \cdot m \leq \left( \frac{2^{63} + 1}{2^{63}} \right)^{2L} \cdot k
  \]
- For N sample, the probability $P_d$ of observing a given dependency $d$ is:
  \[
  P_d = 1 - (1 - p_d)^N \geq 1 - \left( 1 - \left( \frac{2^{63}}{2^{63} + 1} \right)^{2L} \cdot k \right)^N \approx 1 - (1 - k)^N
  \]
Probabilistic Guarantees

• Loop unrolling = 15

• Probabilities of observing the the dependence from `strcpy` to line 1:
  • Sample 5 times
  \[
p_d \geq 1 - \left(1 - \frac{8}{15}\right)^5 = 0.967
\]
  • Sample 50 times
  \[
p_d \geq 1 - \left(1 - \frac{8}{15}\right)^{50} = 1 - 2.2 \times 10^{-14}
\]
Evaluation

• We implemented BDA in **Rust**

• The system is available at
  
  [https://github.com/bda-tool/bda/]
Evaluation

• Code and Intra-procedural Path Coverage
• Program Dependence Analysis
• Necessity of Posterior Analysis
• Effect of Sampling
• Analysis Overhead
• Downstream Analysis
## Evaluation: Program Dependence Analysis (Compared with ALTO)

<table>
<thead>
<tr>
<th>PROGRAM</th>
<th>#REFER</th>
<th>#FOUND</th>
<th>MISS(%)</th>
<th>#EXTRA</th>
<th>MISTYPED(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.gzip</td>
<td>3,580</td>
<td>2,229,749</td>
<td>0.00</td>
<td>2,226,169</td>
<td>13.55</td>
</tr>
<tr>
<td>175.vpr</td>
<td>13,042</td>
<td>36,840,012</td>
<td>0.00</td>
<td>36,826,970</td>
<td>72.45</td>
</tr>
<tr>
<td>181.mcf</td>
<td>2,050</td>
<td>588,076</td>
<td>0.00</td>
<td>586,026</td>
<td>55.20</td>
</tr>
<tr>
<td>186.crafty</td>
<td>30,777</td>
<td>44,139,556</td>
<td>0.00</td>
<td>44,108,779</td>
<td>11.16</td>
</tr>
<tr>
<td>197.parser</td>
<td>15,196</td>
<td>32,905,403</td>
<td>0.00</td>
<td>32,890,207</td>
<td>89.21</td>
</tr>
<tr>
<td>252.eon</td>
<td>4,401</td>
<td>994,655</td>
<td>0.00</td>
<td>990,264</td>
<td>98.02</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>57,507</td>
<td>102,068,477</td>
<td>0.00</td>
<td>100,349,485</td>
<td>92.69</td>
</tr>
<tr>
<td>254.gap</td>
<td>7,935</td>
<td>10,611,636</td>
<td>0.00</td>
<td>10,603,701</td>
<td>94.06</td>
</tr>
<tr>
<td>255.vortex</td>
<td>29,971</td>
<td>265,981,817</td>
<td>0.00</td>
<td>265,951,846</td>
<td>89.66</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>4,306</td>
<td>2,466,876</td>
<td>0.00</td>
<td>2,462,570</td>
<td>28.71</td>
</tr>
<tr>
<td>300.twolf</td>
<td>16,710</td>
<td>44,735,257</td>
<td>0.00</td>
<td>44,718,440</td>
<td>75.42</td>
</tr>
</tbody>
</table>

**Avg.**  
| 16.861 | 49,414,683 | 0.00 | 49,246,769 | 65.47 |

**Benchmark:** *SPECTINT 2000*  
**Timeout Budget:** *12 hours*
## Evaluation: Program Dependence Analysis (Compared with ALTO)

<table>
<thead>
<tr>
<th>PROGRAM</th>
<th>#REFER</th>
<th>ALTO</th>
<th>#FOUND</th>
<th>MISS(%)</th>
<th>#EXTRA</th>
<th>MISTYPED(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.gzip</td>
<td>3,580</td>
<td></td>
<td>2,229,749</td>
<td>0.00</td>
<td>2,226,169</td>
<td>13.55</td>
</tr>
<tr>
<td>175.vpr</td>
<td>13,042</td>
<td></td>
<td>36,840,012</td>
<td>0.00</td>
<td>36,826,970</td>
<td>72.45</td>
</tr>
<tr>
<td>181.mcf</td>
<td>2,050</td>
<td></td>
<td>588,076</td>
<td>0.00</td>
<td>586,026</td>
<td>55.20</td>
</tr>
<tr>
<td>186.crafty</td>
<td>30,777</td>
<td></td>
<td>44,139,556</td>
<td>0.00</td>
<td>44,108,779</td>
<td>11.16</td>
</tr>
<tr>
<td>197.parser</td>
<td>15,196</td>
<td></td>
<td>32,905,403</td>
<td>0.00</td>
<td>32,890,207</td>
<td>89.21</td>
</tr>
<tr>
<td>252.eon</td>
<td>4,401</td>
<td></td>
<td>994,655</td>
<td>0.00</td>
<td>990,264</td>
<td>98.02</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>57,507</td>
<td></td>
<td>102,068,477</td>
<td>0.00</td>
<td>100,349,485</td>
<td>92.69</td>
</tr>
<tr>
<td>254.gap</td>
<td>7,935</td>
<td></td>
<td>10,611,636</td>
<td>0.00</td>
<td>10,603,701</td>
<td>94.06</td>
</tr>
<tr>
<td>255.vortex</td>
<td>29,971</td>
<td></td>
<td>265,981,817</td>
<td>0.00</td>
<td>265,951,846</td>
<td>89.66</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>4,306</td>
<td></td>
<td>2,466,876</td>
<td>0.00</td>
<td>2,462,570</td>
<td>28.71</td>
</tr>
<tr>
<td>300.twolf</td>
<td>16,710</td>
<td></td>
<td>44,735,257</td>
<td>0.00</td>
<td>44,718,440</td>
<td>75.42</td>
</tr>
<tr>
<td><strong>Avg.</strong></td>
<td><strong>16.861</strong></td>
<td><strong>49,414,683</strong></td>
<td><strong>0.00</strong></td>
<td><strong>49,246,769</strong></td>
<td><strong>65.47</strong></td>
<td></td>
</tr>
</tbody>
</table>
### Evaluation: Program Dependence Analysis (Compared with ALTO)

<table>
<thead>
<tr>
<th>PROGRAM</th>
<th>#REFER</th>
<th>#FOUND</th>
<th>MISS(%)</th>
<th>#EXTRA</th>
<th>MISTYPED(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.gzip</td>
<td>3,580</td>
<td>2,226,169</td>
<td>0.00</td>
<td>2,226,169</td>
<td>13.55</td>
</tr>
<tr>
<td>175.vpr</td>
<td>13,042</td>
<td>36,826,970</td>
<td>0.00</td>
<td>36,826,970</td>
<td>72.45</td>
</tr>
<tr>
<td>181.mcf</td>
<td>2,050</td>
<td>586,026</td>
<td>0.00</td>
<td>586,026</td>
<td>55.20</td>
</tr>
<tr>
<td>186.crafty</td>
<td>30,777</td>
<td>44,108,779</td>
<td>0.00</td>
<td>44,108,779</td>
<td>11.16</td>
</tr>
<tr>
<td>252.eon</td>
<td>4,401</td>
<td>990,264</td>
<td>0.00</td>
<td>990,264</td>
<td>98.02</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>57,507</td>
<td>100,349,485</td>
<td>0.00</td>
<td>100,349,485</td>
<td>92.69</td>
</tr>
<tr>
<td>254.gap</td>
<td>7,935</td>
<td>10,603,701</td>
<td>0.00</td>
<td>10,603,701</td>
<td>94.06</td>
</tr>
<tr>
<td>255.vortex</td>
<td>29,971</td>
<td>265,951,846</td>
<td>0.00</td>
<td>265,951,846</td>
<td>89.66</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>4,306</td>
<td>2,462,570</td>
<td>0.00</td>
<td>2,462,570</td>
<td>28.71</td>
</tr>
<tr>
<td>300.twolf</td>
<td>16,710</td>
<td>44,718,440</td>
<td>0.00</td>
<td>44,718,440</td>
<td>75.42</td>
</tr>
<tr>
<td><strong>Avg.</strong></td>
<td><strong>16.861</strong></td>
<td><strong>49,414,683</strong></td>
<td><strong>0.00</strong></td>
<td><strong>49,246,769</strong></td>
<td><strong>65.47</strong></td>
</tr>
</tbody>
</table>

Dependency detected by reference execution but not by the tool.
## Evaluation: Program Dependence Analysis (Compared with ALTO)

<table>
<thead>
<tr>
<th>PROGRAM</th>
<th>#REFER</th>
<th>#FOUND</th>
<th>MISS(%)</th>
<th>#EXTRA</th>
<th>MISTYPED(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>164.gzip</td>
<td>3,580</td>
<td>2,226,169</td>
<td>0.00</td>
<td>32,890,207</td>
<td>89.21</td>
</tr>
<tr>
<td>175.vpr</td>
<td>13,042</td>
<td>36,826,970</td>
<td>0.00</td>
<td>990,264</td>
<td>98.02</td>
</tr>
<tr>
<td>181.mcf</td>
<td>2,050</td>
<td>586,026</td>
<td>0.00</td>
<td>100,349,485</td>
<td>92.69</td>
</tr>
<tr>
<td>186.crafty</td>
<td>30,777</td>
<td>44,108,779</td>
<td>0.00</td>
<td>44,718,440</td>
<td>75.42</td>
</tr>
<tr>
<td>197.parser</td>
<td>15,196</td>
<td>32,905,403</td>
<td>0.00</td>
<td>32,890,207</td>
<td>89.21</td>
</tr>
<tr>
<td>252.eon</td>
<td>4,401</td>
<td>994,655</td>
<td>0.00</td>
<td>990,264</td>
<td>98.02</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>57,507</td>
<td>102,068,477</td>
<td>0.00</td>
<td>100,349,485</td>
<td>92.69</td>
</tr>
<tr>
<td>254.gap</td>
<td>7,935</td>
<td>10,611,636</td>
<td>0.00</td>
<td>10,603,701</td>
<td>94.06</td>
</tr>
<tr>
<td>255.vortex</td>
<td>29,971</td>
<td>265,981,817</td>
<td>0.00</td>
<td>265,951,846</td>
<td>89.66</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>4,306</td>
<td>2,466,876</td>
<td>0.00</td>
<td>2,462,570</td>
<td>28.71</td>
</tr>
<tr>
<td>300.twolf</td>
<td>16,710</td>
<td>44,735,257</td>
<td>0.00</td>
<td>44,718,440</td>
<td>75.42</td>
</tr>
<tr>
<td><strong>Avg.</strong></td>
<td><strong>16.861</strong></td>
<td><strong>49,414,683</strong></td>
<td><strong>0.00</strong></td>
<td><strong>49,246,769</strong></td>
<td><strong>65.47</strong></td>
</tr>
</tbody>
</table>
Evaluation: Program Dependence Analysis (Compared with ALTO)

<table>
<thead>
<tr>
<th>PROGRAM</th>
<th>#REFER</th>
<th>#FOUND</th>
<th>MISS(%)</th>
<th>#EXTRA</th>
<th>MISTYPED(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.gzip</td>
<td>3,580</td>
<td>2,229,749</td>
<td>0.00</td>
<td>3226,169</td>
<td>13.55</td>
</tr>
<tr>
<td>175.vpr</td>
<td>13,042</td>
<td>36,840,012</td>
<td>0.00</td>
<td>36,826,970</td>
<td>72.45</td>
</tr>
<tr>
<td>181.mcf</td>
<td>2,050</td>
<td>588,076</td>
<td>0.00</td>
<td>586,026</td>
<td>55.20</td>
</tr>
<tr>
<td>186.crafty</td>
<td>30,777</td>
<td>44,139,556</td>
<td>0.00</td>
<td>44,108,779</td>
<td>11.16</td>
</tr>
<tr>
<td>197.parser</td>
<td>19,130</td>
<td>32,303,403</td>
<td>0.00</td>
<td>32,060,207</td>
<td>89.21</td>
</tr>
<tr>
<td>252.eon</td>
<td>4,401</td>
<td>994,655</td>
<td>0.00</td>
<td>990,264</td>
<td>98.02</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>57,507</td>
<td>102,068,477</td>
<td>0.00</td>
<td>100,349,485</td>
<td>92.69</td>
</tr>
<tr>
<td>254.gap</td>
<td>7,935</td>
<td>10,611,636</td>
<td>0.00</td>
<td>10,603,701</td>
<td>94.06</td>
</tr>
<tr>
<td>255.vortex</td>
<td>29,971</td>
<td>265,981,817</td>
<td>0.00</td>
<td>265,951,846</td>
<td>89.66</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>4,306</td>
<td>2,466,876</td>
<td>0.00</td>
<td>2,462,570</td>
<td>28.71</td>
</tr>
<tr>
<td>300.twolf</td>
<td>16,710</td>
<td>44,735,257</td>
<td>0.00</td>
<td>44,718,440</td>
<td>75.42</td>
</tr>
<tr>
<td>Avg.</td>
<td>16.861</td>
<td><strong>49,414,683</strong></td>
<td><strong>0.00</strong></td>
<td><strong>49,246,769</strong></td>
<td><strong>65.47</strong></td>
</tr>
</tbody>
</table>

The checker is implemented as an LLVM pass, propagating symbol information to individual instructions, registers and memory locations.
### Evaluation: Program Dependence Analysis (Compared with ALTO)

<table>
<thead>
<tr>
<th>PROGRAM</th>
<th>#REFER</th>
<th>#FOUND</th>
<th>MISS(%)</th>
<th>#EXTRA</th>
<th>MISTYPED(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.gzip</td>
<td>3,580</td>
<td>2,229,749</td>
<td>0.00</td>
<td>2,226,169</td>
<td>13.55</td>
</tr>
<tr>
<td>175.vpr</td>
<td>13,042</td>
<td>36,840,012</td>
<td>0.00</td>
<td>36,826,970</td>
<td>72.45</td>
</tr>
<tr>
<td>181.mcf</td>
<td>2,050</td>
<td>588,076</td>
<td>0.00</td>
<td>586,026</td>
<td>55.20</td>
</tr>
<tr>
<td>186.crafty</td>
<td>30,777</td>
<td>44,139,556</td>
<td>0.00</td>
<td>44,108,779</td>
<td>11.16</td>
</tr>
<tr>
<td>197.parser</td>
<td>15,196</td>
<td>32,905,403</td>
<td>0.00</td>
<td>32,890,207</td>
<td>89.21</td>
</tr>
<tr>
<td>252.eon</td>
<td>4,401</td>
<td>994,655</td>
<td>0.00</td>
<td>990,264</td>
<td>98.02</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>57,507</td>
<td>102,068,477</td>
<td>0.00</td>
<td>100,349,485</td>
<td>92.69</td>
</tr>
<tr>
<td>254.gap</td>
<td>7,935</td>
<td>10,611,636</td>
<td>0.00</td>
<td>10,603,701</td>
<td>94.06</td>
</tr>
<tr>
<td>255.vortex</td>
<td>29,971</td>
<td>265,981,817</td>
<td>0.00</td>
<td>265,951,846</td>
<td>89.66</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>4,306</td>
<td>2,466,876</td>
<td>0.00</td>
<td>2,462,570</td>
<td>28.71</td>
</tr>
<tr>
<td>300.twolf</td>
<td>16,710</td>
<td>44,735,257</td>
<td>0.00</td>
<td>44,718,440</td>
<td>75.42</td>
</tr>
<tr>
<td><strong>Avg.</strong></td>
<td><strong>16.861</strong></td>
<td><strong>49,414,683</strong></td>
<td><strong>0.00</strong></td>
<td><strong>49,246,769</strong></td>
<td><strong>65.47</strong></td>
</tr>
</tbody>
</table>

ALTO reports **49M** dependence, **65%** of them are mis-typed, without missing any.
Evaluation: Program Dependence Analysis (Compared with ALTO)

<table>
<thead>
<tr>
<th>PROGRAM</th>
<th>#REFER</th>
<th>#FOUND</th>
<th>MISS(%)</th>
<th>#EXTRA</th>
<th>MISTYPED(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.gzip</td>
<td>3,580</td>
<td>29,370</td>
<td>0.22</td>
<td>25,798</td>
<td>11.92</td>
</tr>
<tr>
<td>175.vpr</td>
<td>13,042</td>
<td>559,460</td>
<td>0.08</td>
<td>546,428</td>
<td>61.88</td>
</tr>
<tr>
<td>181.mcf</td>
<td>2,050</td>
<td>3,347</td>
<td>0.00</td>
<td>1,297</td>
<td>12.94</td>
</tr>
<tr>
<td>186.crafty</td>
<td>30,777</td>
<td>1,077,346</td>
<td>0.15</td>
<td>1,046,614</td>
<td>7.31</td>
</tr>
<tr>
<td>197.parser</td>
<td>15,196</td>
<td>659,867</td>
<td>0.01</td>
<td>644,673</td>
<td>81.12</td>
</tr>
<tr>
<td>252.eon</td>
<td>4,401</td>
<td>28,855</td>
<td>0.00</td>
<td>24,454</td>
<td>78.11</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>57,507</td>
<td>5,389,973</td>
<td>0.23</td>
<td>5,363,373</td>
<td>82.77</td>
</tr>
<tr>
<td>254.gap</td>
<td>7,935</td>
<td>205,200</td>
<td>0.52</td>
<td>197,306</td>
<td>74.30</td>
</tr>
<tr>
<td>255.vortex</td>
<td>29,971</td>
<td>2,159,444</td>
<td>0.33</td>
<td>2,129,473</td>
<td>64.10</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>4,306</td>
<td>13,917</td>
<td>0.23</td>
<td>9,621</td>
<td>10.84</td>
</tr>
<tr>
<td>300.twolf</td>
<td>16,710</td>
<td>2,285,090</td>
<td>0.34</td>
<td>2,268,436</td>
<td>73.45</td>
</tr>
<tr>
<td><strong>Avg.</strong></td>
<td><strong>16.861</strong></td>
<td><strong>1,128,352</strong></td>
<td><strong>0.19</strong></td>
<td><strong>1,114,316</strong></td>
<td><strong>50.80</strong></td>
</tr>
</tbody>
</table>

BDA reports 1M dependence (48 times smaller than ALTO’s), 50% of them are mis-typed, only with 0.19% missing rate
## Evaluation: Program Dependence Analysis (Compared with VSA)

| PROGRAM  | #REFER |  | BDA |  |
|----------|--------|  | #FOUND | MISS(%) | #EXTRA | MISTYPED(%) |
| 175.vpr  | 13,042 |  | 559,460 | 0.08 | 546,428 | 61.88 |
| 181.mcf  | 2,050  | 3,347 | 0.00 | 1,297 | 12.94 |
| 186.crafty | 30,777 | 1,077,346 | 0.15 | 1,046,614 | 7.31 |

| PROGRAM  | #REFER |  | VSA |  |
|----------|--------|  | #FOUND | MISS(%) | #EXTRA | MISTYPED(%) |
| 175.vpr  | 13,042 |  | TIMEOUT | TIMEOUT | TIMEOUT | TIMEOUT |
| 181.mcf  | 2,050  | 23,068 | 0.00 | 21,018 | 54.33 |
| 186.crafty | 30,777 |  | TIMEOUT | TIMEOUT | TIMEOUT | TIMEOUT |

BAP-VSA only handles 181.mcf within 12 hours.
Evaluation: Program Dependence Analysis (Compared with VSA)

<table>
<thead>
<tr>
<th>PROGRAM</th>
<th>#REFER</th>
<th>BDA</th>
<th>VSA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>#FOUND   MISS(%)  #EXTRA MISTYPED(%)</td>
<td>#FOUND   MISS(%)  #EXTRA MISTYPED(%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>175.vpr</td>
<td>13,042</td>
<td>559,460  0.08      546,428  61.88</td>
<td></td>
</tr>
<tr>
<td>181.mcf</td>
<td>2,050</td>
<td>3,347    0.00      1,297      12.94</td>
<td></td>
</tr>
<tr>
<td>186.crafty</td>
<td>30,777</td>
<td>1,077,346 0.15</td>
<td>1,046,614  7.31</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>175.vpr</td>
<td>13,042</td>
<td>TIMEOUT  TIMEOUT  TIMEOUT  TIMEOUT</td>
<td>TIMEOUT</td>
</tr>
<tr>
<td>181.mcf</td>
<td>2,050</td>
<td>23,068   0.00      21,018    54.33</td>
<td></td>
</tr>
<tr>
<td>186.crafty</td>
<td>30,777</td>
<td>TIMEOUT  TIMEOUT  TIMEOUT  TIMEOUT</td>
<td>TIMEOUT</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

BAP-VSA only handles 181.mcf within 12 hours.
BDA reports 5 times less dependence, with less mistyped ones.
Evaluation: Downstream Analysis
Evaluation: Identify Indirect Control Flow Targets

*IDA* is a widely used commercial disassembling tools.

<table>
<thead>
<tr>
<th>PROGRAM</th>
<th>#INDIRECT JUMP EDGES</th>
<th>#INDIRECT CALL EDGES</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IDA</td>
<td>REFER</td>
</tr>
<tr>
<td>164.gzip</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>175.vpr</td>
<td>49</td>
<td>0</td>
</tr>
<tr>
<td>176.gcc</td>
<td>3,628</td>
<td>324</td>
</tr>
<tr>
<td>181.mcf</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>186.crafty</td>
<td>159</td>
<td>38</td>
</tr>
<tr>
<td>197.parser</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>252.eon</td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>1,454</td>
<td>229</td>
</tr>
<tr>
<td>254.gap</td>
<td>63</td>
<td>5</td>
</tr>
<tr>
<td>255.vortex</td>
<td>247</td>
<td>56</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>300.twolf</td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>Avg.</td>
<td>470</td>
<td>54</td>
</tr>
</tbody>
</table>
### Evaluation: Identify Indirect Control Flow Targets

**IDA** is a widely used commercial disassembling tool.

<table>
<thead>
<tr>
<th>PROGRAM</th>
<th>#INDIRECT JUMP EDGES</th>
<th>#INDIRECT CALL EDGES</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IDA</td>
<td>REFER</td>
</tr>
<tr>
<td>164.gzip</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>175.vpr</td>
<td>49</td>
<td>0</td>
</tr>
<tr>
<td>176.gcc</td>
<td>3,628</td>
<td>324</td>
</tr>
<tr>
<td>181.mcf</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>186.crafty</td>
<td>159</td>
<td>38</td>
</tr>
<tr>
<td>197.parser</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>252.eon</td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>1,454</td>
<td>229</td>
</tr>
<tr>
<td>254.gap</td>
<td>63</td>
<td>5</td>
</tr>
<tr>
<td>255.vortex</td>
<td>247</td>
<td>56</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>300.twolf</td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td><strong>Avg.</strong></td>
<td><strong>470</strong></td>
<td><strong>54</strong></td>
</tr>
</tbody>
</table>
# Evaluation: Identify Indirect Control Flow Targets

*IDA* is a widely used commercial disassembling tool.

<table>
<thead>
<tr>
<th>PROGRAM</th>
<th>#INDIRECT JUMP EDGES</th>
<th>#INDIRECT CALL EDGES</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IDA</td>
<td>REFER</td>
</tr>
<tr>
<td>164.gzip</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>175.vpr</td>
<td>49</td>
<td>0</td>
</tr>
<tr>
<td>176.gcc</td>
<td>3,628</td>
<td>324</td>
</tr>
<tr>
<td>181.mcf</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>186.crafty</td>
<td>159</td>
<td>38</td>
</tr>
<tr>
<td>197.parser</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>252.eon</td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>1,454</td>
<td>229</td>
</tr>
<tr>
<td>254.gap</td>
<td>63</td>
<td>5</td>
</tr>
<tr>
<td>255.vortex</td>
<td>247</td>
<td>56</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>300.twolf</td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td><strong>Avg.</strong></td>
<td>470</td>
<td>54</td>
</tr>
</tbody>
</table>
Evaluation: Identify Indirect Control Flow Targets

BDA performs as good as IDA in inferring indirect jump targets. (470 in average)

BDA reports 767 indirect call edges, without missing any observer ones.

<table>
<thead>
<tr>
<th>PROGRAM</th>
<th>#INDIRECT JUMP EDGES</th>
<th>#INDIRECT CALL EDGES</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IDA</td>
<td>REFER</td>
</tr>
<tr>
<td>164.gzip</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>175.vpr</td>
<td>49</td>
<td>0</td>
</tr>
<tr>
<td>176.gcc</td>
<td>3,628</td>
<td>324</td>
</tr>
<tr>
<td>181.mcf</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>186.crafty</td>
<td>159</td>
<td>38</td>
</tr>
<tr>
<td>197.parser</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>252.eon</td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>1,454</td>
<td>229</td>
</tr>
<tr>
<td>254.gap</td>
<td>63</td>
<td>5</td>
</tr>
<tr>
<td>255.vortex</td>
<td>247</td>
<td>56</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>300.twolf</td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>Avg.</td>
<td>470</td>
<td>54</td>
</tr>
</tbody>
</table>
Evaluation: Malware Analysis

• Malware behavior is largely defined by its system and library calls, together with parameter values.

• BDA performs static constant propagation through dependence, to identify the parameter values.
**Evaluation: Malware Analysis**

*Cuckoo* is the state-of-the-art malware analysis tool. BDA reports 3 times more hidden malicious behaviors than cuckoo.

<table>
<thead>
<tr>
<th>MD5 OF MALWARE</th>
<th>REPORT DATE</th>
<th>#LIBRARY CALL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a0b96488c4be390ce2072735ffb0e49</td>
<td>2019-01-22</td>
<td>50</td>
</tr>
<tr>
<td>3fb857173602653861b4d0547a49b395</td>
<td>2018-07-24</td>
<td>20</td>
</tr>
<tr>
<td>49c178976c50cf77db3f6234efce5eeb</td>
<td>2019-01-23</td>
<td>23</td>
</tr>
<tr>
<td>5e890cb3f6cba8168d078fde090996</td>
<td>2019-01-25</td>
<td>28</td>
</tr>
<tr>
<td>6dc1f557eac7093ee9e5807385dbcb05</td>
<td>2018-12-23</td>
<td>20</td>
</tr>
<tr>
<td>72afccbf455faa4bc1e5f16ee67c6f915</td>
<td>2019-07-02</td>
<td>6</td>
</tr>
<tr>
<td>74124dae8fdbc903bece57d5be31246b</td>
<td>2019-03-21</td>
<td>36</td>
</tr>
<tr>
<td>912bca5947944f4cd09e9620d7aa8c4a</td>
<td>2019-10-04</td>
<td>20</td>
</tr>
<tr>
<td>a664df72a34b863fc0a6e04c96866d4c</td>
<td>2018-12-20</td>
<td>23</td>
</tr>
<tr>
<td>c38d08b904d5e1c7c798e840f1d8f1ee</td>
<td>2018-08-28</td>
<td>34</td>
</tr>
<tr>
<td>c63cef04d931d8171d0c40b7521855e9</td>
<td>2019-01-23</td>
<td>20</td>
</tr>
<tr>
<td>df4db38f6d3c1e751dcf06bea072ba9c</td>
<td>2018-10-23</td>
<td>20</td>
</tr>
</tbody>
</table>

*Avg.* / 25 108
Evaluation: Malware Analysis

BDA reports 3 times more hidden malicious behaviors than cuckoo.

<table>
<thead>
<tr>
<th>REPORT DATE</th>
<th>#LIBRARY</th>
<th>CALL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CUCKOO</td>
<td>BDA</td>
</tr>
<tr>
<td>2019-01-22</td>
<td>50</td>
<td>164</td>
</tr>
<tr>
<td>2018-07-24</td>
<td>20</td>
<td>112</td>
</tr>
<tr>
<td>2019-01-23</td>
<td>23</td>
<td>48</td>
</tr>
<tr>
<td>2019-01-25</td>
<td>28</td>
<td>138</td>
</tr>
<tr>
<td>2018-12-23</td>
<td>20</td>
<td>75</td>
</tr>
<tr>
<td>2019-07-02</td>
<td>6</td>
<td>81</td>
</tr>
<tr>
<td>2018-03-21</td>
<td>36</td>
<td>203</td>
</tr>
<tr>
<td>2019-10-04</td>
<td>20</td>
<td>68</td>
</tr>
<tr>
<td>2018-12-20</td>
<td>23</td>
<td>99</td>
</tr>
<tr>
<td>2018-08-28</td>
<td>34</td>
<td>151</td>
</tr>
<tr>
<td>2019-01-23</td>
<td>20</td>
<td>81</td>
</tr>
<tr>
<td>2018-10-23</td>
<td>20</td>
<td>77</td>
</tr>
<tr>
<td><strong>Avg.</strong></td>
<td>/</td>
<td><strong>108</strong></td>
</tr>
</tbody>
</table>
Closely Related Work

Random abstract interpretation
• Discovering affine equalities using random interpretation [POPL 03]
• Global value numbering using random interpretation [POPL 04]
• Precise inter-procedural analysis using random interpretation [POPL 05]

Path encoding
• Efficient path profiling [MICRO 96]
• Precise Calling Context Encoding [ICSE 10]

Reducing the runtime complexity of path-sensitive analysis
• ESP: Path-sensitive program verification in polynomial time [PLDI 02]
• Sound, complete and scalable path-sensitive analysis [PLDI 08]
Conclusion

• We propose a practical program dependence analysis for binary executables
  • A novel unbiased whole-program path sampling algorithm
  • A per-path abstract interpretation
  • Probabilistic guarantees in disclosing a dependence relation

• Result
  • Improve the state-of-the-art, such as Value Set Analysis
  • Improve performance of downstream applications
Thank you!

Q&A

Email Address
zhan3299@purdue.edu

BDA Repo
https://github.com/bda-tool/bda/