1 Basic Information of Binaries under Evaluation

To assess BDA’s effectiveness and efficiency, we compare it with other dependence analysis techniques on the SPECINT2000 [2] benchmark. Table 1 presents the statistics of the SPECINT2000 binaries, including their size, number of instructions, basic blocks, and functions.

We also apply BDA in several downstream analyses, one of them is to identify hidden malicious behaviors of a set of 12 recent malware samples provided by VirtualTotal [3]. We present Table 2 to show malware ids, size, and report date.

2 Proof of Theorem 4.1

**Theorem 4.1.** Using Algorithm 2, the probability $\tilde{p}$ of any whole-program path being sampled satisfies equation (1) in which $n$ is the total number of whole-program paths and $L$ is the length of the longest path.

$$
\left(\frac{2^{63}}{2^{63} + 1}\right)^{2L} \cdot \frac{1}{n} \leq \tilde{p} \leq \left(\frac{2^{63} + 1}{2^{63}}\right)^{2L} \cdot \frac{1}{n}
$$

\[ (1) \]
Proof. First, for any weight $w_v$, we prove that $\tilde{w}_v$ follows $\frac{2^{63}}{2^{63}+1} \cdot w_v \leq \tilde{w}_v \leq w_v$.

\[
\begin{align*}
exponent &= \max(\lfloor \log w_v \rfloor, 63) - 63 \\
sign &= \lfloor w_v / 2^\exponent \rfloor
\end{align*}
\]  

(2)

According to equation (2) if $w_v < 2^{64}$, $\tilde{w}_v = w_v$. Otherwise, $\sign \leq w_v / 2^\exponent < \sign + 1$, and hence $\sign \cdot 2^\exponent \leq w_v < (\sign + 1) \cdot 2^\exponent$. As $\sign \geq 2^{63}$ when $w_v \geq 2^{64}$, we have $\tilde{w}_v \leq w_v < \frac{2^{63}+1}{2^{63}} \cdot w_v$. Thus, $\frac{2^{63}}{2^{63}+1} \cdot w_v \leq \tilde{w}_v \leq w_v$. As a result, the following holds.

\[
\frac{2^{63}}{2^{63}+1} \cdot \frac{w_1}{w_1+w_0} \leq \frac{\tilde{w}_1}{\tilde{w}_1+\tilde{w}_0} \leq \frac{2^{63}+1}{2^{63}} \cdot \frac{w_1}{w_1+w_0}
\]  

(3)

Let $p_1 = \frac{w_1}{w_1+w_0}$ be the accurate probability of choosing branch 1, the lighter-weight branch. $p_0 = \frac{w_0}{w_1+w_0}$ choosing the other. Thus, we can derive the following from inequality 3.

\[
\frac{2^{63}}{2^{63}+1} \cdot p_1 \leq \frac{\tilde{w}_1}{\tilde{w}_1+\tilde{w}_0} \leq \frac{2^{63}+1}{2^{63}} \cdot p_1
\]  

(4)

Next, we derive the bounds of $\tilde{p}_1$, the probability of Algorithm ?? choosing branch 1. There are two cases.

(a) If $n < 64$, we directly have $\tilde{p}_1 = \frac{\tilde{w}_1}{\tilde{w}_1+\tilde{w}_0}$. According to inequality 4 we have the following.

\[
\frac{2^{63}}{2^{63}+1} \cdot p_1 \leq \tilde{p}_1 \leq \frac{2^{63}+1}{2^{63}} \cdot p_1
\]  

(5)

(b) If $n \geq 64$, $\tilde{p}_1 = \frac{\tilde{w}_1 \cdot \sign}{\tilde{w}_0 \cdot \sign \times 2^n}$. Note that $\frac{\tilde{w}_1}{\tilde{w}_1+\tilde{w}_0} = \frac{\tilde{w}_1 \cdot \sign}{\tilde{w}_0 \cdot \sign \times 2^n}$. Thus, we have $\tilde{p}_1 \geq \frac{\tilde{w}_1}{\tilde{w}_1+\tilde{w}_0}$. Combining with inequality 4 we can have $\tilde{p}_1 \geq \frac{2^{63}}{2^{63}+1} \cdot p_1$. On the other hand, $\tilde{p}_1 = \frac{\tilde{w}_1}{\tilde{w}_1+\tilde{w}_0} \cdot \frac{\tilde{w}_0 \cdot \sign \times 2^n + \tilde{w}_1 \cdot \sign}{\tilde{w}_0 \cdot \sign \times 2^n}$. Because $\tilde{w}_1 \cdot \sign < 2^{64} \leq 2 \cdot \tilde{w}_0 \cdot \sign$, we can have $\frac{\tilde{w}_0 \cdot \sign \times 2^n + \tilde{w}_1 \cdot \sign}{\tilde{w}_0 \cdot \sign \times 2^n} \leq 2^{n-1} + 1$. As $n \geq 64$, we can have $\tilde{p}_1 = \frac{\tilde{w}_1}{\tilde{w}_1+\tilde{w}_0} \cdot \frac{\tilde{w}_0 \cdot \sign \times 2^n + \tilde{w}_1 \cdot \sign}{\tilde{w}_0 \cdot \sign \times 2^n} < \frac{\tilde{w}_1}{\tilde{w}_0+\tilde{w}_1} \cdot \frac{2^{63}+1}{2^{63}}$. Combining with inequality 4 we can have $\tilde{p}_1 < \left(\frac{2^{63}+1}{2^{63}}\right)^2 \cdot p_1$. Thus,

\[
\frac{2^{63}}{2^{63}+1} \cdot p_1 \leq \tilde{p}_1 \leq \left(\frac{2^{63}+1}{2^{63}}\right)^2 \cdot p_1
\]  

(6)

From inequality 3 and 5 the following is true.

\[
\left(\frac{2^{63}}{2^{63}+1}\right)^2 \cdot p_1 \leq \tilde{p}_1 \leq \left(\frac{2^{63}+1}{2^{63}}\right)^2 \cdot p_1
\]  

(7)

Similarly, we can prove the bound for $\tilde{p}_0$.

Note that any sampled path could contain at most $L$ conditional predicates. Thus, the probability $\tilde{p}$ of any whole-program path being sampled satisfies equation 1.

$\square$
3 Algorithms in Posterior Analysis

After the abstract interpretation of all sampled paths, the posterior analysis is performed to complete dependence analysis, via aggregating the abstract values collected from individual path samples in a flow-sensitive, context-sensitive, and path-insensitive fashion. This section will present detailed algorithms of Per-sample Analysis and Handle Memory Read which are elided in [1].

Per-sample Analysis Algorithm 1 traverses each instruction iaddr and the abstract address maddr accessed by the instruction and updates I2M (line 4). If iaddr is a memory write, the previous definition of maddr is killed by iaddr (line 6) and iaddr becomes the latest definition (line 7). If it is a read, a dependence is identified between iaddr and the lastest definition and added to DEP (line 9).

**Algorithm 1 Per-sample Analysis**

| INPUT: | MOS: MemOpSeq | \(\text{▷ memory operation sequence}\) |
| OUTPUT: | I2M: Address \(\rightarrow\) \{AbstractValue\} | \(\text{▷ map an instruction to abstract addresses accessed by it}\) |
| | DEP: Address \(\rightarrow\) \{Address\} | \(\text{▷ map an instruction to the instructions it depends on}\) |
| | KILL: Address \(\rightarrow\) \{Address\} | \(\text{▷ map an instruction to reaching definitions it kills}\) |
| LOCAL: | DEF: AbstractValue \(\rightarrow\) Address | \(\text{▷ map an abstract address to its latest definition}\) |

1: function PerSampleAnalysis(MOS)  
2: while \(\neg\)MOS.empty() do  
3: \(\langle\text{iaddr, maddr}\rangle \leftarrow\) MOS.dequeue () \(\text{▷ acquire an instruction instance and the accessed address}\)  
4: I2M[iaddr] \(\leftarrow\) I2M[iaddr] \(\cup\) \{maddr\}  
5: if is_memory_write(iaddr) then  
6: \(KILL[iaddr] \leftarrow KILL[iaddr] \cup \{DEF[maddr]\}\) \(\text{▷ previous definition of maddr is killed by iaddr}\)  
7: \(DEF[maddr] \leftarrow iaddr\) \(\text{▷ iaddr is the new definition of maddr}\)  
8: else if is_memory_read(iaddr) then  
9: \(DEP[iaddr] \leftarrow DEP[iaddr] \cup \{DEF[maddr]\}\) \(\text{▷ detect a new dependence}\)  
10: end if  
11: end while  
12: return \(\langle I2M, DEP, KILL\rangle\)  
13: end function

Handle Memory Read Similar to handling memory writes in [1], Algorithm 2 specially addresses strong updates, which lead to single dependence (lines 4-5). Otherwise in lines 7-11, for each maddr ever accessed by iaddr in some sample, dependences are introduced between iaddr to all the live definitions of maddr in M2I.
Algorithm 2 Handle Memory Read

<table>
<thead>
<tr>
<th>INPUT:</th>
<th>iaddr: Address</th>
<th>DIP: Address × Address</th>
<th>DIP′: Address × Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>iaddr: Address</td>
<td>DIP: Address × Address</td>
<td>DIP′: Address × Address</td>
</tr>
<tr>
<td></td>
<td>iaddr: Address</td>
<td>DIP: Address × Address</td>
<td>DIP′: Address × Address</td>
</tr>
<tr>
<td></td>
<td>iaddr: Address</td>
<td>DIP: Address × Address</td>
<td>DIP′: Address × Address</td>
</tr>
<tr>
<td></td>
<td>iaddr: Address</td>
<td>DIP: Address × Address</td>
<td>DIP′: Address × Address</td>
</tr>
<tr>
<td></td>
<td>iaddr: Address</td>
<td>DIP: Address × Address</td>
<td>DIP′: Address × Address</td>
</tr>
<tr>
<td></td>
<td>iaddr: Address</td>
<td>DIP: Address × Address</td>
<td>DIP′: Address × Address</td>
</tr>
<tr>
<td></td>
<td>iaddr: Address</td>
<td>DIP: Address × Address</td>
<td>DIP′: Address × Address</td>
</tr>
<tr>
<td></td>
<td>iaddr: Address</td>
<td>DIP: Address × Address</td>
<td>DIP′: Address × Address</td>
</tr>
<tr>
<td></td>
<td>iaddr: Address</td>
<td>DIP: Address × Address</td>
<td>DIP′: Address × Address</td>
</tr>
<tr>
<td></td>
<td>iaddr: Address</td>
<td>DIP: Address × Address</td>
<td>DIP′: Address × Address</td>
</tr>
<tr>
<td></td>
<td>iaddr: Address</td>
<td>DIP: Address × Address</td>
<td>DIP′: Address × Address</td>
</tr>
<tr>
<td></td>
<td>iaddr: Address</td>
<td>DIP: Address × Address</td>
<td>DIP′: Address × Address</td>
</tr>
<tr>
<td></td>
<td>iaddr: Address</td>
<td>DIP: Address × Address</td>
<td>DIP′: Address × Address</td>
</tr>
<tr>
<td></td>
<td>iaddr: Address</td>
<td>DIP: Address × Address</td>
<td>DIP′: Address × Address</td>
</tr>
<tr>
<td></td>
<td>iaddr: Address</td>
<td>DIP: Address × Address</td>
<td>DIP′: Address × Address</td>
</tr>
<tr>
<td></td>
<td>iaddr: Address</td>
<td>DIP: Address × Address</td>
<td>DIP′: Address × Address</td>
</tr>
<tr>
<td></td>
<td>iaddr: Address</td>
<td>DIP: Address × Address</td>
<td>DIP′: Address × Address</td>
</tr>
<tr>
<td></td>
<td>iaddr: Address</td>
<td>DIP: Address × Address</td>
<td>DIP′: Address × Address</td>
</tr>
<tr>
<td></td>
<td>iaddr: Address</td>
<td>DIP: Address × Address</td>
<td>DIP′: Address × Address</td>
</tr>
<tr>
<td></td>
<td>iaddr: Address</td>
<td>DIP: Address × Address</td>
<td>DIP′: Address × Address</td>
</tr>
<tr>
<td></td>
<td>iaddr: Address</td>
<td>DIP: Address × Address</td>
<td>DIP′: Address × Address</td>
</tr>
<tr>
<td></td>
<td>iaddr: Address</td>
<td>DIP: Address × Address</td>
<td>DIP′: Address × Address</td>
</tr>
<tr>
<td></td>
<td>iaddr: Address</td>
<td>DIP: Address × Address</td>
<td>DIP′: Address × Address</td>
</tr>
<tr>
<td></td>
<td>iaddr: Address</td>
<td>DIP: Address × Address</td>
<td>DIP′: Address × Address</td>
</tr>
<tr>
<td></td>
<td>iaddr: Address</td>
<td>DIP: Address × Address</td>
<td>DIP′: Address × Address</td>
</tr>
<tr>
<td></td>
<td>iaddr: Address</td>
<td>DIP: Address × Address</td>
<td>DIP′: Address × Address</td>
</tr>
<tr>
<td></td>
<td>iaddr: Address</td>
<td>DIP: Address × Address</td>
<td>DIP′: Address × Address</td>
</tr>
<tr>
<td></td>
<td>iaddr: Address</td>
<td>DIP: Address × Address</td>
<td>DIP′: Address × Address</td>
</tr>
</tbody>
</table>

1: function HandleMemoryRead(iaddr, DIP, M2I, GI2M, GDEP)
2: if capacity(GDEP[iaddr]) ≡ 1 then
3:     for def in GDEP[iaddr] do
4:         DIP′ ← DIP′ ∪ {⟨iaddr, def⟩}
5:     end for
6: else
7:     for maddr in GI2M[iaddr] do
8:         for def in M2I[maddr] do
9:             DIP′ ← DIP′ ∪ {⟨iaddr, def⟩}
10:     end for
11: end for
12: end if
13: return DIP′
14: end function

References

