Supplementary Material

This document servers as the supplementary material of OOPSLA 2019 publication titled "*BDA: Practical Dependence Analysis for Binary Executables by Unbiased Whole-program Path Sampling and Per-path Abstract Interpretation*"[1].

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.

Table 1: SPECINT2000 programs.

Program	Size	# Insn	# Block	# Func
164.gzip	143,760	7,650	707	61
175.vpr	435,888	32,218	2,845	255
176.gcc	4,709,664	378,261	36,931	1,899
181.mcf	62,968	2,977	213	24
186.crafty	517,952	42,084	4,433	104
197.parser	367,384	24,584	2,911	297
252.eon	3,423,984	40,119	7,963	615
253.perlbmk	1,904,632	133,755	12,933	717
254.gap	1,702,848	91,608	9,020	458
255.vortex	1,793,360	109,739	16,970	624
256.bzip2	108,872	6,859	577	63
300.twolf	753,544	57,460	4,280	167

Table 2: Malware samples.

Malware	Size	Report Date
1a0b96488c4be390ce2072735ffb0e49	1,806,356	2018-03-10
3fb857173602653861b4d0547a49b395	163,099	2018-07-17
49c178976c50cf77db3f6234efce5eeb	116,385	2018-03-12
5e890cb3f6cba8168d078fdede090996	18,112	2018-03-14
6dc1f557eac7093ee9e5807385dbcb05	88,520	2018-07-09
72afccb455faa4bc1e5f16ee67c6f915	729,816	2017-05-17
74124dae8fdbb903bece57d5be31246b	21,804	2018-10-09
912bca5947944fdcd09e9620d7aa8c4a	124,366	2018-10-09
a664df72a34b863fc0a6e04c96866d4c	200,976	2018-07-17
c38d08b904d5e1c7c798e840f1d8f1ee	178,781	2017-02-24
c63cef04d931d8171d0c40b7521855e9	88,436	2018-03-14
dc4db38f6d3c1e751dcf06bea072ba9c	124,154	2018-07-17

2 Proof of Theorem 4.1

Theorem 4.1. Using Algorithm 2, the probability $\widetilde{p}$ 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.

$$(\frac{2^{63}}{2^{63}+1})^{2L} \cdot \frac{1}{n} \leq \widetilde{p} \leq (\frac{2^{63}+1}{2^{63}})^{2L} \cdot \frac{1}{n} \tag{1}$$

Proof. First, for any weight w_v, we prove that $\widetilde{w_v}$ follows $\frac{2^{63}}{2^{63}+1} \cdot w_v \leq \widetilde{w_v} \leq w_v$.

$$\begin{cases} exp = \max\left(\lfloor \log w_v \rfloor, 63\right) - 63 \\ sig = \lfloor w_v / 2^{exp} \rfloor \end{cases} \tag{2}$$

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

$$\frac{2^{63}}{2^{63}+1} \cdot \frac{w_1}{w_1 + w_0} \leq \frac{\widetilde{w_1}}{\widetilde{w_1} + \widetilde{w_0}} \leq \frac{2^{63}+1}{2^{63}} \cdot \frac{w_1}{w_1 + w_0} \tag{3}$$

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 4 from inequality 3.

$$\frac{2^{63}}{2^{63}+1} \cdot p_l \leq \frac{\widetilde{w_1}}{\widetilde{w_1} + \widetilde{w_0}} \leq \frac{2^{63}+1}{2^{63}} \cdot p_l \tag{4}$$

Next, we derive the bounds of $\widetilde{p_1}$, the probability of Algorithm **??** choosing branch 1. There are two cases.

(a) If $n < 64$, we directly have $\widetilde{p_l} = \widetilde{w_1}/(\widetilde{w_1} + \widetilde{w_0})$. According to inequality 4, we have the following.

$$\frac{2^{63}}{2^{63}+1} \cdot p_l \leq \widetilde{p_l} \leq \frac{2^{63}+1}{2^{63}} \cdot p_l \tag{5}$$

(b) If $n \geq 64$, $\widetilde{p_1} = \frac{\widetilde{w_1}.sig}{\widetilde{w_0}.sig \times 2^n}$. Note that $\frac{\widetilde{w_1}}{\widetilde{w_0}+\widetilde{w_1}} = \frac{\widetilde{w_1}.sig}{\widetilde{w_0}.sig \times 2^n + \widetilde{w_1}.sig}$. Thus, we have $\widetilde{p_1} \geq \frac{\widetilde{w_1}}{(\widetilde{w_1}+\widetilde{w_0})}$. Combining with inequality 4, we can have $\widetilde{p_1} \geq \frac{2^{63}}{2^{63}+1} \cdot p_l$. On the other hand, $\widetilde{p_1} = \frac{\widetilde{w_1}}{\widetilde{w_0}+\widetilde{w_1}} \cdot \frac{\widetilde{w_0}.sig \times 2^n + \widetilde{w_1}.sig}{\widetilde{w_0}.sig \times 2^n}$. Because $\widetilde{w_1}.sig < 2^{64} \leq 2 \cdot \widetilde{w_0}.sig$, we can have $\frac{\widetilde{w_0}.sig \times 2^n + \widetilde{w_1}.sig}{\widetilde{w_0}.sig \times 2^n} < \frac{\widetilde{w_0}.sig \times 2^n + \widetilde{w_0}.sig \times 2}{\widetilde{w_0}.sig \times 2^n} = \frac{2^{n-1}+1}{2^{n-1}}$. As $n \geq 64$ here, we can have $\widetilde{p_1} = \frac{\widetilde{w_1}}{\widetilde{w_0}+\widetilde{w_1}} \cdot \frac{\widetilde{w_0}.sig \times 2^n + \widetilde{w_1}.sig}{\widetilde{w_0}.sig \times 2^n} < \frac{\widetilde{w_1}}{\widetilde{w_0}+\widetilde{w_1}} \cdot \frac{2^{63}+1}{2^{63}}$. Combining with inequality 4, we can have $\widetilde{p_1} < (\frac{2^{63}+1}{2^{63}})^2 \cdot p_l$. Thus,

$$\frac{2^{63}}{2^{63}+1} \cdot p_1 \leq \widetilde{p_1} \leq (\frac{2^{63}+1}{2^{63}})^2 \cdot p_1 \tag{6}$$

From inequality 5 and 6, the following is true.

$$(\frac{2^{63}}{2^{63}+1})^2 \cdot p_1 \leq \widetilde{p_1} \leq (\frac{2^{63}+1}{2^{63}})^2 \cdot p_1 \tag{7}$$

Similarly, we can prove the bound for $\widetilde{p_0}$.

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

□

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`	▷ memory operation sequence
OUTPUT:	$I2M$:	`Address` → {`AbstractValue`}	▷ map an instruction to abstract addresses accessed by it
	DEP:	`Address` → {`Address`}	▷ map an instruction to the instructions it depends on
	$KILL$:	`Address` → {`Address`}	▷ map an instruction to reaching definitions it kills
LOCAL:	DEF:	`AbstractValue` → `Address`	▷ map an abstract address to its latest definition

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

INPUT:	$iaddr$:	`Address`	▷ the current instruction
	DIP:	`Address` $\times$ `Address`	▷ dependences
	$M2I$:	`AbstractValue` $\rightarrow$ {`Address`}	▷ map an address to its definitions
	$GI2M$:	`Address` $\rightarrow$ {`AbstractValue`}	▷ map an instruction to its accessed addresses
	$GDEP$:	`Address` $\rightarrow$ {`Address`}	▷ map an instruction to its dependences in samples
OUTPUT:	DIP':	`Address` $\times$ `Address`	▷ updated dependences

function HANDLEMEMORYREAD($iaddr$, DIP, $M2I$, $GI2M$, $GDEP$)
  if `capacity` $(GDEP\,[iaddr]) \equiv 1$ **then** ▷ strong dependence
    for def in $GDEP\,[iaddr]$ **do**
      $DIP' \leftarrow DIP' \cup \{\langle iaddr, def\rangle\}$
    end for
  else
    for $maddr$ in $GI2M\,[iaddr]$ **do**
      for def in $M2I\,[maddr]$ **do**
        $DIP' \leftarrow DIP' \cup \{\langle iaddr, def\rangle\}$
      end for
    end for
  end if
  return DIP'
end function

References

[1] Zhuo Zhang, Wei You, Guanhong Tao, Guannan Wei, Yonghwi Kwon, and Xiangyu Zhang. Bda: Practical dependence analysis for binary executables by unbiased whole-program path sampling and per-path abstract interpretation. In *Proceedings of the ACM on Programming Languages archive Volume 3 Issue OOPSLA*, 2019.

[2] Standard Performance Evaluation Corporation. Specint2000 benchmark. `https://www.spec.org/cpu2000/CINT2000/`, 2003.

[3] VirusTotal. Virustotal. `https://www.virustotal.com/`, 2018.