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

ZHUO ZHANG, Purdue University, USA
WEI YOU*, Renmin University of China, China
GUANHONG TAO and GUANNAN WEI, Purdue University, USA
YONGHWI KWON, University of Virginia, USA
XIANGYU ZHANG, Purdue University, USA

Binary program dependence analysis determines dependence between instructions and hence is important for many applications that have to deal with executables without any symbol information. A key challenge is to identify if multiple memory read/write instructions access the same memory location. The state-of-the-art solution is the value set analysis (VSA) that uses abstract interpretation to determine the set of addresses that are possibly accessed by memory instructions. However, VSA is conservative and hence leads to a large number of bogus dependences and then substantial false positives in downstream analyses such as malware behavior analysis. Furthermore, existing public VSA implementations have difficulty scaling to complex binaries. In this paper, we propose a new binary dependence analysis called BDA enabled by a randomized abstract interpretation technique. It features a novel whole program path sampling algorithm that is not biased by path length, and a per-path abstract interpretation avoiding precision loss caused by merging paths in traditional analyses. It also provides probabilistic guarantees. Our evaluation on SPECINT2000 programs shows that it can handle complex binaries such as gcc whereas VSA implementations from the-state-of-art platforms have difficulty producing results for many SPEC binaries. In addition, the dependences reported by BDA are 75 and 6 times smaller than Alto, a scalable binary dependence analysis tool, and VSA, respectively, with only 0.19% of true dependences observed during dynamic execution missed (by BDA). Applying BDA to call graph generation and malware analysis shows that BDA substantially supersedes the commercial tool IDA in recovering indirect call targets and outperforms a state-of-the-art malware analysis tool Cuckoo by disclosing 3 times more hidden payloads.

CCS Concepts: • **Security and privacy** → *Software reverse engineering*; • **Software and its engineering** → *Interpreters*; *Automated static analysis.*

Additional Key Words and Phrases: Path Sampling, Abstract Interpretation, Binary Analysis, Data Dependence

ACM Reference Format:
Zhuo Zhang, Wei You, Guanhong Tao, Guannan Wei, Yonghwi Kwon, and Xiangyu Zhang. 2019. BDA: Practical Dependence Analysis for Binary Executables by Unbiased Whole-Program Path Sampling and Per-Path Abstract Interpretation. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 137 (October 2019), 31 pages. https://doi.org/10.1145/3360563

*Corresponding author

Authors' addresses: Zhuo Zhang, Purdue University, USA, zhan3299@purdue.edu; Wei You, Renmin University of China, China, youwei@ruc.edu.cn; Guanhong Tao; Guannan Wei, Purdue University, USA, {taog,guannanwei}@purdue.edu; Yonghwi Kwon, University of Virginia, USA, yongkwon@virginia.edu; Xiangyu Zhang, Purdue University, USA, xyzhang@cs.purdue.edu.

2475-1421/2019/10-ART137
https://doi.org/10.1145/3360563

1 INTRODUCTION

Binary analysis is a key technique for many applications such as legacy software maintenance [Gallagher and Lyle 1991; Loyall and Mathisen 1993], reuse [Sæbjørnsen et al. 2009; Zeng et al. 2013], hardening [Payer et al. 2015; Wang et al. 2017], debloating [Ferles et al. 2017; Quach et al. 2018], commercial-off-the-shelf software security testing [Li et al. 2017; Rawat et al. 2017], malware analysis [Song et al. 2008; Yin et al. 2007], and reverse engineering (e.g., communication protocol reverse engineering) [Caballero et al. 2007; Lin et al. 2008]. A key binary analysis is program dependence analysis that determines if there is dependence between two instructions. Binary program dependence analysis is much more challenging than source level dependence analysis as symbol information (e.g., types and variables) is lost during compilation and source level data structures, variables, and arguments are compiled down to registers and memory accesses (through registers), which are very generic and difficult to analyze. The analysis is further confounded by indirect control flow (e.g., call instructions with non-constant targets, often induced by virtual methods in object oriented programs), as call targets are difficult to derive statically without type information. The critical challenge in binary dependence analysis is memory alias analysis that determines if memory access instructions may access a same memory location.

Given the importance of binary analysis, there are a number of widely used binary analysis platforms such as IDA [Hex-Rays 2008], CodeSurfer [GrammaTech 2008], BAP [Brumley et al. 2011], and ANGR [UCSB 2008]. Some of them leverage dynamic dependence analysis, which is highly effective when inputs are available. However, inputs or input specifications are largely lacking in many security applications. While symbolic execution and fuzzing may be used to generate inputs, they have difficulties scaling to lengthy program paths and execution states for complex binaries with complicated input constraints. Therefore, most of these platforms additionally adopt the *Value Set Analysis* (VSA), a static analysis method, to address the memory alias problem (and hence the dependence analysis problem). VSA was proposed by Balakrishnan et al. in [Balakrishnan and Reps 2004]. It computes the set of possible values for the operands of each instruction. Aliases of two memory accesses can be determined by checking if their value sets share common (address) values. VSA uses a *strided interval* to denote a set of values. Each strided interval specifies the lower bound, the upper bound and the stride. While being compact, strided intervals feature conservativeness. In many cases, they may become simple value ranges (i.e., intervals with stride 1). As such, even though VSA is sound, it has a number of limitations while being used in practice. Specifically, the possible addresses of many memory accesses often degenerate to the entire memory space such that substantial bogus dependences are introduced; when the set of possible address values of a memory write is inflated, the write becomes extremely expensive as it has to update the value set for all the possible addresses. More discussion can be found in Section 2.1. According to our experiment (see Section 7.2), most publicly available implementations of VSA fail to run on many SPECINT2000 programs [ATA 2018]. In addition, they produce substantial false positives in dependence analysis.

While soundness (i.e., never missing any true positive program dependence) is critical for certain applications such as semantic preserving binary code transformation, for which VSA aims, probabilistic guarantees (i.e., the analysis has a very low likelihood of missing any true positive) are sufficient for many practical applications. For example, a critical and fundamental application of VSA (and dependence analysis) is to derive indirect control flow transfer targets such that precise call graphs can be constructed. The sound and conservative VSA inevitably has a large number of bogus call edges, rendering the resulted call graph not that useful. In contrast, an analysis that can disclose most true (indirect) call edges and have a low chance of missing some may be more useful in practice. Malware behavior analysis [Cozzi et al. 2018] aims to understand hidden payloads of malware samples by reporting the system calls performed by the samples and the

corresponding concrete arguments of these system calls (e.g., file delete system call with directory argument "/home"). Missing a few dependences (by chance) may not critically impact the generated behavior report whereas having a large number of bogus dependences would lead to substantial false positives, significantly enlarging the human inspection efforts. In other applications including functional component identification (for binary debloating) [Ferles et al. 2017], static analysis guided vulnerability detection/fuzzing [Li et al. 2017], and protocol reverse engineering [Lin et al. 2008], dependence analysis with probabilistic guarantees may provide the appropriate trade-offs between effectiveness and practicality.

Therefore in this paper, we propose a binary level program dependence analysis technique with probabilistic guarantees, enabled by a novel randomized abstract interpretation technique. Specifically, our technique samples the space of whole program paths in a fashion that the likelihood of different paths being taken are evenly distributed, not biased by path length. Note that tossing a fair coin at each conditional statement yields a very biased path distribution such that long paths can hardly be reached. Abstract interpretation is performed on individual sample path, which is different from VSA that operates like a data-flow analysis that computes/merges the abstract values from all possible paths at each step of interpretation. To avoid using value ranges or strided intervals for external inputs, our abstract interpretation samples input values from pre-defined distribution. Probabilistic guarantees can be provided depending on the number of samples taken when certain assumptions are satisfied. A context-sensitive and flow-sensitive posterior dependence analysis is performed based on the abstract values computed by the large number of sample interpretations. The analysis is able to reduce the possible false negatives caused by incomplete path sampling. It also features strong updates such that false positives can be effectively suppressed.

Our contributions are summarized as follows.

- We propose a novel whole program path sampling algorithm for general path exploration. We also identify the probabilistic guarantees of our sampling algorithm with certain assumptions. We devise a per-path abstract interpretation technique that is critical for avoiding bogus abstract values and dependences, and a posterior analysis to compensate the possible incompleteness in path sampling.
- We address a number of practical challenges such as handling loops, recursions, indirect jumps, and indirect calls.
- We propose a new binary program dependence analysis enabled by a novel randomized abstract interpretation technique.
- We develop a prototype BDA and evaluate it on SPECINT2000 binaries. Our evaluation shows that it scales to complex binaries including gcc, whereas VSA implementations from popular platforms such as BAP and ANGR fail to produce results for many binaries. When compared to dynamic dependences observed during running these binaries on standard inputs, BDA misses only 0.19% dependences on average. The dependences reported by BDA are 6 times smaller than those by VSA (when it produces results) and 75 times smaller than Alto (another binary dependence analysis tool that scales). We also evaluate BDA in two downstream analysis, one is to identify indirect control flow transfer targets and the other is to study hidden malware behaviors on 12 recent malware samples. In the former analysis, BDA is equally effective as a state-of-the-art commercial tool IDA in identifying indirect jump targets and substantially outperforms in identifying indirect call targets (4 found by IDA on average versus 767 found by BDA on average). In the malware analysis, BDA substantially outperforms a commercial state-of-the-art malware analysis tool Cuckoo [Cuckoo 2014] by reporting 3 times more hidden malicious behaviors. The project is publicly available at [Zhang et al. 2019a].

```
1  #define MAX_LEN 56
2  #define WORD_CNT 1000
3
4  struct Trie{Word *word; Trie *child[26];};
5  struct Word{char val[MAX_LEN]; Trie *node;};
6  struct Dict{long cap; Word words[WORD_CNT];};
7  Dict *dict;
8  Trie *trie;
9
10 void init_dict() {
11   long idx = 0; //index of the longest word
12   dict->cap = WORD_CNT;
13   read_words(dict->words, &idx);
14   output_word(dict->words[idx].val);
15 }
16 void read_words(Word* words, long *idx) {
17   int i, j;
18   for (i = 0; i < WORD_CNT; i++) {
19     words[i].node = trie;
20     for (j = 0; j < MAX_LEN; j++) {
21       words[i].val[j] = read_char();
22       /* Do the following things:
23       1.break_if_line_end(words[i].val[j]);
24       2.update_trie(words[i], j);
25       3.update_longest_idx(idx, j);
26       */
27     }
28     words[i].node->word = &(words[i]);
29   }
30 }
```

Fig. 1. Example to explain the limitations of existing techniques.

2 MOTIVATION

At the binary level, program dependences induced by registers can be easily inferred. The challenge lies in identifying those induced by memory, due to the difficulty of (statically) determining the locations accessed by memory operations. As such, a key challenge in binary dependence analysis, and also in binary analysis in general, is to determine the points-to relations for memory access instructions. In this section, we explain the limitations of existing techniques, present our observations of program dependences (through memory), and motivate the idea of BDA.

2.1 Limitations of Existing Techniques

We use 197.parser from the SPEC2000INT benchmark [ATA 2018] as an example to illustrate the limitations of existing techniques. 197.parser is a word processing program that analyzes the syntactical structure of a given input sentence based on a pre-defined dictionary. Figure 1 presents the simplified code of its dictionary initialization logic. In particular, it sets the the number of words in the dictionary to a pre-defined value (line 12), reads words from the dictionary file (line 13), and then outputs the longest word (line 14). During the process of reading words, 197.parser maintains a dictionary tree (lines 24) and records the index of the longest word (line 25).

The core of memory alias analysis (and also the downstream dependence analysis) is to statically determine the possible runtime values (PRV) of the address operand of a memory access instruction, which could be a register or a memory location. We call such operands *variables* for easy description. While the problem is undecidable in general, a large collection of approximation algorithms have been proposed to provide various trade-offs between efficiency and precision. Among all these efforts, Alto [Debray et al. 1998] and VSA [Balakrishnan and Reps 2004] are two prominent existing efforts. The latter has been the standard for more than a decade.

Alto. Alto abstracts the PRV of a variable as an *address descriptor* $\mathtt{h}insn;OFFSET\mathtt{i}$, where $insn$ is the instruction that computes a base value and $OFFSET$ denotes a set of possible offsets to the base value. For example, assume in line 13 in function `init_dict()` (Figure 1), the address of `dict->words` is loaded to register `rdi` by two instructions. The first instruction `i` loads the base `dict` and the second instruction `j` adds the offset of field `words`, which is 8. The address descriptor of `rdi` after `i` and `j` is hence `hi;f0x8gi`. Alto only models PRV computation through register operations, not through memory reads and writes. For an instruction `i` that loads a value from a memory location to a register, Alto resets the PRV of the register to a new address descriptor `hi;f0x0gi`, not being able to inherit the address descriptor stored by the latest memory write to the location. As such, it has to conservatively consider a memory read with a new descriptor can

read from any address, and have dependence with any memory write, causing substantial false positives. For example in function `init_dict()`, Alto considers the read of `dict->words` at line 13 is dependent on the write of `dict->cap` at line 12.

VSA. VSA computes PRV by abstract interpretation, modeling operations through both registers and memory. It abstracts the PRV of a variable as a *strided interval* $s\,[lb, ub]$, where lb and ub specify the lower bound and upper bound of the interval and s is the stride between values in the interval. Intuitively, the strided interval represents the set of integers $\{lb, lb+s, lb+2s, \ldots, ub\}$. Each strided interval may be associated with a memory region, which could be heap (denoted as H_a where a is the allocation site), stack (denoted as S_f where f is the corresponding function), or general for non-heap and non-stack values (denoted as G). There is a special value $\top$, which indicates all possible values. VSA computes strided intervals following a set of rules. For example, the `addition` rule is defined as follows. Let $SI_1 = s_1\,[lb_1, ub_1]$ and $SI_2 = s_2\,[lb_2, ub_2]$ be two strided intervals, and $SI_3 = SI_1 + SI_2$. Then we have the following equation (1), with ∂cd^{10} the greatest common divisor. Observe that the rule is conservative, meaning SI_3 is a super-set of the all the possible sums of the values in SI_1 and SI_2.

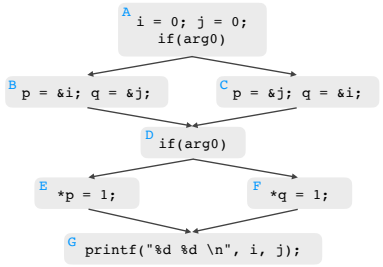
$$SI_3 = \partial cd^{10}(s_1, s_2)\,[lb_1 + lb_2, ub_1 + ub_2] \tag{1}$$

The major limitation of VSA is over-approximation. According to equation (1), abstract interpretation may induce bogus PRV at each instruction, due to both the ∂cd^{10} operation and the simple approximation of lower and upper bounds. Since there are typically a large number of interpretation steps in whole-program analysis, the bogus dependences are aggregated and magnified, making end results not usable. For example, the write of `words[i].node->word` at line 28 has false dependence with any following memory read according to VSA. Appendix A will discuss this in details.

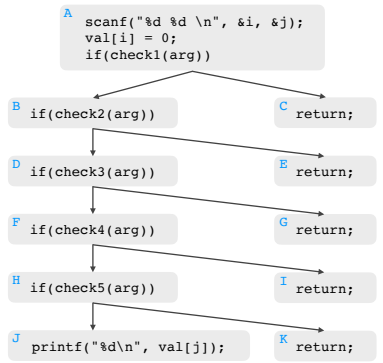
2.2 Observations

Different analyses entail different kinds of sensitivity. For example, the simplest type inference could be path-insensitive, context-insensitive, and even flow-insensitive. As one of the most complex analyses, dependence analysis is flow-sensitive, context-sensitive, and path-sensitive. However, a key observation is that *a dependence relation, which means dependence through memory in our context, can be disclosed by many whole-program paths.* In other words, even though it is context- and path-sensitive, the level of sensitivity is limited. Intuitively, given a program with n statements, the number of dependences is $O(n^2)$, whereas the number of paths could be $O(2^n)$, assuming all branching statements have only two branches. Hence, a dependence may be exposed by many paths. Consider the code snippet `example1` in Figure 2, whose control flow graph is shown in Figure 2a. There are four possible paths, three of which can expose the dependence between lines 12 and 6 regarding variable `i`. Similarly, three paths can expose the dependence between lines 12 and 7 regarding `j`. Essentially, a dependence is likely exposed if one of its exhibition paths is taken. Program dependences are also input sensitive, meaning that a dependence may or may not be present along a same program path depending on input values. Consider `example2` in Figure 2. Variables `i` and `j` denote input and are used as array indices. Note that the code has only one path, and the dependence between lines 17 and 18 may or may not be exercised depending on the values of `i` and `j`. According to [Yang and Gupta 2002], *run time values of program variables likely fall into a small range.* In our example, assuming both variables have a uniform distribution in range $[0, c]$, the likelihood of the dependence being exercised is $\frac{1}{c}$. If the path is taken n times with randomly sampled `i` and `j` values, the likelihood becomes $1 \quad 1 \quad \frac{1}{c}^{n}$, which is close to 1 when n is large.

```
1  #define MAX_LEN 56
2  char val[MAX_LEN];
3  int i, j, *p, *q;
4
5  void example1(int arg0, int arg1) {
6    i = 0;
7    j = 0;
8    if (arg0) { p = &i; q = &j; }
9    else { p = &j; q = &i; }
10   if (arg1) *p = 1;
11   else *q = 1;
12   printf("%d %d\n", i, j);
13 }
14
15 void example2(char arg) {
16   scanf("%d %d\n", &i, &j);
17   val[i] = arg;
18   printf("%d\n", val[j]);
19 }
20
21 void example3(int *arg) {
22   scanf("%d %d\n", &i, &j);
23   val[i] = 0;
24   if (check1(arg)) return;
25   if (check2(arg)) return;
26   if (check3(arg)) return;
27   if (check4(arg)) return;
28   if (check5(arg)) return;
29   printf("%d\n", val[j]);
30 }
```



(a) CFG of example1.



(b) CFG of example3.

Fig. 2. Examples to illustrate our observations and our technique.

2.3 Our Technique

We propose a sampling based abstract interpretation technique for dependence analysis. Specifically, following a novel algorithm, BDA samples inter-procedural program paths in a way that the likelihood of different paths being sampled follows a uniform distribution, without being biased by path length. In other words, BDA is able to sample as many unique paths as possible given a limited budget. For each sample path, abstract interpretation is performed to compute the possible values for individual instructions. During abstract interpretation, external inputs (e.g., user inputs) are randomly sampled from pre-defined distributions; calling contexts are explicitly denoted as call strings; stack memory is denoted as a stack frame with offset; heap memory is denoted by its allocation site; abstract values are updated based on instruction semantics; memory reads/writes are modeled through an abstract store; and path feasibility is partially modeled (details can be found in Section 5). Note that the abstract interpretation in BDA is not based on strided intervals. Instead, it is to-some-extent similar to concrete execution, computing a single abstract value at each instruction instance. The values associated with a static instruction is the union of all the values derived for individual instances of the instruction. In the mean time, it is still quite different from concrete execution, which has extreme difficulty ensuring memory safety when path feasibility is not fully modeled, or concrete external inputs are not available. After aggregating the values derived from individual samples, BDA performs an additional posterior analysis to mitigate the possible incomplete path coverage during sampling. The analysis merges values computed along different branches at each control flow joint point and then cross-checks the address values of memory access instructions to detect dependences. The value merge allows dependences that belong to un-sampled paths to be disclosed with high likelihood.

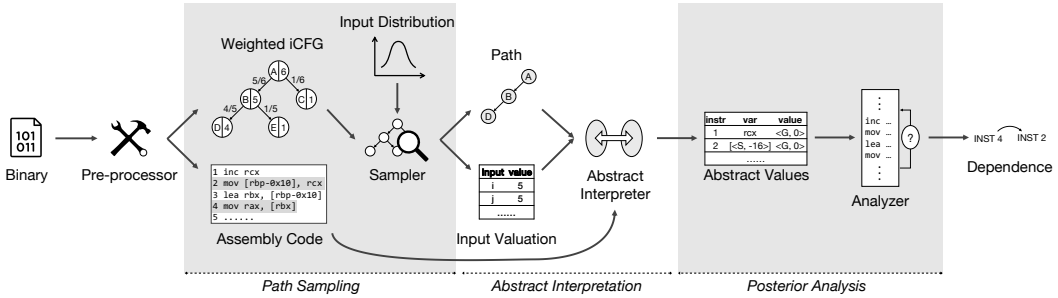


Fig. 3. Architecture of BDA.

Note that a naive sampling algorithm that tosses a fair coin at each conditional jump instruction does not work. Consider `example3` in Figure 2 with CFG in Figure 2b. With naive sampling, the path A ! C gets $\frac{1}{2}$ chance to be taken, while the path A ! B ! D ! F ! H ! J has only $\frac{1}{32}$. With the sampling algorithm in BDA, the six paths in the code have an equal chance to be taken. Assuming i and j have the range of »0; 99¼, BDA guarantees that the dependence between lines 23 and 29 is covered with 99.74% when 60 sample paths are taken. Coming back to our 197.parser example in Figure 1, BDA is able to disclose all the true positive dependences in the two functions without generating any false positives.

3 DESIGN

The architecture of BDA is shown in Figure 3. It consists of four components: including pre-processor, sampler, abstract interpreter and analyzer. The pre-processor disassembles the given binary to get its assembly code and generates its inter-procedural control flow graph (iCFG) with call edges and return edges explicitly represented. Each basic block of iCFG is weighted by the number of possible inter-procedural paths starting from the block. The sampler samples path based on the weights of blocks and samples external input values based on the pre-defined distributions.

Given a sampled path and input valuation, the abstract interpreter interprets the instructions along the path and computes the abstract values of operands at each instruction. The abstract values for individual instructions are passed to the analyzer for posterior memory dependence analysis. At last, BDA outputs a list of pairs of memory-dependent instructusions as analysis results. In the next a few sections, we discuss the details of the individual components.

4 PATH SAMPLING

In the sampling step, BDA takes a binary executable and its inter-procedural control flow graph (*iCFG*), generates a given number of whole-program path samples. The sampling follows a uniform distribution of the space of unique paths. As mentioned in Section 2, a simple sampling algorithm that tosses a fair coin at each predicate has strong bias towards short paths. Note that we use an iterative method to handle *iCFG* in the presence of indirect calls, which will be discussed in Section 4.3.

The basic idea of our sampling algorithm is as follows. For each branching instruction, BDA computes the number of inter-procedural program paths starting from the branch. Sampling bias for the instruction is hence computed from the path counts. Intuitively, a branch leading to more paths has a higher probability to be taken. In order to realize the idea, we address the following two prominent challenges: (1) how to compute the number of inter-procedural paths (in the presence of function calls, loops, and even recursion); and (2) how to sample a strongly-biased distribution as it often occurs that one branch of a conditional statement has a very small number of paths

(e.g., those exit upon an error condition) while the other branch has a huge number of paths (e.g., beyond the maximum integer that can be represented in 64 bits). We also study the probabilistic guarantee of our sampling algorithm.

4.1 Path Counting

Our path counting algorithm is inspired by the seminal path encoding algorithm in [Ball and Larus 1996a]. In Ball-Larus (BL) path encoding, the number of paths starting from a node is the sum of the numbers of paths of its children. It transforms a CFG to its acyclic version (e.g., by removing back-edges) and then computes the the path count for each node in a reverse topological order. Figure 4 shows the path count for each node (called *node weight* from this point on) for the code in Figure 2b. Each node is annotated with node id and its weight. Observe that the leaf nodes have weight 1. Then node H is computed to have weight 2, F has weight 3, and so on. The fractions along edges denote the sampling bias. For example, at node A, the chance to take $A\ !\ B$ is $\frac{5}{6}$ whereas $A\ !\ C$ is $\frac{1}{6}$. The probabilities of taking the 6 different paths are all $\frac{1}{6}$. However, the BL path counting algorithm is intra-procedural and does not consider loop iterations. Hence, we propose a new whole-program path counting algorithm. To simplify our discussion, we assume the subject program is loop-free and recursion-free, but has calls and returns. Moreover, each callee must return to its caller and there are no indirect calls. In Section 4.3, we will explain how to address these practical issues.

In order to handle inter-procedural path counting, we have to precisely determine the weight (i.e., the number of paths) of an invocation instruction. The key observation is that the weight of an invocation to a callee function `foo()` is the product of the number of inter-procedural paths from the entry of `foo()` to the exit of `foo()`, including paths in the callees of `foo()`, and the weight of the instruction right after the invocation instruction in the caller. The former is called the *callee paths* and the latter is called the *continuation paths*.

The procedure is explained in details in Algorithm 1. It takes the inter-procedural CFG of the binary, and computes the weight for each node, *which denotes the number of inter-procedural paths from the node to the exit of its enclosing function*. Since the input *iCFG* does not have loops or recursion, we can perform topological sort on the graph. Intuitively, one can consider that we first sort the call graph and then sort the nodes inside each function. The loop in lines 2-15 traverses each node in the reverse topological order. If it is a return instruction, its weight is set to 1 (line 4). If it is a call, the weight is computed as the product of the weight of the return address and the weight of the entry point of the callee (i.e., the number of inter-procedural paths inside the callee). Since a method may have a huge number of such paths, which we assume to be bounded by 2^K, the complexity of such product is $O^1K\ I o \partial^1 K^{\infty}$. In practice, we find using $K = 600{;}000$ bits to represent weights is enough. In lines 10-13, if the node is neither call nor return, its weight is the sum of the children weights.

Example. Consider the example in Figure 5, which has three functions `main()`, `gee()`, `foo()`, with both `main()` and `foo()` calling `gee()`. The weighted *iCFG* is shown in Figure 6. Following reverse topological order, `gee()` is processed first. As such, $W\,\text{»}A¼ = 1$ and $W\,\text{»}D¼ = 2$ as there are two paths inside `gee()`. Inside `foo()`, $W\,\text{»}E¼ = 1$ as it is a return; $W\,\text{»}G¼ = W\,\text{»}E¼ + W\,\text{»}F¼ = 2$, and $W\,\text{»}H¼ = W\,\text{»}D¼\quad W\,\text{»}G¼ = 4$. Similarly, in `main()`, $W\,\text{»}N¼ = W\,\text{»}I¼\quad W\,\text{»}K¼ = 4$, $W\,\text{»}M¼ = W\,\text{»}D¼\quad W\,\text{»}L¼ = 2$, and $W\,\text{»}O¼ = W\,\text{»}N¼ + W\,\text{»}M¼ = 6$, meaning there are 6 whole-program paths. The bottom of Figure 6 shows the probability of the red path being taken, which is exactly $\frac{1}{6}$, same for the others.

Note that the computed path counts can be directly used in path sampling, even though the weight of node only denotes *the number of paths from the node to the end of its enclosing function* (denoted as x), not *the number of paths from the node to the end of the program* (denoted as y). The

Algorithm 1 Path Counting

```
Input:  iCFG   . loop-free and recursion-free iCFG of the target binary
Output: W:     . weight (i.e., path count) for each node, a K-bits integer
1: function PathCounting(iCFG)
2:     for iaddr in reverse topological order of iCFG do
3:         if iaddr is a return node then                . for a return node, initialize its weight to 1
4:             W »iaddr¼ 1
5:         else if iaddr is a call node then             . function invocation intruction
6:             callee    call target of iaddr
7:             ret_addr  the instruction right after iaddr
8:             W »iaddr¼ W »ret_addr¼ W »callee¼        . K-digits multiplication, with complexity O ¹K log K º
9:         else                                          . other instructions
10:            W »iaddr¼ 0
11:            for succ in successors of iaddr do
12:                W »iaddr¼ W »iaddr¼ + W »succ¼
13:            end for
14:        end if
15:    end for
16:    return W
17: end function
```

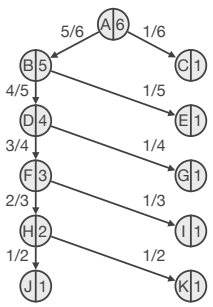


Fig. 4. Weighted CFG for Fig.2b

```
void gee(int *a) {
  if (input()) *a=0;
  else *a=2;
}
void foo(int *a) {
  gee(a);
  if (input()) *a+=1;
}
int main() {
  int a;
  if (input()) gee(&a);
  else foo(&a);
}
```

Fig. 5. Code example with functions

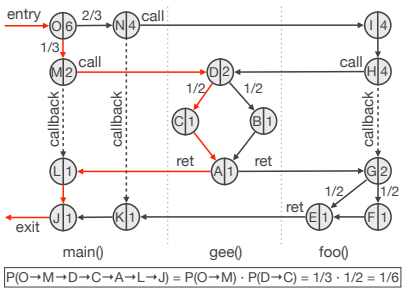


Fig. 6. Weighted iCFG for Fig.5

reason is that y equals to x times *the number of continuation paths of the enclosing function* (denoted as z), multiplying the same z on both branches of a predicate does not change sample bias. Consider the example in Figure 6, nodes C and B have weight 1 (i.e., $x = 1$) although there are 2 paths from either to the end of the program (i.e., $y = 2$). However, using either scheme yields the sampling bias at D (i.e., 1 against 1 versus 2 against 2).

4.2 Path Sampling and Probability Analysis

Given the pre-computed weights, our path sampling is to toss a biased coin at a predicate. The predicate bias is locally computed from the weights of the predicate and its children. Since there are substantial variations in weight values (e.g., 1 versus 2^{1000}), we have to design a special procedure to simulate the biased distribution, which is presented in Algorithm 2. In the subsequent section, we will show how to achieve uniform distribution for whole-program path sampling using this algorithm and demonstrate its correctness and effectiveness. To simplify discussion, we only consider sampling a predicate of two branches, whose weights are w_0 and w_1 with $w_0 > w_1$ without losing generality. The algorithm is to simulate picking branch 0 with the (approximate) probability of $\frac{w_0}{w_0+w_1}$ and branch 1 with $\frac{w_1}{w_0+w_1}$. Sampling more branches can be easily extended. Due to the

frequent invocation of the sampling function (for each predicate), we develop an efficient algorithm with $O(1)$ expected complexity (not worst-time complexity). Observe that what we need is a ratio between weights, instead of precise weights. Inspired by the floating point representation, we introduce an approximate representation of weights. Specifically, each weight is transformed to two 64-bit values: sid and exp, analogous to the significant and exponent in floating point representation, respectively. They satisfy the following, with wf_v an approximation of weight value w_v.

$$\langle sid, exp \rangle \quad sid \quad 2^{exp} \approx wf_v \tag{2}$$

To minimize representation error, sid and exp are derived as follows.

$$\begin{aligned} exp &= \max(\lfloor \log w_v \rfloor - 63, 0) \\ sid &= \lfloor w_v \cdot 2^{exp} \rfloor \end{aligned} \tag{3}$$

Algorithm 2 Branch Selection

```
INPUT:  w0, w1:                                  . weights with w0  w1 without losing generality
OUTPUT: 0 or 1:                                                    . the branch to choose
LOCAL:  wfi : ⟨sid, exp⟩   . approximate representation of weight, consisting of significant bit and exponent
1:  function SELECTBRANCH(w0, w1)                                  . Random pick one ID based on weight
2:      (wf0, wf1) ← (approximate(w0), approximate(w1))
3:      n ← wf0.exp − wf1.exp
4:      if n ≥ 64 then
5:          for i in range(n) do
6:              if random(2) = 0 then                  . random(n) returns k (0 ≤ k < n) with probability 1/n
7:                  return 0
8:              end if
9:          end for
10:         return (random(wf0.sid) < wf1.sid)                     . wf0.sid − 2 must be larger than wf1.sid
11:     else
12:         return (random(wf0.sid − 2^n) + wf1.sid < wf1.sid)
13:     end if
14: end function
```

Taking $2^{65} - 1$ as an example, it is represented as $\langle 2^{64} - 1, 1 \rangle$, which introduces an error of $\frac{(2^{64}-1)\cdot 2^1}{2^{65}-1} = 2.7e-20$. With the representation, Algorithm 2 describes the sampling procedure. Specifically, if the exponent difference between w_0 and w_1 is smaller than 64, in line 12, BDA randomly samples a value in $[0, wf_0.sid - 2^n + wf_1.sid)$ and then checks if it is smaller than $wf_1.sid$. If so, branch 1 is selected; otherwise 0, denoting the probability of $\frac{wf_1.sid}{wf_0.sid - 2^n + wf_1.sid}$. When the exponent difference is larger than 64, it first leverages a loop in lines 5-9 that tosses a fair coin n times and selects 0 when any of the n coins is 0. If all n tries yield 1, which has the probability of $\frac{1}{2^n}$, line 10 further samples with a probability of $\frac{wf_1.sid}{wf_0.sid}$, to approximate the intended probability, as w_1 is very small compared to w_0 and hence it is negligible when added to w_0.

THEOREM 4.1. *Using Algorithm 2, the probability p of any whole-program path being sampled satisfies equation 4, in which n is the total number of whole-program paths and L is the length of the longest path, which can be considered as $O(x)$ with x the number of nodes in iCFG.*

$$\left(\frac{2^{63}}{2^{63}+1}\right)^{2L} \frac{1}{n} \quad p \quad \left(\frac{2^{63}+1}{2^{63}}\right)^{2L} \frac{1}{n} \tag{4}$$

Due to the space limitations, we omit the proof, which can be found in our supplementary material [Zhang et al. 2019b].

By applying Taylor's Theorem to inequality (4), we can derive inequality (5). In practice, the length L of the longest path of any binary executable (without loops or recursion) satisfies $L \quad {}^{1}2^{63} + 1^{o}$, the approximation is hence very tight around $\frac{1}{n}$. For example, 176.gcc's longest path is nearly 40000, such that $\frac{1 \;\; {}^{1}8e \;\; 15^{o}}{n} \quad \text{ℓ} \quad \frac{1+{}^{1}8e \;\; 15^{o}}{n}$.

$$ {}^{1}1 \quad \frac{2L}{2^{63}+1}{}^{o} \;\; \frac{1}{n} \quad \text{ℓ} \quad {}^{1}1 + \frac{2L}{2^{63}+1}{}^{o} \;\; \frac{1}{n} \tag{5} $$

We should note that a simple random number generator would not work because of the limitation of floating point representation. Considering selecting a branch with possibility $1e \;\; 1000$ represented via $1 : 1e+1000$, it would be transformed to $2e \;\; 308$ (the minimal representable positive value in float64), suggesting over $2e+692$ times undesirable amplification of the likelihood. This would lead to heavily biased sampling. Lumbroso [2013] proposed a heavy-weight algorithm to accurately sample from strongly-biased distribution, whose average-case time complexity is $O^{1}\log^{1}p + q^{oo}$ when sampling from $p:q$. In contrast, Algorithm 2 samples in $O^{1}1^{o}$ with negligible precision loss, and hence is more desirable in our context where the sampling function is frequently invoked.

Probabilistic Guarantee for Disclosing Dependence. As mentioned in Section 2, a (memory) dependence may be disclosed by many paths. Assume m out of total n paths disclose a dependence, and let $k = \frac{m}{n}$. Following our path sampling algorithm, in a path sample, the probability p_d of observing a given dependency d satisfies inequality (6).

$$ \frac{2^{63}}{2^{63}+1}{}^{2L} \quad k \quad p_d = \text{ℓ} \quad m \quad \frac{2^{63}+1}{2^{63}}{}^{2L} \quad k \tag{6} $$

For N samples, the probability P_d of disclosing dependency d at least once has a lower bound mentioned in inequality (7).

$$ P_d = 1 \quad {}^{1}1 \quad p_d{}^{oN} \quad 1 \quad 1 \quad \frac{2^{63}}{2^{63}+1}{}^{2L} \quad k \quad {}^{!N} \quad 1 \quad {}^{1}1 \quad k^{oN} \tag{7} $$

Inequality (7) offers a strong guarantee for finding dependency in practice. Taking 176.gcc as an example, if $L = 40000$, $k = 0{:}0005$ and $N = 10000$, we would have $P_d \quad 99{:}32\%$, which means that the chance of missing the dependence is only $0{:}68\%$.

4.3 Addressing Practical Challenges

Handling Loops. Our discussion so far assumes loop-free and recursion-free programs. BDA distinguishes two kinds of loops and handles them differently. The first kind is loops whose iteration numbers are not external input related. We call it *constant loops*. The other kind is input related, called *input-dependent loops*.

For an input-dependent loop, it is intractable to determine how many times it iterates. A standard solution is to compute a fix-point, which often entails substantial over-approximation. Hence, our design is to bound the number of iterations. A naive solution is to give a fixed bound for all input-dependent loops. However, this could cause non-trivial path explosion in the presence of nesting loops. Hence, we bound the total number of iterations across all the nesting loops within a function. Such a design also allows easy computation of weight values. Assume the bound for each function is $t = 3$, Figure 7a illustrates the idea. For each function F, BDA clones the function t times, denotes as $F_0, \vdots\vdots, F_{t \;\; 1}$. For each back-edge in F_i, we reconnect it to the corresponding loop head in F_{i+1}. For example, back-edge a in Figure 7 becomes a_1, a_2 and a_3 connecting different versions of F. Note that in the transformed graph at most $t = 3$ back-edges could be taken (e.g., a 3 times and b 0 times; a 2 times and b 1 time; and so on).

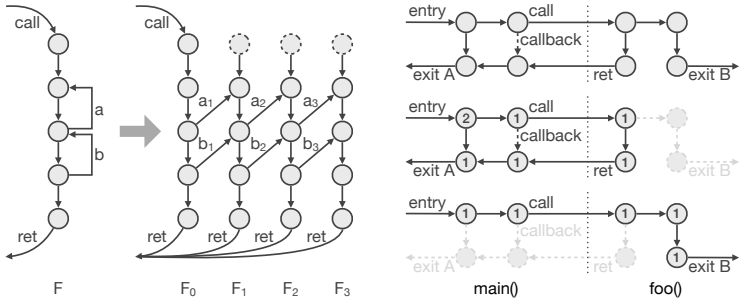


Fig. 7. Example to show how graph transformation works for loop

For constant loops, which are commonly used in initialization, BDA allows them to iterate as many times as them are supposed to. As such, the constant loop predicates are not part of the path samples generated in this phase. We will show in the next section that our abstract interpreter directly handles such loops without referring to path sample. Recursion is handled in a way similar to input-dependent loops. Details are elided.

Handling Multi-exits. So far we assume every function in *iCFG* returns to its caller. In practice, many functions may just exit without returning, posing challenges for path counting. Our solution is to count the paths that must exit without return and those that must return separately. We use the sample graph on the top of Figure 7b to illustrate the basic idea. In the graph, `main()` (on the left) calls `foo()` (on the right), which may exit without return. The essence of our solution is to count the two sub-graphs below separately and sum them up. Specifically, the sub-graph in the middle corresponds to the must-return behavior, whereas the sub-graph on the bottom corresponds to the exit behavior. The number inside each node denotes its weight. As such, there are $2 + 1 = 3$ whole-program paths.

Handling Indirect Calls. We use an iterative method to handle indirect calls. Specifically, after initial path sampling, BDA abstract-interprets the samples. If new call targets are identified during abstract interpretation, the *iCFG* is updated, weights are re-computed, and another round of sampling is performed. The sampling algorithm terminates when no new indirect-calls are found within a time budget. For example, 7836 indirect calls are identified for 254.gap (by BDA) within 5.67 hours, as shown Table 7.

Edge Coverage. If we strictly follow the unbiased whole-program path sampling algorithm, some statements may not be covered. Consider a predicate with two branches, one has weight 1 and the other has weight 2^{1000}. The statement in the short branch may not be covered at all. To address the problem, BDA collects a set of additional samples (usually much smaller than the path samples) to cover control flow edges that have not been covered.

Sampling External Inputs. External input values are sampled from pre-defined value ranges in a uniform fashion. One can consider this input sampling procedure produces an external input valuation that assigns each instance of an external input read instruction with a random value.

Probabilistic Guarantees in Practice. With the additional machinery to handle practical challenges, Theorem 4.1 and bound (7) only hold when the following assumption is satisfied: *the graph transformations to handle loops and recursion must not change k, the probability a dependence is disclosed by a whole-program path.* Furthermore, k may be difficult to derive in practice due to the undecidable nature of the problem. However, our experiment in Section 7 illustrates that the algorithm is effective in practice and the results are consistent with our theoretical analysis.

⟨Program⟩	P	F	S
⟨Statement⟩	S	F	$S_1;S_2 \mid r$ B $e \mid r$ B **R**$(r_a) \mid$ **W**$(r_a, r_v) \mid r$ B **malloc**$() \mid r$ B **input**$()$ $\mid$ **call**$(a) \mid$ **ret** $\mid$ **goto**$(a) \mid$ **if** r **then goto**(a)
⟨Expression⟩	e	F	$r \mid v \mid r_1\ op\ r_2 \mid r\ op\ v$
⟨Operator⟩	op	F	$+ \mid - \mid * \mid / \mid \ldots$
⟨Register⟩	r	F	$\{sp, r_1, r_2, \ldots\}$
⟨AbstractValue⟩	v	F	$\langle m, c \rangle$
⟨MemoryRegion⟩	m	F	$G \mid H_a^c \mid S_a^c$
⟨Const⟩	c	F	{0,1,2,...}
⟨Address⟩	a	F	{0,1,2,...}

Fig. 8. Language.

5 ABSTRACT INTERPRETATION

We explain the abstract interpretation semantics in this section. Given a predefined sample path, represented by a sequence of addresses, and an external input valuation that associates each instruction instance that reads external input with a value sampled from some pre-defined distributions, the abstract interpreter follows the path to compute abstract values for each instruction instance. It models both register and memory reads and writes, e.g., supporting writing an abstract value to an abstract address. If the branch outcome of a loop predicate is not dependent on any external input (e.g., loop predicate with a constant loop bound), BDA does not resort to the path sample, but rather follows the branch based on the abstract value of the predicate. It explicitly represents and updates an abstract call stack, in order to precisely represent stack memory addresses. In addition, the interpretation of arithmetic operations (e.g., additions and subtractions) is precise, without causing any precision loss as that in computing strided intervals in VSA.

Abstract interpretation is essential for BDA. In contrast, an alternative design of using concrete execution to expose dependence is less desirable. Note that in concrete execution, without knowing input specification, the sampled inputs may not satisfy format constraints, leading to early termination. Additionally, concrete execution may have stack/heap reuse, leading to substantial false dependences in the whole-program posterior-analysis, which is necessary and will be explained in Section 6.

Language. To facilitate discussion, we introduce a low-level language to model binary executables. The language is designed to illustrate our key ideas, and hence omits many features (of x86). The implementation of BDA supports these complex features present in real-world binary executables (even though they may not be modeled by our language). The syntax of the language is shown in Figure 8. $\mathbf{R}^1 r_a{}^0$ and $\mathbf{W}^1 r_a; r_v{}^0$ model memory read and write operations, respectively, where register r_a holds the address and register r_v holds the value to write. Heap allocation functions (e.g., `calloc` and `mmap`) are modeled as **malloc**. The allocated size is irrelevant in our analysis and hence elided. External input functions (e.g., `fread` and `scanf`) are modeled by **input**. Other general function calls and returns are modeled by **call** and **ret**. The address of the target function of **call** is a. We assume parameter passing across functions is done explicitly through register and memory read/write instructions. We model the stack pointer register sp to facilitate computing stack related abstract values. In addition, control flow statements (in high-level languages), such as conditional and loop statements, are modeled using **goto** and guarded **goto**.

Abstract values are represented as $\langle m; c \rangle$, where m stands for a memory region and c stands for the offset relative to the base of the region. The memory space is partitioned to three disjoint regions: global, stack and heap. The global region, denoted as G, stands for the locations holding initialized and uninitialized global data, such as the `.data`, `.rodata` and `.bss` segments of an ELF

```
pc 2 ProgramCounter F Address
IS 2 InstructionSize F Address ! Const
IC 2 InvocationCount F Address ! Const
LP 2 LoopPredicate F Address ! Bool
MS 2 MemStore F AbstractValue ! AbstractValue
RS 2 RegStore F Register ! AbstractValue
MT 2 MemTaint F AbstractValue ! Bool
RT 2 RegTaint F Register ! Bool
PA 2 PATH F »Address¼
RV 2 RandomInputValuation F ¹Address Const⁰ ! Const
CS 2 CallStack F »Address Integer Address AbstractValue¼
MOS 2 MemOpSeq F »Address AbstractValue¼
```

```
CalcValue(op, v1, v2) F
  if v1:m G then
    v3 hv2:m; v1:c op v2:ci; t false;
  else if v2:m G then
    v3 hv1:m; v1:c op v2:ci; t false;
  else
    v3 hG; RV »hpc; IC »pc¼i¼i; t true;
  end if
  return hv3; ti;
```

```
NormalizeVal(v) F
  if v:m S then
    CS⁰ CS;
    while v:c > 0 and :CS⁰:empty¹⁰ do
      h ; ; ; vt i CS⁰:pop¹⁰;
      v:m vt:m; v:c v:c + vt:c;
    end while
  end if
  return v;
```

Fig. 9. Definitions.

file. A stack region, denoted as S_a^c, models a stack frame that holds local variable values for the c-th invocation instance of the function at address a. A heap region, denoted as H_a^c, models a memory region allocated in the c-th invocation instance of the allocation instruction at *program counter* (pc) address a. A non-address constant value can be expressed as having $m = G$. Note that in our interpretation, an instruction may be encountered multiple times in a sample path and we distinguish these different instances. In contrast, VSA does not; instead it merges the abstract values for all possible instances, which is an important source of inaccuracy.

Definitions. Figure 9 introduces a number of definitions that are used in the semantic rules. We use pc to denote the program counter that indicates the address of current instruction, IS to denote the size of each instruction, IC to represent the current instance of an instruction, and LP to indicate whether the current instruction is a loop predicate. MS denotes the abstract value store that maps an abstract memory address value to the abstract value stored at that address, and RS denotes the register store that maps a register to its abstract value. MT and RT represent the taint stores for memory and registers, respectively. The taint tag of an abstract value indicates if the value has been directly/transitively computed from some (randomly sampled) external input. In other words, there is data flow from some external inputs to the abstract value. A sample path is denoted by PA, which is a list of addresses ordered by their appearance in the path. A sampled external input valuation RV assigns a sampled value to each instance of an external input instruction. Both PA and RV are generated by the previous sampling phase and provided as inputs to the abstract interpretation process. We use CS to explicitly model call stack. It is a list of four-element tuples, denoting respectively the invocation site, its instance, the return address, and a copy of the abstract value of the sp register which is supposed to be updated upon function invocation. The outcome of abstract interpretation MOS contains the abstract values for each memory access instruction encountered.

Table 1. Interpretation rules.

Rule	Statement	Actions
Read	r:=**R**(r_a)	IC »pc¼++; v:=**NormalizeVal** ¹RS »r_a¼º; RS »r¼:=MS »v¼; RT »r¼:=MT »v¼_RT »r_a¼; MOS:enqueue ¹ħpc; v i º; pc := pc+IS »pc¼;
Write	**W**(r_a, r_v)	IC »pc¼++; v:=**NormalizeVal** ¹RS »r_a¼º; MS »v¼:=RS »r_v¼; MT »v¼:=RT »r_v¼_RT »r_a¼; MOS:enqueue ¹ħpc_0; v i º; pc := pc_E+IS »pc¼;
Malloc	r:=**malloc**()	IC »pc¼++; RS »r¼:= $H_{pc}^{IC\text{»}pc\text{¼}}$; 0x0 ; pc := pc+IS »pc¼;
Input	r:=**input**()	IC »pc¼++; RS »r¼:=RV »ħpc; IC »pc¼i¼; RT »r¼:=true; pc := pc+IS »pc¼;
Goto	**goto**(a)	IC »pc¼++; pc := a;
If-Goto	**if** r **then goto**(a)	IC »pc¼++; a_t := ¹RS »r¼ . ħG; 0i ? a : pc+IS »pc¼º; pc := :RT ¹r º ^ LP ¹pc º ? a_t : PA:pop¹º;
Call	**call**(a)	IC ¹pc º++; pc^0 := pc; pc := PA:pop¹º; if ¹pc == aº fCS:push¹pc^0; IC»pc^0¼; pc^0 + IS»pc^0¼; RS»sp¼º; RS »sp¼:= D $S_{pc}^{IC\text{»}pc\text{¼}}$; 0x0 ; E };
Ret	**ret**	IC »pc¼++; ħ ; ; pc; RS »sp¼i:=CS:pop¹º;
Expr1	r_t:=r_1 *op* r_2	IC »pc¼++; ħRS »r_t¼; ti:=**CalcValue** ¹*op*; RS »r_1¼; RS »r_2¼º; RT »r_t¼:=RT »r_1¼_RT »r_2¼_t; pc := pc+IS »pc¼;
Expr2	r_t:=*r op v*	IC »pc¼++; ħRS »r_t¼; ti:=**CalcValue** ¹*op*; RS »r¼; v º; RT »r_t¼:=RT »r¼_t; pc := pc+IS »pc¼;

Semantics Rules. The semantic rules are presented in Table 1. Upon interpreting an instruction, the instance count IC is incremented by one. Rule **Read** describes the semantics of memory read. It invokes an auxiliary procedure **NormalizeVal()** to normalize the abstract (address) value in register r_a, denoted as RS»r_a¼. As shown in Figure 9, if the value is a global or heap value, it is returned directly. Otherwise, it is checked to identify the enclosing stack frame of the address. Note that it is common for an instruction to access a stack location beyond the current stack frame (e.g., access an argument passed from the caller function). The procedure traverses the stack frames from the top to the bottom till it finds a frame on which the offset becomes negative. After normalization, the abstract value stored in the normalized address is copied to the target register r. The taint bit of r is the union of the taint bits of the normalized address and the address register r_a. At the end, the pc is updated to the next instruction. Rule **Write** describes the semantics of memory write. Similar to memory read, it normalizes the address value and then updates the memory value store MS and the memory taint store MT. Rule **Malloc** creates a new abstract value denoting the allocation site with 0 offset. Note that BDA does not model memory safety and hence the size of allocation is irrelevant. Intuitively, one can consider each allocated heap region has infinite size. This can be achieved during abstract interpretation but not during concrete execution. Rule **Input** loads the abstract value of destination register r from the pre-generated external input sample valuation RV, which is constructed by drawing value samples from predefined distributions during the preceding sampling phase. In addition, the taint bit is set `true` to indicate that the value is related to external input. Rule **Goto** sets the program counter to the target address a.

In Rule **If-Goto**, if the taint bit of r is not set and the current instruction is a loop predicate, that is, r is not directly/transitively computed from external input, the loop branch outcome is certain and independent from the sampled value. Hence, pc is set to a_t, which is either the branch target a specified by the statement when r is true, or the fall-through address. Otherwise, it is loaded from the pre-computed path sample PA. Observe that BDA respects path feasibility when loop predicate outcome is not derived from any external input, e.g.,*constant loops* (in the initialization phase). Taint analysis allows us to identify such predicates. In Rule **Call**, pc is first copied to pc^0, then it is updated by loading from the sample path PA. BDA may determine to skip a function call if it is part of a recursion. If the call is not skipped, indicated by pc being equal to the specified target a, the invocation site pc^0, its instance count, the return address (i.e., the instruction after the invocation), and the current abstract value of sp are pushed to the call stack CS. Then, the abstract value of sp is reset, indicating a new stack frame. Rule **Ret** pops the call stack to acquire the return address and restores the value of sp. Rules **Expr1** and **Expr2** update the resulting register r_t with the value calculated by the **CalcValue**() procedure and record the corresponding taint tags. As

Table 2. Abstract interpretation example (PA = a ! e ! g ! k)

SourceCode	Trace	BDA Trace	Actions		
			RS	*MS*	*CS*
1 . int main() {	2	a. r_1 := **malloc**^{10}	»r_1¼ = ħH^1_a; 0i		
2 . char *s = malloc(2);		b. sp := sp ħG; 4i	»sp¼ = ħS^1_a; -4i		
3 . foo(s);		c. **W**^{1}sp; r_1^{o}		»ħS^1_a; -4i¼ = ħH^1_a; 0i	
4 . }	3	d. **call**^{1}e^o	»sp¼ = ħS^1_e; 0i		»ħS^1_a; -4i¼
5 .	7	e. r_3 := **input**^{10}	»r_3¼ = ħG; 502i		
6 . void foo(char *s) {		f. **if** r_3 **then goto**^{1}p^o			
7 . if(input()) return;	8	g. r_2 := **R**^{1}sp^{o}	»r_2¼ = ħH^1_a; 0i		
8 . gee(s);		h. sp := sp ħG; 4i	»sp¼ = ħS^1_e; -4i		
9 . }		i. **W**^{1}sp; r_2^{o}		»ħS^1_e; -4i¼ = ħH^1_a; 0i	
10 .		j. **call**^{1}k^o	»sp¼ = ħS^1_k; 0i		»ħS^1_a; -4i; ħS^1_e; -4i¼
11 . void gee(char *s) {	11	k. r_4 := **R**^{1}sp^{o}	»r_4¼ = ħH^1_a; 0i		
12 . for(int i=0; i<2; i++)	12	l. r_5 := ħG; 0i	»r_5¼ = ħG; 0i		
13 . s[i] = input();		m. r_6 := r_5 ħG; 2i	»r_6¼ = ħG; 0i		
14 . }		n. **if** r_6 **then goto**^{1}x^o			
	...	...	...		

shown in Figure 9, **CalcValue**() computes the result of operation op on operands v_1 and v_2. If one of the operands belongs to the global region, then the resulting memory region is inherited from the other operand and the resulting offset is derived by performing the operation on the offset fields of the two operands. Otherwise (e.g., both operands denote values in some heap region, which may occur as path feasibility may not be respected by BDA), we use a random value as the result, since we could not obtain a precise result for operations on two non-global abstract values. In this case, the result taint tag is set to `true`.

Example. Consider the example in Table 2. The source code, the source level trace, the trace in our language, and the interpretation actions are shown in the columns from left to right. Observe that instructions a c correspond to the invocation at line 2 that writes the returned value from `malloc()` to stack. In d (i.e., invocation to `foo()` in line 3), the current sp value ħS^1_a; -4i is pushed to CS; sp is updated to denote the stack frame of `foo()`; and the target instruction e is loaded from the sample path PA (in the caption of Table 2). In f (i.e., the conditional in line 7), since r_3 is from external input, the target ∂ is loaded from PA. In ∂ (i.e., passing s in line 8), when sp (i.e., the address of local variables) is read, its value ħS^1_e; 0i is first normalized to ħS^1_a; -4i, which is used to access MS to acquire the value of ħH^1_a; 0i. In n (i.e, the loop predicate in line 12), r_6 is not input related, the interpreter evaluates the predicate and takes the true branch.

6 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. Specifically, it computes *abstract states* for each *program point*, which is an instruction annotated with a calling context. The abstract states represent the set of live abstract addresses at the given program point and their definition instructions (an intuitive correspondence at the source level is that the set of live variables at a program point and the statements that define them). Dependences are detected between a read instruction and all the definitions to the address being read. It is context-sensitive as it considers instructions under different contexts as different program points. It is path-insensitive as it merges the abstract values collected along different branches at control flow joint point (e.g., the instruction where the two branches of a conditional statement meet). This allows addressing incompleteness in path sampling. *However, our analysis is much more accurate than a flow-sensitive and path-insensitive data-flow analysis as it does not compute any new abstract values (e.g., by transfer functions in standard*

```
1 char bar(char *p) {
2   *p = 0;
3   if (input()) {
4     *p = 1;
5     foo(*p);
6   }
7   if (input()) return *p;
8   else return ~(*p);
9 }
```

Fig. 10. Posterior analysis example

```
1 typedef struct node {int val; struct node *next} node_t;
2 node_t list_a[10000], list_b[10000];
3 int foo() { // list_a and list_b are independent
4   for (int i = 0, j = 1; i < 10000; i++, j++){
5     list_a[i].next = &(list_a[j % 10000]);
6     list_b[i].next = &(list_b[j % 10000]);}
7   list_a[input()].next->val = 0;
8   return list_b[input()].next->val;
9 }
```

Fig. 11. Taint tracking example (simplified from 181.mcf)

data flow analysis), but rather just aggregates the collected abstract values. This avoids the substantial precision loss caused by the conservativeness of transfer functions. Intuitively, abstract values collected in individual sample paths are propagated through all paths (by the merge operation) to disclose any missing dependences due to incomplete path sampling. To further mitigate the precision loss caused by the merge operation, our analysis also features strong updates [Lhoták and Chung 2011] and strong kills that preclude bogus abstract states.

Detailed Design. The details of the analysis are shown in Algorithm 3. It takes as input the set of memory operation sequences (*MOSes*), each sequence generated by interpreting a path sample, and the inter-procedural control flow graph (*iCFG*) that maps an instruction to its successor(s), and produces the instruction pairs with (memory) dependence relations (*DIP*). The process consists of two stages. In the first stage (lines 2-7), a per-sample analysis (refer Algorithm 1 in [Zhang et al. 2019b] for details) is performed on each memory operation sequence to derive three pieces of information: the set of abstract addresses accessed by each instruction *I2M*, the set of definitions (i.e., writes) each instruction depends on *DEP*, and the set of definitions killed by each write instruction *KILL*. These results are merged to their global correspondences (lines 4-6). In the second stage (lines 8-32), a work list (*WL*) is used to traverse *iCFG* to compute abstract states *PS* for each program point. Lines 11-19 determine the successors of the current program point and maintain the calling context *cs*. If *iaddr* is a memory write (lines 21-22), the set of live addresses and their definitions *M2I* are updated by the procedure HandleMemoryWrite() (Algorithm 4). Specifically, Algorithm 4 checks if *iaddr* defines the same abstract address in all sample paths (line 4 in Algorithm 4). If so, strong update is performed by resetting the definition of *maddr* to *iaddr*; otherwise, *iaddr* is added to the definition set of *maddr* (line 7). If *iaddr* always kills the same definition in all samples (line 8), the definition is removed from the result set (line 9). Return to Algorithm 3. In lines 23-24, if *iaddr* is a memory read, dependences are derived from *M2I*, the abstract state at *iaddr*, through the procedure HandleMemoryRead() (refer Algorithm 2 in [Zhang et al. 2019b] for details) Lines 26-31 proceed to the succeeding program points. Particularly, a control flow successor is added to the work-list if its abstract state has undertaken any change (lines 27-29). Our analysis terminates when a fixed point is reached. At the end, we want to mention that full context-sensitivity is very expensive. Hence BDA supports configurable call depth. In our experiment, we use depth 2.

Example. Consider the example in Figure 10. For simplicity, we use source code to explain our ideas. Assume BDA collects 2 path samples: `2 ! 3 ! 4 ! 5 ! 6 ! 8` and path `2 ! 3 ! 7`. As such, abstract interpretation exposes dependences from line 7 to 4, from line 5 to 4, and from line 8 to 2, but missing that from line 8 to 4 due to incomplete path coverage. By merging the abstract values from the two branches of the predicate in line 3, the posterior analysis discloses the missing dependence. Additionally, as function `bar()` might be invoked several times, pointer `p` at line 4 might have multiple abstract values. As such, at line 4 traditional analysis like VSA cannot kill the

Algorithm 3 Posterior Dependence Analysis

```
INPUT:   MOSes:  {MemOpSeq}                                          . set of memory operation sequences
         ICFG:   Node  Edge                                          . inter-procedural control flow graph
OUTPUT:  DIP:    {Address  Address}                                  . set of dependent instruction pairs
LOCAL:   GI2M:   Address ! {AbstractValue}    . map an instruction to abstract addresses accessed by it
         GDEP:   Address ! {Address}                . map an instruction to its depending instructions
         GKILL:  Address ! {Address}             . map an instruction to reaching definitions killed by it
         M2I:    AbstractValue ! {Address}                . map an abstract address to its definitions
         WL:     »CallString  Address¼                . work list of program points with calling context
         PS:     ¹CallString  Address° ! ¹AbstractValue ! {Address}°                . abstract state
```

```
1:  function PosteriorDependenceAnalysis(MOSes, ICFG)
2:      for MOS in MOSes do
3:          ⟨I2M; DEP; KILL⟩ := PerSampleAnalysis¹MOS°
4:          GI2M := map_merge¹GI2M; I2M°
5:          GDEP := map_merge¹GDEP; DEP°
6:          GKILL := map_merge¹GKILL; KILL°
7:      end for
8:      WL.enqueue¹⟨nil; entry¹ICFG°⟩°
9:      while ¬WL.empty¹°° do
10:         ⟨cs; iaddr⟩ := WL.dequeue¹°
11:         if is_call¹iaddr° then                    . update calling context upon a call instruction
12:             cs.push¹iaddr°
13:             succs := call_target¹ICFG; iaddr°
14:         else
15:             if is_ret¹iaddr° then
16:                 iaddr := cs.pop¹°
17:             end if
18:             succs := get_succ¹ICFG; iaddr°                  . get the following instruction
19:         end if
20:         M2I := PS »cs; iaddr¼                . the set of reaching definitions at iaddr
21:         if is_memory_write¹iaddr° then
22:             M2I := HandleMemoryWrite¹iaddr; M2I; GI2M; GKILL°
23:         else if is_memory_read¹iaddr° then
24:             DIP := HandleMemoryRead¹iaddr; DIP; M2I; GI2M; GDEP°
25:         end if
26:         for succ in succs do
27:             if ¬map_contains¹PS »cs; succ¼; M2I° then
28:                 PS »cs; succ¼ := map_merge¹PS »cs; succ¼; M2I°
29:                 WL.enqueue¹cs; succ° . additional analysis round is needed when changes detected
30:             end if
31:         end for
32:     end while
33:     return DIP
34: end function
```

definition from line 2, whereas BDA can, by its strong kill. This prevents the bogus dependence from line 5 to 2.

7 EVALUATION

BDA is implemented in Rust, leveraging Radare2 [Pancake 2018] that provides basic disassembling functionalities. For input distribution, we used a fixed normal distribution $\mathcal{N}(\mu = 0; \sigma^2 = 32768^2)$, without assuming prior knowledge [1]. To assess BDA's effectiveness and efficiency, we compare it

[1] We have tried different parameters. The impact is not significant.

Algorithm 4 Handle Memory Write

```
INPUT:  iaddr:   Address                                  . the current instruction
        M2I:     AbstractValue ! fAddressg                . map an abstract address to its definitions
        GI2M:    Address ! fAbstractValueg . map an instruction to the abstract addresses accessed by it
        GKILL:   Address ! fAddressg                      . map an instruction to reaching definitions killed by it
OUTPUT: M2I'0:   AbstractValue ! fAddressg                 . a new map between abstract address to definitions
1: function HANDLEMEMORYWRITE(iaddr, M2I, GI2M, GKILL)
2:     M2I'0  M2I
3:     for maddr in GI2M »iaddr¼ do
4:         if capacity ¹GI2M »iaddr¼º  1 then                                  . strong update
5:             M2I'0 »maddr¼  fiaddrg
6:         else
7:             M2I'0 »maddr¼  M2I'0 »maddr¼ [ fiaddrg
8:             if capacity ¹GKILL »iaddr¼º  1 then                            . strong kill
9:                 M2I'0 »maddr¼  M2I'0 »maddr¼ n GKILL »iaddr¼
10:            end if
11:        end if
12:    end for
13:    return M2I'0
14: end function
```

with Alto and VSA (from state-of-the-art binary analysis platforms) on SPECINT2000, a standard benchmark widely used by binary analysis techniques including the aforementioned two. We also apply BDA in two downstream analyses, one is to identify indirect control flow transfer targets, a critical challenge in constructing call graphs, and the other is to identify hidden malicious behaviors of a set of 12 recent malware samples provided by VirtualTotal [VirusTotal 2018]. In the former experiment, we compare BDA with IDA, an industry standard platform. In the latter, we compare with Cuckoo [Cuckoo 2014], a state-of-the-art malware analysis platform. All experiments were conducted on a server equipped with 32-cores CPU (Intel® Xeon™ E5-2690 @ 2.90GHz) and 128G main memory. The SPEC binaries were generated by LLVM with the default compilation option. We strip all the symbol information before using them. Basic information of the SPECINT2000 binaries and the malware samples can be found in our supplementary material [Zhang et al. 2019b].

7.1 Coverage

Code coverage. In this experiment, we study the code coverage of our unbiased whole-program path sampling algorithm and compare it with a naive algorithm that tosses a fair coin at each predicate. Specifically, we collect 10,000 path samples for each algorithm and report the code coverage. The detailed results are presented in Appendix B. Overall, our algorithm can achieve almost 100% coverage for all programs. On average, it covers 554% more instructions, 529% more basic blocks, and 258% more functions than the naive algorithm. For programs with complex path structures, our algorithm has much better coverage. Take 197.parser as an example. It contains lots of error-handling code that detours from the main processing logic at the beginning of the program. The naive algorithm tends to get stuck in these error handling paths. In contrast, our sampling algorithm appropriately prioritizes the main processing logic that contains many more deep paths.

Path coverage. Next, we study the path coverage. Since there are usually an extremely large number of whole-program paths (even not considering loops and recursion), it is not that useful to report whole-program path coverage. Hence, we report intra-procedure path coverage, in which the paths we consider are the BL paths defined in [Ball and Larus 1996b]. Specifically, these are intra-procedural paths starting at function entry or some loop heads and ending at function exit

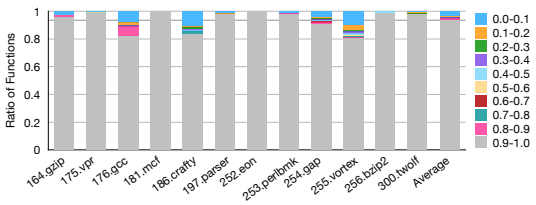


Fig. 12. Path coverage.

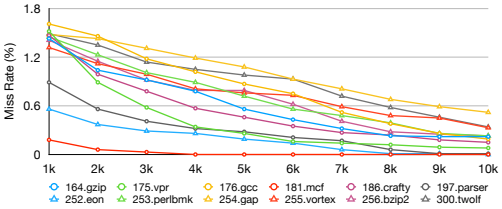


Fig. 13. Effect of sampling.

or a back-edge. The results are shown in Figure 12, which shows the percentage of functions for which BDA has achieved various levels of coverage. As we can see, 93% of the functions have a full or close-to-full path coverage. Those functions whose path coverage is less than 50% have an extremely large number of unique paths (e.g., function `get_method()` in 164.gzip has 4514809836 BL paths). As we will show in the next experiment, according to the observation discussed in Section 2 that a dependence tends to be covered by many paths. Incomplete path coverage does not cause prominent problems for us. In addition, the posterior analysis substantially mitigates the issue as well.

7.2 Program Dependence

In this experiment, we perform dependence analysis on SPECINT2000 programs. We also compare with Alto and VSA. For Alto, we port its original implementation [Muth et al. 1998] on DEC Alpha to x86. There are three popular binary analysis platforms that support VSA, including CodeSurfer [GrammaTech 2008], ANGR [UCSB 2008], and BAP [Brumley et al. 2011]. Among them, CodeSufer is not publicaly available and ANGR's VSA does not handle complex binaries as SPECINT2000 programs (after confirming with the authors). We hence choose BAP's VSA for comparison (called BAP-VSA). Note that it is intractable to acquire the ground truth of program dependencies, even with source code (due to various reasons such as aliasing and loops). Therefore, we use two methods to evaluate the quality of detected dependencies. First, we run the programs with the inputs provided by SPEC and use the observed dependencies as *reference*. Any dependence detected by reference executions but not by the analysis tools is called a *missing* dependence (or a false negative). Any dependence detected by the tools but not observed during reference executions is called an *extra* dependence. Note that the provided inputs achieve 81% code coverage for the SPECINT2000 benchmarks. In addition, we implemented a static type checker to validate if the source and the destination of a (detected) dependence have the same type. The checker is implemented as an LLVM pass, which propagates symbol information to individual instructions, registers and memory locations. As such, we can obtain the type of each binary operation and its operands. Note that such information is much richer than the debugging information generated during compilation. Any dependence whose source and destination have different types is considered a *mis-typed* dependence, which is most likely to be a bogus dependence. For fair comparison, we set a fixed timeout of 12 hours for each program.

Result Summary. Table 3 shows the result summary. Column 2 denotes the number of dependencies observed in the reference execution, columns 3-6 and 7-10 report the number of reported dependencies, the missing ones, the extra ones and the mis-typed ones for Alto and BDA, respectively. Column 11 shows the reduction of the reported dependencies by BDA from Alto (e.g., the reduction of `181.mcf` from Alto is (588076-3347)/3347=17470%). N/A in the table means that the tool times out and hence its analysis result is not available. Note that BAP-VSA only handles `181.mcf`, we list the result separately on the bottom. We have doubled the execution time for other programs but the analysis still cannot terminate. Further inspection shows that when the value set of an

Table 3. Memory Dependence.

Program	# Refer	Alto				BDA				Reduce
		# Found	# Miss	# Extra	# MisTyped	# Found	# Miss	# Extra	# MisTyped	
164.gzip	3,580	2,229,749	0 (0.00%)	2,226,169	302,100 (13.55%)	29,370	8 (0.22%)	25,798	3,502 (11.92%)	7492%
175.vpr	13,042	36,840,012	0 (0.00%)	36,826,970	26,692,177 (72.45%)	559,460	10 (0.08%)	546,428	346,217 (61.88%)	6485%
181.mcf	2,050	588,076	0 (0.00%)	586,026	324,621 (55.20%)	3,347	0 (0.00%)	1,297	433 (12.94%)	17470%
186.crafty	30,777	44,139,556	0 (0.00%)	44,108,779	4,926,267 (11.16%)	1,077,346	45 (0.15%)	1,046,614	78,785 (7.31%)	3997%
197.parser	15,196	32,905,403	0 (0.00%)	32,890,207	29,355,388 (89.21%)	659,867	2 (0.01%)	644,673	535,291 (81.12%)	4887%
252.eon	4,401	994,655	0 (0.00%)	990,264	974,925 (98.02%)	28,855	0 (0.00%)	24,454	22,538 (78.11%)	3347%
253.perlbmk	57,507	102,068,477	0 (0.00%)	100,349,485	94,603,019 (92.69%)	5,389,973	130 (0.23%)	5,363,373	4,461,094 (82.77%)	1794%
254.gap	7,935	10,611,636	0 (0.00%)	10,603,701	9,981,368 (94.06%)	205,200	41 (0.52%)	197,306	152,470 (74.30%)	5071%
255.vortex	29,971	265,981,817	0 (0.00%)	265,951,846	238,479,881 (89.66%)	2,159,444	98 (0.33%)	2,129,473	1,385,953 (64.10%)	12217%
256.bzip2	4,306	2,466,876	0 (0.00%)	2,462,570	708,163 (28.71%)	13,917	10 (0.23%)	9,621	1,509 (10.84%)	17626%
300.twolf	16,710	44,735,257	0 (0.00%)	44,718,440	33,741,198 (75.42%)	2,285,090	56 (0.34%)	2,268,436	1,678,383 (73.45%)	1858%
Avg.	**16,861**	**49,414,683**	**0 (0.00%)**	**49,246,769**	**40,008,101 (65.47%)**	**1,128,352**	**36 (0.19%)**	**1,114,316**	**787,834 (50.80%)**	**7477%**
176.gcc	435,692	N/A	N/A	N/A	N/A	692M	498 (0.11%)	692M	79.43%	N/A
BAP-VSA[2] on 181.mcf	**# Found: 23,068**		**# Miss: 0 (0.00%)**		**# Extra: 21,018**		**# MisTyped: 12,533 (54.33%)**		**Reduce: 589%**	

address operand is substantially inflated, which happens a lot in practice, each write through the address operand incurs very expensive updates for a very large number of abstract locations.

Observe that although Alto does not have any missing dependence, the number of reported dependence is very large (due to its conservativeness) and 65.47% of which are mis-typed. Such substantial bogus dependences hinder its use in practice. In comparison, the dependences reported by BDA are 75 times smaller, at the prices of a negligible missing rate (0.19%). Note that although in some cases the mis-typed rate of BDA is only slightly lower than that of Alto (e.g., 197.parser), the absolute number of mis-typed dependences is much smaller. We argue the results by our tool are more useful in practice. We should note that the analysis of 176.gcc is very expensive due to its complexity. As such, Alto could not finish in time. Compared to VSA, BDA reports 589% fewer dependences with a much smaller number of mistyped dependences (433 versus 12533).

We also study the reasons of missing dependences and mis-typed dependences. We find that missing dependences are mainly due to loop paths difficult to cover statically. Consider the code snippet from 164.gzip in Figure 14a, BDA misses the dependence from line 6 to line 4 regarding the suffix of `dst` copied from `msg`. The reason is that BDA only iterates loop (lines 5-6) for a small number of times, which allows it to detect the dependence from line 6 to line 3 regarding the prefix of `dst`, but not the suffix. Mis-type dependences are mainly due to the fact that BDA does not model path feasibility when predicates are dependent on external inputs. As such, bogus dependences are introduced along infeasible paths. We want to point out that the same limitation applies to all analyses that do not fully model path feasibility (e.g., data-flow analysis). Consider the code snippet in Figure 14b from 164.gzip. Along some infeasible paths, pointers `err_cnt` and `err_msg` are not allocated and hence have the NULL value, which leads to bogus dependence from line 6 to line 3.

[2]BAP-VSA took 10.9 hours and 8.3GB memory for 181.mcf. It timed out for the rest (exceeding 12 hours).

Table 4. Effect of posterior analysis and taint tracking.

Program	original BDA			w/o analysis			w/o taint-tracking		
	All(K)	Miss	MisTyped	All(K)	Miss	MisTyped	All(K)	Miss	MisTyped
164.gzip	29	0.22%	11.92%	24	4.53%	2.28%	31	0.37%	23.49%
175.vpr	559	0.08%	61.88%	79	5.44%	43.86%	583	0.11%	67.13%
176.gcc	692(M)	0.11%	79.43%	14(M)	7.26%	35.52%	723(M)	0.10%	84.86%
181.mcf	3	0.00%	12.94%	2	1.17%	10.22%	4	0.10%	28.17%
186.crafty	1,077	0.15%	7.31%	124	1.88%	1.13%	1,114	0.14%	13.42%
197.parser	659	0.01%	81.12%	98	8.94%	62.96%	670	0.01%	84.34%
252.eon	28	0.00%	78.11%	10	1.52%	58.58%	28	0.00%	79.95%
253.perlbmk	5,389	0.23%	82.77%	636	5.35%	67.98%	5,524	0.25%	89.36%
254.gap	205	0.52%	74.30%	70	2.08%	36.73%	217	0.49%	81.17%
255.vortex	2,159	0.33%	64.10%	356	4.73%	58.36%	2,227	0.33%	67.30%
256.bzip2	13	0.23%	10.84%	10	2.90%	6.19%	15	0.46%	23.53%
300.twolf	2,252	0.34%	73.45%	294	6.21%	67.01%	2,375	0.35%	79.21%
Avg.	**58,697**	**0.18%**	**53.18%**	**1,308**	**4.57%**	**37.56%**	**61,324**	**0.22%**	**60.16%**

Table 5. Runtime overhead.

Program	Time[3] (h)				Memory
	Total	PP	AI	PA	(GB)
164.gzip	1.59	0.15	1.13	0.31	3.8
175.vpr	6.80	0.54	2.75	3.51	21.4
176.gcc	10.06	1.63	7.54	0.89	103.3
181.mcf	0.83	0.06	0.71	0.06	1.6
186.crafty	7.39	0.36	2.47	4.56	15.6
197.parser	5.62	0.29	2.17	3.16	12.5
252.eon	5.98	0.84	3.51	1.63	5.7
253.perlbmk	11.35	0.68	4.24	6.43	73.5
254.gap	5.67	0.21	2.61	2.85	4.0
255.vortex	11.75	0.63	4.13	6.99	58.1
256.bzip2	2.32	0.18	1.27	0.87	4.2
301.twolf	11.68	0.57	3.99	7.12	47.9
Avg.	**6.75**	**0.51**	**3.03**	**3.21**	**29.3**

Necessity of Posterior Analysis and Taint Tracking. We study the necessity of posterior analysis and taint tracking. Table 4 shows the effect of the posterior analysis by comparing the number of missing dependences when using the posterior analysis and when simply aggregating the dependences collected in individual samples (0.18% versus 4.57%). We also report the total dependences. Observe that the posterior analysis produces much more dependences in total. Due to the lack of ground-truth, it is difficult to infer how many are true dependences. However, the comparison of mis-typed dependences (53.18% versus 37.56%) demonstrates the posterior analysis substantially reduces the false negative rate while only incurring a relatively modest growth of false positives (compared to the explosion incurred in Alto and VSA). Table 4 shows the effects of taint tracking as well. Observe that the comparisons of missing and mis-typed dependences (0.18% versus 0.22% and 53.18% versus 60.16%, respectively) indicate the necessity of taint tracking. The root cause of additional bogus dependences is that without taint tracking, constant loops are not properly interpreted (i.e., only the first a few unrolled iterations are interpreted). Consider a simplified code snippet from 181.mcf in Figure 11. The for-loop at line 4 is a constant loop, in which two independent node lists `list_a` and `list_b` are initialized. Without taint tracking, BDA cannot recognize it as a constant loop and hence only interprets the first a few iterations. As a result, the `next` field of the remaining `list_a` and `list_b` entries are not initialized and all have a null value. Consequently, BDA considers there is a dependence between lines 7 and 8, which is false. Since 181.mcf has many such lists and many constant initialization loops, bogus dependences are introduced between a large number of accesses through uninitialized pointers.

Effect of Sampling We also study the effects of having different number of samples. Figure 13 shows the effect of sampling. Observe that the missing rate decreases as the number of samples increases. When the number of samples reaches 10k, the missing rate is reduced to less than 0.5% for all programs and 0 for some (e.g., mcf, eon and parser). Note that the experimental results are consistent with our probabilistic analysis in Section 4.2.

Analysis Overhead. Table 5 presents the time and memory consumption of BDA in analyzing each SPEC2000 program. On average, BDA takes 6.75 hours to analyze a program, with 7%, 45% and 48% spending on the pre-processing, abstract interpretation and posterior analysis, respectively. The sampling stage takes relatively small amount of time even with the cost of dealing with large weight values. The time consumption for abstract interpretation is the sum of individual samples. The memory consumption of BDA ranges from 1.6GB to 103.3GB (29.3GB on average), depending on the complexity of the target programs. As a comparison, Alto has similar memory consumption

[3]PP for pre-processing, AI for abstract interpretation and PA for posterior analysis.

```
1 char* encode_msg(char *msg, int n) {
2   char *dst = malloc((n + PREFIX_LEN);
3   strcpy(dst, PREFIX_STR);
4   strcat(dst, msg);
5   for (int i = 1; i < n + PREFIX_LEN; i++)
6     dst[i] = dst[i] ^ dst[i-1];
7   return dst;
8 }
```

(a) missing dependence

```
1 void error(char* err_msg, int *err_cnt) {
2   if (!err_cnt)
3     *err_cnt += 1; // Written type: int
4     // Potentially write to address NULL
5   if (!err_msg)
6     puts(err_msg); // Read type: char
7     // Potentially read from address NULL
8 }
```

(b) mis-typed dependence

Fig. 14. Code snippet on missing and mistyped dependence.

as BDA (21.9GB vs. 22.6GB) and is 27.7% slower (8.3h vs. 6.5h) on SPECINT2000 excluding 176.gcc. We argue that since dependence graph generation is a one-time effort, the entailed overhead is reasonable.

7.3 Applications

We evaluate BDA in two downstream analysis, one is to identify indirect control flow transfer targets (conducted on the SPEC programs), the other is to disclose hidden malware behaviors (conducted on 12 recent malware samples).

Inferring Indirect Control Transfer Targets. With program dependences, we can infer the potential targets of an indirect jump/call instruction by backward slicing from its target register. Table 7 in Appendix C shows the results. For comparison, we also present the analysis result of IDA and the indirect targets observed when running with inputs provided in SPEC. Observe that BDA performs as good as IDA in inferring indirect jump targets and substantially outperforms IDA in inferring indirect call targets (4 found by IDA on average versus 767 found by BDA). We should note that indirect jump targets are easier to infer than indirect call targets, as indirect jumps are always intra-procedural and have fixed patterns when they are generated by mainstream compilers. IDA leverages such patterns whereas we leverage dependences. None of the observed call targets is missed by BDA. In addition, the set of indirect call targets reported by BDA is comparable to those reported in [Peng et al. 2014], a very expensive *concrete* execution engine that forcefully executes along a large number of paths. The results demonstrate the practical use of BDA.

Exposing Malware Behaviors. Program dependence can be used to study malware behaviors. In the literature of malware analysis [Cozzi et al. 2018], the behavior of a malware sample is largely defined by the system calls performed by the sample, together with parameter values. With dependencies, we can perform (static) constant propagation through dependence edges to identify the parameter values of critical library functions. We also compare BDA with Cuckoo, a state-of-the-art malware analysis tool. Cuckoo reports behavior on the system call level, while BDA reports on the library call level. To be comparable, we map library calls to system calls. Table 8 in Appendix C shows the results. Observe that BDA reports 3 times more hidden malicious behaviors.

Case study. We take the malware sample *a664df72a34b863fc0a6e04c96866d4c* as a case to study how our dependence analysis can help detect hidden malicious behaviors. It is a bot malware that waits for commands from a remote server. Figure 15 shows the simplified code of its initialization logic. In particular, it tries to connect to a remote server every 5 seconds until success (line 2), then executes the binary files stored in some pre-defined directories (e.g., `/dev/netslink`) to setup the running environment (lines 9-14). The behavior of running the binary files will not be triggered by the sandbox execution in Cuckoo, since the remote server is down. Hence, Cuckoo fails to detect such behavior. In contrast, we perform static constant propagation through dependences to extract critical library calls with *concrete* parameters. Figure 16 presents the static slice of `system()` call

```
1  int main() {
2    while (!cnc_server_connected()) sleep(5);
3    initialization();
4    ......
5  }
6  void initialization() {
7    char *dirs[10], cmd[256];
8    memcpy(dirs, rodata_41176, 0x50);
9    for (int i = 0; i < 10; i++) {
10     sprintf(cmd, "cd %s && "
11       "for a in `ls -a %s`; do >$a; done;",
12       rodata_4112A0[i], rodata_4112A0[i]);
13     system(cmd);
14   }
15 }
```

Fig. 15. simplified code

```
a lea rdi, [rbp + local_60]
b mov esi, rodata_411760 ; list of bin dirs:
  "/dev/netslink/", "/var/", ..., "/usr/"
c mov edx, 0x50
d call memcpy
e ...
f lea rdi, [rbp + local_1F0]
g lea rsi, rodata_4112A0 ; format string:
  "cd %s && for a in `ls -a %s`; do >$a; done;"
h mov rdx, [rbp + rax * 8 + local_60]
i mov rcx, rdx
j call sprintf
k ...
l lea rdi, [rbp + local_1F0]
m call system
```

Fig. 16. slicing with the dependence information

Fig. 17. Malware case study.

(line 13), whose parameter depends on the invocation of the `sprintf` library function, which fills the format string buffer indicated by `rbp + local_1F0`. It further depends on the format string stored in the global memory `rodata_4112A0`, which has the value `"cd %s && for a in ‘ls -a %s‘; do >$a; done;"`. Hence, we detect the behavior of running the binary files under pre-defined directories without executing the malware.

8 FUTURE WORK

Widening and Using Concrete Values. BDA currently does not use widening, which approximates abstract values to reduce search space. The reason is that in the binary analysis context, widening may cause substantial precision loss, as indicated by the results of classic VSA. However, limited and selective widening may be feasible with our sampling technique. For instance, we will explore per-path widening in our future work, which may potentially provide a good trade-off between cost and precision. In addition, per-path interpretation opens the door of using concrete values instead of abstract values. In fact, some of BDA's abstract values closely resemble concrete values (e.g., a stack address is a function entry with concrete offset). We will explore leveraging concrete values to preclude taint tracking that approximates runtime property related to loops.

Other Applications of Path Sampling. Our path sampling algorithm is general. We plan to use it in symbolic execution and fuzzing, whose path exploration strategy is mainly edge or statement coverage driven. We will also develop randomized techniques based on path sampling for other analysis with limited path sensitivity such as type inference and shape analysis.

9 RELATED WORK

Binary Analysis. Our work is related to binary analysis, including static [Lee et al. 2011; Sutter et al. 2000; Theiling 2000] and dynamic analysis [Kolbitsch et al. 2010; Lin et al. 2010; Slowinska et al. 2011]. Alto [Debray et al. 1998] and VSA [Balakrishnan and Reps 2004] aim to provide a sound solution to identifying aliases among memory accesses. Compared to these two, BDA is sampling based and per-sample abstract interpretation based, and hence features better precision (with probabilistic guarantee under assumption) and scalability, as shown by our results. Force-execution [Peng et al. 2014] concretely executes a binary along different paths, by force-setting branch outcomes. It features an expensive execution engine that recovers from exceptions caused by violations of path feasibility. Due to its cost, force-execution has difficulty covering long paths. To address the above limitation, You et al. [2020] propose a light-weight force-execution technique with

probabilistic memory pre-planning. However, their context-insensitive path exploration strategy only focuses on predicates, leading to accuracy loss in dependence analysis. Recently, machine learning is extensively used in binary analysis, e.g., identifying function boundary [Shin et al. 2015], pinpointing function type signature [Chua et al. 2017], and detecting similar binary code [Ding et al. 2019; Xu et al. 2017]. In particular, Guo et al. [2019] use LSTM to distinguish the different types of memory regions in VSA analysis. However, it does not change the core of VSA.

Random Interpretation. BDA is also related to random interpretation, a well-known probabilistic program analysis technique used in precise inter-procedural analysis [Gulwani and Necula 2005], global value numbering [Gulwani and Necula 2004] and discovering affine equalities [Gulwani and Necula 2003]. It features a randomized abstract interpretation that executes both branches of a conditional predicate on each run and performs a randomized affine combination at join points. However, such an affine combination is limited for numerical operations and hard to scale to binary program dependence analysis. Compared with these works, our per-path interpretation is more like concrete execution with higher accuracy and scalability.

Other Program Analysis. Our technique is related to program dependence analysis [Bell et al. 2015; Bergeretti and Carré 1985; Clause et al. 2007; Ferrante et al. 1987; Myers 1999; Newsome and Song 2005; Olmos and Visser 2005; Palepu et al. 2013; Sabelfeld and Myers 2003; Zhu et al. 2015]. These techniques require source code. In addition, our technique only focuses on data dependence whereas many existing works also consider control dependence. BDA is also related to points-to analysis [Deutsch 1994; Emami et al. 1994; Hirzel et al. 2007; Kahlon 2008; Liang and Harrold 1999; Steensgaard 1996; Thiessen and Lhoták 2017; Xu and Rountev 2008; Zheng and Rugina 2008] that addresses a similar problem. The difference lies in that our analysis does not require symbol information and hence is more difficult. Some techniques aim to reduce the runtime complexity of path-sensitive analyses [Das et al. 2002; Dillig et al. 2008]. In contrast, our technique is sampling based. We believe BDA is complementary to existing work.

Probabilistic Program Analysis. Probabilistic techniques have been increasingly used in program analysis in recent years. Probabilistic symbolic execution [Borges et al. 2015; Geldenhuys et al. 2012] quantifies how likely it is to reach certain program points. Probabilistic model checking [Donaldson et al. 2009; Filieri et al. 2011; Kwiatkowska et al. 2011] encodes the probability of making a transition between states and entails computation of the likelihood that a target system satisfies a given property. Probabilistic disassembling [Miller et al. 2019] computes a probability for each address in the code space, which indicates the likelihood of the address representing a true positive instruction. Probabilistic type inference uses probabilistic graph models to infer data type [Xu et al. 2016]. There are also works on using MCMC type of sampling to derive analysis information such as memory access pattern for race detection [Cai et al. 2016] and leak detection [Hauswirth and Chilimbi 2004], and runtime events for program understanding [Toronto et al. 2015; Zhong and Chang 2008]. Most of them are concrete execution based. In comparison, BDA features a novel unbiased path sampling algorithm and leverages abstract interpretation.

10 CONCLUSION

We propose a practical program dependence analysis for binary executables. It features a novel unbiased whole-program path sampling algorithm and per-path abstract interpretation. Under certain assumptions, our technique has probabilistic guarantee in disclosing a dependence relation. Our experiments show that our technique has substantially improved the state-of-the-art, such as value set analysis. It also improves performance of downstream applications in indirect call target identification and malware behavior analysis.

A DETAILS OF VSA

Table 6 illustrates how VSA works on the `read_words()` function. Specifically, the strided interval for register `r12` at instruction `b` is `0x0»0x8,0x8¼`, denoting a constant 8, and the strided interval for register `r14` at instruction `d` is `0x40»0x0; 0xfa00¼`. Intuitively, `r12` corresponds to the `dict->words` variable passed from the `init_dict` function, and `0x8` is the offset of the `words` field in the `Dict` structure. The strided interval of `r14` represents all the possible loop count `i` values at the binary level. Note that the stride is 40, which is the size of `Word`. These two strided intervals are propagated to instruction `g`, where we have the strided interval for `r12+r14` (corresponding to `&(words[i])` at the source level) as `0x40»0x8,0xfa08¼` according to the addition rule.

The computation of strided intervals is conservative, which may lead to substantial bogus values in PRV. For example, consider the strided interval for `r12+r14+r15` at instruction `m`. The strided interval for `r12+r14` is `0x40»0x8,0xfa08¼` as mentioned earlier. The strided interval for `r15`, which corresponds to the counter `j` of the inner loop, is `0x1»0x0,0x38¼`. According to the addition rule, the resulted strided interval is `0x1»0x8,0xfa40¼`. As we can see that the resulted strided interval is an over-approximation, covering all possible addresses in the memory region of `dict->words`, while only the addresses corresponding to `dict->words[i].vals[j]` should be included. As such, when instruction `m` writes a value read from input, which is denoted as `>` due to the lack of input pre-condition, VSA essentially updates the abstract value for all addresses in `dict->words` to `>`. Specifically, `words[i]->node` holds a `>` value such that when the later instruction `r` writes a value to `words[i].node->word`, which writes to a field of the memory region denoted by `words[i].node`, VSA conservatively writes the value to the entire address space. As a result, any following memory read would have (bogus) dependence with `r`. Moreover, since VSA needs to update the strided interval for all possible addresses, which could be 2^{64} for the 64-bit system, the analysis becomes extremely time-consuming. According to our experience, such phenomenon happens quite often in practice, substantially hindering the applicability of VSA. In Section 7.2, our evaluation shows that the state-of-the-art public VSA implementations fail on many SPEC2000 programs.

Table 6. How VSA works on the `read_words` function.

SourceCode	AsmCode	Variable	VSA
`void read_words(Word* words, long *idx) {`	`a. sub rsp, 0x30`	`rsp ;Stack Pointer`	S_f : `0x0»-0x30,-0x30¼`
	`b. mov r12, rdi`	`r12 ;words`	H_α : `0x0»0x8,0x8¼`
`for (int i = 0; i < WORDS_CNT; i++) {`	`c. xor r14, r14`	`r14 ;i*sizeof(Word)`	G : `0x40»0x0,0xfa00¼`
	`d. cmp r14, 0xfa00`		
	`e. jge u.`	N/A	N/A
`words[i].node = trie;`	`f. mov r13, [0x601110]`	`r13 ;trie`	H_β : `0x0»0x0,0x0¼`
	`g. lea rbx, [r12+r14]`	`r12+r14 ;&(words[i])`	H_α : `0x40»0x8,0xfa08¼`
	`h. mov [rbx+0x38], r13`	`rbx+0x38 ;&(words[i].node)`	H_α : `0x40»0x40,0xfa40¼`
`for (int j = 0; j < MAX_LEN; j++) {`	`i. xor r15, r15`	`r15 ;j`	G : `0x1»0x0,0x38¼`
	`j. cmp r15, 0x38`		
	`k. jge p.`	N/A	N/A
`words[i].val[j] = read_char();`	`l. call read_char`	N/A	N/A
	`m. mov [r12+r14+r15], rax`	`r12+r14+r15 ;&(words[i].val[j])`	H_α : `0x1»0x8,0xfa40¼`
		`rax ;read_char()`	`>`
`} // Inner Loop`	`n. inc r15`	`r15 ;j`	G : `0x1»0x0,0x38¼`
	`o. jmp j.`	N/A	N/A
`words[i].node->word = &(words[i]);`	`p. mov rax, [r12+r14+0x38]`	`r12+r14+0x38 ;&(words[i].node)`	H_α : `0x40»0x40,0xfa40¼`
		`rax ;words[i].node`	`>`
	`q. lea rbx, [r12+r14]`	`rbx ;&(words[i])`	H_α : `0x40»0x8,0xfa08¼`
	`r. mov [rax], rbx`	`rax ;words[i].node->word`	`>`
`} // Outer Loop`	`s. add r14, 0x40`	`r14 ;i*sizeof(Word)`	G : `0x40»0x0,0xfa00¼`
	`t. jmp d.`	N/A	N/A

B COVERAGE

Figures 18a, 18b and 18c present the code coverage of our algorithm (in dark gray bars) and the naive algorithm (in the light gray bars) at the instruction level, basic block level and the function level, respectively.

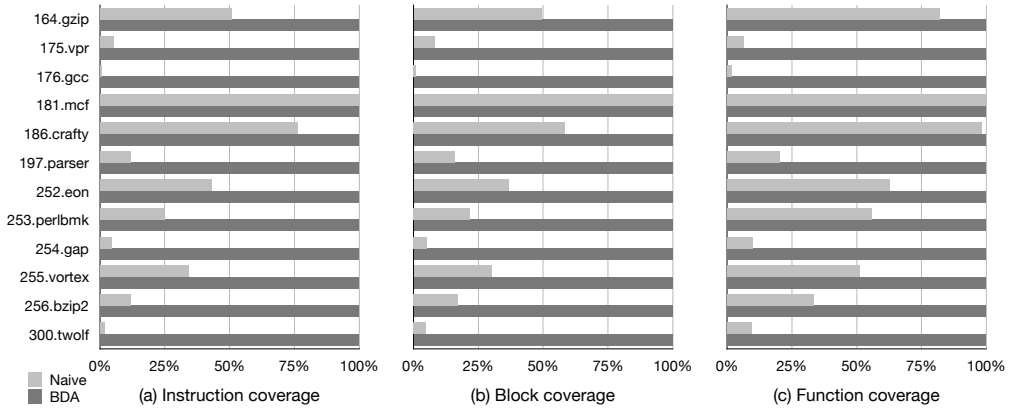


Fig. 18. Code coverage.

C RESULTS OF DOWNSTREAM ANALYSIS

Table 7 presents the result of inferring indirect control transfer targets. Table 8 presents the result of malware behavior detection.

Table 7. Inferring indirect jump/call targets.

Program	# Indirect Jump Edges			# Indirect Call Edges		
	IDA	Dynamic	BDA	IDA	Dynamic	BDA
164.gzip	0	0	0	0	3	3
175.vpr	49	0	49	0	1	1
176.gcc	3,628	324	3,628	25	214	853
181.mcf	0	0	0	0	1	1
186.crafty	159	38	159	0	1	1
197.parser	0	0	0	0	1	1
252.eon	17	0	17	0	183	215
253.perlbmk	1,454	229	1,454	24	243	261
254.gap	63	5	63	2	1,438	7,836
255.vortex	247	56	247	0	24	27
256.bzip2	0	0	0	0	1	1
301.twolf	17	0	17	0	1	1
Avg.	470	54	470	4	176	767

Table 8. Malware behavior analysis.

Malware	# Library Calls	
	Cuckoo	BDA
1a0b96488c4be390ce2072735ffb0e49	50	164
3fb857173602653861b4d0547a49b395	20	112
49c178976c50cf77db3f6234efce5eeb	23	48
5e890cb3f6cba8168d078fdede090996	28	138
6dc1f557eac7093ee9e5807385dbcb05	20	75
72afccb455faa4bc1e5f16ee67c6f915	6	81
74124dae8fdbb903bece57d5be31246b	36	203
912bca5947944fdcd09e9620d7aa8c4a	20	68
a664df72a34b863fc0a6e04c96866d4c	23	99
c38d08b904d5e1c7c798e840f1d8f1ee	34	151
c63cef04d931d8171d0c40b7521855e9	20	81
dc4db38f6d3c1e751dcf06bea072ba9c	20	77
Avg.	25	108

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their constructive comments. Also, the authors would like to express their thanks for Le Yu and Yapeng Ye for proofreading and Yu Shi for her help in illustration. Purdue authors were supported in part by DARPA FA8650-15-C-7562, NSF 1748764, 1901242 and 1910300, ONR N000141410468 and N000141712947, and Sandia National Lab under award 1701331. UVA authors were supported in part by NSF 1850392. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of the sponsors.

REFERENCES

ATA. 2018. SPEC2000. http://www.spec2000.com/.

Gogul Balakrishnan and Thomas Reps. 2004. Analyzing memory accesses in x86 executables. In *International conference on compiler construction*. Springer, 5–23.

Thomas Ball and James R Larus. 1996a. Efficient path profiling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 46–57.

Thomas Ball and James R. Larus. 1996b. Efficient Path Profiling. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 29, Paris, France, December 2-4, 1996*. 46–57.

Jonathan Bell, Gail E. Kaiser, Eric Melski, and Mohan Dattatreya. 2015. Efficient dependency detection for safe Java test acceleration. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. 770–781.

Jean-Francois Bergeretti and Bernard Carré. 1985. Information-Flow and Data-Flow Analysis of while-Programs. *ACM Trans. Program. Lang. Syst.* 7, 1 (1985), 37–61.

Mateus Borges, Antonio Filieri, Marcelo d'Amorim, and Corina S. Pasareanu. 2015. Iterative distribution-aware sampling for probabilistic symbolic execution. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. 866–877.

David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. 463–469.

Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Xiaodong Song. 2007. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*. 317–329.

Yan Cai, Jian Zhang, Lingwei Cao, and Jian Liu. 2016. A deployable sampling strategy for data race detection. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. 810–821.

Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. 2017. Neural Nets Can Learn Function Type Signatures From Binaries. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. 99–116.

James A. Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007, London, UK, July 9-12, 2007*. 196–206.

Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. 2018. Understanding Linux Malware. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. 161–175.

Cuckoo. 2014. Cuckoo Sandbox. https://cuckoosandbox.org/.

Manuvir Das, Sorin Lerner, and Mark Seigle. 2002. ESP: Path-Sensitive Program Verification in Polynomial Time. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*. 57–68.

Saumya K. Debray, Robert Muth, and Matthew Weippert. 1998. Alias Analysis of Executable Code. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*. 12–24.

Alain Deutsch. 1994. Interprocedural May-Alias Analysis for Pointers: Beyond k-limiting. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994*. 230–241.

Isil Dillig, Thomas Dillig, and Alex Aiken. 2008. Sound, complete and scalable path-sensitive analysis. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. 270–280.

Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. 2019. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In *40th IEEE Symposium on Security and Privacy, S&P 2019*.

Alastair F. Donaldson, Alice Miller, and David Parker. 2009. Language-Level Symmetry Reduction for Probabilistic Model Checking. In *QEST 2009, Sixth International Conference on the Quantitative Evaluation of Systems, Budapest, Hungary, 13-16 September 2009*. 289–298.

Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. 1994. Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994*. 242–256.

Kostas Ferles, Valentin Wüstholz, Maria Christakis, and Isil Dillig. 2017. Failure-directed program trimming. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8,*

2017. 174–185.

Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (1987), 319–349.

Antonio Filieri, Carlo Ghezzi, and Giordano Tamburrelli. 2011. Run-time efficient probabilistic model checking. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. 341–350.

Keith Brian Gallagher and James R. Lyle. 1991. Using Program Slicing in Software Maintenance. *IEEE Trans. Software Eng.* 17, 8 (1991), 751–761.

Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. 2012. Probabilistic symbolic execution. In *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*. 166–176.

GrammaTech. 2008. CodeSurfer. https://www.grammatech.com/products/codesurfer.

Sumit Gulwani and George C. Necula. 2003. Discovering affine equalities using random interpretation. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*. 74–84.

Sumit Gulwani and George C. Necula. 2004. Global value numbering using random interpretation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*. 342–352.

Sumit Gulwani and George C. Necula. 2005. Precise interprocedural analysis using random interpretation. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*. 324–337.

Wenbo Guo, Dongliang Mu, Min Du, Xinyu Xing, and Dawn Song. 2019. DEEPVSA: Facilitating Value-set Analysis with Deep Learning for Postmortem Program Analysis. In *28th USENIX Security Symposium, USENIX Security 2019*.

Matthias Hauswirth and Trishul M. Chilimbi. 2004. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2004, Boston, MA, USA, October 7-13, 2004*. 156–164.

Hex-Rays. 2008. IDA. https://www.hex-rays.com/products/ida.

Martin Hirzel, Daniel von, Dincklage, Amer Diwan, and Michael Hind. 2007. Fast online pointer analysis. *ACM Trans. Program. Lang. Syst.* 29, 2 (2007), 11.

Vineet Kahlon. 2008. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. 249–259.

Clemens Kolbitsch, Thorsten Holz, Christopher Kruegel, and Engin Kirda. 2010. Inspector Gadget: Automated Extraction of Proprietary Gadgets from Malware Binaries. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. 29–44.

Marta Z. Kwiatkowska, Gethin Norman, and David Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. 585–591.

JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled Reverse Engineering of Types in Binary Programs. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*.

Ondrej Lhoták and Kwok-Chiang Andrew Chung. 2011. Points-to analysis with efficient strong updates. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. 3–16.

Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. 627–637.

Donglin Liang and Mary Jean Harrold. 1999. Efficient Points-to Analysis for Whole-Program Analysis. In *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, Proceedings*. 199–215.

Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, and Xiangyu Zhang. 2008. Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*.

Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Automatic Reverse Engineering of Data Structures from Binary Execution. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010*.

Joseph P. Loyall and Susan A. Mathisen. 1993. Using Dependence Analysis to Support the Software Maintenance Process. In *Proceedings of the Conference on Software Maintenance, ICSM 1993, Montréal, Quebec, Canada, September 1993*. 282–291.

Jérémie Lumbroso. 2013. Optimal discrete uniform generation from coin flips, and applications. *arXiv preprint arXiv:1304.1916* (2013).

Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. 2019. Probabilistic Disassembly. In *Proceedings of the 41st ACM/IEEE International Conference on Software Engineering (ICSE 2019)*.

Robert Muth, Saumya Debray, Scott Watterson, Koen De Bosschere, and Vakgroep Elektronica En Informatiesystemen. 1998. alto: A link-time optimizer for the DEC Alpha. (1998).

Andrew C. Myers. 1999. JFlow: Practical Mostly-Static Information Flow Control. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*. 228–241.

James Newsome and Dawn Xiaodong Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA*.

Karina Olmos and Eelco Visser. 2005. Composing Source-to-Source Data-Flow Transformations with Rewriting Strategies and Dependent Dynamic Rewrite Rules. In *Compiler Construction, 14th International Conference, CC 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*. 204–220.

Vijay Krishna Palepu, Guoqing (Harry) Xu, and James A. Jones. 2013. Improving efficiency of dynamic analysis with dynamic dependence summaries. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*. 59–69.

Pancake. 2018. Radare2. https://rada.re/r/.

Mathias Payer, Antonio Barresi, and Thomas R. Gross. 2015. Fine-Grained Control-Flow Integrity Through Binary Hardening. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings*. 144–164.

Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. 2014. X-Force: Force-Executing Binary Programs for Security Applications. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. 829–844.

Anh Quach, Aravind Prakash, and Lok-Kwong Yan. 2018. Debloating Software through Piece-Wise Compilation and Loading. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. 869–886.

Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*.

Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 21, 1 (2003), 5–19.

Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel J. Quinlan, and Zhendong Su. 2009. Detecting code clones in binary executables. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*. 117–128.

Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing Functions in Binaries with Neural Networks. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*. 611–626.

Asia Slowinska, Traian Stancescu, and Herbert Bos. 2011. Howard: A Dynamic Excavator for Reverse Engineering Data Structures. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*.

Dawn Xiaodong Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Information Systems Security, 4th International Conference, ICISS 2008, Hyderabad, India, December 16-20, 2008. Proceedings*. 1–25.

Bjarne Steensgaard. 1996. Points-to Analysis in Almost Linear Time. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*. 32–41.

Bjorn De Sutter, Bruno De Bus, Koenraad De Bosschere, P. Keyngnaert, and Bart Demoen. 2000. On the Static Analysis of Indirect Control Transfers in Binaries. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2000, June 24-29, 2000, Las Vegas, Nevada, USA*.

Henrik Theiling. 2000. Extracting safe and precise control flow from binaries. In *7th International Workshop on Real-Time Computing and Applications Symposium (RTCSA 2000), 12-14 December 2000, Cheju Island, South Korea*. 23–30.

Rei Thiessen and Ondrej Lhoták. 2017. Context transformations for pointer analysis. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 263–277.

Neil Toronto, Jay McCarthy, and David Van Horn. 2015. Running Probabilistic Programs Backwards. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint*

Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings. 53–79.

UCSB. 2008. ANGR. https://angr.io/.

VirusTotal. 2018. VirusTotal. https://www.virustotal.com/.

Shuai Wang, Wenhao Wang, Qinkun Bao, Pei Wang, XiaoFeng Wang, and Dinghao Wu. 2017. Binary Code Retrofitting and Hardening Using SGX. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation, FEAST@CCS 2017, Dallas, TX, USA, November 3, 2017*. 43–49.

Guoqing (Harry) Xu and Atanas Rountev. 2008. Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*. 225–236.

Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. 363–376.

Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. 2016. Python probabilistic type inference with natural language support. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. 607–618.

Jun Yang and Rajiv Gupta. 2002. Frequent value locality and its applications. *ACM Transactions on Embedded Computing Systems (TECS)* 1, 1 (2002), 79–105.

Heng Yin, Dawn Xiaodong Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*. 116–127.

Wei You, Zhuo Zhang, Yonghwi Kwon, Yousra Aafer, Fei Peng, Yu Shi, Carson Harmon, and Xiangyu Zhang. 2020. PMP: Cost-effective Forced Execution with Probabilistic Memory Pre-planning. In *2020 IEEE Symposium on Security and Privacy, SP 2020, Proceedings, 18-20 May 2020, San Francisco, California, USA*. IEEE Computer Society.

Junyuan Zeng, Yangchun Fu, Kenneth A. Miller, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2013. Obfuscation resilient binary code reuse through trace-oriented programming. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*. 487–498.

Zhuo Zhang, Wei You, Guanhong Tao, Guannan Wei, Yonghwi Kwon, and Xiangyu Zhang. 2019a. BDA. https://github.com/bda-tool/bda.

Zhuo Zhang, Wei You, Guanhong Tao, Guannan Wei, Yonghwi Kwon, and Xiangyu Zhang. 2019b. BDA Supplementary Material. https://github.com/bda-tool/bda/blob/master/Supplementary_Material.pdf.

Xin Zheng and Radu Rugina. 2008. Demand-driven alias analysis for C. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. 197–208.

Yutao Zhong and Wentao Chang. 2008. Sampling-based program locality approximation. In *Proceedings of the 7th International Symposium on Memory Management, ISMM 2008, Tucson, AZ, USA, June 7-8, 2008*. 91–100.

Erzhou Zhu, Feng Liu, Zuo Wang, Alei Liang, Yiwen Zhang, Xuejian Li, and Xuejun Li. 2015. Dytaint: The implementation of a novel lightweight 3-state dynamic taint analysis framework for x86 binary programs. *Computers & Security* 52 (2015), 51–69.