to-BPA. Also, the combination of Arity with BPA can further increase the precision rate substantially.

C. *Profiling-based precision and recall*

Next we present a method of evaluating CFG construction precision and soundness based on runtime profile data. The idea is to profile a benchmark under a set of test cases and collect the targets of indirect calls. Then intuitively the precision rate is the percentage of CFG-predicted targets that actually appear as targets during profiling, and the recall rate is the percentage of those targets appearing in profiling that are predicted by the CFG. Note that since the set of targets with respect to test cases underapproximates the set of targets with respect to all possible inputs, the profiling-based precision rate is a lower bound of the real precision rate, while the profiling-based recall rate is an upper bound of the real recall rate.

For profiling, we collected a set of runtime traces by using Intel's Pin tool [31] on SPEC binaries. We used the extensive reference input datasets of SPEC2k6 to collect the runtime traces; we did not perform this for those security-critical benchmarks because they did not come with reference input datasets. We then used the following formula to calculate the average precision rate over all indirect calls:

$$Pc = \frac{1}{n}\sum_{i=1}^{n} Pc_i \;\; where \;\; Pc_i = \frac{TP_i}{TP_i + FP_i}$$

TP_i and FP_i are the numbers of true positives and false positives at indirect call site i, respectively. A true positive is a target that is predicted by CFG and also appears during profiling; a false positive is a target that is predicted by CFG but does not appear during profiling.

Table III shows profiling based precision rates for different techniques. In summary, AT, Arity, BPA, and Hybrid have the arithmetic mean of precision rates of 25.3%, 35.1%, 54.0% and 57.6% respectively, over all benchmarks and optimization levels. Thus, on average, BPA achieves a 18.9% higher precision rate than Arity.

As noted earlier, profiling based precision rates are underapproximated, as reference datasets may not trigger all program behavior, resulting in incomplete ground truth.

The formula for calculating the average recall rate is:

$$Rc = \frac{1}{n}\sum_{i=1}^{n} Rc_i \;\; where \;\; Rc_i = \frac{TP_i}{TP_i + FN_i}$$

TP_i and FN_i are the numbers of true positives and false negatives at indirect call site i, respectively. A false negative is a target that appears during profiling but is not predicted by CFG. By our experiments, BPA and AT achieve 100% recall rates, and Arity achieves 99.8% on perlbench, 98.9% on gcc, and 100% on all other benchmarks. BPA's 100% recall rate provides a validation that its generated CFGs are sound and enforcing them via CFI does not prevent the legitimate execution of an application.

D. *Case studies*

According to previous results, BPA can sometimes generate significantly better results than the arity-based method. To

TABLE III: Profiling based precision rates (for GCC 9.2).

Program	Opt Level	Precision (%)			
		AT	Arity	BPA	Hybrid
401.bzip2	O0	30.0	60.0	30.0	60.0
	O1	30.0	60.0	30.0	60.0
	O2	30.0	60.0	30.0	60.0
	O3	30.0	60.0	30.0	60.0
458.sjeng	O0	85.7	85.7	85.7	85.7
	O1	85.7	85.7	85.7	85.7
	O2	85.7	85.7	85.7	85.7
	O3	85.7	85.7	85.7	85.7
433.milc	O0	100.0	100.0	100.0	100.0
	O1	100.0	100.0	100.0	100.0
	O2	100.0	100.0	100.0	100.0
	O3	100.0	100.0	100.0	100.0
482.sphinx3	O0	4.2	66.7	80.0	80.0
	O1	4.2	54.2	80.0	80.0
	O2	2.4	59.5	88.6	88.6
	O3	2.4	59.5	88.6	88.6
456.hmmer	O0	4.5	4.5	90.5	90.5
	O1	4.5	4.5	90.5	90.5
	O2	4.5	4.5	90.5	90.5
	O3	7.1	7.1	100.0	100.0
464.h264ref	O0	0.9	1.7	14.6	14.9
	O1	0.8	1.1	11.9	12.1
	O2	0.8	1.1	3.1	3.4
	O3	0.8	1.1	3.1	3.4
445.gobmk	O0	1.8	2.3	37.3	37.7
	O1	1.8	2.3	22.2	22.7
	O2	1.8	4.3	24.5	24.5
	O3	1.8	2.3	17.6	17.7
400.perlbench	O0	0.4	0.7	29.2	29.5
	O1	0.4	0.8	30.3	30.7
	O2	0.5	0.9	34.7	35.1
	O3	0.3	0.5	24.4	24.6
403.gcc	O0	0.1	0.2	27.6	27.7
	O1	0.1	0.2	30.6	32.2
	O2	0.1	0.2	33.3	34.7
	O3	0.1	0.2	27.5	31.7

```
static int (*qcmp)();
int hit_comparison(_, _) {...}
void FullSortTophits(_ {
 specqsort(_, _, _, hit_comparison);
}
void specqsort(_, _, _, int (*compar)())
{ qcmp = compar;
  if ((*qcmp)(j, lo) > 0) ...
}
```

Fig. 10: Simplified code snippet from `456.hmmer`.

understand in what kinds of scenarios this happens, we did manual inspection on selected benchmarks.

We next discuss a typical case in `456.hmmer`. According to Sec. VII-B and Sec. VII-C, BPA generates much higher precision CFG on `456.hmmer` than Arity. It turns out that all the address-taken functions in `456.hmmer` are non-void functions and expect exactly two arguments; as a result, the arity-based method cannot refine the set of targets for an indirect call. In contrast, BPA's block-based