TABLE IV: Execution time by BPA and VSA. ∞ means timeout (exceeding 10 hours).

	Opt	Execution runtime (s)													
		bzip2	sjeng	milc	sphinx3	hmmer	h264ref	gobmk	perlbench	gcc	thttpd	memcached	lighttpd	exim	nginx
BPA	O0	17	158	35	24	62	350	1221	6919	33658	19	43	81	2554	2656
BPA	O1	8	116	32	31	68	332	1946	3756	23573	13	91	95	2121	2027
BPA	O2	8	131	33	36	79	379	1933	4006	27619	15	113	112	2728	2793
BPA	O3	12	152	34	42	75	1118	2290	4903	25246	17	131	151	2892	2855
VSA [11]	O0–O3	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

points-to analysis achieves high precision. Consider the code snippet adapted from `456.hmmer` in Fig. 10. In function `FullSortTophits`, there is a function call to `specqsort` with `hit_comparison`'s address passing to an argument in instruction 4. Then, instruction 7 assigns the argument's function address to a global variable `qcmp`, which is used in instruction 8 to load the function pointer. BPA precisely performs points-to analysis on this code pattern with well partitioned stack and global memory blocks. This is where BPA shows better results.

Our manual inspection also revealed where precision can be further improved. Currently BPA partitions the heap only at the granularity of heap allocation sites. This turns out to be the major avenue for precision loss, as applications often store control data and control-relevant data into a heap region allocated at a site, and also often use heap wrappers. We leave the work of balancing between better heap modeling and scalability for future work.

E. CFI evaluation

We integrated BPA's CFGs with a CFI implementation. We ran CFI-enforced binaries of SPEC2k6 benchmarks with their reference inputs, like we did for computing recall rates in Sec. VII-C. From the experiments, we observed no soundness violation, which shows BPA's applicability on CFI enforcement. Our prototype CFI is based on Intel's Pin [31] (v3.16) for dynamically instrumenting binaries. For CFI checks, we followed MCFI [33] and converted BPA-produced CFGs into a bitmap representation. The performance overhead of running SPEC2k6 binaries inside Pin with CFI checks compared to the case of running inside Pin without CFI checks is 11.7%, by calculating the geometric mean of the performance overheads of the binaries with the optimization level of O2. Note that Pin itself adds a considerable overhead on top of vanilla binaries (around 110% by our measurements). Further optimizations of our CFI prototype can be performed by adopting a more efficient binary instrumentation framework such as Dyninst [9] or UROBOROS [47], but we view this as mostly engineering work as there has been plenty of work of efficient CFI instrumentation with less than 5% overhead.

F. Performance evaluation

We conducted our experiments on Ubuntu 18.04 with 500GB of RAM and 32-cores CPU (Intel Xeon Gold 6136 with 3.00GHz). The machine has a large amount of RAM for the reason that Souffle [24] (v2.0.2), the Datalog engine we use, requires a large memory consumption. We collected memory usage data from the large benchmarks with more than 50K assembly instructions. Table V shows the results. Not surprisingly, `403.gcc` from SPEC2k6 consumes the largest amount of memory. Except for `403.gcc`, less than 64GB of RAM is sufficient for evaluation of other benchmarks.

TABLE V: Memory consumption by BPA for O2.

Prog	hmmer	h264ref	gobmk	perlbench	gcc	exim	nginx
Mem (GB)	0.6	3.6	28	57	352	48	24

Points-to analysis is known to be a hard problem, and the scalability issue rises even at the source level [27], [41]. As previously discussed, binary-level points-to analysis is even more challenging to scale due to the limited amount of information and the complexity of assembly code. Table IV presents the execution time of BPA for the benchmarks we use. Given that there is currently no practical points-to analysis framework at the binary level for resolving indirect-call targets, we believe the runtime results are acceptable and demonstrate BPA's scalability. Similar to memory usage, `403.gcc` takes the longest time (around 9.3 hours) to finish, due to its large size and its complexity in the usage of control-relevant data.

Execution time comparison with VSA. We chose BAP's [11] VSA implementation for the scalability comparison; this decision was motivated by a previous system called BDA [52]. The BDA paper claims that BAP-VSA is the only publicly available VSA framework for complicated benchmarks; other frameworks with VSA such as CodeSurfer [5] and ANGR [38] are either not publicly available or not suitable for complicated benchmarks. For example, ANGR only supports intraprocedural analysis, and therefore not suitable for inferring indirect call targets by points-to analysis, where interprocedural analysis is necessary in most cases. We tested BAP-VSA on our benchmarks; as Table IV shows, none of the benchmarks terminated within 10 hours. The results were consistent with the results reported by the previous paper [52]; when they ran their experiments with BAP-VSA on SPECINT2k (not SPEC2k6), only `181.mcf` terminated in 10.9 hours and others did not terminate within 12 hours. Note that runtime data from the original VSA paper [6] cannot be directly compared with our results, mainly due to the unsoundness in that implementation. The paper lists multiple reasons for possible unsound issues of their analysis, including the failure to resolve indirect call and jump edges that prevent further analysis.

VIII. Discussions and Future Work

BPA's static analysis has two major components: (1) memory block generation, (2) value-tracking analysis. The component of value-tracking analysis is sound, given a set of memory blocks. The first component, memory block generation, sometimes relies on heuristics (when partitioning the global data region). These heuristics may not be sound under all circumstances, meaning that the generated blocks may violate