BPA-X: An Architecture-Agnostic Block-Based Points-to Analysis for Stripped Binaries

Bokai Zhang
Pennsylvania State University
bzz5205@psu.edu

Monika Santra
Pennsylvania State University
monikas@psu.edu

Syed Rafiul Hussain
Pennsylvania State University
hussain1@psu.edu

Gang Tan
Pennsylvania State University
gtan@psu.edu

***Abstract*—Sound indirect-call resolution for stripped binaries is critical for security applications such as CFI enforcement, debloating, and large-scale vulnerability discovery, yet it remains challenging in the absence of symbol and type information. A recent work, Block-Based Points-to Analysis (BPA) addresses this problem with a scalable block memory model, but its implementation is tightly coupled to 32-bit x86 through an ISA-specific disassembly pipeline.**

To overcome this limitation, we present BPA-X, an architecture-agnostic block-based points-to analysis framework for stripped binaries across multiple ISAs. BPA-X preserves the core soundness assumptions of BPA's block memory model while replacing x86-specific components with an architecture-agnostic VEX IR via binary analysis platform *angr*. It generalizes local and global memory-block partitioning using VEX semantics instead of x86-specific patterns, lifts VEX IR into SSA form, and performs fixpoint computation on interprocedural value tracking and reachability analysis.

Our evaluation on SPEC CPU 2006 and real-world server binaries shows that BPA-X improves memory-block partitioning, reduces AICT on many x86 programs compared to BPA, and extends the analysis to x64 without degrading much precision. BPA-X also reduces memory consumption by 25% and improves runtime on large benchmarks.

I. Introduction

Constructing high-precision Control-Flow Graphs (CFGs) for commercial off-the-shelf (COTS) stripped binaries is fundamental in several security applications, including Control-Flow Integrity (CFI) and software debloating. However, accurately resolving the targets of indirect calls remains a significant challenge since the operand of a call instruction remains unknown until runtime. The absence of source-level information like symbols and types further complicates static inference.

Current reverse engineering frameworks often face a difficult trade-off between performance and completeness. Heavyweight approaches, such as `angr`'s CFGEmulated [1], utilize symbolic execution to resolve targets, but this method is often hampered by state space explosion, rendering it impractical for large-scale applications. Conversely, frameworks like `Ghidra` [2] and `angr`'s CFGFast [1] primarily rely on heuristics and intra-procedural constant propagation, which are inaccurate in complex scenarios. While these tools are efficient, they frequently fail to resolve complex indirect branches. Platforms like Binary Ninja attempt a middle ground using Intermediate Language (IL)-based data-flow analysis, yet soundly resolving indirect calls in stripped binaries remains an open problem.

Prior research has introduced various techniques to address this gap. Recent static approaches like BinDSA [3] and MANTA [4] attempt to infer types from stripped binaries and resolve indirect call targets, but they prioritize precision and efficiency over soundness to minimize false positives. Hybrid approaches such as TypeSqueezer [5], SchedExec [6], and CBench [7] have limitations on coverage, especially on stripped binaries, which lack symbol tables and type information. In comparison, ML-based approaches such as CALLEE [8], AttnCall [9], and DISA [10] utilize neural networks to identify targets of indirect calls, but these ML-driven models have to handle information loss when encoding complex instruction semantics into fixed vector representations, and these solutions are not sound as shown in their evaluation. High-precision tools like KallGraph [11], TFA [12], MLTA [13], and DeepType [14] achieve better results but rely on source-level metadata that is absent in COTS binaries.

One approach to this problem is BPA [15], which proposes a block-based memory model to perform static points-to analysis on stripped binaries. By leveraging this model, it provides a sound and scalable solution. However, BPA [15] and derivative tools like BinPointer [16] are limited to 32-bit x86 binaries. This is because they rely on the RockSalt disassembler [17], which produces an intermediate representation called Register-Transfer Language (RTL); however, there are currently no RTL models for x64 nor ARM. Modern malware and vulnerabilities are no longer confined to the 32-bit x86 ecosystem, and threats target a heterogeneous range of platforms from ARM-powered mobile devices to x64 servers. To counter this, defensive analysis tools must abstract away hardware specifics, prioritizing semantic logic over instruction-set implementation.

In this paper, we present BPA-X, an architecture-agnostic, block-based points-to analysis framework designed to resolve indirect call targets in stripped binaries. Based on the VEX IR lifted by the `angr` framework, BPA-X decouples the analysis logic from architecture-specific instruction sets. We extend the

Workshop on Binary Analysis Research (BAR) 2026
27 February 2026, San Diego, CA, USA
ISBN 978-1-970672-08-4
https://dx.doi.org/10.14722/bar.2026.23064
www.ndss-symposium.org

original block memory model to handle VEX IR semantics, and implement efficient value set analysis. Overall, BPA-X makes the following contributions:

1) Architecture-Agnostic Analysis: Based on VEX IR generated by `angr`, BPA-X performs portable points-to analysis across multiple architectures without requiring ISA-specific assumptions.
2) Enhanced Memory Block Model: We generalize the local and global memory block heuristics to be more conservative in partitioning memory regions for the soundness of indirect call resolution.
3) Evaluation: We evaluate BPA-X on SPEC CPU 2006 benchmarks and real-world server applications across both x86 and x64 platforms. Our results demonstrate that BPA-X successfully extends analysis to x64 binaries without compromising precision compared to architecture-specific tools. Furthermore, BPA-X significantly optimizes performance, achieving up to a 25% reduction in memory consumption of programs compiled with -O2 optimization level and improving execution time on most large binaries.

II. Related Work

A. Analysis-Based Approaches

TypeArmor [18] represents a foundational and influential approach to resolving indirect call targets in stripped binaries with static analysis. It utilizes liveness and use-def analysis to reconstruct conservative approximations of function prototypes and call site signatures. While it establishes a sound many-to-many mapping based on argument counts and return value usage, TypeArmor [18] inherently over-approximates to maintain binary-level compatibility. Its results are often considered coarse-grained compared to the increased precision achieved by more recent, sophisticated frameworks.

BPA [15] is another tool for discovering indirect call targets with static analysis. It addresses the scalability bottlenecks of traditional value set analysis by introducing a block memory model, which partitions the memory space into disjoint blocks to avoid expensive offset tracking. While this abstraction allows for scalable points-to analysis, the framework's portability is severely constrained. BPA is implemented on top of the RockSalt [17] disassembler, and since RockSalt [17] targeted only the 32-bit x86 architecture, the analysis rules and heuristics are tightly coupled to this specific IR, making the tool unable to analyze binaries from other architectures without significant re-engineering.

BinDSA [3] prioritizes precision and efficiency over strict soundness. It employs a "soundy" design philosophy, utilizing field- and context-sensitive unification algorithms alongside heap reconstruction to filter out false positives. However, this gain in precision comes at the cost of recall. By relying on recovered type information to prune conflicting points-to sets, BinDSA [3] sacrifices soundness for precision, as shown in its evaluation. Consequently, while highly precise, BinDSA [3] cannot guarantee the exhaustive discovery of all control flow transfers, which is a critical requirement for security applications like CFI enforcement.

TypeSqueezer [5] introduces a hybrid approach for refining indirect call targets in stripped binaries by combining static analysis with dynamic profiling. It operates on a "squeeze" philosophy, using runtime observations to cluster indirect call sites and address-taken functions, thereby tightening the lower and upper bounds of their arity and argument widths. However, this approach is fundamentally limited by its dependence on code coverage of test cases, which is common in hybrid approaches.

Recent research has demonstrated that leveraging rich type information can significantly improve indirect call resolution. Tools such as KallGraph [11], TFA [12], DeepType [14], and MLTA [13] achieve high precision by enforcing multi-layer type consistency or integrating value-flow analysis to filter targets. However, these approaches fundamentally rely on source-level metadata or intermediate representations (IR) for precise type definitions, which are typically absent in COTS binaries. While type information can be partially reconstructed [19], [20], their accuracy remains insufficient for the complex composite types required to effectively refine indirect call targets in stripped binaries.

B. Machine-Learning-Based Approaches

To mitigate the over-approximation issue in static analysis, recent ML-based approaches like CALLEE [8] and AttnCall [9] utilize neural networks to learn the contextual relationships between indirect callers and callees. However, finding the features which can distinguish different callers and callees remains a significant challenge. Furthermore, these models often struggle to derive meaningful representations from small functions with sparse semantic information and fail to capture long-range dependencies across function boundaries.

Alternatively, DISA [10] integrates deep learning with static analysis by employing a transformer-based model to predict memory block boundaries, aiming to refine the precision of BPA [15]. However, its reliance on intraprocedural analyses limits its capability to resolve global memory accesses that involve complex pointer arithmetic or are passed through function arguments. Consequently, its improvement on global memory partitioning remains marginal compared to local memory, restricting the precision of indirect call target resolution in programs that heavily utilize global data structures. In addition, it is also limited to 32-bit x86 programs due to reliance on the original BPA [15] framework.

III. Background

A. Background of `angr`

`angr` [1] is a multi-architecture binary analysis platform designed to perform complex tasks such as Control-Flow Graph (CFG) recovery and symbolic execution. In the context of BPA-X, its primary role is as a binary-to-IR lifter that translates machine code into VEX IR. VEX is a RISC-style

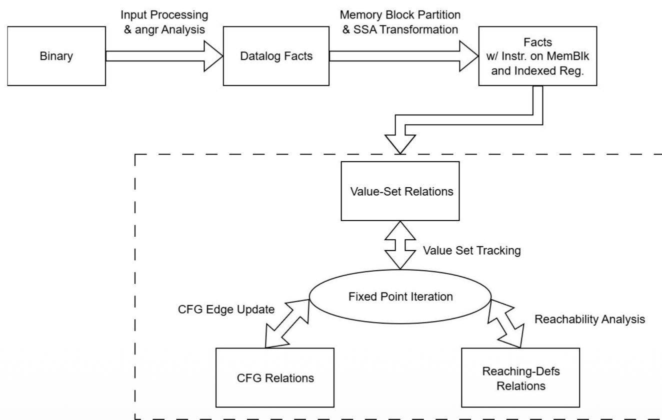

Fig. 1. Workflow of BPA-X

intermediate representation from Valgrind [21] that decomposes complex CISC instructions into a series of simple operations. A key advantage of using VEX IR is its architecture-agnostic design; it abstracts away hardware-specific details by representing all side effects. This abstraction provides a stable, uniform interface for pointer analysis across different instruction set architectures (ISAs) without necessitating ISA-specific rules for every supported platform.

B. Background of Datalog

Datalog is a declarative logic programming language that is often used for implementing large-scale static analyses [15], [16], [22]–[24]. It is a Prolog subset that guarantees termination on all inputs. Analysis in Datalog is expressed as a set of recursive rules that derive new facts from a base set of input relations. This language is particularly well-suited for BPA-X binary analysis because it inherently supports fixed-point computation. In a points-to analysis framework, this allows for the seamless integration of mutually dependent components. Its declarative nature also decouples the technical specification of the points-to rules from the underlying search and optimization strategies of the solver, enabling high-performance evaluation.

IV. Methodology

A. Input Processing

As shown in Fig. 1, BPA-X first lifts the input binary into VEX IR and constructs the CFG without indirect call edges by using the `angr` binary analysis framework [1]. The processing pipeline consists of three parts: IR lifting, function boundary identification, and Control Flow Graph (CFG) construction.

VEX IR Lifting. To maintain architecture-agnostic analysis, BPA-X uses `angr` [1] to translate binaries into VEX IR code. Fig. 2 shows an example of how `angr` lifts a snippet of x64 binary into the IR. This lifting process translates complex operations into RISC-style operations. For example, at location `0x112d` in Fig. 2, `push` is translated into multiple simple operations on `rbp` and `rsp`. `angr` also abstracts side effects into explicit IR operations. At location `0x1134` in Fig. 2, flag

```
0x112d: push   %rbp
0x112e: mov    %rsp,%rbp
0x1131: mov    %edi,-0x4(%rbp)
0x1134: cmpl   $0x0,-0x4(%rbp)
```

```
------ IMark(0x112d, 1, 0) ------
t0 = GET:I64(rbp)
t8 = GET:I64(rsp)
t7 = Sub64(t8,0x0000000000000008)
PUT(rsp) = t7
STle(t7) = t0
------ IMark(0x112e, 3, 0) ------
PUT(rbp) = t7
PUT(rip) = 0x0000000000401131
------ IMark(0x1131, 3, 0) ------
t10 = Add64(t7,0xfffffffffffffffc)
t13 = GET:I64(rdi)
t25 = 64to32(t13)
STle(t10) = t25
PUT(rip) = 0x0000000000401134
------ IMark(0x1134, 4, 0) ------
t14 = Add64(t7,0xfffffffffffffffc)
t5 = LDle:I32(t14)
PUT(cc_op) = 0x0000000000000007
t26 = 32Uto64(t5)
PUT(cc_dep1) = t26
PUT(cc_dep2) = 0x0000000000000000
PUT(rip) = 0x0000000000401138
```

Fig. 2. An Example of `angr` Lifting x64 to VEX IR

updates caused by `cmpl` is translated into updates on `cc_op`, `cc_dep1`, and `cc_dep2`, which are usually read in another conditional jump instruction. These abstractions are essential for subsequent analyses.

Function Boundary Identification. BPA-X includes interprocedural analysis to find value sets of related subjects, but the accuracy depends heavily on the correct identification of function boundaries. While BPA [15] originally relies on heuristics, identifying these boundaries in stripped binaries is still a very difficult task for static tools. Although recent studies use machine-learning-based approaches to recover function boundaries in stripped binaries, such as XDA [25] and DeepDi [26], they still remain probabilistic and inherently suffer from false positives and negatives. As a result, we treat function discovery as an orthogonal challenge. To ensure the integrity of our points-to results, we utilize ground truth metadata from the symbol table of non-stripped version of the same binary for the function boundaries. However, it is important to note that BPA-X is not strictly bound to this approach, and it is designed to be compatible with any function boundary detection tools.

Address-Taken Functions. Once function boundaries are established, address-taken functions are identified by scanning the data regions and immediate operands in the instructions that point to valid function entry addresses, marking them as potential targets for indirect calls.

Initial CFG Construction. BPA-X initiates its analysis by constructing a Direct Control-Flow Graph (DCFG), a partial graph that serves as the baseline structure by capturing only direct control flows and excluding indirect call edges.

This DCFG is generated based on angr's CFGFast [1] for scalability. During this process, BPA-X relies on angr's built-in heuristics to resolve indirect jumps within jump tables. To address compiler-induced tail-call optimizations, the system applies logic similar to BPA [15], where each detected tail-call jump is substituted with a call CFG edge and a return CFG edge pointing to the immediate next location. Additionally, for analytical convenience, an intraprocedural edge is explicitly added connecting a call instruction to its follow-up basic block. Furthermore, like BPA [15], edges are removed for calls to non-returning library functions or functions lacking reachable return instructions.

B. Memory Block Partition

Similar to BPA [15], BPA-X partitions memory regions into memory blocks for scalable value tracking analysis with some trade-off on precision. A memory block holds, e.g., a local variable, an array, or a struct of fields. The boundaries of memory blocks must respect a pointer-arithmetic assumption, which states that adding an offset to a pointer that points to some memory block must result in a pointer pointing to the same memory block. This assumption keeps the value tracking analysis sound when an analysis is not tracking offsets within memory blocks. The data memory has 3 disjoint regions: stack frame of each function, global data regions shared by all functions, and the dynamically allocated heap region. For each memory region, BPA-X avoids relying on compiler-specific 32-bit x86 instruction patterns (as BPA [15] does); instead, it leverages the semantics of the lifted VEX IR, as explained below.

Local Memory Blocks. Within each function, BPA-X analyzes its VEX IR instructions to compute what stack addresses are used and to determine which stack addresses are boundaries of local memory blocks. The goal is to identify the starting addresses of local variables that are defined in a program or allocated by the compiler, and discard the addresses in the middle of local variables, to satisfy the assumption of memory block models for sound analyses.

For each function, the stack address is represented as `sp + offset`. Here, `sp` denotes the value of the stack frame pointer (e.g., `esp` in x86, `rsp` in x64, or `SP` in ARM) at the moment of function entry. The `offset` is the displacement relative to `sp` as resolved during analysis. BPA-X identifies the following stack addresses as memory block boundaries:

1) **Function Arguments:** When the `offset` resolves to a positive constant, any `sp + offset` is identified as a function argument. This is because stack-passed arguments are located at higher addresses relative to the stack frame pointer's position at the start of function execution.
2) **Caller-Allocated Callee Arguments:** When the `offset` resolves to a negative constant and the address is utilized in instructions preparing for a function call, `sp + offset` identifies a stack slot allocated for outbound arguments. These slots are used when the number of arguments exceeds the capacity of architecture-specific registers. For example, this includes the 7th and subsequent arguments in x64 (System V ABI), the 9th and subsequent in ARM64, and all arguments in standard x86 calling conventions. Different from BPA [15], we explicitly include those addresses as local memory block boundaries.
3) **Start Address of a Primitive or Composite Data Structure:** When `offset` resolves to a negative constant, and the derived stack address `sp+offset` (computed from the stack-frame pointer or stack-base pointer) is moved into a general-purpose register or stored to memory, we identify `sp+offset` as a local memory block boundary. This heuristic relies on the observation that compilers typically load the base address of a composite typed variable into a register before accessing its internal fields. Compared to BPA [15], BPA-X adopts a more conservative approach to local memory blocks by focusing exclusively on memory locations related to stack pointers in this category. The reason is that memory locations with constant offsets can often be a location of an array element when its index is fixed. For example, in 32-bit x86, `lea 0x1(%eax),%ebx` can be the location of the second element in an array indexed by `%eax`. In BPA [15], it is treated as a new local memory block boundary, which violates the memory block model, but this address is discarded in BPA-X. Although this design may miss some memory locations of the beginning of a local variable, it satisfies the assumption of the memory block model for soundness.

Global Memory Blocks. BPA-X performs global memory partitioning similarly to BPA [15], but with refinements tailored to architecture-agnostic VEX IR. When a constant value falls within the range of global data regions (e.g., `.data`, `.bss`, or `.rodata`), the following rules are applied to identify candidate global memory block boundaries:

1) **Constants stored into memory or registers:** Specifically `STle/STbe(temp) = const` or `PUT(reg) = const` in VEX IR. This reflects scenarios where a global pointer is initialized or passed as a function argument, indicating the start of a referenced data structure or object.
2) **Constants used as direct addresses:** Specifically `STle/STbe(const)` or `LDle/LDbe(const)` in VEX IR. This reflects scenarios where the program accesses a global scalar variable or the first element of a global array via absolute addressing.
3) **Constants used in base-offset arithmetic:** Specifically `const + expr`. This reflects accesses to fields within a global struct or array, where the constant represents the base address of the object and the expression represents a dynamic offset.
4) **Constants used in scaled-index arithmetic:** Specifically `const + (expr * scale)`. This reflects array indexing operations, where the constant serves as the base address of the array and the scaled expression

calculates the offset for a specific element index.

Unlike BPA [15], which is limited to matching complex 32-bit x86 specific addressing modes within individual instructions, BPA-X operates on VEX IR. This allows it to analyze a broader context across the entire function, ensuring the analysis is portable to multiple architectures such as x64 and ARM.

After identifying these candidates, BPA-X filters out "false candidates" that point to the middle of composite data structures. Like BPA [15], when scales are involved in addressing, BPA-X estimates the access range using the set of possible values derivable via pointer arithmetic from each candidate boundary. If a candidate falls within the range generated by another boundary, it is an internal offset rather than a unique block boundary and is removed. This refinement process ensures the analysis remains sound by correctly identifying the true heads of arrays of composite-typed elements, even in the presence of aggressive compiler optimizations.

Heap Memory Blocks. To partition the heap, BPA-X follows a methodology similar to the original BPA [15] by adopting an allocation-site approach. In this model, all memory dynamically allocated at a specific call site to an allocation function, such as `malloc`, `calloc`, or `realloc`, is grouped into a single memory block.

Each heap block is assigned a unique allocation site ID, which is determined by the address of the call instruction that performed the allocation. This approach relies on the assumption that the binary is dynamically linked, allowing the analysis to identify calls to standard library memory management functions even when symbol tables are stripped. By treating each unique allocation site as a distinct block, BPA-X maintains the core pointer-arithmetic assumption of the block memory model: that a pointer originating from one heap allocation cannot undergo pointer arithmetic to point to a different heap block.

C. SSA Transformation

Before performing value set analysis, BPA-X transforms the program into the Single Static Assignment (SSA) form to ensure each variable is assigned exactly once. This transformation is critical for balancing analysis precision and scalability, as a flow-insensitive analysis on variables in the SSA form is equivalent to a flow-sensitive analysis on the original variables before the SSA transform.

Scope of SSA Indexing. In the context of VEX IR, BPA-X distinguishes between components that are SSA-indexed and those that remain unindexed:

1) SSA-Indexed Components: The analysis renames and indices all machine locations, specifically architecture-specific registers (e.g., `rax`, `rbx`). Additionally, BPA-X extends BPA [15] by indexing function arguments on the stack, treating the local memory blocks representing these arguments as versioned variables to track interprocedural data flow.
2) Non-SSA-Indexed Components: To maintain scalability, BPA-X does not index temporary registers created by `angr` in VEX IR (which are already unique and assigned only once in each basic block), global memory blocks, heap memory blocks, or local memory blocks representing internal local variables. This choice trades flow-sensitivity of memory blocks for scalability.

Efficient Incremental SSA Construction. To optimize the transformation across diverse architectures, BPA-X utilizes an architecture-agnostic classification of registers:

1) Callee-Saved (Non-Volatile) Registers: These registers are expected to preserve their values across function calls. BPA-X performs a reaching definition analysis for these registers only once, as their definitions are stable even when new indirect call related CFG edges are added.
2) Caller-Saved (Volatile) Registers: These registers may be overwritten during function calls. BPA-X maintains a separate reaching definition analysis for these registers. Importantly, it employs an incremental analysis approach by placing additional ϕ-functions at all merge points. By adapting an efficient, incremental SSA algorithm, BPA-X can update the possible values of the affected registers dynamically when more indirect call edges are discovered in the CFG. It allows scalable interprocedural value set analysis while maintaining the correctness of the interprocedural data flow.

D. Value Tracking Analysis

BPA-X performs an interprocedural value tracking analysis to compute the set of possible values—specifically function start addresses and memory block/chunk addresses—that each abstract location may hold. This analysis is implemented using Datalog logic rules that process the VEX IR program in its SSA form.

Tracked Objects and Values. Following the design of BPA [15], the analysis tracks values for abstract locations, which include SSA-indexed registers, temporary registers created by `angr`, and the various memory blocks generated during the partitioning phase (stack, global, and heap). The analysis is flow-insensitive because the SSA transformation effectively records control-flow information through its renaming. The primary goal is to maintain a Value Set for each location, consisting solely of potential code addresses (address-taken function addresses) and base addresses of memory blocks. Internal constant offsets and non-pointer values are abstracted away to maintain the scalability of the analysis.

CFG Updates with Indirect Calls. In BPA-X, the Control-Flow Graph (CFG) is dynamically expanded as the value sets of indirect call destinations are updated. For example, in the VEX IR instruction `PUT(rip) = t5` (lifted from an x64 instruction `call *%rax`), the temporary register `t5` represents the value of `rax` at the call site. If the value set of `t5` is found to contain the start address of function `foo`, BPA-X inserts the following edges into the CFG:

1) A Call Edge: From the basic block containing the call site to the entry point of function `foo`.

TABLE I
ACCURACY OF LOCAL MEMORY BLOCK BOUNDARIES

Platforms	Opt	BPA-X		BPA	
		Wrong (%)	Miss (%)	Wrong (%)	Miss (%)
x86	O0	0.69	50.01	3.51	33.45
	O1	1.52	9.82	4.42	17.18
	O2	2.42	11.30	5.88	18.45
	O3	1.71	11.75	8.07	19.87
x64	O0	0.66	79.77	-	-
	O1	5.99	16.83	-	-
	O2	5.53	26.30	-	-
	O3	5.55	26.44	-	-

TABLE II
ACCURACY OF GLOBAL MEMORY BLOCK BOUNDARIES

Platforms	Opt	BPA-X		BPA	
		Wrong (%)	Miss (%)	Wrong (%)	Miss (%)
x86	O0	0.39	30.61	0.26	31.37
	O1	10.86	33.00	35.06	32.95
	O2	11.71	33.36	36.22	33.59
	O3	12.76	32.21	33.77	32.43
x64	O0	4.16	30.52	-	-
	O1	15.20	30.67	-	-
	O2	15.23	32.42	-	-
	O3	16.91	31.97	-	-

2) Return Edges: From the returning basic blocks of `foo` back to the basic block immediately following the call site in the caller.

Reachability Analysis and Value Set Updates. The addition of new CFG edges also affects the program data flow. When new caller-callee relationships are established, values in argument-passing local memory blocks and caller-saved registers are propagated between the caller and the new callee. This occurs because the new CFG edges reveal additional definitions for these objects that were previously unreachable. Furthermore, the value sets of other dependent objects are updated on-demand whenever they have a data-flow dependency on these newly updated sets.

Fixed Point Iteration. The construction of the CFG and the tracking of value sets are mutually dependent. The analysis begins with an initial CFG containing only direct branch edges. The process then enters a recursive cycle:

1) Value set analysis identifies new potential targets for indirect calls.
2) Incremental CFG updates insert these new edges into the graph.
3) Expanded reachability introduces new definitions for tracked objects, which in turn adds more values to their respective value sets.

Following the methodology of BPA [15], this entire workflow is implemented in Datalog. It leverages the Datalog engine to automatically perform the fixed-point computation, terminating only when no further values can be added to the value sets and the CFG remains stable.

V. EVALUATION

In this section, we evaluate the effectiveness and efficiency of BPA-X. We aim to answer the following research questions:

TABLE III
AICT COMPARISON.

Programs	Platforms	Opt	BPA-X	BPA
hmmer	x86	O0	2.8	2.9
		O1	2.5	4.3
		O2	13.4	2.8
		O3	14.0	1.0
	x64	O0	17.1	-
		O1	9.8	-
		O2	3.2	-
		O3	2.7	-
h264ref	x86	O0	5.3	4.3
		O1	4.8	4.1
		O2	23.4	26.4
		O3	15.0	18.0
	x64	O0	6.0	-
		O1	4.3	-
		O2	4.4	-
		O3	4.8	-
gobmk	x86	O0	1053.7	884.6
		O1	1026.9	1334.8
		O2	1043.3	1297.2
		O3	1008.9	1376.5
	x64	O0	1626.4	-
		O1	1042.2	-
		O2	1117.9	-
		O3	1079.4	-
perlbench	x86	O0	458.5	400.3
		O1	361.1	364.2
		O2	352.7	363.7
		O3	369.2	453.4
	x64	O0	463.3	-
		O1	366.5	-
		O2	352.5	-
		O3	451.5	-
exim	x86	O0	29.5	31.3
		O1	25.3	29.6
		O2	25.6	30.6
		O3	37.0	40.6
	x64	O0	36.0	-
		O1	31.9	-
		O2	26.0	-
		O3	35.2	-
nginx	x86	O0	466.9	444.0
		O1	435.3	463.4
		O2	487.5	525.1
		O3	474.7	511.0
	x64	O0	434.7	-
		O1	414.9	-
		O2	444.4	-
		O3	394.5	-

- **RQ1 (Memory Block Accuracy):** How accurate are the memory block boundaries identified by BPA-X's architecture-agnostic heuristics compared to ground truth and prior architecture-specific approaches in BPA [15]?
- **RQ2 (Indirect Call Resolution):** How precise is BPA-X in resolving indirect call targets, and does the architecture-agnostic design maintain the precision levels as BPA [15]?
- **RQ3 (Performance):** What is the execution time and memory consumption of BPA-X?

A. Experimental Setup

All experiments were conducted on a machine with 16 cores of CPU (Intel Xeon Gold 6136 with 3.00GHz) and 350GB of RAM. We implemented BPA-X with Souffle version 2.4 and

TABLE IV
PROFILING-BASED PRECISION (%) AND RECALL (%) COMPARISON

Programs	Platforms	Opt	BPA-X	BPA	angr	
			Prec	Prec	Prec	Rec
hmmer	x86	O0	80.3	80.2	N/A	0
		O1	83.9	82.8	N/A	0
		O2	7.7	90.5	N/A	0
		O3	7.1	100	N/A	0
	x64	O0	4.7	-	0	0
		O1	8.7	-	0	0
		O2	28.4	-	0	0
		O3	30.0	-	0	0
h264ref	x86	O0	10.0	9.9	N/A	0
		O1	10.2	10.1	N/A	0
		O2	0.1	0.0	0	0
		O3	1.9	1.7	0.3	0.8
	x64	O0	9.9	-	0	0
		O1	10.3	-	0	0
		O2	10.3	-	0.3	0.9
		O3	10.1	-	0.3	0.8
gobmk	x86	O0	21.1	35.1	N/A	0
		O1	24.3	20.0	N/A	0
		O2	23.9	19.9	N/A	0
		O3	23.9	13.1	N/A	0
	x64	O0	4.0	-	0.0	0
		O1	24.0	-	0.0	0
		O2	21.8	-	0.0	0
		O3	21.8	-	0.0	0
perlbench	x86	O0	23.9	27.1	24.5	34.3
		O1	25.6	26.6	29.5	41.4
		O2	30.0	32.3	34.6	50.7
		O3	15.2	16.4	19.4	43.0
	x64	O0	24.0	-	25.0	36.1
		O1	25.2	-	29.3	42.3
		O2	27.3	-	31.0	45.6
		O3	13.5	-	16.7	37.7

`angr` version 9.2.173. We evaluated BPA-X using the C programs from the SPEC CPU 2006 benchmark, `thttpd-2.29`, `memcached-1.5.4`, `lighttpd-1.4.48`, `exim-4.89`, and `nginx-1.10`, which are consistent with the dataset used in the BPA evaluation [15]. We compiled these programs using GCC-9.2 with optimization levels ranging from `-O0` to `-O3` on both x86 and x64 platforms. Due to memory constraints, we cannot finish `gcc` in SPEC benchmark; note that the original BPA implementation used more memory than 350GB when analyzing `gcc`.

B. Memory Block Accuracy

A core contribution of BPA-X is its architecture-agnostic memory block partitioning. Since incorrect partitions (unsound boundaries) can lead to violations of the block memory model, and missed partitions (coarse granularity) lead to precision loss, validating these heuristics is critical.

To examine **RQ1**, we compare the boundaries generated by BPA-X against the ground truth derived from DWARF information [27] of the unstripped version of those binaries. We classify a boundary as *Wrong* if it splits a single variable into multiple blocks (threatening soundness) and *Miss* if it fails to separate distinct variables (reducing precision). For global memory blocks, we exempt some *Wrong* boundaries when the corresponding global constants are never involved in pointer arithmetic, implying that the pointer-arithmetic assumption relied upon by the block memory model is not violated. Conservatively, we do not exempt constants that are moved into registers or stored in memory.

Table I and Table II present the average accuracy of memory block partitions on all of the programs tested. As BPA [15] does not support x64 binaries, some entries are blank. Evaluation of heap memory blocks is omitted as it does not involve heuristics. On x86 programs, BPA-X demonstrates a lower wrong rate compared to BPA [15] across almost all optimization levels, validating our conservative heuristic design.

C. Accuracy of Indirect Call Analysis

To examine **RQ2**, the first metric that we use is average indirect call targets (AICT), which is commonly used in previous studies [3], [8], [10], [15]. As shown in Table III, BPA-X demonstrates accuracy comparable to the baseline BPA. We omit programs where BPA-X and BPA yield identical AICT values. We find that BPA-X also achieves lower AICT values on complex benchmarks such as `h264ref` and `gobmk` under higher optimization levels (O2 and O3). This indicates that extending the analysis to x64 does not compromise the precision of the underlying algorithm on x86 targets.

Another way of evaluation is using profiling to collect the actual targets of indirect calls under test inputs and treating them as the pseudo ground truth when measuring precision and recall. This pseudo ground truth is dependent on the coverage of the test inputs, and we use the reference input in SPEC benchmarks and Intel's Pin tool [28] to collect the targets of indirect calls. This approach of estimating the ground truth of indirect call targets by dynamic tracing is also used by [11], [15], [16]. Table IV shows the precision and recall of BPA-X, BPA [15], and `angr`'s CFGFast [1] (v9.2.173). As BPA-X is built based on `angr`'s CFGFast [1], for comparison, we also include its metrics. Different from the results in BPA [15], we independently evaluated it as its original ground truth is not publicly available. Similarly, we omit programs in the table where BPA-X and BPA [15] achieve identical precision and recall. Additionally, we note that both BPA-X and BPA achieve 100% recall on these programs and thus the recall rates for them are omitted from the table; further, "N/A" in the `angr` columns indicates no indirect call targets are predicted.

The metrics of AICT and precision show that BPA-X successfully extends BPA [15] to x64 platforms without compromising the precision. As evidenced in Table III, BPA-X achieves AICT values comparable to BPA on x86 programs and yields even lower average target counts at higher optimization levels. Regarding the profiling-based precision in Table IV, the results are largely consistent with BPA, with the notable exception of `hmmer` on x86. The divergence in precision for `hmmer` is attributed to the limitations of the profiling-based ground truth. BPA-X identifies potential targets that were not covered by the reference test inputs, consequently marked as false positives, whereas BPA failed to predict the targets of these call sites in `hmmer`. Thus, BPA-X provides a more comprehensive solution compared to

TABLE V
EXECUTION TIME COMPARISON ON BINARIES OF OPTIMIZATION LEVEL -O2

Tools	Platforms	Execution Time (s)												
		bzip2	sjeng	milc	sphinx3	hmmer	h264ref	gobmk	perlbench	thttpd	memcached	lighttpd	exim	nginx
BPA-X	x86	10	22	23	41	75	193	1043	4194	14	37	133	3914	3153
	x64	12	23	21	30	69	332	5182	8014	12	26	105	4893	1359
BPA	x86	8	131	33	36	79	379	1933	4006	15	113	112	2728	2793

TABLE VI
MEMORY CONSUMPTION ON BINARIES OF OPTIMIZATION LEVEL -O2

Tools	Platforms	Memory Consumption (GB)					
		hmmer	h264ref	gobmk	perlbench	exim	nginx
BPA-X	x86	0.4	1.9	16.2	61.3	32.7	28.9
	x64	0.4	3.1	53.1	130.2	41.2	21.4
BPA	x86	0.6	3.6	28	57	48	24

the baseline and strictly outperforming angr's CFGFast [1] in terms of accuracy.

D. Performance Evaluation

To examine **RQ3**, we evaluate the end-to-end execution time and memory consumption.

Execution Time. We compare the execution time of BPA-X on x86 binaries against the BPA [15] to assess the overhead of our architecture-agnostic design. We also report the performance on x64 binaries to demonstrate the scalability of BPA-X on 64-bit architectures, which BPA does not support.

On the x86 programs of the benchmark, compared with BPA [15], BPA-X has average speedups of 1.16x, 1.32x, 1.34x, and 1.15x on binaries compiled with `-O0`, `-O1`, `-O2`, and `-O3` optimization levels, respectively. Table V presents the details of execution time for programs compiled with `-O2` optimization. Results for other optimization levels are omitted.

We attribute this efficiency to the optimized Datalog rules on the VEX IR abstraction, which avoids pattern matching on the x86 semantics. While BPA-X is slower on certain benchmarks like `exim` and `nginx`, it remains within a comparable magnitude.

For x64 binaries, the analysis time generally increases due to the increased number of registers associated with 64-bit architecture. However, for smaller benchmarks like `bzip2` and `sjeng`, the performance difference between x86 and x64 is negligible. This validates that BPA-X is scalable to analyze binaries on different architectures.

Memory Consumption. Table VI details the peak memory consumption during the analysis. BPA-X exhibits a significant reduction in memory usage compared to BPA [15] on x86 binaries. For example, BPA-X consumes only 16.2 GB on `gobmk` x86 binaries, which is a 42% reduction compared to the 28 GB required by BPA.

This improvement validates our design choice in SSA transformation, where we selectively avoid SSA-indexing for temporary registers and fix reachability analysis for callee-saved registers.

On x64 platforms, memory consumption naturally rises due to more registers used in programs. Since all machine registers are SSA-indexed, it will create more copies during analysis, but the overall memory footprint remains within a manageable range.

VI. CONCLUSION

This paper presented BPA-X, an architecture-agnostic redesign of block-based points-to analysis for stripped binaries. By lifting binaries to VEX IR and expressing interprocedural value tracking over an SSA-form IR with incremental updates, BPA-X decouples BPA-style indirect-call resolution from ISA-specific disassembly and enables portable analysis across architectures while maintaining the core soundness assumptions of the block memory model. Empirically, BPA-X delivers AICT comparable to BPA on x86, extends the analysis to x64, and can substantially improve performance on large programs.

Our current evaluation treats function boundary recovery as an orthogonal problem and therefore uses ground-truth boundaries from non-stripped binaries; integrating and stress-testing BPA-X with state-of-the-art boundary recovery (including probabilistic ML-based tools) is a natural next step. Additional future work includes further improving scalability on very large x64 binaries (where register width and calling-convention effects increase value-set propagation pressure), extending evaluation to additional ISAs (e.g., ARM), exploring complementary refinements that tighten precision without weakening soundness, and applying sound points-to analysis in downstream applications such as control flow integrity and software debloating.

ACKNOWLEDGMENT

The authors thank the anonymous BAR reviewers for their time and invaluable feedback to improve this work. This project has been supported by NSF under grants 2243632 and 2145631.

REFERENCES

[1] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 138–157.

[2] N. S. Agency, "Ghidra software reverse engineering framework," https://github.com/NationalSecurityAgency/ghidra.

[3] L. Gao and H. Yin, "Bindsa: Efficient, precise binary-level pointer analysis with context-sensitive heap reconstruction," *Proceedings of the ACM on Software Engineering*, vol. 2, no. ISSTA, pp. 1190–1211, 2025.

[4] C. Ye, Y. Cai, A. Zhou, H. Huang, H. Ling, and C. Zhang, "Manta: Hybrid-sensitive type inference toward type-assisted bug detection for stripped binaries," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, 2024, pp. 170–187.

[5] Z. Lin, J. Li, B. Li, H. Ma, D. Gao, and J. Ma, "Typesqueezer: When static recovery of function signatures for binary executables meets dynamic analysis," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 2725–2739.

[6] Y. Shi, L. Tian, L. Chen, Y. Yang, and G. Shi, "Scheduled execution-based binary indirect call targets refinement," in *European Symposium on Research in Computer Security*. Springer, 2024, pp. 3–23.

[7] Y. Li, M. Wang, C. Zhang, X. Chen, S. Yang, and Y. Liu, "Finding cracks in shields: On the security of control flow integrity mechanisms," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1821–1835.

[8] W. Zhu, Y. Wang, Y. Cui, C. Zhang, and X. Han, "Callee: Recovering call graphs for binaries with transfer and contrastive learning," *IEEE Transactions on Dependable and Secure Computing*, 2025.

[9] R. Sun, Y. Guo, Z. Wang, and Q. Zeng, "Attncall: Refining indirect call targets in binaries with attention," in *European Symposium on Research in Computer Security*. Springer, 2023, pp. 391–409.

[10] P. Wang, M. Santra, M. Liu, C. Sun, D. Zeng, and G. Tan, "Disa: Accurate learning-based static disassembly with attentions," in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, 2025, pp. 843–857.

[11] G. Li, M. Sridharan, and Z. Qian, "Redefining indirect call analysis with kallgraph," in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2025, pp. 2957–2975.

[12] D. Liu, S. Ji, K. Lu, and Q. He, "Improving {Indirect-Call} analysis in {LLVM} with type and {Data-Flow}{Co-Analysis}," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 5895–5912.

[13] K. Lu and H. Hu, "Where does it go? refining indirect-call targets with multi-layer type analysis," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1867–1881.

[14] T. Xia, H. Hu, and D. Wu, "{DEEPTYPE}: Refining indirect call targets with strong multi-layer type analysis," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 5877–5894.

[15] S. H. Kim, C. Sun, D. Zeng, and G. Tan, "Refining indirect call targets at the binary level." in *NDSS*, 2021.

[16] S. H. Kim, D. Zeng, C. Sun, and G. Tan, "Binpointer: towards precise, sound, and scalable binary-level pointer analysis," in *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, 2022, pp. 169–180.

[17] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan, "Rocksalt: better, faster, stronger sfi for the x86," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, 2012, pp. 395–404.

[18] V. Van Der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, "A tough call: Mitigating advanced code-reuse attacks at the binary level," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 934–953.

[19] C. Zhu, Z. Li, A. Xue, A. P. Bajaj, W. Gibbs, Y. Liu, R. Alur, T. Bao, H. Dai, A. Doupé *et al.*, "{TYGR}: Type inference on stripped binaries using graph neural networks," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 4283–4300.

[20] Y. Wang, R. Liang, Y. Li, P. Hu, K. Chen, and B. Zhang, "Typeforge: Synthesizing and selecting best-fit composite data types for stripped binaries," in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2025, pp. 1–18.

[21] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[22] A. Flores-Montoya and E. Schulte, "Datalog disassembly," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1075–1092.

[23] Y. Liu, S. Mechtaev, P. Subotić, and A. Roychoudhury, "Program repair guided by datalog-defined static analysis," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1216–1228.

[24] S. Abeysinghe, A. Xhebraj, and T. Rompf, "Flan: An expressive and efficient datalog compiler for program analysis," *Proceedings of the ACM on Programming Languages*, vol. 8, no. POPL, pp. 2577–2609, 2024.

[25] K. Pei, J. Guan, D. Williams-King, J. Yang, and S. Jana, "Xda: Accurate, robust disassembly with transfer learning," *arXiv preprint arXiv:2010.00770*, 2020.

[26] S. Yu, Y. Qu, X. Hu, and H. Yin, "{DeepDi}: Learning a relational graph convolutional network model on instructions for fast and accurate disassembly," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2709–2725.

[27] D. D. I. F. Committee, "Dwarf version 5," https://dwarfstd.org/dwarf5std.html.

[28] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.