Refining Indirect Call Targets at the Binary Level

Sun Hyoung Kim*, Cong Sun†, Dongrui Zeng*, Gang Tan*
*The Pennsylvania State University
†Xidian University
*{szk450, dxz16, gtan}@psu.edu, †suncong@xidian.edu.cn

***Abstract*—Enforcing fine-grained Control-Flow Integrity (CFI) is critical for increasing software security. However, for commercial off-the-shelf (COTS) binaries, constructing high-precision Control-Flow Graphs (CFGs) is challenging, because there is no source-level information, such as symbols and types, to assist in indirect-branch target inference. The lack of source-level information brings extra challenges to inferring targets for indirect calls compared to other kinds of indirect branches. Points-to analysis could be a promising solution for this problem, but there is no practical points-to analysis framework for inferring indirect call targets at the binary level. Value set analysis (VSA) is the state-of-the-art binary-level points-to analysis but does not scale to large programs. It is also highly conservative by design and thus leads to low-precision CFG construction. In this paper, we present a binary-level points-to analysis framework called BPA to construct sound and high-precision CFGs. It is a new way of performing points-to analysis at the binary level with the focus on resolving indirect call targets. BPA employs several major techniques, including assuming a block memory model and a memory access analysis for partitioning memory into blocks, to achieve a better balance between scalability and precision. In evaluation, we demonstrate that BPA achieves a 34.5% precision improvement rate over the current state-of-the-art technique without introducing false negatives.**

I. Introduction

Control-Flow Integrity (CFI) constrains attackers' ability of control-flow manipulation by enforcing a predetermined control-flow graph (CFG). One key step in CFI is constructing its policy, that is, the CFG it enforces. The major challenge of CFG construction is to infer the targets of indirect control-transfer instructions, especially indirect calls (calls through register or memory operands).

A program can have multiple CFGs, some fine-grained and some coarse-grained. For instance, a coarse-grained CFG may allow an indirect call to target all possible functions in a program, even though at runtime only a subset of functions can be actually called. A coarse-grained CFG can allow an attacker too much freedom in manipulating the control flow within the CFG. Control Jujutsu [15], CFB [12], and TROP [16] have shown the importance of constructing precise CFGs; imprecisely refined indirect call targets are more likely to remain in the attack surface. Therefore, precise yet correct (sound) indirect call refinement has been recognized as a major research track for enforcing CFI. This research is also important to other security applications, such as bug finding. In bug finding, coarse grained CFGs result in imprecise and unscalable analysis, while ignoring indirect calls results in an incomplete analysis and failing to detect hidden bugs. In particular, some bug detection tools [23], [25], [49] stop analysis at indirect calls, constraining their abilities to find bugs through indirect calls.

Realizing the importance of constructing high-precision CFGs for security applications, many systems have been designed with the assumption of having access to source code [1], [15], [16], [19], [27], [29], [33], [43], [45]. Source code provides rich semantic information such as symbols and types, which enable sophisticated static analysis. In the meantime, the demand for CFI is also critical in commercial off-the-shelf (COTS) binaries and legacy code, where source code is not available. The state-of-the-art CFG construction for stripped binaries is a combination of TypeArmor [46] and PathArmor [44], which enforce a backward-context-sensitive arity-based CFG [34]. An arity-based CFG allows an indirect call to target any function whose number of parameters is compatible with the number of arguments supplied at the call. Compared with source-level approaches, the relatively slow research progress for stripped-binary CFG construction is due to the challenges in performing high-precision program analysis at the binary level without the assistance of source-level information.

In this paper, we focus on generating high-precision CFGs for stripped binaries. To resolve targets for an indirect call, we utilize a binary-level static points-to analysis to decide what functions may be pointed to by the code pointer used in the call. Binary-level points-to analysis brings extra challenges compared to source-level analysis, due to the lack of source-level information as well as the complexity of assembly instructions. C-like variables at the binary level are represented by memory accesses usually through registers (indirect memory accesses). To infer what variables are used in the indirect memory accesses of instructions, alias analysis is required, which further complicates points-to analysis. One possible choice for such a points-to analysis is Value-Set Analysis (VSA) [6], which partitions memory into memory regions (e.g., a stack frame for a function) and tracks the set of numeric values in abstract locations (alocs) via the strided-interval abstract domain. For a pointer value, its value set tracks both which memory regions the pointer points to and the *offsets* that the pointer has from the beginnings of the regions. Since those offsets are numbers, VSA has to track the numeric values in storage locations such as registers; e.g., its numeric analysis may decide that `rax` has values 1, 3, 5, etc. On the other hand, the numeric analysis is time consuming and as a result VSA does not scale to large programs, as discussed in

Network and Distributed Systems Security (NDSS) Symposium 2021
21-24 February 2021
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.24386
www.ndss-symposium.org

prior work [52].

Therefore, the key challenge is to design a points-to analysis that is both scalable and sufficiently precise for high-precision CFG construction of large stripped binaries. To address the challenge, we propose a novel block-based points-to analysis called BPA. It assumes a block memory model [28], in which a memory region is divided into disjoint *memory blocks* and a pointer to a memory block cannot be made to point to other blocks via pointer arithmetic. Taking advantage of the block memory model, BPA only tracks what memory blocks a pointer can point to, but not the offsets of the pointer from the beginnings of blocks. In this way, expensive numeric analysis (as in VSA) is avoided in BPA. This is the key for BPA to scale to large programs. Further, our experiments show that the decision of not-tracking offsets still allows BPA to precisely track code addresses in memory blocks, which enables high-precision CFG construction.

Overall, BPA makes the following contributions:

- We propose a novel binary-level block-based points-to analysis (BPA) for stripped binaries. It takes as input a stripped binary, and starts with a memory access analysis to divide memory into memory blocks. BPA then performs the block-based value tracking analysis to perform alias analysis on memory accesses and infer the targets of indirect branches. This analysis is scalable and precise in terms of CFG generation.
- We have built BPA via Datalog, a modular logic programming language that has been used for static analysis at the source-code level in the past [10], [17], [22], [39], [48]. Our system supports binaries compiled by both GCC and Clang as well as different compiler optimization levels, including **-O0** to **-O3**. On a set of real-world benchmarks, BPA achieves 34.5% higher precision on detecting indirect-call targets than the conservative arity matching technique. Our experiments show that, unlike VSA, BPA can scale to large stripped binaries, when given sufficient resources; for example, for 403.gcc from SPEC2k6, BPA can finish its analysis within 10 hours on a machine with around 350GB of RAM.

II. Related work

We discuss related work in CFG generation, binary-level points-to analysis, and Datalog-based static analysis.

CFG generation from source code. Most systems that generate CFGs require source code or source-level information, such as relocation information and debugging information [1], [15], [16], [19], [27], [29], [33], [43], [45], [50]. For example, the original CFI [1] allows an indirect call to target all address-taken functions, which are identified by relocation information. Forward-CFI [43] matches indirect calls and functions by the arity information with the help of the GCC compiler. MCFI [33] and TypeDive [29] generate high-precision CFGs by utilizing source-level type information for the matching. Another track of high-precision CFG generation performs static analysis to infer the targets of indirect branches with the help of source-level information. For example, Kernel CFI [19] applies source-level taint tracking to infer what code addresses can flow to what indirect branches. Control Jujutsu [15] employs a source-level alias analysis (DSA, Data Structure Analysis) to construct the CFG. Another CFG construction system [50] starts with compiler-generated meta information including debugging information, performs a binary-level type inference to infer types of function pointers used in indirect calls, and uses the inferred types to build a CFG. Although this work performs binary-level type inference, it relies on source information such as debugging information.

CFG generation from binary code. CFG construction without source code or source-level information is challenging and its research progress is generally limited. CCFIR [51] analyzes all code addresses in a stripped binary to determine what can be targeted by indirect calls; it is a migration of the original CFI's scheme to binary code, which allows an indirect call to target address-taken functions. TypeArmor [46] performs several binary-level analyses to infer the arity information for both indirect call sites and functions, which is an implementation of Forward-CFI's CFG policy at the binary level.

Binary level points-to analysis. VSA [6] is the most common points-to analysis technique adapted by other binary-analysis platforms such as BAP [11], ANGR [38] and CodeSurfer [5]. VSA tracks the value sets of abstract locations (alocs), which can be registers, CPU flags, and memory locations [6]. However, VSA often does not scale to large programs; a detailed discussion that compares the designs between VSA and BPA will be offered in the overview section. BDA [52] proposes a path sampling algorithm with probabilistic guarantees to perform a scalable binary-level dependency analysis, which only tracks dependent values and has customized tracking rules. While BDA achieves higher precision on dependency analysis than VSA and IDA [21], its path sampling does not cover all paths and cannot guarantee the completeness of the analysis result. Thus, to our best knowledge, there is no practical points-to analysis framework at the binary level for generating precise and sound CFGs.

Datalog static analysis. Points-to analysis frameworks at source level have been implemented in Datalog for Java and C/C++ programs [7], [10], [20], [26], [40], where they have enjoyed benefits of modularity, high-performance, and precise static analysis offered by Datalog. A recent work [18] implemented a binary disassembly and reassembly framework in Datalog, which can be used for providing inputs to our tool. To the best of our knowledge, Datalog has not been used to implement practical binary-level points-to analysis frameworks.

III. Overview

We start with an overview of the block memory model adopted in BPA and then present the workflow of our system.

A. The block memory model

Designing a sound and precise points-to analysis is known to be difficult even with source-level information. Performing points-to analysis on stripped binaries introduces extra challenges. The first challenge is how to model memory to have a good balance between scalability and precision. Modeling the memory as an array of one-byte slots is unscalable. The other

extreme is to model memory as a whole, meaning that reads and writes at different memory addresses are not distinguished. This, however, is highly imprecise.

BPA adopts a block memory model, which is inspired by the CompCert project [28]. It used the block memory model to specify the semantics of C-like languages and verify correctness of program transformations in compilers. We observe that the block memory model can be used for performing scalable binary-level points-to analysis. In this model, memory is divided into a set of disjoint *memory blocks*. Each block is comprised of a logically cohesive set of memory slots. For example, the stack frame for a function that has an integer local variable and a local array can be divided into two blocks: one for the integer and the other for the entire array.

Pointer arithmetic is allowed within one block, but the block model assumes that it is not possible to make a pointer to one block point to a different block via pointer arithmetic. For instance, if p points to block b_1, for any offset o, the result of pointer arithmetic $p + o$ must also point to block b_1, not any other block b_2. Therefore, BPA by design eliminates the consideration of pointer arithmetic during points-to analysis. Conceptually, the block model makes two blocks separate by an infinite amount of space; so adding a finite offset to a pointer cannot make it point to a different block. This infinite-separation view is sound for points-to analyses if blocks are formed properly, for the following reason. Memory blocks should delineate the boundaries of where legal pointer accesses should be; out-of-block accesses imply memory errors (e.g., accessing an array out of bound). Since program behavior after a memory error is undefined, a points-to analysis should capture only those points-to relations in executions without memory errors. Therefore, when performing a points-to analysis, we can safely assume pointer arithmetic does not make a pointer go outside of a block. This assumption of memory-safe executions is also in previous work of formalizing points-to analysis [14], [42].

One key benefit of the block model is that it enables BPA to not track offsets during its points-to analysis. Since adding an offset to a pointer does not change the block the resulting pointer points to, for a pointer BPA tracks only what blocks it might point to, not the offsets the pointer has from the beginnings of blocks. By not tracking offsets, BPA treats a block as a whole, meaning that memory reads and writes through pointers to the block at different offsets are not distinguished. This enables BPA to be much more scalable than previous binary-level points-to analysis such as VSA. Furthermore, this design also accommodates CFG construction with good precision, as in general it matches well with how code addresses are stored and retrieved from memory. As an example, suppose a program stores into a memory block two function pointers: fp_1 stored at $p + o1$ and fp_2 stored at $p + o2$, assuming p points to the block; the program then performs an indirect call using a function pointer retrieved from $p + o3$. BPA's analysis then ignores all those offsets and decides the block has two function pointers (fp_1 and fp_2) inside and the indirect call can target either one of them. This overapproximation yields good results on stripped binaries, as our experiments show.

Comparison with VSA. VSA in theory can achieve high precision for performing points-to analysis but it comes with a large cost. Even the original VSA algorithm [6] has several design choices to trade precision for scalability, such as using a strided-interval abstraction to represent a set of offset values. However, their proposed trade-off actions are not sufficient for generating high-precision CFGs in large real-world applications. One bottleneck is their memory modeling.

VSA partitions memory into memory regions and each region is further partitioned into a set of abstract memory locations (alocs). Unlike the block memory model, in VSA a pointer to one aloc can point to another aloc after pointer arithmetic. This design requires numeric analysis to track offsets so that the analysis can know what alocs a pointer might point to after pointer arithmetic; this incurs a large computational burden. Also, for soundness, the numeric analysis may yield a large value set for a pointer in the worst case, effectively making the pointer point to all possible alocs within a region. Such an over-approximation may result in a large target set for an indirect call and leads to low-precision CFG generation. Prior work [38], [52] also criticized VSA in terms of its scalability and precision.

Deciding block boundaries. The block memory model requires an algorithm for dividing memory into blocks. At the source-code level, it is relatively easy to perform this division. For example, the original block memory model work [28] uses source code information in a C-like language to put each local variable into a separate memory block.

For stripped binaries, it is much more difficult to decide the boundaries of blocks. Stripped binaries do not carry symbol nor type information to aid this process. As a result, BPA has to rely on a separate memory access analysis to recover boundaries. At high level, this memory-access analysis is similar to the process of recovering variable-like entities in IDAPro [21] and alocs in VSA [6]. However, since the block memory model makes the strong assumption about the separation among blocks, BPA's boundary recovery algorithm has to be more conservative, relying on its own set of rules and semantic-based heuristics for block boundary discovery. Our experiments demonstrate that the blocks generated by this algorithm obey the pointer-arithmetic assumption.

B. System overview

BPA is designed for resolving indirect-branch targets in stripped binaries by a block-based points-to analysis. It takes as input a stripped binary, and performs a disassembly to produce a CFG that contains targets for direct branches, but not indirect branches. The targets of indirect branches are then determined by BPA's points-to analysis. Fig. 1 describes the workflow of BPA, which consists of three major components: input processing, memory-block generation, and value-tracking analysis.

Input processing. This component generates facts encoded in Datalog to represent the input disassembly. It requires the input disassembly being represented in a Direct Control-Flow Graph (DCFG), which includes edges for direct branches but not indirect branches. This component first translates the input's assembly instructions into an intermediate representation. Then, it identifies function boundaries. Based on function

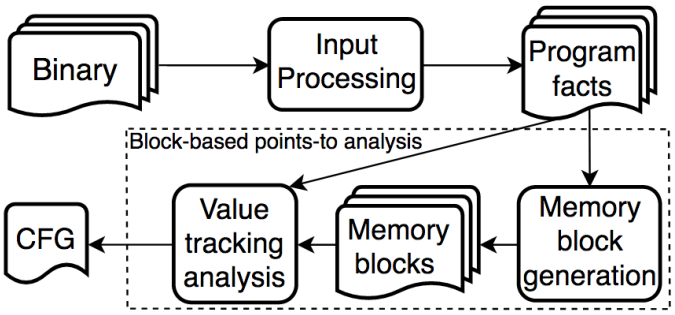

Fig. 1: System flow.

boundaries, functions whose addresses are taken are identified, which can be considered candidates of indirect call targets.

Memory-block generation. This component performs a memory-access analysis to generate memory blocks from the DCFG in all memory regions, including data sections, function stack frames, and the heap. It computes block boundaries to partition each memory region. For different memory regions, we partition them at different granularity levels, for balancing between precision and scalability.

Block-based value tracking analysis. This component takes as input the DCFG, the address-taken functions, and the generated memory blocks to add new control-flow edges discovered for indirect branches into the CFG. In detail, this component tracks values over abstract locations (registers and memory blocks) according to the input CFG and determines the targets of indirect branches. First, it transforms the program into a memory-block access intermediate representation. Then, it resolves the targets of indirect branches by a fixed point computation that merges SSA transformation, value tracking analysis, and CFG update that terminates when the CFG does not change anymore. Since the total number of indirect-branch targets is finite, there exists a fixed point to guarantee termination. At the end, the final CFG is generated.

IV. Input Processing: Datalog Fact Generation

To perform our block-based points-to analysis, we disassemble the input binary and process the disassembly results to generate Datalog facts that represent the program and necessary information for the later components in BPA. We require the disassembly being represented by a DCFG, a CFG whose nodes are basic blocks of assembly instructions and whose edges are for the targets of direct branches. During input processing, BPA first converts the disassembled code into an intermediate representation, which is then encoded into Datalog facts. Afterwards, BPA identifies function boundaries and performs DCFG refinements.

A. Background: Datalog

Datalog is a declarative logic programming language and has lately found applications in static analysis. It is a subset of Prolog that guarantees program termination. A Datalog program consists of a set of logical rules in the form of "c :- $a_1, ..., a_n$", where c is the conclusion and a_1 to a_n are assumptions; that is, if assumptions a_1 to a_n all hold, then conclusion c holds. Both the conclusion and assumptions are expressed with *predicates* that describe relations. Consider the following Datalog example about graph reachability.

```
path(x, y) :- edge(x, y).
path(x, z) :- edge(x, y), path(y, z).
```

By the first rule, if there is an edge from x to y, then from x we can reach y. By the second recursive rule, if there exists a node y so that there is a edge from x to y and y can reach z, then x can reach z. A datalog rule without assumptions is called a *fact*. A Datalog execution engine then uses the rules and the facts to derive tuples that are true. For the above example, if the facts are edge(1, 2) and edge(2, 3), then the engine generates path(1, 2), path(2, 3), and path(1, 3).

$$
\begin{array}{rcl}
v & \in & \text{Integers} \\
sz & \in & \text{Integers} \\
reg & := & \text{EAX} \mid \text{EBX} \mid \ldots \\
flag & := & \text{CF} \mid \text{ZF} \mid \ldots \\
loc & := & \text{PC} \mid reg \mid flag \mid \ldots \\
bvop & := & \text{add} \mid \text{and} \mid \text{shl} \mid \ldots \\
cmp & := & \text{lt} \mid \text{eq} \mid \text{gt} \mid \ldots \\
exp & := & \text{bitvec}(sz,\ v) \mid \text{arith}(bvop,\ exp,\ exp) \\
 & \mid & \text{test}(cmp,\ exp,\ exp) \\
 & \mid & \text{ite}(exp,\ exp,\ exp) \\
 & \mid & \text{load\_loc}(loc) \mid \text{load\_mem}(exp) \\
instr & := & loc = exp \mid \text{Mem}[exp] = exp \\
 & \mid & \text{IF } exp \text{ DO } instr
\end{array}
$$

Fig. 2: The major syntax of RTL [32].

B. Program representation with RTL

BPA converts assembly instructions into an intermediate language called RTL [32]. The RTL language is a RISC-like language, with a small set of instructions. An assembly instruction is translated to a sequence of RTL instructions.

Fig. 2 presents the major syntax of the RTL language. Locations *loc* represent storage in the CPU, including the program counter, registers (*reg*), CPU flags (*flag*), and others. Expressions are pure and produce values when evaluated. They include bit-vectors of certain sizes (bitvec), common bit-vector operations (arith), comparison between two expressions (test), conditional expressions (ite, if-then-else), and loads from locations and memory. Instructions have side effects. Instruction "$loc = exp$" updates location loc with the value of expression exp; instruction "$\text{Mem}[exp_1] = exp_2$" updates memory at address exp_1 with the value of exp_2. There is also a conditional instruction. However, BPA's analysis is path insensitive and ignores the conditions in conditional instructions. The input DCFG, after its instructions are translated into RTL, is encoded into Datalog facts. Most of the encoding is straightforward and explained in Appendix B; we only note that the encoding is compact in that an expression is always assigned the same ID even if the expression appears multiple times (e.g., in different instructions).

C. Identifying function boundaries

BPA's interprocedural points-to analysis requires function boundary information, which is not present in stripped binaries. To identify function boundaries, the main idea is to find the start addresses of functions from the targets of direct calls and code addresses used in the program. These code addresses contain all targets of indirect calls/jumps. We perform heuristic based static analysis, similar to [51] and [46], to classify code addresses into function addresses and local addresses within functions. The function addresses are candidate targets of indirect calls and the local addresses within functions are candidate targets of indirect jumps. Then, BPA partitions the code region into a set of functions based on function addresses. Note that this function boundary identification does not rely on any metadata, such as symbol tables and debugging information. In general, it can be replaced by any function boundary recovery tool, such as [3], [8], [35], [37].

With function boundaries identified, the binary is scanned to identify a set of functions whose start addresses appear as constants in the binary's code/data sections. Those functions are determined as Address-Taken (AT) functions and they are potential targets of indirect calls.

D. DCFG refinement

After function boundaries are identified, BPA performs a couple of refinements on the CFG, to make the following steps easier to proceed. The first refinement is to reverse the effect of tail-call optimization, which is often performed by an optimizing compiler. A tail call in a function is a call that is the last action of the function. In a tail-call optimization, the compiler turns a tail call into a jump instruction. BPA classifies the following jumps as tail-call jumps: (1) direct jumps targeting function start addresses, and (2) indirect jumps that are not classified as table-jumps. BPA then replaces each detected tail-call jump to a call instruction followed immediately by a return instruction. Reversing the effect of tail-call optimization is important for interprocedural analysis, which needs to know what edges are interprocedural edges, and is also important for detecting functions that do not return, discussed next.

The second refinement is to add an intraprocedural edge from a call instruction to its follow-up basic block. In static analysis, such intraprocedural edges are added for the convenience of analysis (e.g., summary information for the call can be associated with these edges). However, we do not add such edges for calls to non-returning functions. BPA treats `exit`, `abort`, and `assert_fail` as non-returning functions. Further, if a function does not have any reachable return instruction in its intraprocedural CFG, it is also treated as a non-returning function.

V. Memory Block Generation

BPA adopts a memory-access analysis to generate memory blocks for value tracking analysis. As previously discussed, the granularity of memory blocks is the key for achieving a good balance between scalability and precision. In addition, for soundness all pointer arithmetic should stay within blocks. Achieving this is in general challenging for stripped binaries. As we will discuss when presenting experiments in Sec. VII, the memory blocks generated by BPA obey the pointer-arithmetic assumption and allow good scalability. At a high level, the data memory has three kinds of disjoint memory regions: stack frames, the heap (for dynamically allocated data), and the global data regions. For each memory region, BPA analyzes the memory accesses in the input program and partitions it into memory blocks. We next discuss this process in detail.

```
1    ; {esp=top}
2  push ebp
3    ; {esp=top-4}
4  mov  ebp, esp
5    ; {ebp=top-4, esp=top-4}
6  mov  [ebp-4], $1000
7  mov  eax, [ebp+8]
8  mov  [ebp-8], eax
9  lea  eax, [ebp-8]
10 push eax
11 call <store_by_pieces_2>
12 ...
```

Fig. 3: An example for partitioning the stack.

A. Stack partitioning

For a function, BPA analyzes its code to partition its stack frame into a set of memory blocks. This is similar to how other systems (e.g., [2], [6], [21]) recover variable-like entities on the stack, except that BPA's partitioning must be coarse grained to uphold the pointer-arithmetic assumption of the block memory model. Therefore, it proceeds in two steps. The first step is about gathering a set of boundary candidates through a stack layout analysis. In the second step, it removes candidates that may split a compound data structure, such as an array or a struct. The remaining candidates are used to partition the stack frame into memory blocks.

Collecting boundary candidates. For a function, BPA analyzes its code to determine what stack addresses instructions might use and collects those stack addresses as boundary candidates. The process is similar to how variable-like entities are recovered in previous systems [2], [21]; so we will discuss it only briefly with an example. At a high level, it is an intraprocedural analysis that infers the relationship between registers and the esp value at the function entry (stack top). At every point in the function, it computes a set of equations in the following form: $\{r_1 = \text{top} + c_1, ..., r_n = \text{top} + c_n\}$, meaning that r_1 equals the entry esp value (top) plus constant c_1, ..., and r_n equals top plus constant c_n. If a register does not point to the stack, then no equation for the register is included in the above set. Fig. 3 provides a simple example and the stack layout inference result for the first two instructions. At the entrance, esp points to the stack top; after "push ebp", esp gets to be top-4; after "mov ebp, esp", ebp also gets to be top-4. For the rest of the code, the equations for ebp and esp do not change and are not shown in the figure.

With the result of stack layout analysis, BPA then goes through all instructions, extracts stack addresses from instruction operands, and uses them as boundary candidates.

For the example in Fig. 3, instructions from line 6 to 9 all involve stack addresses. For example, instruction 6 accesses the address $\mathrm{top}-8$, since ebp is $\mathrm{top}-4$. Collectively, instructions 6 to 9 produce the following boundary candidates: $\{\mathrm{top}-8, \mathrm{top}+4, \mathrm{top}-12\}$.

Removing boundary candidates. The boundary candidates collected in the previous step may be in the middle of compound data structures, such as an array or a struct, due to compiler optimizations. Those candidates would be fine for partitioning the stack frame into alocs in VSA, as it does not rely on the assumption that pointer arithmetic stays within alocs; it tracks offsets of pointers and can determine whether pointer arithmetic makes the resulting pointers point to different alocs. However, if BPA partitioned the stack frame with a boundary candidate that is in the middle of a compound structure, pointer arithmetic would cross the boundary of generated blocks, breaking the assumption of the block model. Therefore, such boundary candidates have to be removed.

BPA adopts the following set of heuristics for deciding what boundary candidates from the first step should be kept: (1) $\mathrm{top}+0$ is always a boundary as the return address is stored from top to $\mathrm{top}+3$ in x86; (2) $\mathrm{top}+c$, where c is positive, is a boundary as in most calling conventions of x86 such stack slots are used to pass parameters; (3) $\mathrm{top}+c$, where c is negative, is kept only if it is stored into a general-purpose register or a memory location. Category (3) addresses are kept because programs often store the base address of a compound structure into a general-purpose register or a memory location, before accessing the components in that structure; addresses extracted in this way approximate the base addresses of compound structures. We note that IDAPro [21] also uses this observation to perform type recovery (although it considers only movement into registers, not memory).

For the example in Fig. 3, only instruction at line 9 is moving a stack address to a register; therefore, only $\mathrm{top}-12$ is in category (3). This example was adapted from `403.gcc` in SPEC2k6. By inspecting the source code, we find that instructions 6 to 8 are accessing different fields of the same struct and $\mathrm{top}-12$ is the base address of the struct.

At last, the remaining boundary candidates are used to partition the stack frame. Note that our stack partitioning is conservative in a few ways. First, the stack layout analysis in the step that collects boundary candidates may not discover all possible stack accesses in instructions; e.g., it is intraprocedural and tracks data flow of stack addresses through registers, but not memory. Second, the heuristics in the step for removing boundary candidates may fail to discover boundaries of some compound data structures; as a result, two compound data structures may be put into the same memory block. Being conservative harms the precision and scalability but not soundness, as it leads to a more coarse-grained partitioning; coarse-grained partitioning has less chance to violate the pointer-arithmetic assumption of the block memory model.

B. Global memory region partitioning

BPA partitions global memory regions (i.e., the .DATA, .RODATA, and .BSS data sections) by analyzing related memory accesses. Methodology-wise, it is also a two-step process; it discovers block boundary candidates in step 1; in step 2,

```
1  struct discard_rule {
2    void (*condition)();
3    int flag;
4  } rules[] = {
5    {fptr1, 0}, {fptr2, 1}, {fptr3, -1}
6  };
7  void discard_moves(...){
8    int i, user_f;
9    for (i = 0; i < 3; i++){
10     rules[i].condition();
11     user_f = rules[i].flag;
12   }
13   rules[2].condition();
14   gtp(rules);
15 }
16 void gtp(struct discard_rule rules[]){
17   int j = rules[1].flag;
18 }
```

Fig. 4: Source code for global region partitioning example.

```
1  <discard_moves>:
2  mov  ebp, esp
3  mov  [ebp-4], 0      ; i=0
4  mov  [ebp-8], 0
5  Loop:
6    mov  edx, [ebp-4]   ; get i
7    mov  eax, [$1000+(edx*8)]
8    call eax ; call rules[i].condition
9    mov  ecx, [$1004+(edx*8)]
10   mov  [ebp-8], ecx
11   add  [ebp-4], 1
12   cmp  [ebp-4], 1
13   jl   Loop
14 mov  eax, [$1016]
15 call eax   ; call rules[2].condition
16 push $1000 ; argument passing
17 call <gtp>
18
19 <gtp>:
20 mov  ebp, esp
21 mov  eax, [ebp+8] ; load from parameter
22 mov  edx, [eax+12] ; load rules[1].flag
23 mov  [ebp-4], edx ; store to variable j
```

Fig. 5: Assembly for global region partitioning example.

candidates that are in the middle of compound structures are removed. However, since programs access global regions using patterns different from those accessing the stack, different global-memory partitioning techniques are needed.

Collecting boundary candidates. In step 1, block boundary candidates are collected as follows: (i) extract all base addresses ($addr$) from memory-accessing instructions with indexed operands, in the form of "$addr + r$" (constant address plus a register), or "$addr + r * scale$" (constant address plus a register multiplied by a scale factor such as four), or "$addr + r + r' * scale$" (constant address plus a register and another register multiplied by a scale factor such as four); (ii) in addition, we extract constant addresses stored into registers

and memory in the program; we call addresses extracted in (i) and (ii) *pointer-arithmetic-base* (PAB) addresses. Then all PAB addresses that fall into the address ranges of global memory regions are regarded as boundary candidates; the intuition is that such an address is likely to be the start address of a compound data structure. This process is similar to how IDAPro decides on the boundary of alocs in global regions.

We next present an example adapted from `445.gobmk` in SPEC2k6 to illustrate step 1. For easy understanding, we present source code (Fig. 4) and assembly code (Fig. 5) of the example (even though BPA works on binaries). The code has an array of three structs; each has a function pointer and an integer flag. The `discard_moves` function iterates through the array, calls each function pointer, and accesses the flag. Then it calls the function pointer at index 2 of the array; finally, it calls `gtp` with argument `rules`, and the flag at index 1 from `rules` is stored into `j`. In correspondence, the assembly code uses pointer arithmetic to access function pointers and flags in the array, starting from global addresses 1000 and 1004, respectively. Also, the constant global address 1000 is stored into an argument. As a result, the step 1 would collect these two addresses as boundary candidates. Note that 1016 is not collected as a candidate even if in instruction 14 (Fig. 5) there is a memory access using that constant address, because it is accessed without any pointer arithmetic and the address is not stored into a register or memory.

However, the boundary candidates produced in step 1 might still be in the middle of compound structures. This may happen due to compiler optimizations. In the example, the compiler decides to use different base addresses to access the two fields in the struct. If we used those two base addresses (1000 and 1004) in step 1 as block boundaries, it would partition the array into two blocks: b_1 with the first function pointer (`fptr1`), and b_2 for the rest of the array. If such blocks were used in determining the targets of the indirect call at instruction 8 (Fig. 5), BPA's later steps would decide that the indirect-call targets are read from block b_1. The reason is that instruction 7 loads the target for the indirect call through an operand with the base address 1000 and the address is associated with block b_1. Thus, with the assumption that pointer arithmetic stays within blocks, BPA would determine that only `fptr1` can be the target of the indirect call, since `fptr2` and `fptr3` would be in a different block. This would clearly be unsound.

Removing boundary candidates. In step 2, BPA filters out addresses that may split a compound data structure. The idea is to estimate for each boundary candidate an address range that starts with the boundary candidate and that should fall into the same data structure. Any boundary candidate that is strictly within another candidate's estimated range should be eliminated, because such a candidate must be in the middle of a compound data structure. Essentially, the range estimation is a data-flow analysis to estimate a set of possible values that can be computed by pointer arithmetic from each boundary candidate.

If the boundary candidate is a category (i) PAB address (see earlier discussion on collecting boundary candidates), the analysis estimates the possible values of the index register used in an instruction with the PAB address. For the example, it decides that the index register edx in instructions 7 and 9 have at least values of 0 and 1 since they are in a loop. As a result, instruction 7 accesses at least the range of $[1000, 1007]$ and instruction 9 accesses at least the range of $[1004, 1011]$. Because 1004 is in the middle of the first range, it is eliminated as a block boundary. For the example, the end result is that only 1000 is used as a block boundary, treating the whole array as a single block. When the boundary candidate is a category (ii) PAB address, BPA applies data-flow analysis to track pointer arithmetic on the PAB address to related memory accesses, and uses the result to estimate the range. For the example in Fig. 5, starting with 1000 in instruction 16, BPA performs data-flow analysis on registers and discovered stack memory locations from Sec. V-A to compute where the global addresses are stored to. After knowing that eax in instruction 22 points to 1000, BPA generates a filtering range for the data structure to be $[1000, 1015]$; note that even though the instruction accesses only $[1012, 1015]$, BPA treats it as part of a bigger data structure that starts from the base address 1000. By this address range, 1004 is again selected to be removed from the boundary candidates. Our experimental results in Sec. VII-C show that such a design does not introduce false negatives.

With block boundaries determined by remaining PAB addresses, BPA further splits blocks into *memory chunks*. The motivation of introducing memory chunks is based on the fact that there are many read-only constants in the global region and programs often use constant addresses to directly access these read-only constants. To increase analysis precision, BPA uses these constant addresses to partition blocks into chunks. Reading from a chunk would return values only from the chunk, when the surrounding block is read-only. For example, the precision improvement can be seen from instructions 14-15 of Fig. 5; it reads from a memory chunk, and thereby the indirect call at instruction 15 only calls fptr3, which increases precision compared to reading from the entire block that contains the chunk. Note that pointer arithmetic can access different chunks within a block; so memory chunks do not carry the pointer-arithmetic assumption. This makes value set tracking more complicated; as we will discuss in Sec. VI-C, when a memory block gets updated, the analysis needs to update the values of all memory chunks inside the block.

C. Heap partitioning

We partition the heap using the allocation-site approach, used in many static analysis (e.g., [39]) for analyzing the heap. In particular, all memory allocated at a particular allocation site (i.e., a call to memory allocation functions such as *malloc*, *calloc*, and *realloc*) are put into one memory block. Each such block is identified with a unique allocation site ID, determined by the address of the call instruction to a memory allocation function that allocates the block. This means we assume the binary is dynamically linked and stripped. Further dividing heap memory blocks could increase precision, but would come at the cost of analysis overhead.

VI. Block-based value tracking analysis

This section discusses BPA's core component, the block-based value tracking analysis, which takes as input (1) a DCFG with RTL instructions, (2) memory blocks generated in the previous step, and (3) the address-taken functions. It

outputs a final CFG with resolved targets for indirect branches. The value tracking analysis consists of two stages. First, it transforms memory accesses into memory-block accesses, resulting in a memory-block access intermediate representation (MBA-IR). The second stage contains three mutually recursive processes: an SSA transformation, a flow-insensitive value tracking analysis, and a process for discovering indirect branch targets to add to the CFG. Since SSA transformation and value tracking analysis both depend on the CFG, the second stage is a mutually recursive process and computes a fixed point until no indirect branch targets can be further added to the CFG. The second stage is guaranteed to terminate because edges can only be added and the set of possible indirect branch targets is finite. Value tracking analysis utilizes MBA-IR to both compute values for abstract locations and perform memory alias analysis for indirect memory accesses.

A. Memory-block access transformation

BPA translates the RTL program into a memory-block access intermediate representation (MBA-IR). This translation improves the runtime efficiency of the next step, value tracking analysis, in two ways. First, memory accesses in the RTL program are converted into accesses to memory blocks/chunks, avoiding repeating the process that determines the corresponding memory blocks/chunks for memory accesses. Second, since BPA's value tracking analysis tracks only part of the program state, the translation also abstracts away unnecessary RTL instructions and expressions. For example, since BPA avoids path sensitivity for scalability, it does not track path conditions. As a result, instructions, expressions, and RTL locations that are related to path conditions are translated away. Moreover, constants that are not function start addresses or memory block/chunk start addresses are abstracted away, for the reason that an internal constant address of a block cannot be used to compute the start address of a different block, by the pointer-arithmetic assumption of the block memory model.

MBA-IR's syntax is defined in Fig. 6. MBA-IR has five types of instructions:

(1) register-update instructions, $reg \leftarrow exp$;
(2) memory-location-update instructions, $mloc \leftarrow exp$;
(3) indirect memory-update instructions, $*reg \leftarrow exp$;
(4) nondeterministic memory-update instructions, $(mloc \lor *reg) \leftarrow exp$;
(5) and no-ops, SKIP.

A register-update instruction computes the value of exp and stores it into reg. Note that we also treat the program counter as a reg. A memory-location-update instruction has a similar effect except that the destination is a memory block/chunk; recall that BPA uses memory chunks to increase the analysis precision on global data regions. An indirect memory-update instruction ($*reg \leftarrow exp$) stores the computed value of exp to a memory location pointed to by reg. A nondeterministic memory-update instruction directly updates a memory location or indirectly updates a memory location through a register; such nondeterminism is necessary to achieve soundness of the MBA-IR translation. Note that a nondeterministic memory-update instruction cannot directly update more than one memory block, which will be explained during the description of translation rules. An expression, exp, can be (1) a register,

$$
\begin{array}{rcl}
func & \in & \text{Functions} \\
gblk & \in & \text{Global memory blocks} \\
sblk & \in & \text{Stack memory blocks} \\
hblk & \in & \text{Heap memory blocks} \\
gchk & \in & \text{Global memory chunks} \\
reg & \in & \text{Registers and program counter} \\
mloc & := & mblk \mid sblk \mid hblk \mid gchk \\
exp & := & reg \mid mloc \mid *reg \mid \&mloc \mid \&func \\
 & \mid & exp \lor exp \\
ins & := & reg \leftarrow exp \mid mloc \leftarrow exp \mid *reg \leftarrow exp \\
 & \mid & (mloc \lor *reg) \leftarrow exp \mid \text{SKIP}
\end{array}
$$

Fig. 6: The syntax of the MBA-IR.

which produces the value stored in the register when evaluated, (2) a memory location, which produces the value stored in the memory location when evaluated, (3) a dereference operation on a register ($*reg$), which loads the value stored in the memory location pointed to by reg, (3) the memory address of a memory location ($\&mloc$), (4) a function start address ($\&func$), or (5) a nondeterministic choose expression ($exp \lor exp$) to represent a nondeterministic expression evaluation for achieving translation soundness.

We next briefly explain the high-level intuition of translating from RTL to MBA-IR and leave details and a formalization to Appendix A. The core is to translate address expressions used in RTL memory instructions into MBA-IR memory locations (i.e., memory blocks and chunks). This translation is written as $\mathrm{TransAddr(a)}$ and proceeds by pattern matching address a. BPA's implementation of this pattern matching is specialized to RTL code translated from x86 instructions and assumes there are four cases of x86 address-mode operands: (1) c, (2) $c + reg$, (3) $c + reg * sc$, and (4) $c + reg + reg' * sc$. We next discuss only the case of $c + reg$ and leave the rest of the discussion to the appendix. When translating $c + reg$, we consider two possibilities: (1) c is used as the base address of a memory block and reg is used as the offset to the block; (2) reg is used as the base address and c as the offset. Therefore, the translation translates $c + reg$ to a nondeterministic choice "$mloc \lor *reg$", assuming c is mapped to $mloc$ (either a global memory block or a global memory chunk) during global memory partitioning. The actual memory locations associated with the second choice $*reg$ will be known during value tracking analysis. Note that $*reg$ may yield multiple memory locations. For example, a register may point to either a stack block or a heap block, depending on which path the program is taking. Also, as we motivated at the beginning, since BPA's value tracking analysis is path insensitive, the translation ignores instructions, expressions, and RTL locations that are related to path conditions. Thus, the translation removes $\mathrm{test}(cmp,\ exp,\ exp)$, $\mathrm{ite}(exp,\ exp,\ exp)$, and RTL locations other than registers and the program counter from the CFG to reduce analysis cost. Then, the remaining RTL instructions are transformed into aloc-update instructions.

B. SSA transformation

In this step, BPA performs SSA (Single Static Assignment) transformation on the program so that every variable is defined

only once. It is easier to analyze programs in the SSA form. For example, performing a flow-insensitive analysis on the program after SSA is equivalent to a flow-sensitive analysis on the program before SSA. Since every variable is defined only once, SSA transformation essentially renames all variables in a program by adding indices to variable names based on the CFG. As a result, the control flow information is recorded by the indexing system. During SSA transformation, ϕ-instructions are introduced into the program to represent the merge points of multiple control flows. For example, in the following simple C code,

if(x == 1) x = x + 1; **else** x = x + 2;

the variable x after the execution of the if-else statement may hold either the value of x in the then branch or the value of x in the else branch. In SSA transformation, an index is introduced for every variable and a ϕ-instruction is introduced after the statement. In detail, the same program in SSA form is as follows:

if(x_0 == 1) x_1 = x_0 + 1; **else** x_2 = x_0 + 2;
x_3 = ϕ (x_1, x_2);

The ϕ-instruction selects either x_1 or x_2, depending on whether the control flow reaches the instruction through the then branch or the else branch.

BPA performs SSA on the MBA-IR, where it treats only the registers as variables. We design our SSA transformation in Datalog by adapting an efficient SSA algorithm [4]. Here, for brevity, we do not elaborate on the theoretical details of the original algorithm but only summarize our adaption. The key step of our approach is to first perform the SSA transformation for each function according to the intraprocedural edges in the input CFG. Then, we utilize the interprocedural control-flow information to connect intraprocedural SSA forms into a complete SSA form for the whole program. When constructing ϕ-instructions, BPA applies only the first phase (called the really crude phase) to incrementally add them instead of starting from scratch when new edges are detected. This kind of incremental analysis is significantly more scalable [30], [36]. As a result, our SSA transformation achieves better scalability compared with the original algorithm [4] while maintaining the correctness.

C. Value tracking analysis

After transforming MBA-IR code into its SSA form, BPA performs an interprocedural value tracking analysis. The analysis is implemented by Datalog logic rules on instructions in the MBA-IR SSA form. A Datalog engine then scans through all instructions to perform the value tracking analysis according to the rules. To ease the explanation of our full value tracking rules, we first show the formalization of a basic value tracking analysis that does not consider global memory chunks as memory locations. After that, we will show how to incorporate global memory chunks and an instruction reachability detection module, which improves analysis precision.

Basic value tracking analysis.

To formalize the basic value tracking analysis, we design rules for the five types of MBA-IR instructions. In the rules, we use the notation $ireg$ to represent an indexed register (i.e., a register with an index) after the SSA transformation. For readability we present Datalog rules in the inference rule notation, which puts the conclusion below a horizontal bar and the assumptions above the bar.

TABLE I: Definition of $\mathrm{VSet}(-)$.

$\mathrm{VSet}(ireg) \coloneqq \{v \mid \mathrm{AlocVal}(ireg, v)\}$
$\mathrm{VSet}(mloc) \coloneqq \{v \mid \mathrm{AlocVal}(mloc, v)\}$
$\mathrm{VSet}(*ireg) \coloneqq$
$\quad \{v \mid \mathrm{AlocVal}(ireg, mloc) \ \wedge \ \mathrm{AlocVal}(mloc, v)\}$
$\mathrm{VSet}(\&mloc) \coloneqq \{\&mloc\}$
$\mathrm{VSet}(\&func) \coloneqq \{\&func\}$
$\mathrm{VSet}(exp_1 \ \vee \ exp_2) \coloneqq \mathrm{VSet}(exp_1) \cup \mathrm{VSet}(exp_2)$

The rules for the register-update instructions and nondeterministic memory-update instructions are presented in Fig. 7. Further, for brevity, we do not list the formal rules for memory-location-update instructions and indirect memory-update instructions, because they can be adapted from rules for register-update and nondeterministic memory-update instructions, which we will explain; and SKIP is omitted since it has no influence on the value tracking.

We use the predicate $\mathrm{AlocVal}(aloc, val)$ to record the value set information for *abstract locations*; an abstract location is defined to be either an indexed register or a memory location. Also, only start addresses of functions and memory blocks/chunks are tracked. For example, $\mathrm{AlocVal}(\mathrm{EAX}_1, 1000)$ means that register EAX with index 1 holds value 1000. To shorten the rules, we introduce $\mathrm{VSet}(-)$ in Table I and use the notation $val \in \mathrm{VSet}(exp)$ to mean that val belongs to the value set of exp. We use $\mathrm{InCFG}(ins)$ for the predicate that determines the existence of an instruction in the current CFG. Thanks to SSA, BPA's value tracking analysis is flow insensitive. The input program is viewed as a collection of instructions; knowing if an instruction is in the CFG is sufficient for the analysis.

Rules AddrMloc and AddrFunc are the base cases and initialize all abstract locations with the values that are tracked. Rules Ireg and Mloc capture the value flow to an indexed register from another indexed register or a memory location by making the destination indexed register have the same set of values as the source abstract location. Rule DIreg is about the dereference operation via an indexed register. The semantics of the dereference is to first retrieve the memory blocks stored in the indexed register and then transfer the values of the retrieved memory blocks into the destination indexed register. Therefore, in the rule, $\mathrm{AlocVal}(ireg', \&mloc)$ tells that the $ireg'$ stores a memory location $\&mloc$; $\mathrm{AlocVal}(mloc, val)$ states that the memory location $mloc$ holds value val. As a result, the destination $ireg$ should also hold val. Rules AltMloc and AltFunc are for a register-update instruction with a nondeterministic choose expression. Essentially, if one of the operands holds a memory location $mloc$ or a function address $func$, as a result, the destination indexed register also holds the same memory location or function address. Rule ToAlt is for a nondeterministic memory-update instruction. The rule finds a possible destination and makes it to hold the value val, which is evaluated from the source expression. Rule Phi deals with ϕ-instructions introduced by

$$\textsc{AddrMloc}\ \frac{\text{InCFG}(ireg \leftarrow \&mloc)}{\text{AlocVal}(ireg, \&mloc)}$$

$$\textsc{AddrFunc}\ \frac{\text{InCFG}(ireg \leftarrow \&func)}{\text{AlocVal}(ireg, \&func)}$$

$$\textsc{Ireg}\ \frac{\text{InCFG}(ireg \leftarrow ireg') \qquad \text{AlocVal}(ireg', val)}{\text{AlocVal}(ireg, val)}$$

$$\textsc{Mloc}\ \frac{\text{InCFG}(ireg \leftarrow mloc) \qquad \text{AlocVal}(mloc, val)}{\text{AlocVal}(ireg, val)}$$

$$\textsc{DIreg}\ \frac{\begin{array}{c}\text{InCFG}(ireg \leftarrow *ireg') \\ \text{AlocVal}(ireg', \&mloc) \qquad \text{AlocVal}(mloc, val)\end{array}}{\text{AlocVal}(ireg, val)}$$

$$\textsc{AltMloc}\ \frac{\begin{array}{c}\text{InCFG}(ireg \leftarrow exp_1 \ \vee \ exp_2) \\ \&mloc \in \text{VSet}(exp_1) \ \vee \ \&mloc \in \text{VSet}(exp_2)\end{array}}{\text{AlocVal}(ireg, \&mloc)}$$

$$\textsc{AltFunc}\ \frac{\begin{array}{c}\text{InCFG}(ireg \leftarrow exp_1 \ \vee \ exp_2) \\ \&func \in \text{VSet}(exp_1) \ \vee \ \&func \in \text{VSet}(exp_2)\end{array}}{\text{AlocVal}(ireg, \&func)}$$

$$\textsc{ToAlt}\ \frac{\begin{array}{c}\text{InCFG}(mloc' \ \vee \ *ireg \leftarrow exp) \qquad val \in \text{VSet}(exp) \\ mloc = mloc' \ \vee \ \text{AlocVal}(ireg, \&mloc)\end{array}}{\text{AlocVal}(mloc, val)}$$

$$\textsc{UpdMloc}\ \frac{\begin{array}{c}\text{InCFG}(*ireg \leftarrow exp) \qquad val \in \text{VSet}(exp) \\ \text{AlocVal}(ireg, \&mloc)\end{array}}{\text{AlocVal}(mloc, val)}$$

$$\textsc{Phi}\ \frac{\begin{array}{c}\text{InCFG}(ireg \leftarrow \phi(ireg_1, \ldots, ireg_n)) \\ \bigvee_{i=1,\ldots,n} \text{AlocVal}(ireg_i, val)\end{array}}{\text{AlocVal}(ireg, val)}$$

Fig. 7: Representative rules of the basic value tracking analysis.

SSA transformation. The idea is similar to rule ALTMLOC; if one of the n operands of the ϕ-instruction holds value val, the destination $ireg$ should also hold val. The notation $\bigvee_{i=1,\ldots,n}$ represents a sequence of logical or operations on n predicates. Next, we show how to adapt the rules of register-update instructions to construct rules for the memory-location-update instructions and indirect-mloc-update instructions. In detail, for memory-location-update instructions, we can simply substitute all $ireg$ with $mloc$ in each rule's conclusion and also in the $\text{InCFG}(ins)$ assumption. For indirect memory-update instructions, we need to add $\text{AlocVal}(ireg', \&mloc)$ as a new assumption to each rule in Fig. 7 and replace $ireg$ in the rule's conclusion with $mloc$. Note that the interprocedural ϕ-instructions capture the data flows between functions for return values as well as function parameters; so our analysis is interprocedural.

Supporting global memory chunks. To consider global memory chunks as memory locations, a new assumption needs to be added to every rule in Fig. 7. For brevity, we use DIREG as a representative to show how the rules can be adapted. The new rule is shown in Fig. 8. First, how a memory-location load is treated is unchanged. That is, it should still load the values of the target memory location, no matter whether it is a memory block or a global memory chunk. Second, rules for memory location updates should now consider more locations to update; since no pointer offsets are tracked, an update to a memory block should also update all the memory chunks inside, to overapproximate the update effect. Furthermore, an update to a memory chunk should also change the memory block that holds the chunk, since the analysis needs to track all possible values in the block. Therefore, we introduce an aloc-equivalence relation to determine if two alocs are equivalent. The formal definition of aloc equivalence is shown as follows:

$$\textsc{DIreg}\ \frac{\begin{array}{c}\text{InCFG}(aloc' \leftarrow *ireg) \qquad \text{AlocVal}(ireg, \&mloc) \\ aloc' \equiv aloc \qquad \text{AlocVal}(mloc, val)\end{array}}{\text{AlocVal}(aloc, val)}$$

Fig. 8: New rule for $*ireg$ to support memory chunks.

Definition (Aloc Equivalence)**.** We write $aloc_1 = aloc_2$ if the two alocs are the same. An abstract location $aloc_1$ is covered by another abstract location $aloc_2$, written as $aloc_1 \sqsubset aloc_2$, if $aloc_1$ is a global memory chunk, $aloc_2$ is a global memory block, $aloc_1$ is inside $aloc_2$. Then, two allocs are equivalent, written as $aloc_1 \equiv aloc_2$, iff $aloc_1 = aloc_2 \ \vee \ aloc_1 \sqsubset aloc_2 \ \vee \ aloc_2 \sqsubset aloc_1$

As we argued, every aloc update instruction should update all equivalent alocs of its original target aloc. Therefore, we simply add the aloc-equivalence relation into the rule as a condition; in this way, memory chunk accesses are supported.

Instruction reachability detection. An optimization of BPA is to perform analysis only over reachable instructions. Given a CFG of a binary, BPA computes instructions that are reachable in the CFG from the entry point of the program. BPA analyzes only reachable instructions and resolves the targets of only reachable indirect branches. As a result, the target set of an unreachable indirect branch (in dead code) is empty, which increases BPA's analysis precision. To enable this feature, $\text{InCFG}(ins)$ in the rules is replaced with $\text{Reachable}(ins)$, which determines the reachability of an instruction in the CFG.

D. Discovering indirect branch targets

BPA relies on the value tracking result to add targets for indirect branches. In this section, we explain how BPA resolves targets for each kind of indirect branch instructions.

Indirect call. The operand of an indirect call in MBA-IR is an aloc. Thus, BPA checks the inferred value set of the aloc operand by its value tracking analysis. All address-taken functions in the value set are treated as targets of the indirect call; BPA uses address taken functions to filter the value set for the targets of indirect calls. One could use other strategies such as arity-matching or type-matching to further improve precision. We will discuss that BPA and arity-matching can be combined to achieve even higher precision of indirect-call targets resolution in Sec. VII.

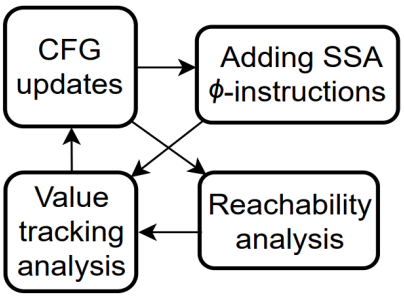

Fig. 9: Workflow of fixed point recursion for updating a CFG. The components are recursively performed until no more indirect branch targets are found.

Indirect jumps and return instructions. In BPA's input processing component, it performs a heuristic-based indirect-jump target detection to identify the function boundaries and refine the DCFG. Although this detection mechanism resolves jump targets that use jump tables, it does not resolve the targets of indirect tail-call jumps. Since the tail-call optimization is supported by BPA, we keep a record for all indirect calls that are transformed from indirect tail-call jumps during DCFG refinement (Sec. IV-D) and use this record to convert such calls back to indirect jumps. These tail-call indirect jumps then take the targets of the corresponding indirect calls in the refined DCFG. For return instructions, BPA constructs a call graph according to the CFG and uses that call graph to resolve the targets of return instructions.

Fixed point recursion. Intuitively, the CFG construction, which focuses on resolving indirect branch instructions' targets, and value-tracking analysis should be mutually recursive: the initial CFG contains only targets of direct branches. Value-tracking analysis then incrementally discovers indirect branch targets and adds those edges to the CFG; the addition of new CFG edges triggers the addition of new $\mathrm{Reachable}(ins)$ facts, which in turn triggers more computation of value-tracking analysis. This process should continue until reaching a fixed point, where a final CFG is generated. To realize the above intuition, after the input DCFG RTL program is converted into MBA-IR, BPA's design makes the rest of the components mutually recursive: (1) SSA transformation depends on the current CFG; (2) value tracking analysis relies on SSA transformation and uses predicates such as $\mathrm{Reachable}(ins)$, which depends on the CFG; (3) after new indirect call targets are discovered, they are added to the CFG, which triggers incremental SSA to add new interprocedural ϕ-instructions or add more operands to existing ϕ-instructions; (4) the new ϕ-instructions enable the value tracking analysis to discover new values in alocs; (5) those new values enable the discovery of new targets of indirect calls. The workflow of this mutually recursive process is shown in Fig. 9. The beauty of Datalog is that it allows all these components to be separately specified as logic rules; the Datalog engine then performs a fixed point calculation to compute the final outcome in an iterative fashion, without programmer involvement.

VII. Evaluation

Our evaluation aims to answer a few major questions: (1) how much does BPA improve over previous techniques in terms of precision? (2) Are the CFGs generated by BPA sound? An unsound CFG (with missing edges) would prevent the execution of legitimate control-flow transfers. (3) how efficient is BPA in CFG generation?

Intuitively, indirect call precision reflects the effectiveness on refining indirect call targets, which is the key to CFI's security guarantee. In general, more precise CFGs (with less spurious indirect-branch edges) provide less freedom for attackers to discover gadgets for exploit generation. Most previous CFI papers rely on graph-based metrics, such as AIR (average indirect-target reduction) and AICT (average indirect-call targets) for evaluations. We follow this tradition to compute AICT for evaluating the precision of BPA's CFGs. In addition, another way of evaluating precision we adopt is to use profiling to collect the actual targets of indirect calls under a set of test cases and treat that as the pseudo ground truth. Based on that, we can approximate precision rates for indirect calls. Furthermore, the targets of indirect calls collected during profiling are used to validate the soundness of CFGs, by checking whether all targets during profiling are actually included in BPA's CFGs. We note that during our evaluation we focus on indirect calls, since BPA uses similar heuristic-based algorithms as others for resolving indirect jumps' targets and return targets are resolved by call graph construction.

Benchmarks. We applied BPA on SPEC2k6's C benchmarks, except for `429.mcf`, `470.lbm`, and `462.libquantum`, which do not have any indirect calls in the compiled binaries. For further evaluation, we also collected five security-critical applications: `thttpd-2.29`, `memcached-1.5.4`, `lighttpd-1.4.48`, `exim-4.89`, and `nginx-1.10`. To prepare the input disassembly, we used RockSalt [32] to disassemble x86 ELF binaries, compiled by GCC-9.2 and Clang-9.0 with optimization levels **-O0**, **-O1**, **-O2**, and **-O3**. We then ran BPA on binaries by different compilers (GCC and Clang) and different optimization levels and collected statistics for all settings. Since similar results and trends are observed from different settings, for brevity, the following sections provide only data for binaries compiled by GCC-9.2.

A. Recovering arity information

As discussed earlier, Address-Taken (AT) functions are considered as possible indirect-call targets in BPA. Indirect call sites are expected to target at only these functions. Besides AT functions, the arity information can be utilized to further refine the set of indirect-call targets.

TypeArmor [46] recovers arity information from a stripped binary to perform arity matching to resolve indirect-call targets. To our best knowledge, this is the state-of-the-art technique that statically detects indirect-call targets for stripped binaries. To compare with BPA and also check whether combining arity matching with BPA would improve precision, we implemented the conservative matching mechanism proposed by TypeArmor [46]. With the recovered arity information, we construct an arity-based CFG for each binary. Further, we refine BPA's output CFG by removing caller-callee pairs that

do not satisfy the conservative arity-matching criterion; we call it the *hybrid* strategy.

Principles for Arity. We explain the conservative matching mechanism of TypeArmor. First, targets of indirect calls can only be AT functions. Second, an indirect call site that prepares n arguments is allowed to target functions that expect n or fewer parameters. Third, among all functions that satisfy the first two criteria, an indirect call site that expects a return value should only target the ones that do provide return values. These principles are designed to be conservative to prevent false negatives when identifying indirect call targets.

Implementing arity-based CFGs. TypeArmor is designed for x64 binaries, which has a different calling convention from x86. Thus, we designed our own arity information recovery algorithm for x86 stripped binaries to facilitate the conservative arity-matching based CFG construction.

Our design is similar to TypeArmor's approach except for the argument count analysis due to a different calling convention for x86, where call-site arguments and function parameters are passed through the stack. We utilize our stack layout analysis discussed in Sec. V-A to infer the number of expected parameters of each AT function. In detail, we find the maximum positive stack memory offset that has been used to read from a stack location in each AT function and divide this offset by four to infer the maximum number of parameters. We can also compute the number of arguments for each indirect call site by leveraging our stack layout analysis. The intuition is that all arguments are pushed onto the stack only after entering the basic block where the indirect call instruction resides; meanwhile, all arguments should stay in a consecutive range of memory. Thus, to infer the number of arguments, we first inspect all memory write instructions that write to the stack within the call site's basic block and record the accessed stack offsets, based on which we find the largest consecutive memory range that starts with the top offset of the stack. Then, we divide the size of the memory range by four to calculate the number of arguments. Our experiments show that this algorithm is compatible with different compilers such as GCC and Clang.

TABLE II: AICT evaluation results (for GCC 9.2).

Program	Opt Level	Instrs	I-Calls	AICT			
				AT	Arity	BPA	Hybrid
401.bzip2	O0	21K	20	2.0	1.0	2.0	1.0
	O1	11K	20	2.0	1.0	2.0	1.0
	O2	11K	20	2.0	1.0	2.0	1.0
	O3	15K	20	2.0	1.0	2.0	1.0
458.sjeng	O0	32K	1	7.0	7.0	7.0	7.0
	O1	22K	1	7.0	7.0	7.0	7.0
	O2	22K	1	7.0	7.0	7.0	7.0
	O3	32K	1	7.0	7.0	7.0	7.0
433.milc	O0	31K	4	2.0	2.0	2.0	2.0
	O1	22K	4	2.0	2.0	2.0	2.0
	O2	23K	4	2.0	2.0	2.0	2.0
	O3	33K	4	1.0	1.0	1.0	1.0
482.sphinx3	O0	45K	8	6.0	2.3	1.3	1.3
	O1	33K	8	6.0	2.4	1.3	1.3
	O2	35K	7	6.0	1.9	0.7	0.7
	O3	39K	7	6.0	1.9	0.7	0.7
456.hmmer	O0	88K	9	22.0	22.0	2.9	2.9
	O1	58K	11	22.0	22.0	4.3	4.3
	O2	60K	10	22.0	22.0	2.8	2.8
	O3	69K	9	14.0	14.0	1.0	1.0
464.h264ref	O0	161K	369	39.0	30.6	4.3	3.3
	O1	100K	353	39.0	28.7	4.1	3.2
	O2	100K	352	39.0	28.9	26.4	17.3
	O3	164K	355	39.0	28.5	18.0	15.6
445.gobmk	O0	213K	44	1790.0	1395.7	884.6	846.3
	O1	154K	44	1786.0	1392.0	1334.8	1191.5
	O2	157K	44	1788.0	1413.3	1297.2	1198.7
	O3	189K	44	1785.0	1460.3	1376.5	1270.1
400.perlbench	O0	306K	139	721.0	580.4	400.3	328.2
	O1	221K	139	721.0	560.9	364.2	282.2
	O2	226K	110	721.0	536.6	363.7	261.8
	O3	273K	237	718.0	523.7	453.4	322.1
403.gcc	O0	969K	459	1211.0	650.0	534.8	323.5
	O1	652K	473	1207.0	566.6	491.0	250.6
	O2	647K	450	1208.0	581.3	427.8	209.8
	O3	763K	727	1198.0	518.6	544.3	247.7
thttpd	O0	18K	1	17.0	17.0	8.0	8.0
	O1	13K	1	17.0	17.0	8.0	8.0
	O2	13K	1	17.0	17.0	14.0	14.0
	O3	14K	1	17.0	17.0	14.0	14.0
memcached	O0	37K	75	24.0	20.8	1.0	1.0
	O1	25K	75	24.0	21.1	1.3	1.3
	O2	26K	72	25.0	21.6	1.4	1.3
	O3	29K	78	24.0	20.5	0.7	0.7
lighttpd	O0	66K	126	52.0	27.8	35.5	17.1
	O1	46K	126	52.0	26.4	35.5	16.5
	O2	48K	109	52.0	24.7	33.9	14.6
	O3	54K	120	52.0	24.3	34.2	14.6
exim	O0	168K	89	85.0	40.4	31.1	17.3
	O1	135K	89	85.0	41.4	29.6	16.2
	O2	139K	106	85.0	38.0	30.6	17.6
	O3	155K	181	85.0	42.1	40.6	23.0
nginx	O0	232K	409	753.0	441.6	444.0	253.8
	O1	151K	409	753.0	422.2	463.4	251.9
	O2	153K	331	753.0	420.5	525.1	274.6
	O3	164K	365	754.0	432.4	511.0	273.8

B. AICT comparison

We evaluate different techniques' CFG precision by measuring the average indirect call target (AICT). Table II shows the AICT statistics of different techniques on the benchmarks, for all optimization levels. It also lists the number of x86 instructions and the number of indirect call instructions in the binaries. In the group of AICT columns, AT is a system that allows an indirect call to target all address-taken functions [51]; Arity is the conservative arity-matching based CFG construction, which is our implementation of TypeArmor [46] for x86; BPA is our system; Hybrid resolves indirect-call targets by a combination of BPA and Arity.

As BPA does not consider unreachable indirect calls, some indirect call sites do not have any targets in their results. Thus, some AICT numbers by BPA or the Hybrid approach are less than 1.0. Also, for `456.hmmer` and `memcached`, BPA produces significantly better results than Arity, and the results are at the same level or even better than those produced by a source-type-based approach [33], [50] based on our manual inspection. Through these AICT numbers, we compare precision improvement for four situations: (1) from AT to Arity (2) from AT to BPA, (3) AT to Hybrid, and (4) Arity to Hybrid. We choose to conduct these four comparisons to show BPA's abilities on improving precision over the state-of-the-art techniques. For each comparison, we compute the geometric mean of a precision improvement rate by each benchmark listed in Table II.

From (1) to (4), the geometric mean of AICT reduction rates are 20.1%, 37.0%, 62.7% and 34.5% respectively, considering all the benchmarks with all optimization levels in Table II, excluding `458.sjeng` and `433.milc` where all techniques precisely resolve the targets. These rates show that BPA achieves higher AICT reduction rate (precision increase rate) than Arity when comparing the 20.1% precision increase rate of AT-to-Arity and the 37.0% precision increase rate of AT-

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 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

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

```
1  static int (*qcmp)();
2  int hit_comparison(_, _) {...}
3  void FullSortTophits(_ {
4   specqsort(_, _, _, hit_comparison);
5  }
6  void specqsort(_, _, _, int (*compar)())
7  { qcmp = compar;
8    if ((*qcmp)(j, lo) > 0) ...
9  }
```

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

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

the pointer-arithmetic assumption of the block memory model. On the other hand, the profiling based soundness validation showed that BPA-generated CFGs achieved a 100% recall rate for SPEC2k6 benchmarks. The result showed that all runtime indirect call targets were included in BPA-generated CFGs, including 403.gcc, which is notoriously complicated. Further, if we are willing to sacrifice the ability to handle stripped binaries, we can use compiler-generated meta information such as symbol tables to get sound block boundaries.

BPA is a new way of performing points-to analysis at the binary level with the purpose of finding indirect-call targets on stripped binaries. Although we changed traditional points-to analysis to not track offsets, we believe the block memory model can be applicable to traditional points-to analysis. For example, one can track offsets in a block-based points-to analysis based on the pre-computed boundaries to avoid over-approximation of alias analysis on memory accesses. As discussed earlier, our manual inspection suggests that better heap modeling would further enhance the analysis precision. Supporting flow sensitivity on memory blocks and context sensitivity on function calls are also possible. Although scalability may be decreased, such improvements on the block memory model would make points-to analysis more precise.

BPA's block memory model enables scalable analysis of the *legal behavior* of binaries and is geared toward applications such as CFI and DFI [13]. In these applications, we first analyze the legal behavior of a target program to extract an integrity property (e.g., data-flow integrity), and then enforce the property through runtime monitoring. In addition, CFGs constructed by BPA are also useful for any binary-level static analysis, as a CFG is a prerequisite for any such analysis (e.g., data flow analysis, binary debloating, etc.). There are applications such as malware analysis and exploit generation that require analyzing the *illegal behavior* of binaries; e.g., analyze what happens after a buffer overflow. The block memory model, as it currently stands, is not a good fit for analyzing illegal behavior.

BPA's current prototype utilizes RockSalt [32] to convert binary code into an RTL IR. RockSalt supports only x86 binaries. Recently, we implemented a component for converting the BAP [11] IR to RTL IR for x64 binaries; as future work, we plan to integrate that component with BPA to handle x64 binaries. Another note is that source code is less likely to be available for legacy x86 binaries, which highlights the importance of analyzing x86 binaries.

BPA's current design and implementation target binaries compiled from C programs. To our best knowledge, there is no practical points-to analysis for stripped binaries generated from C++ code. We believe BPA's block memory model still applies to C++ generated binaries, but supporting them may require a substantial amount of both conceptual and engineering efforts, particularly in tracking virtual tables (vtables) and object references. For vtables, we believe our global region partitioning method can be extended to generate memory blocks for vtables; one observation to support our belief is that for safety concerns vtables are encoded in the read-only data section, and pointers to vtables are stored at the beginning of the object's memory region. Further, our value tracking analysis needs to be extended to track the assignment of vtable pointers into heap memory blocks generated at allocation sites (i.e., call sites of the `new` function), and to track how object references are created and propagated. We leave the C++ support as interesting future work.

IX. Conclusions

High-precision CFG generation is the key to improving CFI's security strength. However, performing that for stripped binaries is still lacking. Hence, we propose a block-based points-to analysis (BPA) to construct high-precision CFGs on stripped binaries. It assumes a block memory model and designs algorithms to partition memory regions so that BPA can achieve a balance among soundness, precision, and scalability. BPA performs a block-based value tracking analysis, the core of BPA, which relies on the block memory model to perform a fixed point computation to resolve targets of indirect branches. We formalize the block-based value tracking as inference rules and implement the analysis in Datalog, which allows us to conveniently switch between different design choices of memory-region partitioning. Our experiment shows that BPA can substantially improve the CFG precision compared with previous approaches, with acceptable runtime efficiency, and without introducing false negatives.

Acknowledgment

The authors would like to thank anonymous reviewers for their insightful comments. This work was supported by ONR research grant N00014-17-1-2539.

References

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity," in *12th ACM Conference on Computer and Communications Security (CCS)*, 2005, pp. 340–353.

[2] N. S. Agency, "Ghidra reverse engineering tool," https://www.nsa.gov/resources/everyone/ghidra/, 2017.

[3] J. Alves-Foss and J. Song, "Function boundary detection in stripped binaries," in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, pp. 84–96.

[4] J. Aycock and N. Horspool, "Simple generation of static single-assignment form," in *International Conference on Compiler Construction (CC)*. Springer, 2000, pp. 110–125.

[5] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum, "Codesurfer/x86—a platform for analyzing x86 executables," in *International Conference on Compiler Construction (CC)*. Springer, 2005, pp. 250–254.

[6] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *International Conference on Compiler Construction (CC)*, 2004, pp. 5–23.

[7] G. Balatsouras and Y. Smaragdakis, "Structure-sensitive points-to analysis for C and C++," in *International Static Analysis Symposium*, 2016, pp. 84–104.

[8] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "BYTEWEIGHT: learning to recognize functions in binary code," in *23rd Usenix Security Symposium*, 2014, pp. 845–860.

[9] A. R. Bernat and B. P. Miller, "Anywhere, any-time binary instrumentation," in *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on program analysis for software tools*, 2011, pp. 9–16.

[10] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2009, pp. 243–262.

[11] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: a binary analysis platform," in *Computer Aided Verification (CAV)*, 2011, pp. 463–469.

[12] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *24th Usenix Security Symposium*, 2015, pp. 161–176.

[13] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006, pp. 147–160.

[14] C. L. Conway, D. Dams, K. S. Namjoshi, and C. Barrett, "Pointer analysis, conditional soundness, and proving the absence of errors," in *15th International Symposium on Static Analysis*, 2008, pp. 62–77.

[15] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *22nd ACM Conference on Computer and Communications Security (CCS)*, 2015, pp. 901–913.

[16] R. M. Farkhani, S. Jafari, S. Arshad, W. Robertson, E. Kirda, and H. Okhravi, "On the effectiveness of type-based control flow integrity," in *Annual Computer Security Applications Conference*, 2018, pp. 28–39.

[17] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of android malware through static analysis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 576–587.

[18] A. Flores-Montoya and E. Schulte, "Datalog disassembly," in *29th Usenix Security Symposium*, 2020.

[19] X. Ge, N. Talele, M. Payer, and T. Jaeger, "Fine-grained control-flow integrity for kernel software," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2016, pp. 179–194.

[20] N. Grech and Y. Smaragdakis, "P/taint: unified points-to and taint analysis," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–28, 2017.

[21] Hex-Rays, "The IDA Pro disassembler and debugger," https://www.hex-rays.com/products/ida/, 2008.

[22] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 1036–1046.

[23] S. Jana, Y. J. Kang, S. Roth, and B. Ray, "Automatically detecting error handling bugs using error specifications," in *25th Usenix Security Symposium*, 2016, pp. 345–362.

[24] H. Jordan, B. Scholz, and P. Subotić, "Soufflé: On synthesis of program analyzers," in *International Conference on Computer Aided Verification*. Springer, 2016, pp. 422–430.

[25] Y. Kang, B. Ray, and S. Jana, "Apex: Automated inference of error specifications for c apis," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 472–482.

[26] G. Kastrinis and Y. Smaragdakis, "Hybrid context-sensitivity for points-to analysis," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 423–434, 2013.

[27] M. R. Khandaker, W. Liu, A. Naser, Z. Wang, and J. Yang, "Origin-sensitive control flow integrity," in *28th Usenix Security Symposium*, 2019, pp. 195–211.

[28] X. Leroy and S. Blazy, "Formal verification of a C-like memory model and its uses for verifying program transformations," *Journal of Autom. Reasoning*, vol. 41, no. 1, pp. 1–31, 2008.

[29] K. Lu and H. Hu, "Where does it go? refining indirect-call targets with multi-layer type analysis," in *26th ACM Conference on Computer and Communications Security (CCS)*, 2019, pp. 1867–1881.

[30] Y. Lu, L. Shang, X. Xie, and J. Xue, "An incremental points-to analysis with CFL-reachability," in *International Conference on Compiler Construction (CC)*, 2013, pp. 61–81.

[31] 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," in *Programming Language Design and Implementation (PLDI)*, 2005, pp. 190–200.

[32] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan, "Rocksalt: Better, faster, stronger SFI for the x86," in *Programming Language Design and Implementation (PLDI)*, 2012, pp. 395–404.

[33] B. Niu and G. Tan, "Modular control-flow integrity," in *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2014, pp. 577–587.

[34] W. Qiang, Y. Huang, D. Zou, H. Jin, S. Wang, and G. Sun, "Fully context-sensitive cfi for cots binaries," in *Australasian Conference on Information Security and Privacy*. Springer, 2017, pp. 435–442.

[35] R. Qiao and R. Sekar, "Function interface analysis: A principled approach for function recognition in cots binaries," in *International Conference on Dependable Systems and Networks (DSN)*, 2017, pp. 201–212.

[36] D. Saha and C. Ramakrishnan, "Incremental and demand-driven points-to analysis using logic programming," in *International conference on Principles and practice of declarative programming*, 2005, pp. 117–128.

[37] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *24th Usenix Security Symposium*, 2015, pp. 611–626.

[38] 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 *IEEE Symposium on Security and Privacy (S&P)*, 2016, pp. 138–157.

[39] Y. Smaragdakis and G. Balatsouras, "Pointer analysis," *Foundations and Trends in Programming Languages*, vol. 2, no. 1, pp. 1–69, Apr. 2015.

[40] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras, "Introspective analysis: context-sensitivity, across the board," in *Programming Language Design and Implementation (PLDI)*, 2014, pp. 485–495.

[41] Y. Sui and J. Xue, "On-demand strong update analysis via value-flow refinement," in *International Conference on Software engineering (ICSE)*, 2016, pp. 460–473.

[42] G. Tan and T. Jaeger, "CFG construction soundness in control-flow integrity," in *ACM SIGSAC Workshop on Programming Languages and Analysis for Security (PLAS)*, 2017, pp. 3–13.

[43] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in GCC & LLVM," in *23rd Usenix Security Symposium*, 2014.

[44] V. Van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical context-sensitive CFI," in *22nd ACM Conference on Computer and Communications Security (CCS)*, 2015.

[45] V. van der Veen, D. Andriesse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrdia, "The dynamics of innocent flesh on the bone: Code reuse ten years later," in *24th ACM Conference on Computer and Communications Security (CCS)*, 2017, pp. 1675–1689.

[46] 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 *IEEE Symposium on Security and Privacy (S&P)*, 2016, pp. 934–953.

[47] S. Wang, P. Wang, and D. Wu, "Uroboros: Instrumenting stripped binaries with static reassembling," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 236–247.

[48] J. Whaley, D. Avots, M. Carbin, and M. S. Lam, "Using Datalog with binary decision diagrams for program analysis," 2005, pp. 97–118.

[49] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim, "Precise and scalable detection of double-fetch bugs in OS kernels," in *IEEE Symposium on Security and Privacy (S&P)*, 2018, pp. 661–678.

[50] D. Zeng and G. Tan, "From debugging-information based binary-level type inference to CFG generation," in *8th ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2018, pp. 366–376.

[51] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *22nd Usenix Security Symposium*, 2013, pp. 337–352.

[52] Z. Zhang, W. You, G. Tao, G. Wei, Y. Kwon, and X. Zhang, "BDA: practical dependence analysis for binary executables by unbiased whole-program path sampling and per-path abstract interpretation," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–31, 2019.

APPENDIX A
FORMALIZATION OF MBA-IR TRANSFORMATION

Before we explain the rules, we first discuss notation. We represent the results of memory block generation as four functions: gblk(c), gchk(c), sblk(f, c), and hblk(c). In particular,

gblk(c) returns the global memory block for global memory address c; gchk(c) returns the global memory chunk for global memory address c; sblk(f, c) returns the stack memory block for function f at its stack frame offset c; and hblk(c) returns the heap memory block for a `malloc`-family function call site c. In addition, we use func(c) for mapping the start address c of function *func* to &*func*; recall that Sec. IV-C discusses how function boundaries are identified.

Table VI formalizes the translation. We write TransAddr(a) for translating address expressions; it is designed according to address a's pattern. In total, there are four patterns, which correspond to the four cases of x86 address-mode operands: (1) c, (2) $c + reg$, (3) $c + reg * sc$, and (4) $c + reg + reg' * sc$. The first pattern is a constant c, which may represent a global memory address, a function address, or a value that is not tracked. So the translation relies on the result of memory block generation to either map it to a global memory location or a unique function ID, or treat it as an unnecessary value. When translating $c + reg$, we consider two possibilities: (1) c is used as the base address of a memory block and reg is used as the offset to the block; (2) reg is used as the base address and c as the offset. Therefore, the translation translates $c + reg$ to a nondeterministic choice "$mloc \ \vee \ *reg$", assuming c is mapped to $mloc$ (either a global memory block or a global memory chunk) during global memory partitioning. The translation for constant c is deterministic because one constant address can be the start address of only one memory block/chunk. The actual memory locations associated with the second choice $*reg$ will be known during value tracking analysis. Note that $*reg$ may yield multiple memory locations. For example, a register may point to either a stack block or a heap block, depending on which path the program is taking. For the third pattern, $c + reg * sc$, it shares the same translation rule as pattern c, since the part of $reg * sc$ in the pattern is assumed to stay within the memory block associated with the constant c, according to the block memory model. Thus, it recursively applies TransAddr(c). The same reasoning applies to pattern $c + reg + reg' * sc$; it applies TransAddr(c + reg).

With TransAddr(a) defined, the rest of translation is straightforward. We next discuss TransIns($-$). The location-update instruction $reg = e$ is translated to $reg \leftarrow$ TransExp(e). During expression translation, it generates a "None" value for a constant that is not tracked. Thus, if TransExp(e) = None, it converts the instruction into a SKIP instruction. Finally, a memory-update instruction $\text{Mem}[a] = e$ is translated to TransAddr(a) $\leftarrow$ TransExp(e). As an optimization, during expression translation, values that are not tracked are removed; an RTL instruction that uses only non-tracked values is considered unnecessary and removed from the CFG. At last, for an if-do instruction, IF e DO ins, the translation ignores the condition e and recursively applies TransIns($-$) on instruction ins.

TransExp(e) translates RTL expression e into an MBA-IR expression. A location-load expression is directly converted to the corresponding register in MBA-IR. For a memory-load expression, load_mem(a), it applies the address expression translation (TransAddr(a)) to map it to a memory location or a dereference operation; an RTL bit-vector is treated in the same way as an address expression. For arithmetic expressions,

TABLE VI: MBA-IR transformation rules.

Transformation rules for RTL address expressions:
TransAddr(c) :=
if c is defined in gblk then gblk(c)
elif c is defined in func then func(c)
else None
TransAddr(c + reg) :=
if TransAddr(c) = None then $*reg$
else TransAddr(c) $\vee$ $*reg$
TransAddr(c + reg * sc) := TransAddr(c)
TransAddr(c + reg + reg' * sc) := TransAddr(c + reg)
Transformation rules for RTL instructions:
TransIns(reg = e) :=
if TransExp(e) = None then SKIP
else $reg \leftarrow$ TransExp(e)
TransIns(Mem[a] = e) :=
if TransExp(e) = None then SKIP
else TransAddr(a) $\leftarrow$ TransExp(e)
TransIns(IF e DO ins) := TransIns(ins)
Transformation rules for RTL expressions:
TransExp(load_loc(reg)) := reg
TransExp(load_mem(a)) := TransAddr(a)
TransExp(arith(_, e_1, e_2)) :=
if TransExp(e_1) = None then TransExp(e_2)
elif TransExp(e_2) = None then TransExp(e_1)
else TransExp(e_1) $\vee$ TransExp(e_2)
TransExp(bitvec(_, c)) :=
if TransAddr(c) = None then None
else &TransAddr(c)

arith($bvop$, exp, exp), it ignores the operator and conservatively and recursively translates each operands into values we track and use the nondeterministic choose expression to connect them. Such a translation is valid because we only track base addresses for functions and memory blocks. Either operand could be a base address; and the operator is ignored to be conservative.

APPENDIX B
RTL TO DATALOG

We describe the major predicates for encoding RTL instructions and expressions in Datalog. In total, we have 6 predicates to represent 2 instruction types and 4 expression types:

- **set_loc_rtl**($addr$, $order$, loc_eid, src_eid) is designed for the location-update instructions, which modifies location loc_eid with the value of expression src_eid; the instruction is at address $addr$ and with order $order$.
- **set_mem_rtl**($addr$, $order$, mem_eid, src_eid) is designed for the memory-update instructions, which modifies memory at address mem_eid with the value of expression src_eid; the instruction is at address $addr$ and with order $order$.
- **arith_rtl_exp**(eid, $bvec$, exp_l, exp_r) is designed for arithmetic expressions; the expression ID is eid, of which the value is computed by a bit-vector operation $bvec$ between expressions exp_l and exp_r.
- **get_mem_rtl_exp**(eid, mem_exp) is designed for memory-load expressions; the ID is eid, and its value

is computed by getting the memory content at address mem_exp.

- **get_loc_rtl_exp**(eid, loc) is designed for location-load expressions; the ID is eid, and the value is computed by getting the value from location loc.
- **imm_rtl_exp**(eid, int) is designed for bit-vector expressions; the ID is eid, and the value is int.

Essentially, RTL instructions and expressions that are not represented by predicates are unnecessary for BPA due to our design choice of avoiding path sensitivity for better scalability, which is detailed in Sec. VI-A.

The major complexity lies in the encoding of RTL expressions, which can be nested. For example, in "$e_1 + e_2$" (abbreviation for $\mathrm{arith}(+,\ e_1,\ e_2)$), both e_1 and e_2 are expressions and have their own structures. Such kinds of recursive structures cannot be directly encoded in Datalog, which lacks support for inductive datatypes as other languages do. The main idea of our encoding is to give unique IDs for all subexpressions and use those IDs to encode expressions one level at a time. For $e_1 + e_2$, the encoding gives some eid to the entire expression, some eid_1 to e_1, and some eid_2 to e_2; then a fact "$arith\_rtl\_exp(eid, +, eid_1, eid_2)$" is generated to encode equation "$eid = eid_1 + eid_2$"; e_1 and e_2 are encoded similarly, in a recursive way. Another way of interpreting this process is that it treats an expression as a tree structure, gives an ID to every node in the tree, and adds facts that relate nodes to their child nodes. During the encoding, BPA also detects duplicate subexpressions and reuses IDs for them. For $e_1 + e_2$, if e_1 and e_2 are structurally equivalent, the same ID is used for both e_1 and e_2. This duplicate detection happens for all expressions in a program and as a result turns the encoding of trees into DAGs. RTL instructions are straightforward to encode as there is no nesting. Since an assembly instruction is translated into a sequence of RTL instructions, the encoding for each RTL instruction also includes information about the address of the assembly instruction and the order of the RTL instruction in the sequence. Consider the translation example below:

```
Assembly:
  100: mov edx, [ebp-8]
RTL:
  100: edx = *(ebp-8)
Datalog facts:
  set_loc_rtl(100, 1, e1, e2)
  get_loc_rtl_exp(e1, "edx")
  get_mem_rtl_exp(e2, e3)
  arith_rtl_exp(e3, "-", e4, e5)
  get_loc_rtl_exp(e4, "ebp")
  imm_rtl_exp(e5, 8)
```

It shows an example of how an assembly instruction is translated into RTL instructions, and how corresponding Datalog facts are generated. In this example, the assembly instruction is translated to a single RTL instruction. This instruction loads from memory through an indirect address of `[ebp-8]` and stores the loaded value to `edx`. In the Datalog facts, predicate `set_loc_rtl` represents an RTL instruction that loads a value from expression with ID `e2` and stores it to the register with expression ID `e1`, which represents `edx`. Note how "`*(ebp-8)`" is represented through multiple datalog facts: one dereference at the top level, one subtraction operation at the next level, and more for `ebp` and 8.