BinPointer: Towards Precise, Sound, and Scalable Binary-Level Pointer Analysis

Sun Hyoung Kim
Pennsylvania State University
USA
szk450@psu.edu

Dongrui Zeng
Pennsylvania State University
USA
dxz16@psu.edu

Cong Sun
Xidian University
China
suncong@xidian.edu.cn

Gang Tan
Pennsylvania State University
USA
gtan@psu.edu

Abstract

Binary-level pointer analysis is critical to binary-level applications such as reverse engineering and binary debloating. In this paper, we propose BinPointer, a new binary-level interprocedural pointer analysis that relies on an offset-sensitive value-tracking analysis to achieve high precision. We also propose a soundness and precision evaluation methodology based on runtime memory accesses triggered by reference input data. Our experimental results demonstrate that BinPointer has higher precision over prior work, while maintaining acceptable scalability. The soundness of BinPointer is also validated through runtime data.

CCS Concepts: • **Security and privacy → Software reverse engineering**.

Keywords: static analysis, pointer analysis, binary analysis

ACM Reference Format:
Sun Hyoung Kim, Dongrui Zeng, Cong Sun, and Gang Tan. 2022. BinPointer: Towards Precise, Sound, and Scalable Binary-Level Pointer Analysis. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction (CC '22), April 02–03, 2022, Seoul, South Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3497776.3517776

1 Introduction

The demand for analyzing binary programs has been increasing. Even when source code is available, analyzing binary programs directly has the benefit of not trusting compiler passes such as complicated compiler optimizations. Precise and practical binary-level pointer analysis, is critical in static binary analysis, yet it is challenging due to the existence of pointer aliases. Lacking source-level information such as types, it is challenging for binary-level pointer analysis to achieve high precision, soundness, and efficiency.

Value set analysis (VSA [5]) is the first general-purpose binary-level pointer analysis. It discovers abstract locations during pointer analysis and employs the abstract domain of *strided intervals* to represent the value set of each abstract location. However, recent work [14, 19, 28, 29] criticized VSA for being too conservative and interprocedural VSA were unable to scale to large binaries. For example, Shoshitaishvili et al. [19] discussed that the original VSA design [5] did not perform well on real world binaries; Zhang et al. [29] criticized that VSA often overapproximated a memory access to point to the entire memory space of a program.

Therefore, practical applications of VSA either limit the scope of analysis to be intraprocedural or customize the memory modeling and value tracking analysis based on application domain knowledge to scale to large programs. For example, to balance between precision and scalability, block-based pointer analysis (BPA) [14] tracks function pointers interprocedurally to generate sound CFGs for Control-Flow Integrity (CFI [1]) policies. BPA relies on a *block memory model* to discover and abstract each compound data structure as a block, which is treated as a single variable during pointer analysis. By modeling the memory as blocks that correspond to the program-defined data structures, BPA assumes that a pointer to one memory block cannot be made to point to another memory block. Thanks to this assumption, pointer arithmetics do not need to be modeled during pointer analysis, which is the key to scalability. Moreover, since in good coding practices related function pointers are typically stored in the same compound data structure, BPA is a good fit for generating CFI policies by resolving the targets of indirect branches. However, BPA's memory block generation may cause soundness violations and its design of not tracking offsets within a block may lead to significant overapproximation for a general-purpose pointer analysis.

CC '22, April 02–03, 2022, Seoul, South Korea

ACM ISBN 978-1-4503-9183-2/22/04…$15.00
https://doi.org/10.1145/3497776.3517776

In this paper, we propose BinPointer, a new binary-level interprocedural general-purpose pointer analysis that relies on an offset-sensitive block memory model to achieve good scalability while maintaining high precision and soundness. In detail, while BinPointer still performs pointer analysis based on the block memory model, the major difference is that BinPointer tracks offsets within memory blocks during pointer analysis, which is the key to improving the precision.

We also propose a dynamic analysis based strategy for evaluating the soundness and precision of pointer analysis. The idea is to collect runtime memory reference traces of binaries with respect to test inputs and use that as the "pseudo" ground truth to compute precision and recall rates of BinPointer. Such kind of evaluation is missing in prior systems.

Contributions. BinPointer is implemented in Datalog, a logic programming language that has been widely used for pointer analysis [7, 13, 14, 20]. Overall, BinPointer makes the following contributions:

- BinPointer proposes a new binary-level interprocedural pointer analysis that relies on an offset-sensitive block memory model. It includes a 0-base abstraction, which is offset sensitive and improves the granularity of abstract locations during pointer analysis to achieve higher precision than prior work, while being conservative for soundness.
- We introduce two evaluation methods for evaluating binary-level pointer analysis that uses a block memory model. The first method collects runtime memory access traces to compute the precision and the recall rate of a binary-level pointer analysis. Its key component is a dynamic analysis that maps concrete memory addresses accessed during runtime into block-offset pairs in the block memory model. The conversion component of our dynamic analysis is also applicable for evaluating recall rates of other kinds of binary pointer analysis that does not use the block memory model. Considering that runtime memory accesses may be incomplete (depending on test data), we propose a second method that relies solely on the output of a pointer analysis, which aims to demonstrate BinPointer's reduction of overapproximation compared to BPA.
- We have successfully applied BinPointer to real-world binaries, which demonstrates acceptable scalability. In evaluation, by using the first evaluation method, BinPointer achieved 100% recall rate, which validated the soundness of the offset-sensitive block-based pointer analysis. Moreover, our experimental results demonstrated a 28.5% average precision improvement of BinPointer over BPA according to the collected runtime data. By the second evaluation method, we have seen a 41.6% overapproximation reduction of BinPointer compared to BPA on average.

2 Related Work

Source-Level Pointer Analysis. Pointer analysis, also known as points-to analysis, is a fundamental building block of program analysis. Most pointer analysis is formulated at the source level or the intermediate-representation (IR) level, where there is a rich set of information such as types that can be exploited for better analysis precision and scalability. Classic pointer analysis for C code include Anderson's algorithm [3] and Steensgaard's algorithm [22]. Recent pointer analysis for LLVM IR code include examples such as DSA [15] and SVF [23]. Several source-level pointer analysis frameworks for Java and C/C++ code [6, 7, 10, 13, 21] have been implemented in Datalog, where they have enjoyed benefits of modularity, high-performance, and precise static analysis offered by Datalog.

Binary-Level Pointer Analysis. VSA [5] is the most widely adopted binary-level pointer analysis in reverse engineering platforms such as BAP [8], ANGR [19] and CodeSurfer [4]. VSA is the first work that proposes the theoretical basis of binary-level pointer analysis. However, the original VSA does not scale to large programs and produces too many false positives (i.e., spurious points-to relations) [19, 29]. Thus, VSA needs to be customized for individual applications, where the design of abstract locations and the representation of values are the critical components for customization. For example, ANGR adopts the signedness-agnostic-domain to avoid heavily over-approximated results. BPA [14] is another example of customization by using a block memory model and tracking function pointers for the purpose of sound CFG generation. It demonstrates the possibility of utilizing the block memory model to balance the scalability and accuracy of VSA, which motivates BinPointer. BDA [29] aims at a scalable binary-level dependency analysis on a set of sampled paths. While BDA achieves higher precision on dependency analysis than VSA and IDA [11], its path sampling does not cover all paths and cannot guarantee the soundness of the underlying pointer analysis.

Applications of Binary-Level Pointer Analysis. Many binary reverse engineering tasks require the result of pointer analysis, including disassembly, binary-level CFG construction, binary debloating, binary diffing, among many others. As an example, one of the applications of the original VSA [5] was binary-level CFG construction (specifically, determining the control-flow targets of indirect calls and indirect jumps).

3 Background: Block Memory Model

A major challenge of binary-level pointer analysis is to model memory to balance scalability and precision. The most precise approach models the memory as an array of one-byte slots; however, it is unscalable. The most scalable design is to model the whole memory as one single abstract slot that

holds a set of values, with a memory write adding the written value to the set (i.e., a weak update) and a read getting the value set; however, this is highly imprecise. Therefore, a practical binary-level pointer analysis needs to find a good balance between scalability and precision.

The block memory model was first proposed by the CompCert project [16] for specifying the semantics of C-like languages and verifying correctness of program transformations at source level. BPA [14] showed that the block memory model could be used to improve the scalability of binary-level pointer analysis. In this model, memory is partitioned into a set of disjoint *memory blocks*. Each block contains a logically cohesive set of memory slots. A critical assumption is that adding an offset o to an pointer p results in a pointer within the same block as the block of p; that is, $p + o$ can never be made to point to a different block. This is called the *pointer-arithmetic assumption*. This assumption is sound if memory blocks are formed properly to accommodate legal pointer accesses allowed by source language semantics [16], since out-of-block accesses imply memory errors (e.g., when accessing an array out of bound). A static pointer analysis is supposed to capture points-to relations of legal executions of the input program; when there is a memory error, the program behavior is undefined and pointer analysis should not capture points-to relations happening only in illegal executions. Therefore, with the right block boundaries, the pointer-arithmetic assumption is upheld. Previous work of formalizing pointer analysis [9, 25] also considers only memory-safe executions.

However, BPA ignores the slots within memory blocks, saving the cost of tracking offsets within blocks to improve scalability.[1] That is, BPA determines what memory blocks a pointer may point to, without differentiating the slots in blocks. During the analysis, each memory block is treated as a whole and mapped to one value set. As a result, memory reads and writes through pointers to the same block are referring to the same value set. This design is a middle-ground design, making BPA scalable in analyzing function pointers for large benchmarks while maintaining decent accuracy. However, by avoiding tracking offsets, BPA sacrifices precision, which motivates the design of BinPointer.

Memory Block Generation. An important piece of BPA is how blocks are generated, as block boundaries should uphold the pointer-arithmetic assumption. BPA's memory block generation is customized for function pointer analysis. We next give a quick summary, while leaving details to BPA's paper [14]. BPA requires a complete disassembly of the target binary as input. Then it performs a heuristic based memory-access analysis to generate memory blocks for different memory regions, including stack frames, global data sections, and the heap (for dynamically allocated data), explained in more detail next.

[1]Note that BPA has a minor global chunk mechanism, which divides read-only global blocks into slots; however, the mechanism is limited to only read-only blocks, which are rare outside global blocks. Thus it is ignored in the rest of the paper.

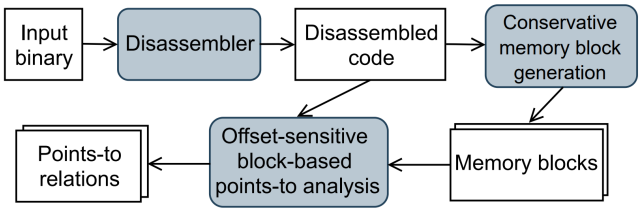

Figure 1. BinPointer's system flow.

Heap memory blocks are identified via static allocation sites. A `malloc`-family library call is identified with a unique allocation site ID and memory regions allocated at that site belong to the same memory block with the ID.

For a function stack frame, BPA analyzes the function's instructions related to stack layout changes and stack memory accesses to generate stack blocks; this is similar to how other systems (e.g., [2, 5, 11, 26]) decompile stack slots to variable-like entities. The stack block generation has two steps. The first step gathers a set of boundary candidates through a stack layout analysis. At a high level, it infers the relationship between register values and the initial stack top (the stack pointer value at the beginning of the function) on the instructions in each function. Then, stack accesses can be identified and the stack addresses in those accesses are used as candidate boundaries. In the second step, candidates that may split a compound data structure, such as an array or a struct, are removed according to stack access patterns. The remaining boundaries partition the stack frame of the target function into stack blocks.

Similar to how stack frames are partitioned, BPA partitions global data sections (i.e., .DATA, .RODATA, and .BSS sections) to memory blocks by following a dataflow analysis and a set of heuristics. Details can be found in the paper [14].

4 System Overview

BinPointer is a general-purpose interprocedural pointer analysis framework that infers the targets of memory-access instructions at the binary level. Fig. 1 presents the workflow of BinPointer. It takes a binary as input and disassembles the binary. After the disassembly, it translates the input program's assembly instructions into RTL [18] instructions. Next, BinPointer analyzes RTL instructions to perform a conservative generation of memory blocks. Then, BinPointer performs an offset-sensitive pointer analysis, using memory blocks with offsets as abstract locations. At the end, BinPointer outputs a set of pointer relations, each of which is a tuple (i, b, o) meaning that the memory-access instruction i may access memory block b with an offset o.

Conservative Memory Block Generation. Both BPA and BinPointer model memory as a set of disjoint memory blocks. BPA targeted CFG construction with pointer analysis. Its soundness was validated with only the targets of indirect calls; the soundness of general points-to relations was not validated. In fact, we have found instances that break the soundness of BPA. Therefore, BinPointer's memory block generation is redesigned to be more conservative, resulting in larger memory blocks than BPA's to pursue a sound general-purpose pointer analysis. We will discuss BinPointer's memory block generation in Sec. 6.

Offset-Sensitive Block-based Pointer Analysis. After memory blocks are generated, BinPointer performs a fixed point computation that recursively interleaves three steps. The first step is to transform the use of registers into the Single-Static-Assignment form based on a CFG by adding ϕ-instructions; the second step performs pointer analysis to infer points-to relations based on the current CFG with the SSA form; the final step discovers targets of indirect branches (indirect calls and jumps) and adds them to the CFG; note that BinPointer takes as input a CFG that contains only edges for direct branches. Since both SSA transformation and pointer analysis depend on the CFG, the three steps are mutually recursive. We implement all three steps by Datalog rules and Datalog's execution engine computes a fixed point until no new points-to relations and CFG edges are discovered. We note that BinPointer's steps of SSA transformation and discovering new CFG edges are identical to BPA's. However, BinPointer's rules of value tracking analysis enable the tracking of offsets in memory blocks, which are formalized and elaborated in Sec. 5; this is the key contribution of BinPointer. Note that BinPointer is a context-insensitive interprocedural pointer analysis. It supports flow sensitivity on registers through SSA, but is flow insensitive on memory locations.

5 Pointer Analysis

BinPointer's key contribution is to compute offsets within memory blocks to achieve higher precision on pointer analysis, while maintaining soundness. This section discusses how its pointer analysis is performed, assuming a set of memory blocks have already been generated (discussed in Sec. 6). The pointer analysis performs value tracking analysis to track values of abstract locations. As noted earlier, this is part of a fixed point computation that also performs SSA and discovers new targets of indirect branch instructions.

5.1 Motivating Example

Tracking all offsets would be too costly for pointer analysis as it would maintain a value set for every abstract location $b[o]$, with b being the ID of a block and o being an offset; o can be an integer or $\top$ (indicating any offset within the block). Hence BinPointer adopts a so-called *0-base abstraction*, which precisely tracks only those offsets resulting from pointer arithmetics using $b[0]$ as base addresses but overapproximates other kinds of offsets to $\top$.

```
(1) call malloc        (4) add edx, 4
(2) mov edx, eax       (5) mov [edx], 1
(3) mov [edx+4], 0     (6) add edx, 4
```

Figure 2. Motivating example for the 0-base abstraction.

We illustrate this abstraction with an example in Fig. 2. In this code, a heap block for allocation site at (1) is created and the address $b[0]$ is stored in eax, assuming b is the allocation-site ID for (1). Then $b[0]$ is stored into edx at (2), followed by a pointer arithmetic edx + 4 at (3); since this pointer arithmetic's base address $b[0]$ has offset 0, its result is tracked precisely and determined to be $b[4]$. Similarly, at instruction (4), the pointer arithmetic is also performed precisely and determined to be $b[4]$. At instruction (5), though edx does not hold a 0-offset address, there is no pointer arithmetic and BinPointer determines it is writing to the abstract location $b[4]$. However, at (6), there is a pointer arithmetic with a non-zero-offset base address; here BinPointer loses precision and edx's value gets to be $b[\top]$.

The motivation for this design choice is to balance between precision and scalability. Many memory operations in programs use 0-offset base addresses to access memory. E.g., imagine a memory block holds a C struct and a field can be accessed through the base address of the struct with some offset; similarly, if a memory block holds an array, an element can be accessed through the base address of the array with some offset. Our 0-base abstraction can reason about these patterns precisely. For each address resulting from a 0-base pointer arithmetic, BinPointer treats it as an abstract location and tracks its value set separately. Furthermore, our abstraction avoids creating too many offsets, especially when a pointer arithmetic occurs in a loop. Imagine a loop is used to access elements in an array with p initialized to be the base address and each iteration uses a pointer arithmetic to move p to the next array element; in this case, BinPointer would not create an abstract location for every array element and would treat p as having offset $\top$ after two iterations.

5.2 Offset-Sensitive Value Tracking Analysis

We next formalize the core of BinPointer's pointer analysis. A binary is first disassembled and translated into RTL [18], an IR with a RISC-like instruction set. The semantics of an x86 instruction is captured by a series of RTL instructions. Fig. 3 shows a simplified RTL syntax necessary for BinPointer. Also, the syntax is in the SSA form for registers. We write *ireg* for an indexed register after the SSA transform; it is a register with an integer index. ϕ-functions are also introduced to capture data flows among indexed registers.

VExp is the type of expressions that evaluate to values. One case is $\text{libcall}_{@c}(f, arg_1, \ldots, arg_n)$, which evaluates to

$$
\begin{array}{rcl}
i, c & \in & \text{Integers} \\
f & \in & \text{Library functions} \\
reg & := & \text{ESP} \mid \text{EBP} \mid \text{EAX} \mid \ldots \\
Bvop & := & + \mid - \mid \times \mid \ldots \\
ireg & := & reg_i \\
VExp & := & c \mid ireg \mid \text{Mem}[AExp] \\
 & \mid & \text{arith}(VExp,\ Bvop,\ VExp) \\
 & \mid & \text{libcall}_{@c}(f, arg_1, \ldots, arg_n) \\
AExp & := & c \mid c + ireg \mid c + ireg * c' \\
 & \mid & c + ireg + ireg' * c' \\
Instr & := & ireg = VExp \mid \text{Mem}[AExp] = VExp \\
 & \mid & ireg = \phi(ireg_1, \ldots, ireg_n)
\end{array}
$$

Figure 3. Simplified RTL syntax in the SSA form.

the return value of the call to an external library function f (e.g., malloc); further, the call is at address c, which will be used to identify the heap block returned by a call to a malloc-like function. *AExp* is the type of expressions that represent memory addresses. They include four x86 address modes: (1) c represents a constant memory address, (2) $c+reg$ represents either an address c plus an offset in reg, or an address with reg as the base address and c as the constant offset, (3) $c + reg * c'$ represents an address c plus an offset in $reg * c'$, where c' is a scale, and (4) $c + reg + reg' * c'$ has two possibilities as in case (2): c is a base address and the rest are an offset, or reg is a base address and the rest are an offset.

There are three types of instructions: assignments to an indexed register ($ireg = VExp$), memory writes ($\text{Mem}[AExp] = VExp$), and ϕ instructions ($ireg = \phi(ireg_1, \ldots, ireg_n)$). BinPointer abstracts away other kinds of instructions (e.g. CPU flag updates) that are not memory related.

Recall that our pointer analysis assumes memory blocks have been generated. We write GB_a for the ID of a global block that starts at global address a; we write $\text{SB}_{(f,a)}$ for the ID of a stack block of function f with the starting offset being a (the offset from the initial stack top of f); and we write HB_c for the heap block with allocation site c. Information about global and stack memory block boundaries are encoded in a pair of auxiliary functions: GBMap and SBMap, explained in Table 1. We further assume FNMap, which tells the function a code address belongs to. In addition, our pointer analysis assumes a preprocessing step: a stack layout analysis that analyzes a function and tracks the offset from registers to the stack top. This is a must analysis and produces a StkOffMap, which is explained in Table 1. For instance, if esp_0 is the initial stack top and $\text{esp}_1 = \text{esp}_0 - 32$, then $\text{StkOffMap}(\text{esp}_1) = -32$; and if further $\text{eax}_1 = \text{esp}_1 + 4$, then $\text{StkOffMap}(\text{eax}_1) = -28$.

With the help of those maps, Fig. 4 and Table 2 define the following relations/functions for pointer analysis:

- Relation $\text{AlocVal}(loc, val)$ holds if an abstract location loc is determined to hold a value val.

Table 1. A set of maps assumed by pointer analysis.

GBMap(c)	returns the start address of the global block that global address c belongs to.
SBMap(f, o)	returns the start address of the stack block that contains the offset o in f's stack frame.
FNMap(c)	returns a function name f, if f's start code address is c.
StkOffMap$(ireg)$	returns (f, o), where f is the function that contains $ireg$ and o the definite offset from $ireg$ to the initial stack top of f, if $ireg$ points to the stack.

$$
\frac{\text{Reachable}(ireg = VExp) \qquad val \in \text{ValSet}(VExp)}{\text{AlocVal}(ireg, val)}\ \textsc{ToIreg}
$$

$$
\frac{\begin{array}{c}\text{Reachable}(\text{Mem}[AExp] = VExp) \\ val \in \text{ValSet}(VExp) \qquad loc' \in \text{LocSet}(AExp) \\ loc' \lozenge loc\end{array}}{\text{AlocVal}(loc, val)}\ \textsc{ToMem}
$$

$$
\frac{\begin{array}{c}\text{Reachable}(ireg = \phi(ireg_1, \ldots, ireg_n)) \\ \text{AlocVal}(ireg_i, val) \text{ for some } i\end{array}}{\text{AlocVal}(ireg, val)}\ \textsc{Phi}
$$

Figure 4. Value tracking rules defined over instructions.

- Relation Reachable(*Instr*) holds if an instruction *Instr* is reachable in the CFG being built along with the value tracking analysis, which enables BinPointer to avoid analyzing dead code. Note that BinPointer incrementally updates the CFG when new indirect branch targets are found.
- Function LocSet(*AExp*) takes an address expression *AExp* and returns its set of abstract locations.
- Function ValSet(*VExp*) takes an expression *VExp* and returns its value set.

Fig. 4 shows a set of flow-insensitive rules that track value sets for different kinds of instructions[2]. Rule ToIReg transfers the value set of *VExp* to the left-hand side of the assignment. Rule ToMem transfers the value set of *VExp* to any abstract location represented by *AExp*. This rule uses relation $loc \lozenge loc'$ when two abstract locations may overlap in their concrete address sets. E.g., $b[3]$ and $b[\top]$ satisfy the relation since $\top$ covers all offsets, including offset 3. When there is a memory write to an abstract location, for soundness all overlapping abstract locations should contain the written value. Rule Phi

[2]Since SSA is applied to registers, it achieves flow sensitivity on value sets in registers even with flow-insensitive rules.

Table 2. Definitions of ValSet(*VExp*) and LocSet(*AExp*).

$S \oplus c :=$
$\{b[c] \mid b[0] \in S\} \cup \{b[\top] \mid b[o] \in S \wedge o \neq 0\}$

LocSet(c) := $\{\text{GB}_a[c-a] \mid a = \text{GBMap}(c)\}$
LocSet($c + ireg$) :=
  if *ireg* in StkOffMap then
    let O = StkOffMap(*ireg*) in
    $\{\text{SB}_{(f,a)}[c+o-a] \mid (f,o) \in O \wedge a = \text{SBMap}(f, c+o)\}$
  else if $c = 0$ then $\{b[o] \mid \text{AlocVal}(ireg, b[o])\}$
  else let $S = \{v \mid \text{AlocVal}(ireg, v)\} \oplus c$ in
    if c in GBMap then $\{\text{GB}_a[\top] \mid a = \text{GBMap}(c)\} \cup S$
    else S
LocSet($c + ireg * c'$) := $\{\text{GBMap}(c)[\top]\}$
LocSet($c + ireg + ireg' * c'$) :=
  if *ireg* in StkOffMap then
    let O = StkOffMap(*ireg*) in
    $\{\text{SB}_{(f,a)}[\top] \mid (f,o) \in O \wedge a = \text{SBMap}(f, c+o)\}$
  else let $S = \{b[\top] \mid \text{AlocVal}(ireg, b[o])\}$ in
    if c in GBMap then $\{\text{GB}_a[\top] \mid a = \text{GBMap}(c)\} \cup S$
    else S

ValSet(c) :=
  if c in FNMap then $\{\text{FNMap}(c)\}$
  else if c in GBMap then $\{\text{GB}_a[c-a] \mid a = \text{GBMap}(c)\}$
  else $\emptyset$
ValSet(*ireg*) :=
  if *ireg* in StkOffMap then $\emptyset$ else $\{v \mid \text{AlocVal}(ireg, v)\}$
ValSet(Mem[*AExp*]) :=
  $\{v \mid loc \in \text{LocSet}(AExp) \wedge \text{AlocVal}(loc, v)\}$
ValSet(arith(*VExp*$_1$, *Bvop*, *VExp*$_2$)) :=
  match *VExp*$_1$, *Bvop*, *VExp*$_2$ with
  | c, +, c' -> $\{b[o] \mid b[o] \in \text{ValSet}(c + c')\}$
  | c, −, c' -> $\{b[o] \mid b[o] \in \text{ValSet}(c - c')\}$
  | c, +, _ -> $(\text{ValSet}(VExp_2) \oplus c) \cup \{b[\top] \mid b[o] \in \text{ValSet}(c)\}$
  | _, +, c -> $(\text{ValSet}(VExp_1) \oplus c) \cup \{b[\top] \mid b[o] \in \text{ValSet}(c)\}$
  | c, −, _ -> $\{b[\top] \mid b[o] \in \text{ValSet}(c)\}$
  | _, −, c -> $\text{ValSet}(VExp_1) \oplus -c$
  | _, +, _ -> $\{b[\top] \mid b[o] \in \text{ValSet}(VExp_1) \cup \text{ValSet}(VExp_2)\}$
  | _, −, _ -> $\{b[\top] \mid b[o] \in \text{ValSet}(VExp_1)\}$
  | _, _, _ -> $\emptyset$
ValSet(libcall$_{@c}(f, arg_1, \ldots, arg_n)$) :=
  if $f \in \{malloc, calloc\}$ then $\{\text{HB}_c[0]\}$
  else if $f = realloc$ then $\{\text{HB}_{c'}[0] \mid \text{AlocVal}(arg_1, \text{HB}_{c'}[0])\}$
  else $\emptyset$

transfers the value sets of argument indexed registers to the destination indexed register.

Definition 5.1 (Location overlapping relation)**.** Two abstract locations b$[o]$ and b$'[o']$ are overlapping, written as b$[o]$ ◊ b$'[o']$, if (1) $b = b'$ and (2) $o = o' \vee o = \top \vee o' = \top$.

Definition $S \oplus c$ in Table 2 models pointer arithmetics based on BinPointer's 0-base abstraction. S is a set of abstract locations and c is a constant. When an abstract location represents the base address of a block (with offset 0), the definition tracks the new address's offset precisely. In other cases, the resulting address is approximated with offset $\top$.

LocSet(−) returns a set of memory abstract locations for an *AExp*. We assume that only global data sections can be accessed in a memory operation using a constant base address (with some possible offset), since heap blocks and stack frames are dynamically allocated and cannot be safely accessed via addresses with constant base addresses. Therefore, the case of LocSet(c) returns a global abstract location using GBMap to compute the global block ID and "c − GBMap(c)" as the offset, assuming c is in the domain of GBMap. The case of LocSet(c + *ireg*) considers three main cases: (1) if *ireg* must point to the stack according to StkOffMap, it returns a stack block based on information in SBMap; (2) if c is a base address of a global block, it returns the global block with a $\top$ offset for approximation, and (3) if *ireg* holds a base address of some block, then pointer arithmetic is performed using c as the offset. The other two cases of LocSet(−) follow a similar design as the first two cases.

ValSet(*VExp*) returns a set of values for *VExp*. For scalability, the pointer analysis tracks only two kinds of values: (1) a memory address in the form of $b[o]$, and (2) the start code address of some function f in the form of $\&f$. For the case of ValSet(c), it returns $\&f$ if c is the start code address of some function f according to FNMap, and returns a global location if c belongs to GBMap; if neither satisfies, it returns the empty set as the pointer analysis does not track other kinds of values including constants. The definitions of ValSet(*ireg*) and ValSet(Mem[*AExp*]) are self explanatory. For ValSet(arith(−)), the goal is to track memory addresses that are computed through pointer arithmetics via either the plus or the minus operator. For other operators, the definition returns the empty set as memory addresses should not be computed through those operators. We next discuss the case of ValSet(arith(c, +, _)) and other cases are similar. Definition ValSet(arith(c, +, _)) considers two possibilities: (1) c used as a base address of a global block, and (2) *VExp*$_2$ used as a base address. In case (1), it returns the global block with offset $\top$. For case (2), BinPointer's abstract $\oplus$ operator is performed on the value set of *VExp*$_2$ and c. Finally, the case for libcall$_{@c}(f, arg_1, \ldots, arg_n)$ models how heap blocks are created.

6 Implementation

As noted earlier, BinPointer's major components including its pointer analysis are implemented in Datalog, a declarative language that enables a modular implementation and efficient fixed point computation. We use the Datalog engine Souffle [12], and BinPointer's core analysis components are implemented by approximately 3,600 lines of Datalog code.

Another component in BinPointer's implementation is its memory block generation. For heap memory blocks, BinPointer adopts the same allocation-site ID approach as BPA.

For partitioning stack and global data sections, BinPointer has different designs for pursuing soundness.

Partitioning Global Data Sections. BPA leverages a data-flow analysis and heuristics to partition global data sections; this method, however, may result in memory blocks that violate the pointer-arithmetic assumption. BinPointer instead relies on symbol tables to identify block boundaries for globals. Symbol tables are generated by compilers to contain information about symbols (e.g., function and variable names) from source code and are embedded in binaries. From symbol tables BinPointer can know global symbols' base addresses, which are used to partition a global data section into memory blocks. Since compiler-generated symbol tables are used by linkers for static/dynamic linking and debuggers for debugging, we can assume the correctness of symbol tables. Therefore, this method of generating memory blocks is trustworthy. Since BinPointer's focus is its offset-sensitive pointer analysis, we rely on symbol tables to ease the preprocessing step for global block boundary generation. To support stripped binaries we can generate global boundaries without relying on symbol tables. One way is to use heuristic-based methods suggested by the BPA paper[14], though it may decrease precision/recall rates depending on the input programs. Generating sound and precise global block boundaries without symbol tables is an interesting research direction.

Stack Partitioning. BinPointer partitions stack frames into memory blocks in two steps: (1) gather a set of boundary candidates by a stack layout analysis and (2) remove candidates that may split a compound data structure. This is similar to BPA, but BinPointer takes a more conservative approach in step (2), meaning that less boundary candidates are kept as the final boundaries. In detail, only the address mode that has a scale component is used to determine final boundaries. For example, in BinPointer, the stack address computed from esp-420 in the memory write instruction "`mov [esp+(esi*4)-420], 0`" is a final boundary. In contrast, boundaries considered by BPA may result in an unsound general-purpose pointer analysis on optimized code. For example, BPA considers the stack address computed from esp-416 in the address-loading instruction "`lea eax, esp-416`" as a boundary, which could be taking the address of the second element of an array starting at esp-420.

For a fair comparison with BPA, during evaluation we compare BinPointer with a version of BPA that uses the same memory block generation design as BinPointer's.

7 Evaluation Strategy

It is challenging to evaluate the precision and soundness of a binary-level pointer analysis, since collecting ground truth for large benchmarks is difficult. Motivated by previous work [14, 27] that relies on profiling to evaluate static CFG construction, we collect runtime memory accesses triggered by test inputs to validate BinPointer's soundness and to evaluate its precision; the workflow is visualized in Fig. 5. To adapt the evaluation to the block memory model, our runtime memory access collection converts the target address of a memory read/write into a pair (b, o), where b is the ID of a memory block, and o is an offset into the block.

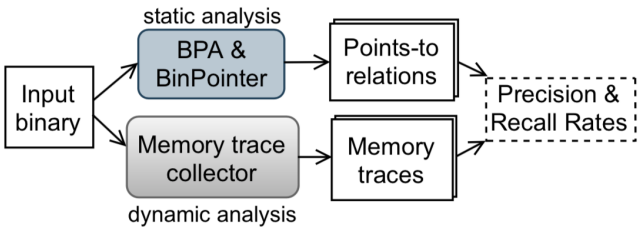

Figure 5. Runtime-data driven evaluation strategy.

We use Intel's Pin [17] to construct a dynamic analysis tool, which collects and converts accessed memory addresses; we call this tool the dynamic collector. In Sec. 7.1, we explain how the tool dynamically converts memory addresses into a format that can be used to evaluate pointer analysis. In Sec. 7.2, we introduce a runtime-data based precision metric. Moreover, to ameliorate the incompleteness problem of runtime memory accesses, in Sec. 7.3 we propose another metric that relies only on the points-to relations of BinPointer and BPA to compare their degrees of overapproximation.

7.1 Dynamic Conversion of Memory Addresses

The key step of our dynamic collector is to convert memory addresses into a format that can be used to evaluate static pointer analysis, including but not limited to BPA and BinPointer. At runtime the information for a memory-access instruction is of the form (i, a), where i is a memory-access instruction and a is a memory address in the flat memory model. However, BinPointer produces results of the form (i, b, o), where i is a memory-access instruction, b the block, and o the offset. Therefore, there is a need to decompose a flat memory address into a block and an offset.

For a memory address in a global data section, the decomposition is straightforward and can be statically performed since a global data section is partitioned via a set of static global memory addresses. In contrast, the decomposition cannot be performed statically for memory addresses to stack frames and dynamically allocated heap regions. For example, if a is an address to a heap region, the base address of the heap region is needed to perform the decomposition; but the base address cannot be known statically and the set of valid base addresses changes during program execution due to memory allocation/deallocation. In general, our Pin-based tool tracks the memory state dynamically (e.g., what heap regions are allocated and their start addresses and sizes) to perform this decomposition.

Converting Heap Memory Accesses. Our dynamic collector tracks the set of currently allocated heap memory regions and their memory ranges. In particular, for a call instruction to *malloc*/*calloc*, the collector records the instruction's code address h, the start address of the allocated heap memory region h_{start}, which is the return value of the call, and the size of the allocated heap memory region h_{size}, which can be retrieved from the argument of the call. That is, h is the allocation site and $[h_{start}, h_{start} + h_{size})$ is the allocated memory region range. If a *malloc*/*calloc* at the same allocation site is invoked multiple times in one run, we distinguish them by indices. Thus, a region allocated at site h is represented by h^n and its range is $[h^n_{start}, h^n_{start} + h^n_{size})$. For a call to *realloc*, which resizes a buffer, the collector retrieves the start address of the resized buffer from the first argument of the call and the new size from the second argument. Then the start address is used to determine the corresponding h^n; and the stored range information of h^n is updated to reflect the new size. Finally, for a call to *free*, its argument holds the start address of a heap memory region that is to be freed. Thus, the collector discards the range information of h^n if h^n_{start} matches the start address.

Information maintained by the collector partitions the heap memory into a set of disjoint heap regions, at any moment during execution. With that information, if a memory instruction i accesses address a, the collector can determine the range $[h^n_{start}, h^n_{start} + h^n_{size})$ that contains a and generates a triple $(i, h, a - h^n_{start})$, where allocation site h is used to identify the static memory block (which represents all regions allocated at that site).

Converting Stack Memory Accesses. The collector maintains a stack of function contexts. In detail, for a function call, the collector pushes the callee's name and the base address of the callee's stack frame as a pair (f, f_{base}) onto the stack maintained by the collector; and the collector pops a pair off the stack when it sees a function return. Tail calls are also considered; they are jump instructions that target functions. In effect, a tail call does not add a new stack frame to the call-stack. However, the context changes. Thus, to reflect the context switch, the collector pops the top pair off its stack and pushes the callee's name and the base address of its stack frame onto the stack. Thus, each adjacent pair on the collector's stack defines the range of a stack frame. As a result, when the collector encounters an instruction at code address i accessing a memory address a within a stack frame f, it retrieves the base address f_{base} of the stack frame and converts the address a into an offset into the stack frame $(i, f, a - f_{base})$. If the stack frame of f is partitioned to blocks, $(i, f, a - f_{base})$ is further converted to be of the form (i, b, o), where b is a block in the f's stack frame; this conversion can be performed statically, since the block partitioning of a stack frame does not depend on dynamic information.

7.2 Runtime-Data based Evaluation Metrics

We use set D for the set of triples (i, b, o) collected during runtime (discussed in Sec. 7.1); in these triples, an offset o is a concrete integer. We further use set S for the set of points-to relations (i, b, o) computed by a static pointer analysis that is based on the block memory model; in these triples, an offset o is an abstract value. E.g., in BinPointer's output, o can be either $\top$ or an integer; in BPA's output, o must be $\top$ as BPA tracks only blocks but not offsets in its pointer analysis.

By comparing D and S, we define notions of recall and soundness. We say a triple $(i, b, o) \in D$ is *covered* by S if either $(i, b, o) \in S$ or $(i, b, \top) \in S$; i.e., the concrete result is covered by the predicted result in S. Then the *recall rate* of S relative to D is the percentage of triples in D that are covered by S. When the recall rate is 100%, we say S is *sound* relative to D, meaning every triple in D is covered by S.

Soundness, however, is only part of the story. For instance, if a pointer analysis's result includes $(i, b, \top)$ for every i and b, it is trivially sound. We therefore introduce a notion of *precision*, relative to runtime data. We first discuss the intuition through examples. Imagine runtime data D contains $(i, b, 0)$ and $(i, b, 1)$. In one case, imagine the pointer analysis result contains $(i, b, 0)$, $(i, b, 1)$, and $(i, b, 2)$; in this case, we say the precision is $2/3$ for (i, b), since two of three predicted outcomes by pointer analysis appear in runtime data. In the previous example, suppose we change the pointer analysis result to be $(i, b, \top)$; since $\top$ represents every possible offset in block b, we say the precision is $2/\text{SizeOf}(b)$, where $\text{SizeOf}(b)$ is the size of block b in terms of bytes. Note that sizes of heap memory blocks can be obtained during runtime data D, while sizes of stack and global memory blocks can be statically determined by the memory block boundaries.

With the precision for a particular pair (i, b) defined, we can define the precision of S relative to D to be the average precision across all blocks and all instructions. We next introduce notations and define it formally.

We write $\text{Instrs}(D)$ for the instructions in D, defined as $\{i \mid \exists b \exists o,\ (i, b, o) \in D\}$. We write $\text{Blocks}(D, i)$ for the blocks associated with i in D, defined as $\{b \mid \exists o,\ (i, b, o) \in D\}$. We write $D(i, b) = \{o \mid (i, b, o) \in D\}$ for the set of offsets associated with (i, b) in D; similarly, we write $S(i, b) = \{o \mid (i, b, o) \in S\}$.

Then the precision of S relative to D is defined as

$$P(S, D) = \frac{1}{|\text{Instrs}(D)|} \sum_{i \in \text{Instrs}(D)} \frac{\sum_{b \in \text{Blocks}(D,i)} P_{i,b}(S, D)}{|\text{Blocks}(D, i)|},$$

where $P_{i,b}(S, D)$ is the precision for (i, b), defined as:

$$P_{i,b}(S, D) = \begin{cases} \frac{|D(i,b) \cap S(i,b)|}{|S(i,b)|}, & \text{if } (i, b, \top) \notin S. \\ \\ \frac{|D(i,b)|}{\text{SizeOf}(b)}, & \text{if } (i, b, \top) \in S \end{cases}$$

7.3 Overapproximation Degree

One weakness of the runtime data based precision metric is that it is relative to runtime data D and the quality of the metric depends on how complete D is. We describe another metric, which does not rely on runtime data and quantitatively measures the overapproximation of a system by calculating the percentage of overapproximated points-to relations in its output. A higher percentage indicates a larger overapproximation; we call this metric *overapproximation degree*. Specifically, if $(i, b, \top) \in S$, we say the pointer analysis output S is overapproximated for the pair of (i, b). A more precise points-to analysis should have less overapproximated (i, b) pairs in its output. Therefore, we calculate the number of overapproximated (i, b) pairs in the output S to measure the overapproximation degree (OD) of a system, which is defined as:

$$OD(S) = |\{(i, b) \mid (i, b, \top) \in S\}|.$$

8 Evaluation

BinPointer is evaluated on a set of SPEC CPU 2k6 C benchmarks and also `nginx` (a web server). The binaries are compiled by GCC-9.2 with optimization levels of -O0 to -O3. To save space, we report only the data of -O0 (unoptimized) and -O2 (most commonly used). We exclude `445.gobmk`, `400.perlbench`, and `403.gcc` as BinPointer does not scale to them. Also, since `nginx` does not come with standard reference inputs, we exclude it from recall and precision evaluation, which requires memory-access traces triggered by reference inputs. Our evaluation aims to answer three major questions: (1) how scalable is BinPointer? (2) how much precision is improved by BinPointer's offset-tracking abstraction? (3) Does BinPointer produce any false negatives?

8.1 Soundness and Precision

Through dynamic analysis introduced in Sec. 7, we collected runtime memory traces in the benchmark programs. When running those programs, we used SPEC CPU 2k6 benchmarks' extensive reference inputs.

BinPointer achieves a 100% recall rate, as defined in Sec. 7.2. That means BinPointer's pointer-analysis results for our benchmarks are sound relative to runtime memory traces we collected. As discussed in Sec. 6, we put additional efforts in making block boundary generation more coarse grained to pursue soundness.

For evaluating precision, Table 3 shows the precision results of BPA and BinPointer according to the runtime-trace based metric defined in Sec. 7.2. We exclude results for `482.sphinx3`, as the reference inputs by SPEC CPU 2k6 triggered only few memory references. Also, `n/a` in the table means no memory references were triggered for that type of memory blocks. As shown in the table, BinPointer's precision for stack blocks is close to 100%, since BinPointer recovers most stack memory locations through stack layout analysis and precise offset tracking. In contrast, BPA's precision is much lower as it does not distinguish memory accesses with different offsets within the same stack block. The precision difference between BinPointer and BPA for global blocks is relatively smaller. However, there are notable differences for several benchmarks such as `464.h264ref`, showing the effectiveness of BinPointer's offset tracking. For `464.h264ref`, by our manual investigation, the major reason is due to instruction patterns similar to the one below:

```
(1) mov eax, $0x100
(2) mov edx, [eax+$0x8]
```

For this example, BinPointer determines instruction (2) accesses an address with a block starting at 0x100 and offset 0x8. However, BPA does not distinguish offsets in this case; it assumes that eax at (2) points to every offset within the global block of start address 0x100.

Table 4 shows the overapproximation degree (OD) reduction results of BinPointer over BPA on binaries compiled by -O2 (other optimization levels' results are generally similar to those of -O2). As an example, for the stack blocks of mcf, BinPointer produces 79.5% less $(i, b, \top)$ triples compared to BPA. In general, BinPointer achieves high OD reduction rates in stack and global blocks. In contrast, BinPointer achieves relatively lower reduction results on heap blocks. For example, the OD rate on `401.bzip2` is 0.7%. Our investigation on `401.bzip2`'s results indicates that BinPointer produces many false positive results when analyzing heap blocks.

Table 3. Precision evaluation results of BPA and BinPointer (-O0 and -O2).

Program	Details		Runtime-based precision results (%)					
	Opt Level	Instrs	Stack		Global		Heap	
			BPA	BinP	BPA	BinP	BPA	BinP
mcf	O0	3.3K	18.9	100.0	27.5	71.9	n/a	n/a
	O2	2.4K	26.3	100.0	27.0	85.7	n/a	n/a
lbm	O0	6.5K	17.9	100.0	41.7	100.0	n/a	n/a
	O2	2.2K	22.3	99.5	73.1	100.0	n/a	n/a
lib-quantum	O0	10K	65.6	100.0	100.0	100.0	7.0	7.0
	O2	9.6K	47.9	100.0	100.0	100.0	6.9	6.9
bzip2	O0	21K	38.8	97.5	46.6	46.6	1.4	4.5
	O2	11K	16.9	93.2	51.7	51.7	3.6	21.8
sjeng	O0	32K	24.8	94.3	53.7	53.8	n/a	n/a
	O2	22K	32.7	97.5	55.1	55.6	n/a	n/a
milc	O0	31K	40.0	97.7	74.7	91.1	22.4	22.4
	O2	23K	49.4	99.4	81.2	88.9	23.7	23.7
hmmer	O0	88K	30.0	99.9	80.7	80.7	8.0	40.7
	O2	60K	38.0	99.9	76.4	76.4	7.6	11.5
h264ref	O0	161K	28.9	97.8	6.7	69.3	22.7	38.3
	O2	100K	35.3	97.3	6.2	65.5	24.2	40.8

8.2 Performance Evaluation

BinPointer scales to real world binaries including `nginx` and `464.h264ref` with reasonable computing resources: 128GB of RAM with a 32-core CPU (Intel Xeon Gold 6136 with

Table 4. Overapproximation degree reduction rate of BinPointer over BPA (-O2).

Memory Region	Overapproximation reduction rate (%)							
	mcf	lbm	libq	bzip2	sjeng	milc	hmmer	h264ref
Stack	79.5	99.1	56.1	30.3	81.1	20.4	33.1	18.7
Global	93.9	89.6	98.8	7.4	10.4	49.4	54.1	15.7
Heap	21.0	50.0	28.7	0.7	1.1	17.9	29.7	12.6

Table 5. Performance evaluations of BPA and BinPointer on large binaries (-O2).

Program	Runtime (s)		Memory (GB)		# of relations	
	BPA	BinP	BPA	BinP	BPA	BinP
hmmer	54	510	0.6	1.8	7M	12M
h264ref	612	23020	3.3	8.8	52M	109M
nginx	1723	7692	23	41	374M	525M

3.00GHz) on Ubuntu 18.04. Table 5 shows performance results on BPA and BinPointer on the three largest binaries in our benchmark sets. The binaries were compiled at level -O2. Around 6.5 and 2 hours were spent on `464.h264ref` and `nginx` by BinPointer, respectively. Benchmarks not shown in the table terminated within ten to a few hundred seconds by BinPointer. In general BinPointer's runtime is significantly higher than BPA's. We also present the size of AlocVal(−, −) relations computed by Datalog for both BPA and BinPointer, where M represents millions. In general, BinPointer produces 1.4x ∼ 2.0x more relations than BPA, resulting in more memory consumption (about 3x).

The data shows that BinPointer is not as scalable as BPA; the reason is that BPA tracks one value set per memory block, while BinPointer tracks separate value sets possibly at different offsets. On the other hand, BinPointer achieves higher precision, shown in Sec 8.1.

8.3 Comparison with angr's VSA

We compared BinPointer with the VSA implementation of `angr` [19], a popular open-source binary-analysis framework. To collect `angr`'s VSA output, we constructed the value flow graph (VFG) interface for each benchmark. In the VFG construction, we chose the static binary CFG construction approach as the basis, 3 as the context-sensitivity level, and 4 as the inter-function level. Then each VFG node constructed for user-defined functions was traversed to collect the registers' state (VSA output) in the node. However, we found two issues regarding the VFG results for the benchmarks we considered. First, `angr` produced VSA information for VFG nodes only in the `main` functions. Second, only a few instructions were associated with VSA information in the output; i.e., `angr`'s VSA output was incomplete.

Considering these issues, we turned to micro-benchmarks instead; we used those that were proposed to evaluate SVF [23, 24], an IR-level pointer analysis. We aimed at the `array` directory and the `struct` directory in SVF's GitHub repository[3] and we selected 7 micro-benchmarks that did not crash `angr`'s VSA and that did not have any user-defined functions other than `main`. Then we compared the VSA results of `angr` and BinPointer for `mov` and `add` instructions that involved memory accesses in the `main` functions of the 7 micro-benchmarks. Table 6 shows the results. For each micro-benchmark, the left-side number is the number of `mov` and `add` instructions whose memory references are precisely resolved; and the right-side number is the total number of the `mov` and `add` instructions with memory references.

Table 6. VSA results by ANGR and BinPointer on SVF micro-benchmarks (-O0).

Prog	array1	array2	array4	array5	stride	ben3	ben5
angr-VSA	0/10	0/3	0/4	2/9	1/5	0/3	0/16
BinPointer	10/10	3/3	4/4	9/9	4/5	3/3	16/16

In all, `angr`-VSA can precisely generate the VSA information for 3 out of 50 instructions. In fact, the resolved results are straightforward stack-access instructions through `rbp`. Though `angr`-VSA did produce result for 2 of the 4 instructions in `array4`, the output was incorrect. In contrast, BinPointer outputs precise VSA information for 49 out of 50 instructions. The only imprecision is one memory access to a global array within a loop in `stride`, where BinPointer produces an over-approximated result, while `angr`-VSA gives no information for this memory access.

8.4 Downstream Applications

Resolve Indirect-Call Targets. Static binary-level CFG construction has a wide range of applications in security and software engineering, such as control-flow integrity and binary debloating, where resolving the targets of indirect calls is the major challenge. Since BinPointer is superior to BPA in terms of pointer analysis precision, BinPointer should also be more precise in resolving indirect-call targets. BPA measures the precision of a CFG by computing the average indirect call target (AICT); it is the average number of targets across all indirect calls. Lower AICT means higher CFG precision.

We computed AICT metrics of the CFGs constructed by BPA and BinPointer for the benchmarks listed in Table 3 and `nginx` (v1.10). `464.h264ref` and `nginx` are the only benchmarks where there are more than 20 indirect call sites and BinPointer achieves AICT reductions compared to BPA. For other benchmarks, both BPA and BinPointer achieve the same AICT numbers. Table 7 shows the AICT results. As a result, 37.9% (-O0) and 25.7% (-O2) AICT reduction rates are observed on BinPointer over BPA on `464.h264ref`. On

[3] Available at https://github.com/SVF-tools/Test-Suite/tree/master/src/non_annotated_tests/

Table 7. AICT results for large benchmarks.

Pointer platform	464.h264ref		Nginx	
	O0	O2	O0	O2
BPA	6.6	30.7	337.8	519.6
BinPointer	4.1	22.8	99.5	465.2

```
(1) int *b[100];                <1> mov ebp, esp
 ...                            ...
(2) *b[20] = 300;               <2> mov eax, [ebp-736]
                                <3> mov [eax], 300
```

Figure 6. Instructions causing a segmentation fault in `array5`

`nginx`, 70.5% (-O0) and 10.5% (-O2) AICT reduction rates are observed on BinPointer over BPA. Moreover, we validated the soundness of the constructed CFGs for `464.h264ref` with runtime indirect call targets; however, we did not validate the soundness of CFGs for `nginx` because `nginx` does not provide reference inputs.

Detect Uninitialized Memory Read. We found an uninitialized memory read in the `array5` benchmark in SVF, which causes a segmentation fault. Fig. 6 shows `array5`'s simplified source code (left side) and assembly instructions (right side). At runtime, there is no instruction that writes any value to the memory location `[ebp-736]` before instruction (2). As a result, an arbitrary value is transferred into `eax` at (2). Thus, instruction (3) accessing memory through `eax` can cause a segmentation fault. This error can be statically detected by BinPointer, because it produces an empty value set (the $\perp$ value in our lattice) for `eax` at instruction (3). In contrast, BPA's underlying pointer analysis cannot detect this error due to its over-approximation on stack memory accesses. This result shows a potential application of BinPointer to detecting uninitialized memory accesses.

9 Conclusion

Precise pointer analysis is the key to analyzing binaries, yet it is challenging to design precise, sound and practical solutions. We propose BinPointer, adopting a block memory model and a 0-base abstraction. They enable us to increase the precision of pointer analysis without losing soundness. Our abstractions are formalized by inference rules and the implementation is modular thanks to Datalog, which also enables high performance. We also introduce new strategies for evaluating pointer analysis for binaries through dynamic analysis, which is implemented to produce runtime memory reference traces that are friendly to validating static pointer analysis. This allows us to compute precision and recall rates of pointer analysis.

Acknowledgments

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

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *12th ACM Conference on Computer and Communications Security (CCS)*. 340–353. https://doi.org/10.1145/1102120.1102165

[2] National Security Agency. 2017. Ghidra Reverse Engineering Tool. https://www.nsa.gov/resources/everyone/ghidra/.

[3] Lars Ole Andersen. 1994. *Program Analysis and Specialization for the C Programming Language*. Ph. D. Dissertation. University of Copenhagen.

[4] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. 2005. CodeSurfer/x86—A platform for analyzing x86 executables. In *International Conference on Compiler Construction (CC)*. Springer, 250–254. https://doi.org/10.1007/978-3-540-31985-6_19

[5] Gogul Balakrishnan and Thomas Reps. 2004. Analyzing Memory Accesses in x86 Executables. In *International Conference on Compiler Construction (CC)*. 5–23.

[6] George Balatsouras and Yannis Smaragdakis. 2016. Structure-sensitive points-to analysis for C and C++. In *International Static Analysis Symposium*. 84–104. https://doi.org/10.1007/978-3-662-53413-7_5

[7] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 243–262. https://doi.org/10.1145/1640089.1640108

[8] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *Computer Aided Verification (CAV)*. 463–469. https://doi.org/10.1007/978-3-642-22110-1_37

[9] Christopher L. Conway, Dennis Dams, Kedar S. Namjoshi, and Clark Barrett. 2008. Pointer Analysis, Conditional Soundness, and Proving the Absence of Errors. In *15th International Symposium on Static Analysis*. 62–77. https://doi.org/10.1007/978-3-540-69166-2_5

[10] Neville Grech and Yannis Smaragdakis. 2017. P/Taint: unified points-to and taint analysis. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–28. https://doi.org/10.1145/3133926

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

[12] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On synthesis of program analyzers. In *International Conference on Computer Aided Verification*. Springer, 422–430.

[13] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid context-sensitivity for points-to analysis. *ACM SIGPLAN Notices* 48, 6 (2013), 423–434. https://doi.org/10.1145/2499370.2462191

[14] Sun Hyoung Kim, Cong Sun, Dongrui Zeng, and Gang Tan. 2021. Refining Indirect Call Targets at the Binary Level. In *Network and Distributed System Security Symposium (NDSS)*. https://doi.org/10.14722/ndss.2021.24386

[15] C. Lattner, A. Lanharth, and V. Adve. 2007. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical for the Real World. In *Programming Language Design and Implementation (PLDI)*. 278–289. https://doi.org/10.1145/1273442.1250766

[16] Xavier Leroy and Sandrine Blazy. 2008. Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations. *Journal of Autom. Reasoning* 41, 1 (2008), 1–31. https://doi.org/10.1007/s10817-008-9099-0

[17] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with

dynamic instrumentation. In *Programming Language Design and Implementation (PLDI)*. 190–200. https://doi.org/10.1145/1064978.1065034

[18] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. 2012. RockSalt: Better, Faster, Stronger SFI for the x86. In *Programming Language Design and Implementation (PLDI)*. 395–404. https://doi.org/10.1145/2345156.2254111

[19] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy (S&P)*. 138–157. https://doi.org/10.1109/SP.2016.17

[20] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-sensitivity. In *38th ACM Symposium on Principles of Programming Languages (POPL)*. 17–30. https://doi.org/10.1145/1926385.1926390

[21] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective Analysis: Context-Sensitivity, across the Board. In *Programming Language Design and Implementation (PLDI)*. 485–495. https://doi.org/10.1145/2666356.2594320

[22] Bjarne Steensgaard. 1996. Points-to Analysis in Almost Linear Time. In *23rd ACM Symposium on Principles of Programming Languages (POPL)*. 32–41. https://doi.org/10.1145/237721.237727

[23] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-Flow Analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. 265–266. https://doi.org/10.1145/2892208.2892235

[24] Yulei Sui, Ding Ye, and Jingling Xue. 2014. Detecting Memory Leaks Statically with Full-Sparse Value-Flow Analysis. *IEEE Transactions on Software Engineering* 40, 2 (2014), 107–122. https://doi.org/10.1109/TSE.2014.2302311

[25] Gang Tan and Trent Jaeger. 2017. CFG Construction Soundness in Control-Flow Integrity. In *ACM SIGSAC Workshop on Programming Languages and Analysis for Security (PLAS)*. 3–13. https://doi.org/10.1145/3139337.3139339

[26] Dongrui Zeng, Ben Niu, and Gang Tan. 2021. MazeRunner: Evaluating the Attack Surface of Control-Flow Integrity Policies. In *The 20th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom 2021)*. IEEE.

[27] Dongrui Zeng and Gang Tan. 2018. From Debugging-Information Based Binary-Level Type Inference to CFG Generation. In *8th ACM Conference on Data and Application Security and Privacy (CODASPY)*. 366–376. https://doi.org/10.1145/3176258.3176309

[28] Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wen-chuan Lee, Yonghwi Kwon, Yousra Aafer, and Xiangyu Zhang. 2021. OSPREY: Recovery of Variable and Data Structure via Probabilistic Analysis for Stripped Binary. In *2021 IEEE Symposium on Security and Privacy (SP)*. 813–832. https://doi.org/10.1109/SP40001.2021.00051

[29] Zhuo Zhang, Wei You, Guanhong Tao, Guannan Wei, Yonghwi Kwon, and Xiangyu Zhang. 2019. BDA: Practical Dependence Analysis for Binary Executables by Unbiased Whole-Program Path Sampling and per-Path Abstract Interpretation. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–31. https://doi.org/10.1145/3360563