UNIVERSITY OF CALIFORNIA
UC RIVERSIDE

CS 250 Software Security

Static Binary Analysis

Static Binary Analysis

- Analyzing binaries without running them.
  - No source code, no types, no variable names.
- Challenges:
  - Unknown memory layout
  - Indirect control flow
  - Instruction decoding

- Applications:
  - Vulnerability discovery
  - Malware analysis
  - Binary hardening

Value Set Analysis

- Gogul Balakrishnan and Thomas Reps, “Analyzing memory accesses in x86 executables”, Compiler Construction 2004.

- Foundation for many modern binary analysis tools.

Why Value Set Analysis (VSA)?

- In binaries, we must reason about addresses, pointers, and memory values.

- Key idea:
  - Track the set of possible values for each register or memory location.

- VSA helps with:
  - Resolving indirect jumps/calls
  - Reconstructing control flow
  - Understanding memory accesses

Abstract Domain

- Static analysis needs to track where values live and what values they may have

- Three layers:
  - Memory Regions: Coarse partitioning of memory
  - A-Locs (Abstract Locations): Logic memory objects within regions
  - Abstract Store: Maps a-loc to value sets

Memory Regions

- Also called Abstract Regions (AR)

- Coarse Regions
  - Stack regions (local variables, return address): one for each function
  - Global data regions (static and global variables)
  - Heap regions (allocated by malloc, new): one for each allocation site
  - Memory-mapped regions

Abstract Locations (a-locs)

- A-locs represent logical memory objects within a memory region
- A pair: (AR, offset)
- Examples
  - A local variable inside a main function: (AR_main, -12)
  - A static global variable: (AR_Global, 0x10020)
  - An integer in a dynamic allocated heap: (AR_Heap_0x11030, 12)

Abstract Stores

- Abstract Store:
  - Maps each a-loc to a value set representing possible values
- Value Set:
  - Encoded as strided intervals: [low : high : stride]

- Example:
  - [0x1000 : 0x2000 : 4] → 0x1000, 0x1004, ..., 0x2000

Basic VSA Algorithm

1. Initialize registers and memory.
2. Apply instruction semantics
   - E.g., mov eax, 0x1000 -> EAX = [0x1000: 0x1000: 1]
   - E.g., add eax, 0x10 -> EAX = [0x1010: 0x1010: 1]
3. Merge value sets at control flow joins.
4. Iterate to fixed-point.
5. Use widening for termination.

Transfer Functions

- Transfer Functions model how each instruction updates the abstract store
- Key types of operations:
  - Move (MOV):
    - Transfer value set from source to destination
  - Arithmetic (ADD, SUB):
    - Shift value sets appropriately
  - Memory Load (MOV reg, [addr])
    - Dereference the value set at the memory a-loc
  - Memory Store (MOV [addr], reg)
    - Update the value set at the memory a-loc with the register's value set
  - Conditional Branch
    - Split paths and conditionally update value sets

Handling Relational Conditions

- Relational branches R1<=c, R1>=R2 refine value sets
- True branch:
  - Constrain value sets so that the condition holds
- False branch:
  - Constrain value sets so that the negation of the condition holds
- Example:
  - R1 <=c:
    - True branch: R1 = [MIN, c]
    - False branch R1 = [c+1, MAX]
  - R1 >= R2:
    - True branch: R1 = [min(R2), MAX]; R2=[MIN, max(R1)]
    - False branch: R1 = [MIN, max(R2)-1]; R2=[min(R1)+1, MAX]
- Refinement improves analysis precision

Example Walkthrough

```
start:
    mov eax, 0x1000
    cmp [eax], 0
    je  skip
    add eax, 0x100
skip:
    mov ecx, [eax]
    jmp ecx
```

- Path 1: EAX = [0x1010: 0x1000: 1]
- Path 2: EAX = [0x1100: 0x1100: 1]
- After merging: EAX = [0x1000, 0x1100: 0x100]
- ECX: Memory[EAX]
- Indirect Jump: Depends on ECX's value set.

Key Challenges in VSA

- Precision vs. Efficiency
  - VSA uses approximations (i.e., strided intervals) to summarize large sets of possible values
  - Over-approximation avoids path-sensitive or context-sensitive tracking
  - Still not very efficient for large binaries
- Memory Aliasing
  - Memory aliasing occurs when different computed addresses may refer to overlapping memory regions.
  - VSA handles aliasing by conservatively merging memory effects when address ranges overlap.
  - This merging reduces precision, as unrelated memory accesses can become conflated, leading to imprecise results.
- Complex Pointer Arithmetic
  - Tracking dynamic computations (e.g., base + scaled index + offset) is difficult.
  - Pointer arithmetic can cause value sets to grow and become harder to model accurately.

Applications of VSA

- Control Flow Recovery
  - Resolve indirect jumps)
- Decompilation:
  - Recover high-level structure
- Security Analysis:
  - Detect memory safety vulnerabilities
- Binary Rewriting
  - Enable patching and instrumentation)

Anecdotal Experiences with VSA

- Strengths
  - Highly effective at resolving indirect jumps (jump tables, virtual calls)
  - Drives powerful static taint analysis and vulnerability detection
  - Helps prune infeasible path in symbolic execution
- Challenges
  - Loops cause fast widening and loss of precision
  - Memory aliasing inflates value sets and introduces false positives
  - Dynamically loaded libraries and unknown functions complete memory modeling
  - Aggressive compiler optimizations hinder precise VSA

Limitations and Extensions

- Flow-sensitive but often context-insensitive

- Extensions:
  - Combine with Symbolic Execution:
    - VSA identifies broad possible value ranges quickly
    - Symbolic execution selectively refines paths and memory access where VSA's over-approximation is too coarse.
    - Help avoid unnecessary full symbolic execution, improving overall scalability and precision
- Add Predicate Reasoning:
  - Track simple logical relationships between variables (e.g., inequalities)
- Context-sensitive VSA
  - Track different calling contexts separately to improve analysis precision.

Refine Indirect Call Target at the Binary Level

NDSS 2021

Overview

- Goal: Build precise CFGs by refining indirect call targets

- Method
  - Use Andersen's pointer analysis algorithm with block memory model
  - Partition memory into blocks for easier pointer tracking
  - Recursively update CFG based on refined points-to sets

- Results:
  - Achieved higher precision and better scalability than VSA

Block Memory Model

- Recall Abstract Regions in VSA
  - Global, Heap, and Stack
  - One stack frame is one block
  - One heap object allocated at one callsite is one block
  - The global region is partitioned into multiple blocks
- Assume pointer arithmetic won't cross block boundary
- All pointers within one block is treated as one
  - In other words, A-loc has no offset

Anderson-Style Pointer Analysis

- Inclusion based:
  - If variable x can point to anything y points to, then Pts(x) $\supseteq$ Pts(y)
- Points-to graph
  - Nodes =variables, memory locations
  - Edges="may point to" relations
- Rules:

Operation	Constraint
`x = y`	`Pts(x) ⊇ Pts(y)`
`x = &y`	`Pts(x) ⊇ {y}`
`*x = y`	For all `l ∈ Pts(x)` : `Pts(l) ⊇ Pts(y)`
`x = *y`	For all `l ∈ Pts(y)` : `Pts(x) ⊇ Pts(l)`

- Result: find a fixed-point where all inclusion constraints are satisfied.

How BPA adapts Andersen's algorithm

Aspect	BPA Strategy
Variables	Registers + Memory Blocks (A-locs)
Memory abstraction	Use **block memory model** to avoid per-byte tracking
Constraints extracted	From disassembly (loads, stores, moves)
Pointer dereference	Abstract *x = y and x = *y over blocks + offsets
Handling function calls	Interprocedural modeling (with function summaries or recursion)
Solving	Use a **Datalog solver** for scalability

Context-insensitive, flow-insensitive, field-insensitive pointer analysis algorithm

Evaluation Results

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	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

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

Discussion

- Block Model
  - Fundamentally imprecise
- Soundness guarantee
  - Global data partitioning maybe unsound
  - The algorithm is sound