Falcon

A Fused Approach to Path-Sensitive Sparse Data Dependence Analysis

Peisen Yao, Zhejiang University

Jinguo Zhou, Ant Group

Xiao Xiao*, Ant Group (Presenter)

Qingkai Shi, Nanjing University

Rongxin Wu, Xiamen University

Charles Zhang, HKUST

A History of CODA Project

2018

- Pinpoint [PLDI'18] — Holistic Value Flow Analysis

2019

- Smoke [ICSE'19] — Memory Leak Detection

2020

- Catapult [ICSE'20] — Extensional Scalability
- Coyote [ICSE'20] — Pipelining
- Trident [ISSTA'20] — Bit-Vector Satisfiability

2021

- Canary [PLDI'21] — Inter-Thread Value-Flow Analysis
- Fusion [PLDI'21] — Sparse Analysis without Path Conditions
- Transcode [ASE'21] — Specification Inference

2022

- Dyck Indexing [OOPSLA'22] — Extended Dyck-CFL Reachability
- Cres [OOPSLA'22] — Container Synthesis
- Wit [ICSE'22] — Divide-By-Zero Detection
- Peahen [ICSE'22] — Deadlock Detection

2023

- Coral [OOPSLA'23] — Call Graph Construction
- Lockpick [USENIX Security'23] — Lock Misuse Bug
- Anchor [TOSEM'23] — Container Value-Flow Analysis

2024

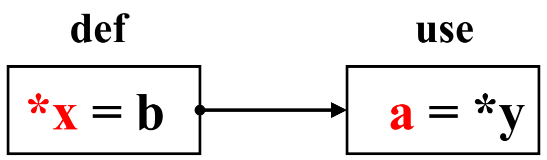
- DAInfer [FSE'24] — Infer Alias Spec via LLM
- Octopus [TOSEM'24] — Parallel Value-Flow Analysis
- Cod [ISSTA'24] — Buffer Overflow Bug Finding
- Current Paper [PLDI'24] — Data Dependence Analysis
- LibAlchemy [ICSE'24] — Static Analysis with Persistence

Data Dependence Analysis

Answer the def-use related queries

Does the value of a rely on the value of b?

```
*x = b
 y = f(x)
 a = *y
```



Problem Statement

High Precision

Build inter-procedural path-sensitive value-flow graph

High Efficiency

Analyze millions of lines of code within 2 hours

High Scalability

Analyze millions of lines of code within 64 GB

Challenge: Aliasing-Path-Explosion Problem

- Assignments to and from indirect memory locations complicate path conditions by the disjunction of the conditions of assignment value, points-to, and statement location in the **Cartesian Product** manner.

$\varphi_1 \quad \varphi_2 \quad \varphi_3 \quad \varphi_4$

$\mathrm{PTS}(\,p\,) = \{\, \varphi_1 : o_1, \varphi_2 : o_2 \,\}$

$\mathrm{PTS}(\,q\,) = \{\, \varphi_3 : o_3\,, \varphi_4 : o_4 \,\}$

σ : *p = q

$$[\,\sigma\,] \times [\,\varphi_1, \varphi_2\,] \times [\,\varphi_3, \varphi_3\,]$$

Path conditions with massive redundancy !!

Existing Works

Neither of existing approaches scale to millions of lines of code.

Bootstrapped Approach	Layered Approach
Symbolic execution such as Focal	Sparse analysis such as SVF
Use caching, pruning, simplification, and searching heuristics to speedup SAT solving	Enable sparse path-sensitive analysis with pre-computed path-insensitive results
Still too many SAT queries and results are represented in a dense manner	Introduce too many spurious value-flow paths and hurt performance and scalability

Falcon Key Design

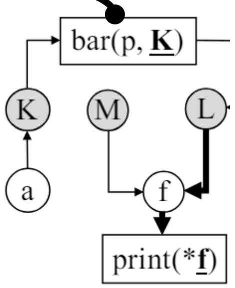
❷ Inline Points-to Sets and Instantiate Symbolic Objects

❶ On-the-Fly Sparse and Semi-Path Sensitive Intra-procedural analysis

Program

Sparse Points-to Sets

\+



Sparse VFG (SEG) with Symbolic Objects [PLDI'18]

❸ Formal/actual matching for SEGs

❹ On-demand Context-sensitive Data Dependence Searching

Client Applications

Stitched SEGs
(Similar to exploded super graph in IFDS)

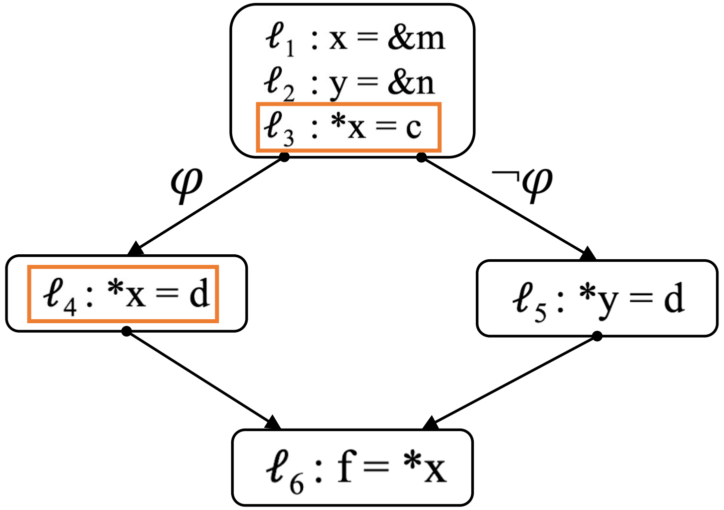
Intra-procedural Analysis

On-the-Fly sparsity

Sparse value-flow graph construction and pointer analysis performed together

Key Idea

A memory location defined at a program point ℓ can only be used at program points dominated by ℓ



Store Rule

$\mathbb{S}_{\ell_3}(\text{alloc}_m) = \{ (\text{true}, \ell_3, c) \}$

$\mathbb{S}_{\ell_4}(\text{alloc}_m) = \{ (\varphi, \ell_4, d) \}$

Dominance Frontiers

{ Store Mapping@l6 }

Load Rule

$\mathbb{S}_{\ell_4}(\text{alloc}_m)$

$\mathbb{S}_{\ell_3}(\text{alloc}_m)$

$\mathbb{S}_{\ell_3}(\text{alloc}_m)$

Walk up the dominator tree

Query value of m@l6?

Query value of m@l5?

Intra-procedural Analysis

Semi-path-sensitive

70% constraints are satisfiable
90% of unsatisfiable constraints are easy to solve

Key Idea

Semi-Decision Procedure

- Solve **easy** constraints that can be determined UNSAT in linear time
- Boolean abstraction + Semi- decision procedure

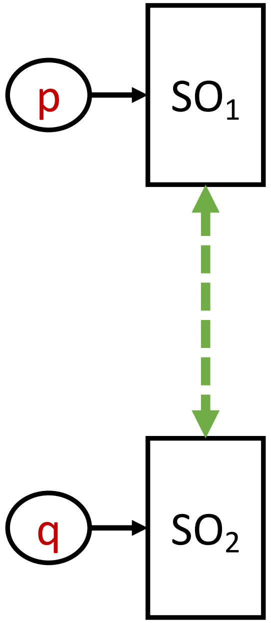
Programs	#SAT Queries	#UNSAT (All)	#UNSAT (Easy)
transmission	26996	6926 (25.7%)	6696 (96.7%)
rats	23897	8297 (34.7%)	8264 (99.6%)
curl	12957	4528 (34.9%)	4463 (98.6%)

Inter-procedural Analysis

- Inline the callee's side-effects of the points-to structure into the caller
- Mark the aliased symbolic objects at the call sites

```
bar(p) {
  *p = 2;
}


foo(q) {
    *q = 1;
L:  bar(q);
    z = *q;
}
```



- Use Symbolic Object (SO) for memory locations accessed by pointers of the formal parameters.
- A way for implementing storeless memory model.

- Mark the symbolic objects SO_1 and SO_2 aliased at the call site L to ***stitch*** the SEGs of foo and bar.
- Create a def of SO_2 after the call site L and perform the ***store rule***.

On-demand Context-Sensitive Searching

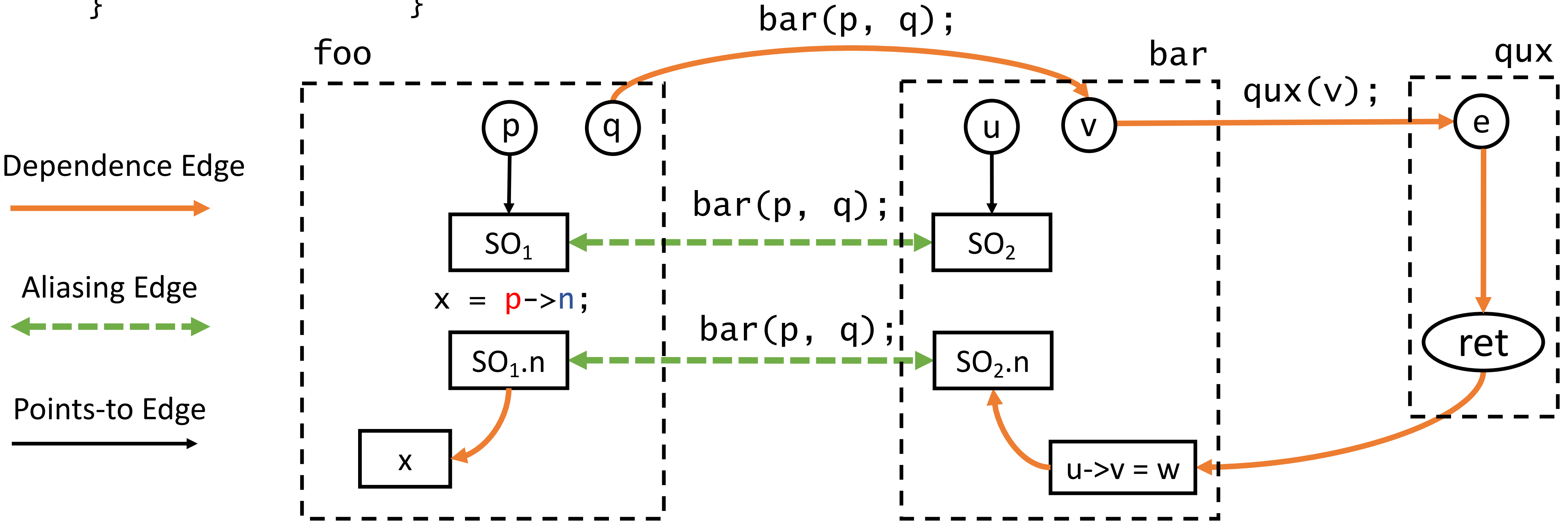
[Query] **Which variable does x depend on?**

```
foo(p, q) {
   bar(p, q);
L: x = p->n;
}
```

```
bar(u, v) {
   w = qux(v);
   u->n = w;
}
```

```
qux(e) {
  return e;
}
```

[Answer] **x depends on q**



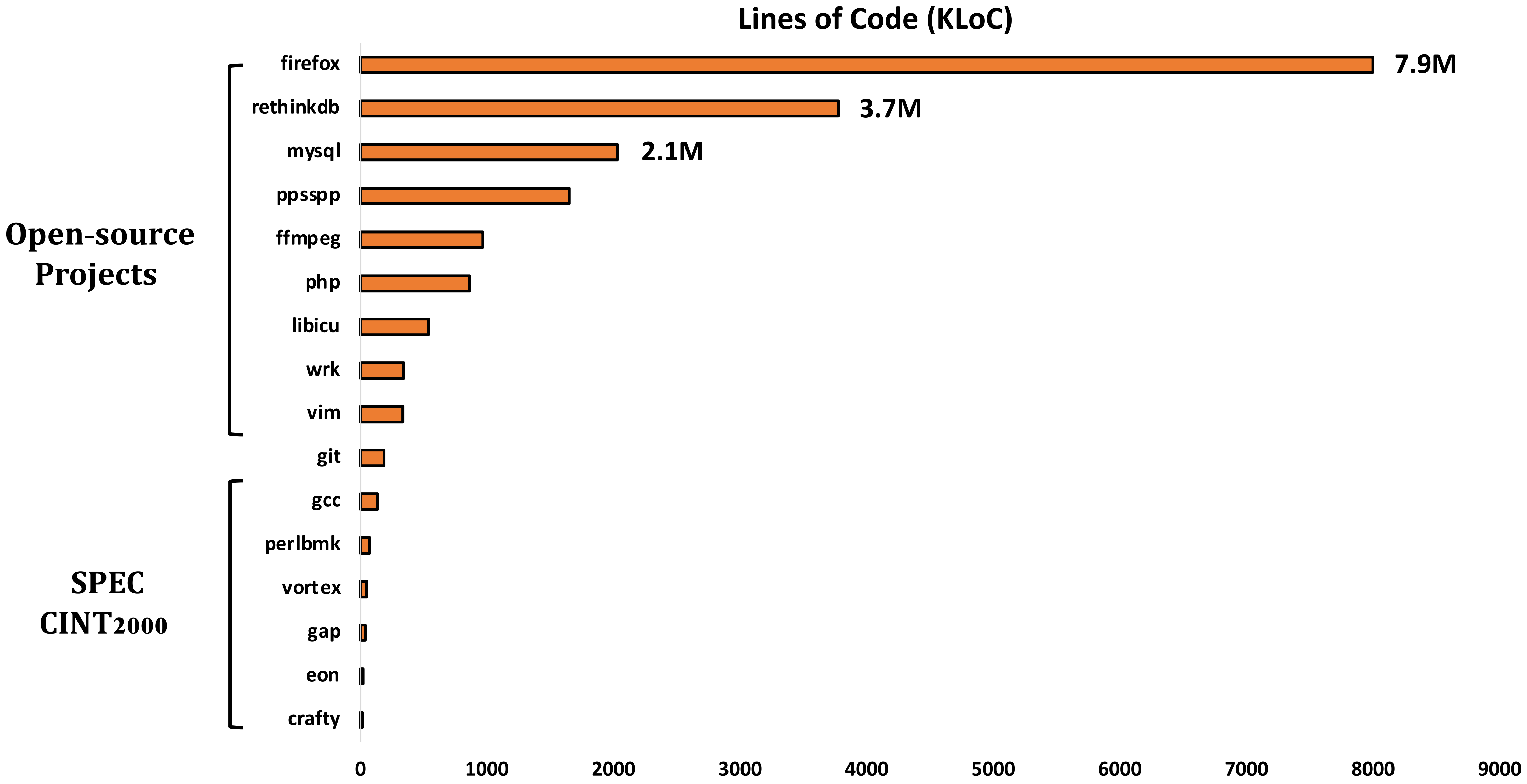
Evaluation Setup

- Implementation
  - Build on top of LLVM and Z3 SMT solver
  - Support most C/C++ features such as unions, arrays, and classes

- Environment
  - 64-bit machine with 40 CPUs@2.20 GHz and 256 GB RAM

- Experiments
  - Value-flow graph construction
  - Thin slicing for program understanding
  - Use-after-free bug detection

Benchmarks



Lines of Code (KLoC)
Open-source Projects
SPEC CINT2000
firefox
rethinkdb
mysql
ppsspp
ffmpeg
php
libicu
wrk
vim
git
gcc
perlbmk
vortex
gap
eon
crafty
7.9M
3.7M
2.1M
0
1000
2000
3000
4000
5000
6000
7000
8000
9000

Evaluating Value-flow Graph Construction

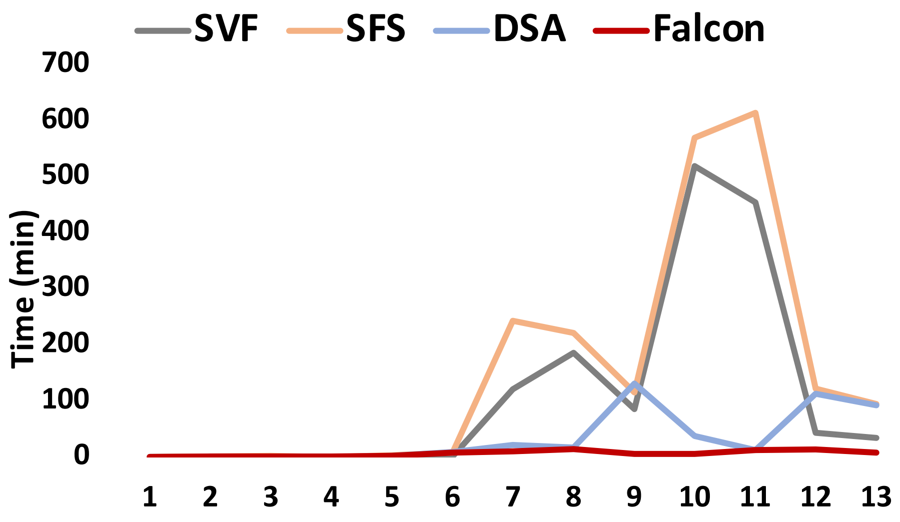
- Goal
  - Examine efficiency and scalability of Falcon for constructing value-flow graphs
- Setting
  - Cutoff time is 12 hours

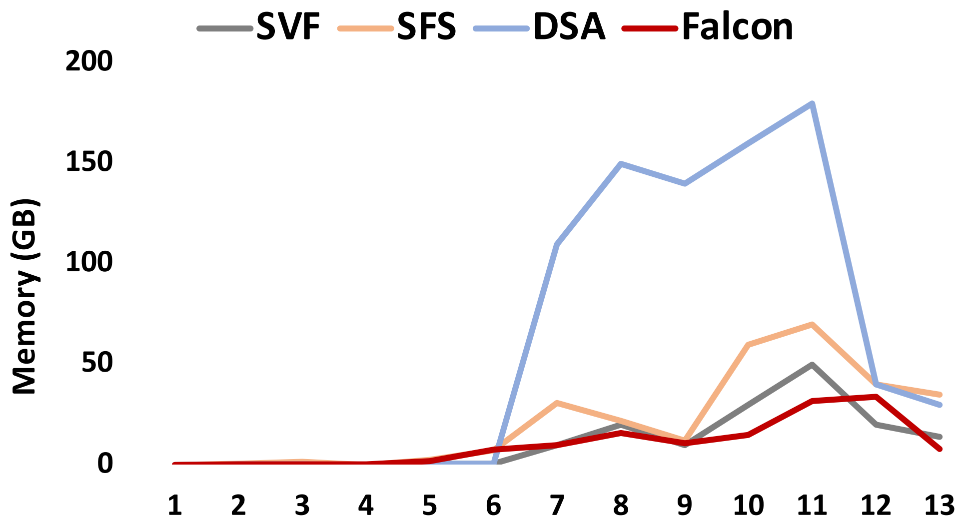
Name	Flow Sensitivity	Context Sensitivity	Exhaustive
SVF [CC'16]	✗	✗	✓
SFS [CGO'10]	✓	✗	✓
SUPA-FS [FSE'16]	✓	✗	✗
SUPA-FSCS [FSE'16]	✓	✓	✗
DSA [PLDI'07]	✗	✓	✓

Evaluating Value-flow Graph Construction

Falcon is More Scalable

- Time: **17×, 25×, 4.4×** faster than SVF, SFS, DSA
- Memory: **1.4×, 1.9×, 4.2×** less memory than SVF, SFS, DSA





- SUPA-FS and SUPA-FSCS only finished analyzing **crafty** and **econ**

Evaluating Thin Slicing

- Goal
  - Measure efficiency and precision of semi-path-sensitive value-flow graphs

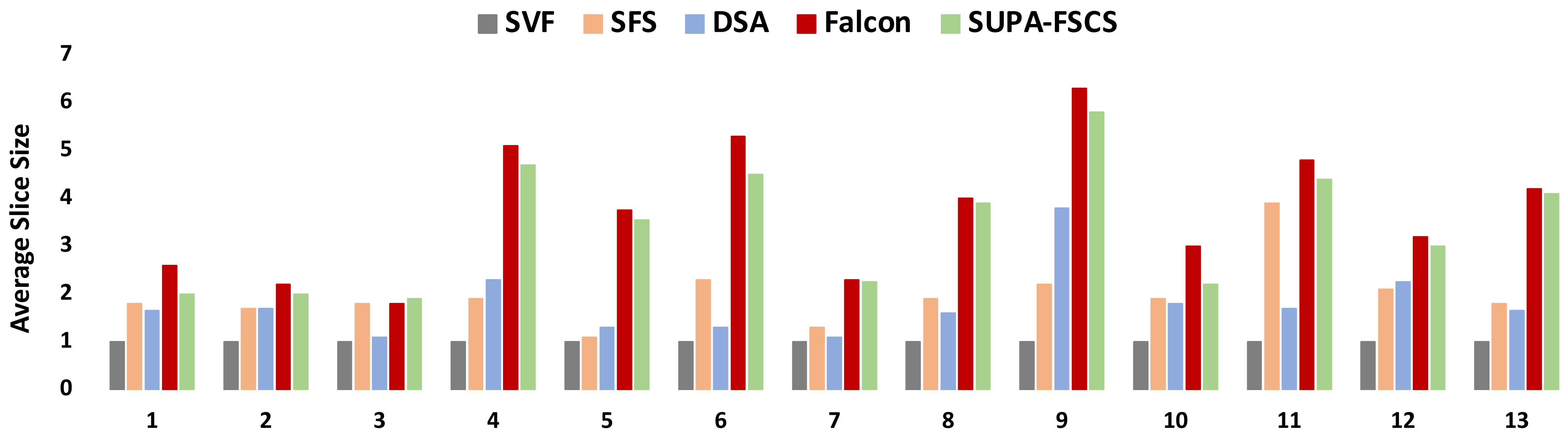
- Setting
  - Exclude time for building value-flow graphs
  - Slicing queries are derived from realistic third-party typestate analysis

- Compare to the same tools as in before

Evaluating Thin Slicing

Falcon is More Efficient and Precise on the premise of Soundiness

- Efficiency: up to **302×** faster than SUPA-FSCS and **54×** on average
- Precision: produce **5.5×, 1.9×, 2.6×, 1.3×** smaller slices than SVF, SFS, DSA, SUPA-FSCS



Evaluating Use-after-free detection

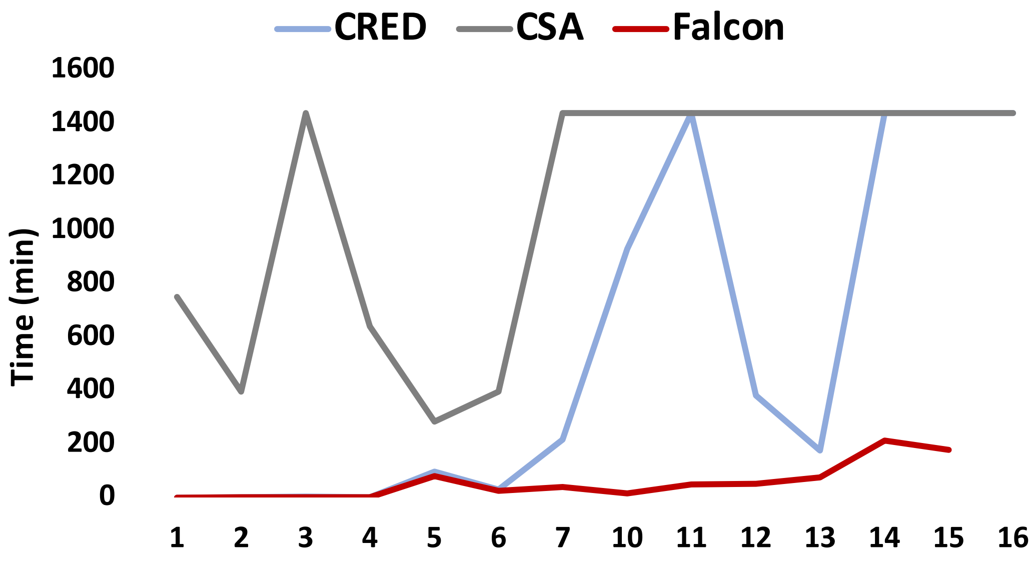
- Goal
  - Investigate efficiency and effectiveness of Falcon for value-flow bug finding
- Setting
  - 15-second time limit for each SMT query
  - Run in single-thread mode with a cutoff time of 24 hours

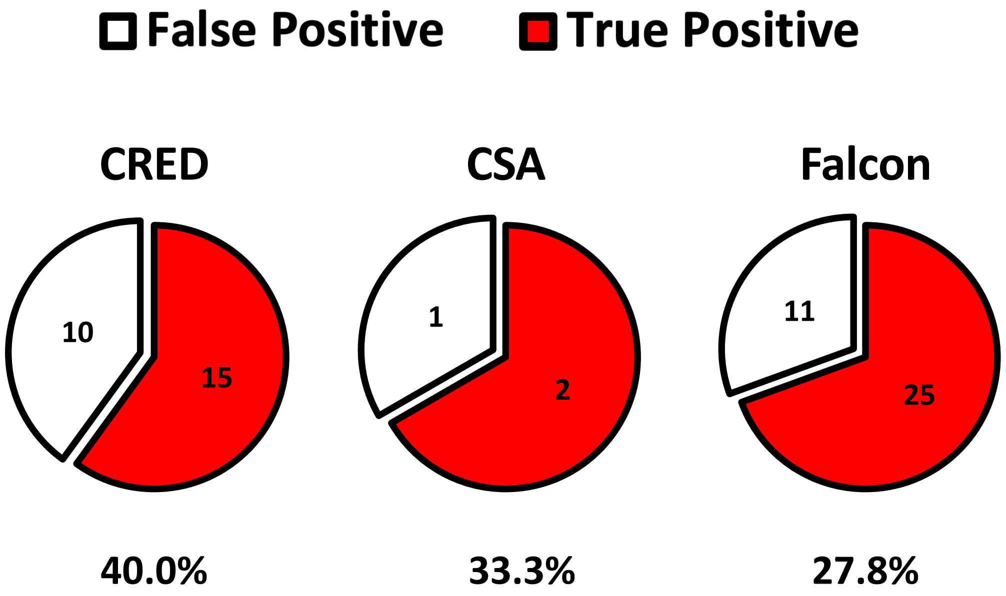
Name	Type	Path Sensitivity
CRED [ICSE'18]	(Layered) Pointer Analysis	Full Path Sensitivity
Clang Static Analyzer (with Z3)	(Bootstrapped) Symbolic Executor	Full Path Sensitivity
Falcon (with Pinpoint)	(Fused) Data Dependence Analysis	Full Path Sensitivity

Evaluating Use-after-free detection

Falcon is More Efficient with Lower False Positive

- Efficiency: **10.3×, 1620.8×** faster than CRED and CSA
- False Positive: **40.0%, 33.3%, 27.8%** for CRED, CSA, Falcon
  - Align with the common industrial requirement of **30%** false positives





Conclusion

❶ On-the-fly sparse

❷ Semi-path-sensitive

❸ On-demand searching

Q & A