Precise Sparse Abstract Execution via Cross-Domain Interaction

ICSE 2024

Xiao Cheng, Jiawei Wang, Yulei Sui

xiao.cheng@unsw.edu.au

Computer Science and Engineering
UNSW Sydney

April 24, 2024

Contribution

- A precise **cross-domain abstract execution/interpretation** over a combined domain through **correlation tracking**.

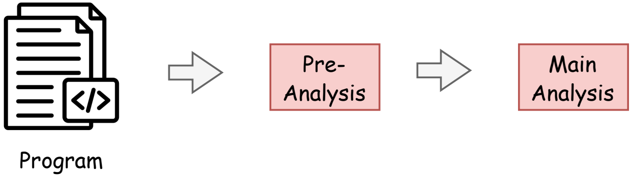
Contribution

- A precise **cross-domain abstract execution/interpretation** over a combined domain through **correlation tracking**.
- An **implication-equivalent (virtual) memory address grouping approach**.

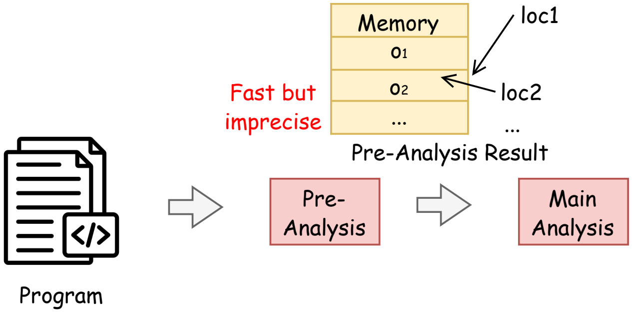
Contribution

- A precise **cross-domain abstract execution/interpretation** over a combined domain through **correlation tracking**.
- An **implication-equivalent (virtual) memory address grouping approach**.
- Significantly **boost the precision and efficiency of assertion-checking clients**, e.g., buffer overflow and null dereference detection.

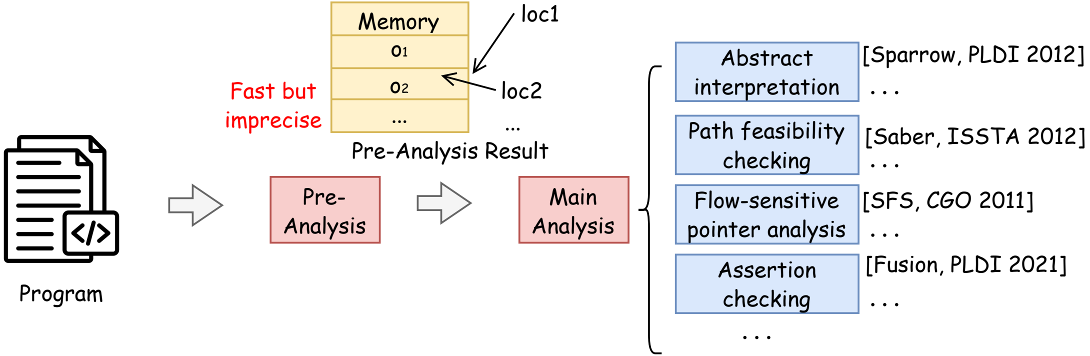
Staged Analysis



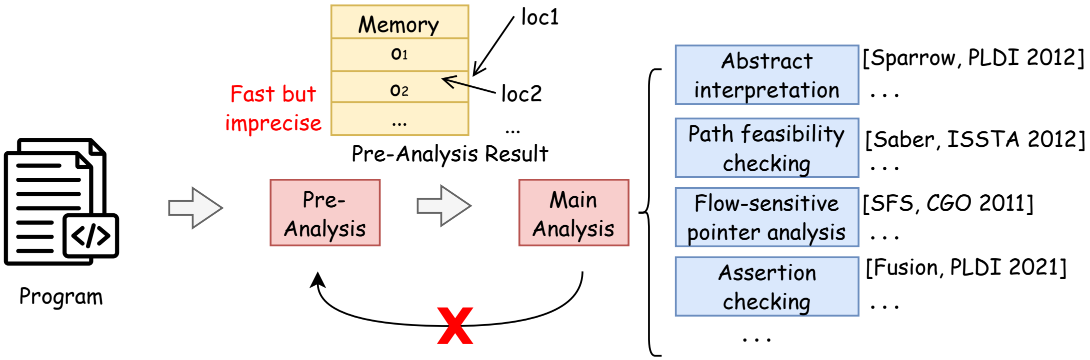
Staged Analysis



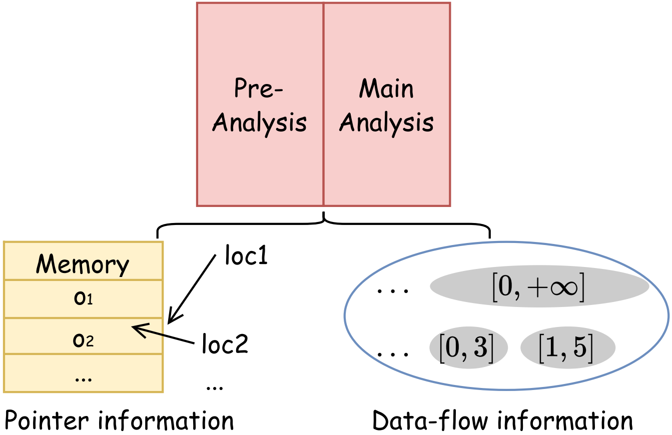
Staged Analysis



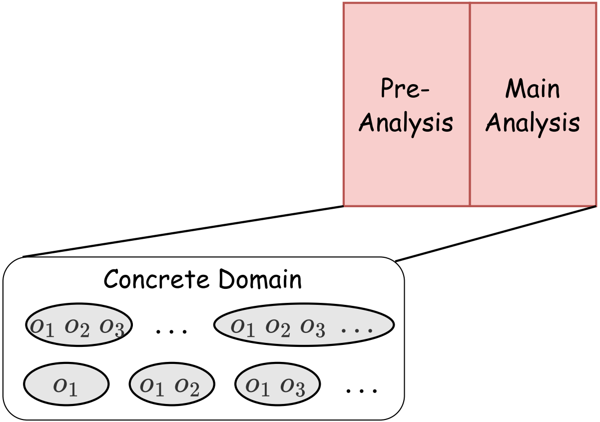
Staged Analysis



Combined Analysis

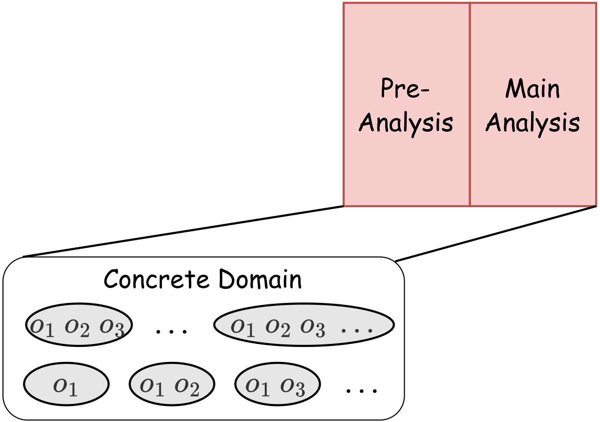


Combined Analysis with Concrete Domain



One concrete domain for both analyses?

Combined Analysis with Concrete Domain



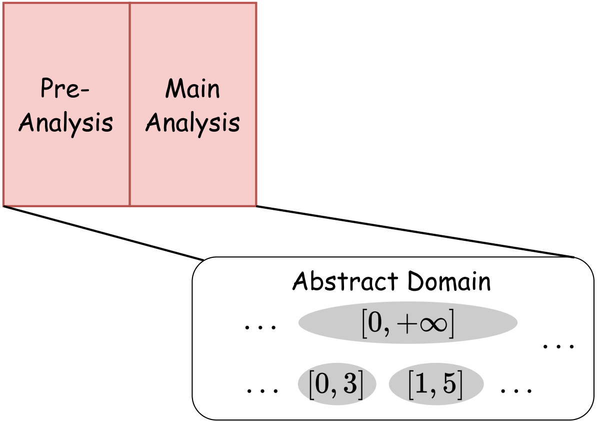
One concrete domain for both analyses?

{1, 2, 3, ..., infinit numbers}

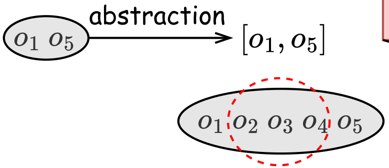
Unscalable!

Combined Analysis with Abstract Domain

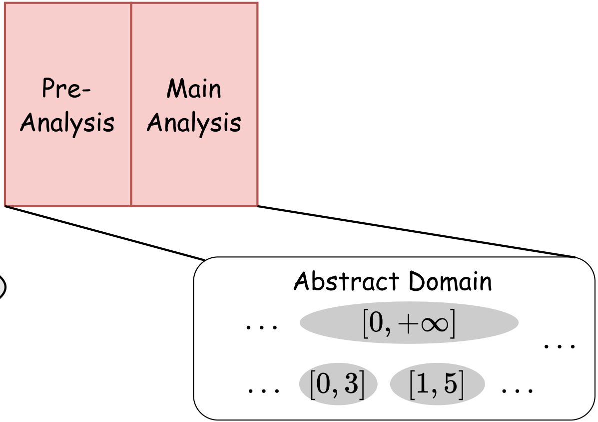
One abstract domain for both analyses?



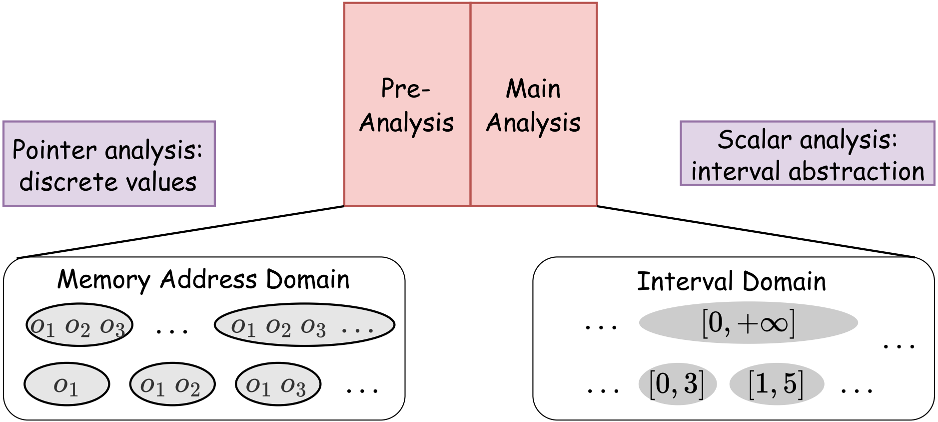
One abstract domain for both analyses?



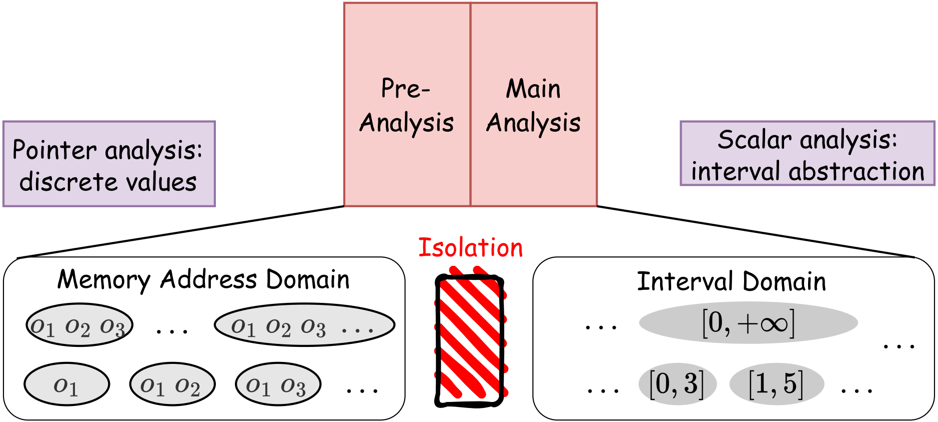
Imprecise!



Combined Analysis with Combined Domain

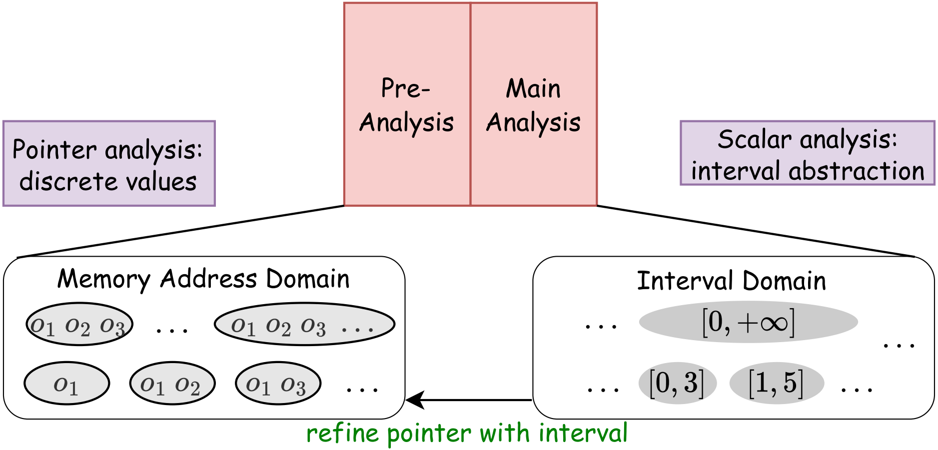


Combined Analysis with Combined Domain

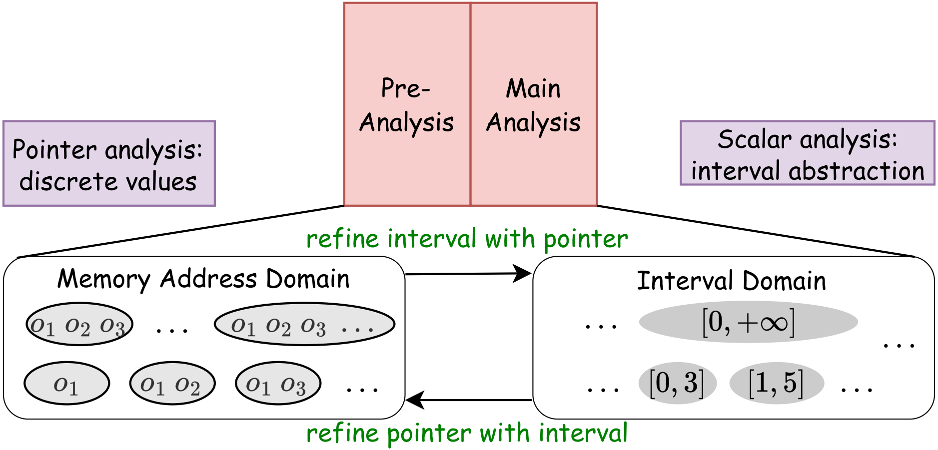


Precision loss without cross-domain interaction

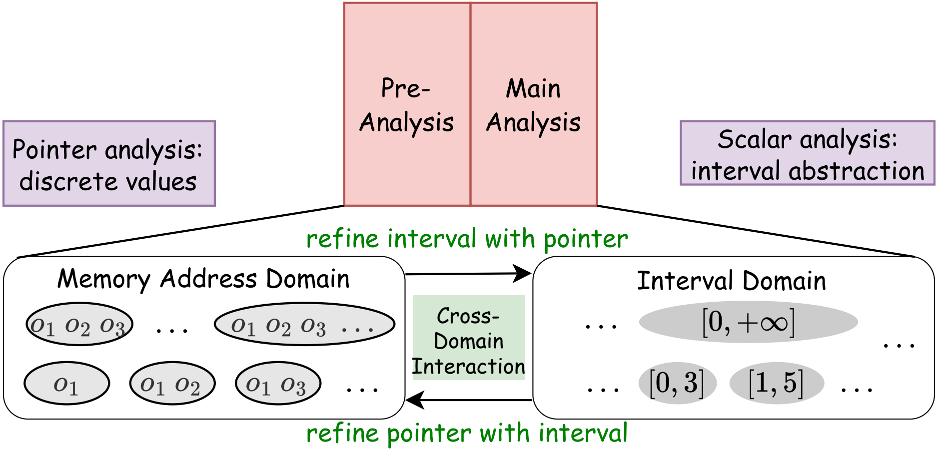
Cross-Domain Online Refinement



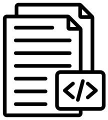
Cross-Domain Online Refinement



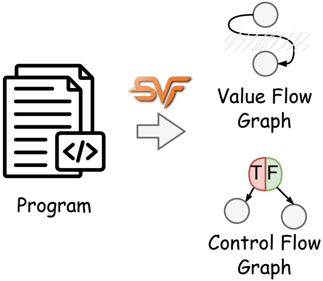
Cross-Domain Online Refinement



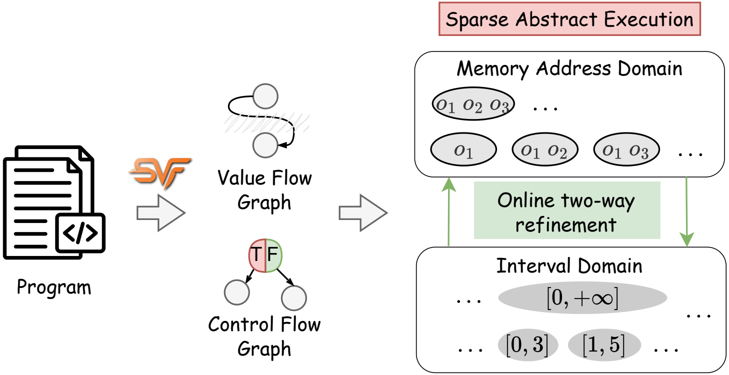
Framework Overview



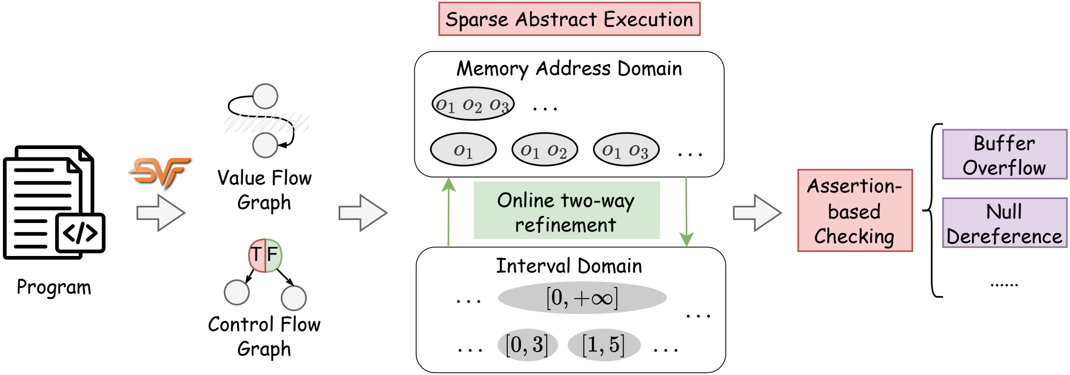
Program


Program
Value Flow Graph
T
F
Control Flow Graph

Framework Overview



Framework Overview



LLVM-like Language

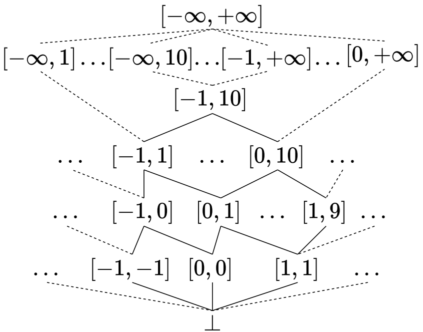
c,fld	$\in \mathcal{C}$	Constants
p,q,r	$\in \mathcal{S}$	Stack virtual registers
g	$\in \mathcal{G}$	Global pointer variables
p,q,r,g	$\in \mathcal{P} = \mathcal{S} \cup \mathcal{G}$	Top-level variables
o, a, a_f, a.fld, a[c]	$\in \mathcal{O}$	Abstract objects
v	$\in \mathcal{V} = \mathcal{P} \cup \mathcal{O}$	Program variables

ℓ ::=		Stmt
	p = c	ConsStmt
	p = alloc_o	AddrStmt
	p = &(q→fld)	GepStmt (Field)
	p = &q[c] (constant)	GepStmt (Array-C)
	p = &q[v] (variable)	GepStmt (Array-V)
	p = *q	LoadStmt
	*p = q	StoreStmt
	p = q	CopyStmt
	p = phi(p_1, p_2, ...p_n)	PhiStmt
	p = ¬q	UnaryStmt
	r = p ⊙ q	BinaryStmt

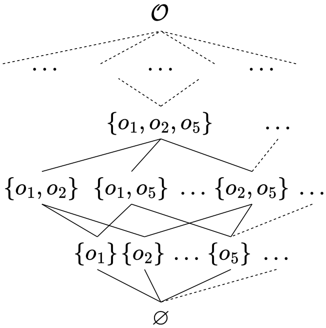
$\odot \in \{+, -, *, /, \%, <<, >>, <, >, \&, \&\&, <=, >=, \equiv, \sim, |, \wedge\}$

Interval and Memory Address Domain

- Interval abstraction (*Interval* domain) for scalar variables.
- Discrete values (*MemAddress* domain) for memory addresses.



(a) Interval domain



(b) Memory address domain

Abstract Trace for Interval and Memory Address Domain

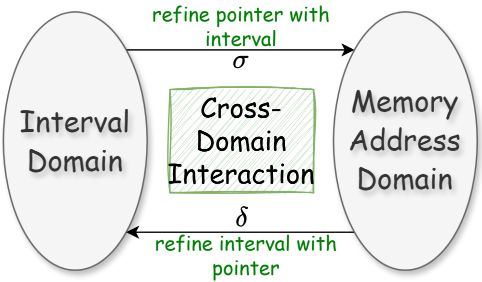
- **Symbol to value mapping**: $\sigma \in \mathcal{P} \rightarrow \mathit{Interval} \times \mathit{MemAddress}$ captures the memory addresses/interval value of top-level pointers/scalar variables.

Abstract Trace for Interval and Memory Address Domain

- **Symbol to value mapping**: $\sigma \in \mathcal{P} \to \mathit{Interval} \times \mathit{MemAddress}$ captures the memory addresses/interval value of top-level pointers/scalar variables.
- **Value to value mapping**: $\delta \in \mathbb{L} \times \mathit{MemAddress} \to \mathit{Interval} \times \mathit{MemAddress}$ captures the correlation between memory objects and memory addresses/interval values at different program locations.

Abstract Trace for Interval and Memory Address Domain

- **Symbol to value mapping**: $\sigma \in \mathcal{P} \rightarrow \mathit{Interval} \times \mathit{MemAddress}$ captures the memory addresses/interval value of top-level pointers/scalar variables.
- **Value to value mapping**: $\delta \in \mathbb{L} \times \mathit{MemAddress} \rightarrow \mathit{Interval} \times \mathit{MemAddress}$ captures the correlation between memory objects and memory addresses/interval values at different program locations.



Analysis Rules

SVFStmt	C-Like form	Abstract Execution Rule
ConsStmt	$\ell : \texttt{p} = \texttt{c}$	$\sigma(\texttt{p}) := \langle[\texttt{c},\texttt{c}], \top\rangle$
CopyStmt	$\ell : \texttt{p} = \texttt{q}$	$\sigma(\texttt{p}) := \sigma(\texttt{q})$
BinaryStmt	$\ell : \texttt{r} = \texttt{p} \otimes \texttt{q}$	$\sigma(r) := \sigma(p)\hat{\otimes}\sigma(q)$
PhiStmt	$\ell : \texttt{r} = \texttt{phi}(\texttt{p}_1, \texttt{p}_2, \ldots, \texttt{p}_n)$	$\sigma(r) := \bigsqcup_{i=1}^{n} \sigma(p_i)$
ValueFlow	$\ell' \overset{o}{\hookrightarrow} \ell$	$\delta_{\overline{\ell}}(o) \sqsupseteq \delta_{\underline{\ell'}}(o)$
AddrStmt	$\ell : \texttt{p} = \texttt{alloc}_{\texttt{o}_\texttt{i}}$	$\sigma(\texttt{p}) := \langle\top, \{o_i\}\rangle$
GepStmt	$\ell : \texttt{p} = \&(\texttt{q} \rightarrow \texttt{i})$ or $\texttt{p} = \&\texttt{q}[\texttt{i}]$	$\sigma(\texttt{p}) := \bigsqcup_{\texttt{o}\in\gamma(\sigma(\texttt{q}))} \bigsqcup_{j\in\gamma(\sigma(\texttt{i}))} \langle\top, \{\texttt{o.fld}_j\}\rangle$
LoadStmt	$\ell : \texttt{p} = *\texttt{q}$	$\sigma(\texttt{p}) := \bigsqcup_{o\in\{o \mid (o\mapsto\_)\in\delta_{\overline{\ell}}\}} \delta_{\overline{\ell}}(o)$
StoreStmt	$\ell : *\texttt{p} = \texttt{q}$	$\delta_{\underline{\ell}} \sqsupseteq (\{o \mapsto \sigma(\texttt{q}) \mid o \in \gamma(\sigma(\texttt{p}))\} \sqcup \delta_{\overline{\ell}} \setminus \mathsf{kill}(\ell))$

$$\mathsf{kill}(\ell : *\mathsf{p} = \mathsf{q}) := \begin{cases} \{\mathsf{o} \mapsto \_ \mid \mathsf{o} \in \gamma(\sigma(\mathsf{p}))\} & \text{if } \sigma(\mathsf{p}) \equiv \langle\top, \{\mathsf{o}\}\rangle \wedge \mathsf{o} \text{ is singleton} \\ \{\mathsf{o} \mapsto \_ \mid \mathsf{o} \in \mathcal{O}\} & \text{if } \sigma(\mathsf{p}) \equiv \langle\top, \varnothing\rangle \\ \varnothing & \text{otherwise} \end{cases}$$

An Example: Analysis Rules

ℓ_1 : slot = input()%4
ℓ_2 : loc1 = &ids[slot]
ℓ_3 :   loc2 = &ids[4]

Abstract trace σ

An Example: Analysis Rules

ℓ_1 : slot = input()%4
ℓ_2 : loc1 = &ids[slot]
ℓ_3 : loc2 = &ids[4]

Abstract trace σ

slot	$\langle [0,3], \top \rangle$

An Example: Analysis Rules

ℓ_1 : slot = input()%4

ℓ_2 : loc1 = &ids[slot]

ℓ_3 :   loc2 = &ids[4]

Abstract trace σ

slot	$\langle [0,3], \top \rangle$
loc1	$\langle \top, \{o_1, o_2, o_3, o_4\} \rangle$

An Example: Analysis Rules

ℓ_1 : slot = input()%4

ℓ_2 : loc1 = &ids[slot]

ℓ_3 : loc2 = &ids[4]

Abstract trace σ

slot	$\langle [0,3], \top \rangle$
loc1	$\langle \top, \{o_1, o_2, o_3, o_4\} \rangle$
loc2	$\langle \top, \{o_5\} \rangle$

An Example: Analysis Rules

ℓ_1 : slot = input()%4

ℓ_2 : loc1 = &ids[slot]

ℓ_3 : loc2 = &ids[4]

Abstract trace σ

slot	$\langle [0,3], \top \rangle$
loc1	$\langle \top, \{o_1, o_2, o_3, o_4\} \rangle$
loc2	$\langle \top, \{o_5\} \rangle$

ℓ_4 : *loc1 = INT_MAX

An Example: Analysis Rules

Abstract trace σ

ℓ_1 : slot = input()%4
ℓ_2 : loc1 = &ids[slot]
ℓ_3 : loc2 = &ids[4]

slot	$\langle [0,3], \top \rangle$
loc1	$\langle \top, \{o_1, o_2, o_3, o_4\} \rangle$
loc2	$\langle \top, \{o_5\} \rangle$

Abstract power trace δ

$\delta_{\overline{\ell_4}}$

o_1, o_2, o_3, o_4	$\langle [0,0], \top \rangle$

ℓ_4 : *loc1 = INT_MAX

An Example: Analysis Rules

Abstract trace σ

ℓ_1 : slot = input()%4

ℓ_2 : loc1 = &ids[slot]

ℓ_3 : loc2 = &ids[4]

slot	$\langle [0,3], \top \rangle$
loc1	$\langle \top, \{o_1, o_2, o_3, o_4\} \rangle$
loc2	$\langle \top, \{o_5\} \rangle$

Abstract power trace δ

$\delta_{\overline{\ell_4}}$

o_1, o_2, o_3, o_4	$\langle [0,0], \top \rangle$

ℓ_4 : *loc1 = INT_MAX

$\delta_{\underline{\ell_4}}$

o_1, o_2, o_3, o_4	$\langle [0,$ INT_MAX $], \top \rangle$

An Example: Analysis Rules

ℓ_1 : slot = input()%4
ℓ_2 : loc1 = &ids[slot]
ℓ_3 : loc2 = &ids[4]

Abstract trace σ

slot	$\langle [0,3], \top \rangle$
loc1	$\langle \top, \{o_1, o_2, o_3, o_4\} \rangle$
loc2	$\langle \top, \{o_5\} \rangle$

Abstract power trace δ

$\delta_{\overline{\ell_4}}$

o_1, o_2, o_3, o_4	$\langle [0,0], \top \rangle$

ℓ_4 : *loc1 = INT_MAX

$\delta_{\underline{\ell_4}}$

o_1, o_2, o_3, o_4	$\langle$[0, INT_MAX], $\top \rangle$

$\delta_{\overline{\ell_5}}$

o_1, o_2, o_3, o_4	$\langle$[0, INT_MAX], $\top \rangle$
o_5	$\langle [0,0], \top \rangle$

ℓ_5 : buf[*loc2]

ℓ_1 : slot = input()%4

ℓ_2 : loc1 = &ids[slot]

ℓ_3 : loc2 = &ids[4]

Abstract trace σ

slot	$\langle [0,3], \top \rangle$
loc1	$\langle \top, \{o_1, o_2, o_3, o_4\} \rangle$
loc2	$\langle \top, \{o_5\} \rangle$

Abstract power trace δ

$\delta_{\overline{\ell_4}}$

o_1, o_2, o_3, o_4	$\langle [0,0], \top \rangle$

ℓ_4 : *loc1 = INT_MAX

$\delta_{\underline{\ell_4}}$

o_1, o_2, o_3, o_4	$\langle [0,$ INT_MAX $], \top \rangle$

$\delta_{\overline{\ell_5}}$

o_1, o_2, o_3, o_4	$\langle [0,$ INT_MAX $], \top \rangle$
o_5	$\langle [0,0], \top \rangle$

ℓ_5 : buf[*loc2]

$\boxed{\delta_{\overline{\ell_5}}(o_5) = \langle [0,0], \top \rangle}$ Safe buffer access

1. A benchmark comprising 7774 programs from NIST Juliet test cases [1], which includes its null dereferences and buffer overflow vulnerabilities.
2. 10 popular open-source C/C++ projects across various application domains: `paste` (file merger), `md5sum` (file verifier), `YAJL` (JSON parsing library), `MP4v2` (MP4 file library), `RIOT` (IoT operating system), `darknet` (neural network framework), `tmux` (terminal multiplexer), `Teeworlds` (online multiplayer game), `NanoMQ` (MQTT broker for IoT edge platform) and `redis` (in-memory database).

UNSW SYDNEY

Table 1: The statistics of the open-source projects. #LOI denotes the number of lines of LLVM instructions. #Method, #Call and #Obj are the numbers of functions, method calls and memory objects, respectively. $|V|$ and $|E|$ are the numbers of ICFG nodes and ICFG edges.

Project	#LOI	#Method	#Call	#Obj	$\|V\|$	$\|E\|$
paste	8,416	53	758	510	9,395	9,922
md5sum	11,483	63	881	606	12,494	13,064
YAJL	20,592	151	561	208	9,253	9,922
MP4v2	39,178	601	610	1,991	15,595	16,733
RIOT	54,597	579	1,614	951	20,176	20,843
darknet	159,205	985	9,776	2,550	136,094	147,852
tmux	446,626	1,967	22,369	3,879	162,879	178,924
Teeworlds	529,737	2,306	28,267	5,754	251,356	246,029
NanoMQ	788,967	3,235	47,646	30,838	358,312	443,670
redis	1,363,507	6,314	68,664	13,958	589,019	704,356
Total	3,422,308	16,254	181,146	61,245	1,564,573	1,791,315

RQ1 **Is CSA effective in detecting existing bugs?** We aim to investigate whether CSA can achieve a better performance than the state-of-the-art on detecting existing bugs.

RQ2 **Can CSA find bugs with a low false positive rate in real-world projects?** We would like to examine the effectiveness and efficiency of CSA using real-world popular applications.

RQ3 **What is the influence of different components in our framework?** We aim to understand RQ3.1: the precision improvement of cross-domain refinement; and RQ3.2: efficiency improvement in terms of time and memory using equivalent correlation tracking.

Table 2: Comparing with five tools and CSA-CP (a variant of CSA without cross-domain interaction) using the NIST benchmark, with true positive rate (#TPR) and precision rate (#PCR) in percentage (%).

Tool	Buffer overflow		Null dereference		Total	
	#TPR (%)	#PCR (%)	#TPR (%)	#PCR (%)	#TPR (%)	#PCR (%)
INFER	19.23	70.57	53.17	50.19	20.20	68.48
CPPCHECK	2.72	100.00	42.86	85.71	3.87	95.00
KLEE	67.78	98.81	91.27	93.12	68.45	98.58
IKOS	49.76	45.83	92.86	92.86	50.99	47.07
SPARROW	44.64	32.49	90.48	52.78	45.95	33.21
CSA-CP	73.84	42.62	100.00	42.64	74.58	42.65
CSA	**73.84**	**84.11**	**100.00**	**100.00**	**74.58**	**84.63**
BugNum	8589		252		8841	

Table 3: Comparing CSA with five open-source tools and CSA-CP using ten popular applications. #TP and #FP are true positive and false positive, respectively. Time (secs), Mem (MB) are running time and memory costs. The – in the Time columns indicates a running time of more than 4h. The – in the Mem columns indicates a cost of more than 100 Gigabytes.

Project	INFER				CPPCHECK				IKOS				KLEE				SPARROW				CSA-CP				CSA			
	Report		Time	Mem	Report		Time	Mem	Report		Time	Mem	Report		Time	Mem	Report		Time	Mem	Report		Time	Mem	Report		Time	Mem
	#TP	#FP	(secs)	(MB)	#TP	#FP	(secs)	(MB)	#TP	#FP	(secs)	(MB)	#TP	#FP	(secs)	(MB)	#TP	#FP	(secs)	(MB)	#TP	#FP	(secs)	(MB)	#TP	#FP	(secs)	(MB)
paste	1	15	7	61	0	17	1	9	3	21	512	1126	4	0	2911	1711	4	35	3	51	3	19	5	92	3	0	9	106
md5sum	2	21	8	80	0	18	1	11	2	35	986	1684	3	0	2824	1642	2	22	2	48	4	26	15	121	4	1	8	110
YAJL	0	17	9	110	0	14	1	12	1	1625	2895	4822	4	16	14400	17333	3	86	6	59	3	35	7	172	3	0	5	102
MP4v2	1	28	313	335	1	26	38	38	1	956	3684	6215	2	3	14400	21358	1	236	214	231	1	25	58	269	1	0	13	384
RIOT	3	29	111	155	2	19	2	22	2	1325	5216	8622	5	2	14400	23654	2	651	315	421	8	38	102	366	8	6	27	346
darknet	25	134	837	282	16	214	10	55	14	1265	9531	23954	25	8	14400	40015	10	842	826	984	21	199	3483	1982	21	10	3507	1875
tmux	5	142	522	909	3	156	30	39	4	1632	11325	38366	2	1	14400	70826	3	1256	1036	1894	12	360	1182	6343	12	10	824	5052
Teeworlds	10	169	684	934	4	187	2	54	2	529	13569	40368	2	1	14400	71865	10	1512	1593	2984	15	244	2754	3485	15	8	2886	2598
NanoMQ	23	154	654	305	10	147	94	38	–	–	–	–	5	2	14400	91465	6	1241	1642	3125	30	292	1801	7063	30	8	1143	6551
redis	6	137	1292	10484	8	136	516	123	–	–	–	–	3	2	14400	101475	5	1152	2654	9211	14	275	8629	4421	14	8	6553	3870
Total	76	846	4437	13655	44	934	695	401	29	7388	47718	125157	55	35	120935	441344	46	7033	8291	19008	111	1513	18036	24314	111	51	14975	20994

Table 4: Comparison between CSA and CSA-NI (a version of CSA without implication-equivalent memory addresses).

Project	CSA-NI		CSA	
	Time (secs)	Mem (MB)	Time (secs)	Mem (MB)
`tmux`	1540 (1.87×)	21016 (4.16×)	824	5052
`Teeworlds`	6176 (2.14×)	14237 (5.48×)	2886	2598
`NanoMQ`	3292 (2.88×)	48805 (7.45×)	1143	6551
`redis`	21232 (3.24×)	32314 (8.35×)	6553	3870
Geo. Mean	(2.47×)	(6.14×)		

Thank You!