Fast Graph Simplification for Path-Sensitive Typestate Analysis through Tempo-Spatial Multi-Point Slicing

FSE 2024

Xiao Cheng, Jiawei Ren, Yulei Sui

xiao.cheng@unsw.edu.au

Computer Science and Engineering
UNSW Sydney

July 26, 2024

Contributions

- A fast graph simplification approach for path-sensitive typestate analysis (PSTA) utilizing tempo-spatial multi-point slicing.

Contributions

- A fast graph simplification approach for path-sensitive typestate analysis (PSTA) utilizing tempo-spatial multi-point slicing.
- A formulation of the multi-point markers extraction as a graph reachability problem based on the IFDS framework.

Contributions

- A fast graph simplification approach for path-sensitive typestate analysis (PSTA) utilizing tempo-spatial multi-point slicing.
- A formulation of the multi-point markers extraction as a graph reachability problem based on the IFDS framework.
- A new multi-point slicing technique that efficiently captures the temporal and spatial correlations necessary for a path-sensitive typestate analysis.

Contributions

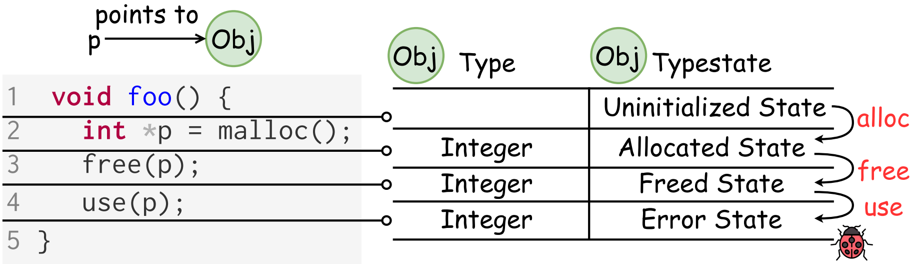
- A fast graph simplification approach for path-sensitive typestate analysis (PSTA) utilizing tempo-spatial multi-point slicing.
- A formulation of the multi-point markers extraction as a graph reachability problem based on the IFDS framework.
- A new multi-point slicing technique that efficiently captures the temporal and spatial correlations necessary for a path-sensitive typestate analysis.
- An implementation and an evaluation to demonstrate the effectiveness and efficiency of graph simplification for PSTA.

Typestate

- Typestate (state of type) represents different **states of a given object type**, which expands the scope of standard **immutable types** to accommodate potential **object typestate changes**.

Typestate

- Typestate (state of type) represents different **states of a given object type**, which expands the scope of standard **immutable types** to accommodate potential **object typestate changes**.



A Use-After-Free (UAF) Bug

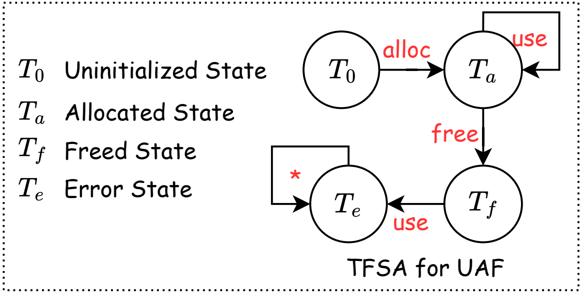
Type vs. Typestate

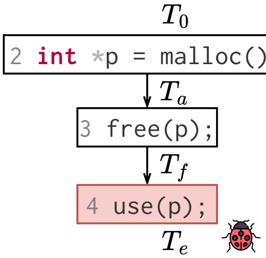
Typestate Analysis

- Typestate analysis determines whether **a sequence of program operations**, e.g., an API calling chain, performed upon an instance of a given type violates safety specifications established by a **finite state automaton (TFSA)**.

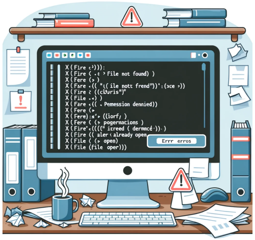
Typestate Analysis

- Typestate analysis determines whether **a sequence of program operations**, e.g., an API calling chain, performed upon an instance of a given type violates safety specifications established by a **finite state automaton (TFSA)**.

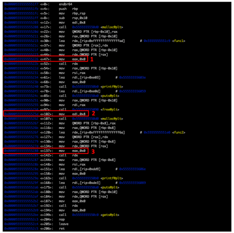




Typestate Analysis



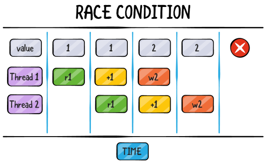
Incorrect file library usage



Use-after-frees

Memory leaks

Access control



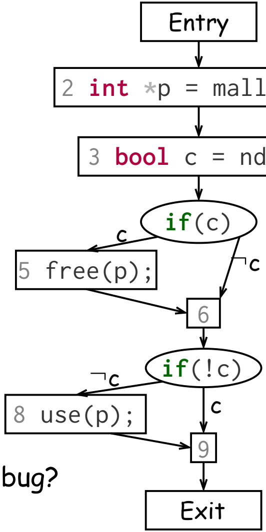
Concurrency bug

API misuse

- Path-sensitive typestate analysis (PSTA) enhances the precision of its path-insensitive counterpart by **capturing correlations between different branches** and eliminating false alerts stemming from infeasible paths.

- Path-sensitive typestate analysis (PSTA) enhances the precision of its path-insensitive counterpart by **capturing correlations between different branches** and eliminating false alerts stemming from infeasible paths.
- In PSTA, the maintenance of an **(abstract) execution state** that captures **program variable values and path constraints** is crucial, and it evaluates the feasibility of paths when encountering branching points.

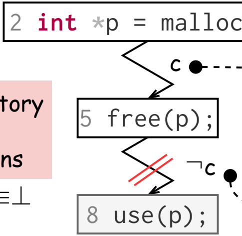
```c
void foo() {
  int *p = malloc();
  bool c = nd();
  if(c) {
    free(p);
  }
  if(!c) {
    use(p);
  }
}
```



A UAF bug?

Path-Sensitive Typestate Analysis (PSTA)

An Example

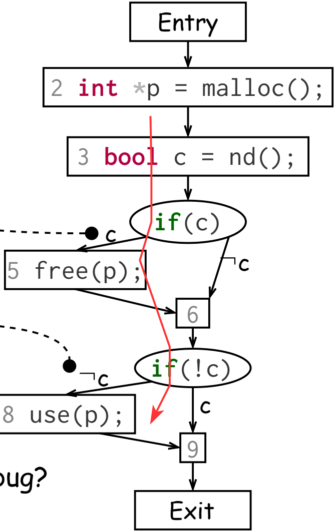


Contradictory path conditions

$c \wedge \neg c \equiv \perp$

~~Yes, a UAF bug!~~

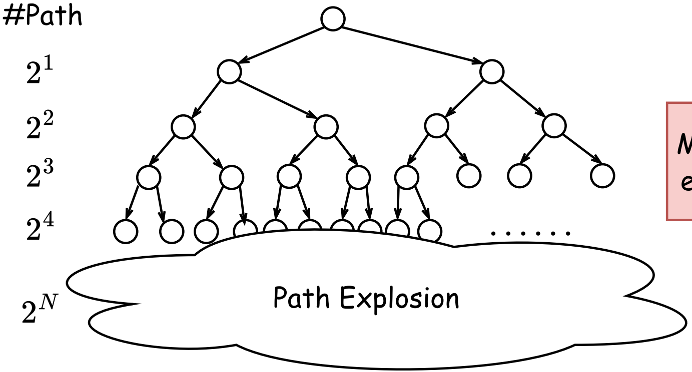
No UAF bug!



A UAF bug?

- Path sensitivity: analyzing each path individually?

- Path sensitivity: analyzing each path individually?
- With each if branch, the possible paths the program can take might **double**. This means the complexity of the program grows **exponentially** as it gets longer.



MOP is too expensive!

- ESP is a representative PSTA working in polynomial time. At a control-flow joint point, ESP merges execution states with identical typestates, yielding a single symbolic state and thus achieving a **maximal-fixed-point (MFP)** solution with **program paths sensitive to typestate preserved**.

[1] Manuvir Das, Sorin Lerner, and Mark Seigle. 2002. ESP: path-sensitive program verification in polynomial time. SIGPLAN Not. 37, 5 (May 2002), 57–68. https://doi.org/10.1145/543552.512538

Symbolic State:

⟨ Typestate, Execution state ⟩

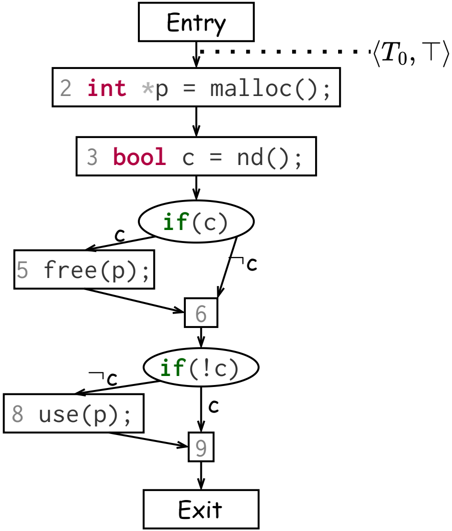
Execution State:

$\top$:All behaviors

$\bot$:No behaviors (infeasible)

c :feasible when c is satisfied

......



Symbolic State:

⟨ Typestate, Execution state ⟩

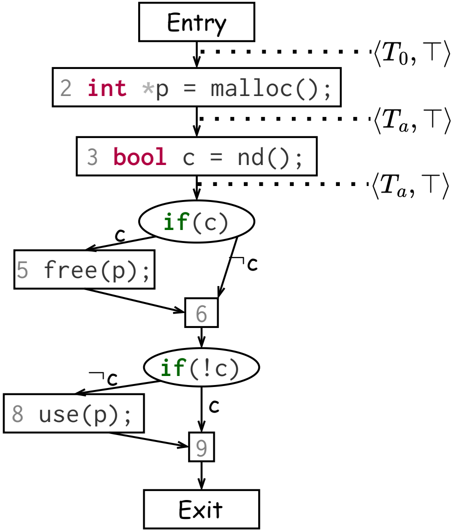
Execution State:

$\top$:All behaviors

$\bot$:No behaviors (infeasible)

c :feasible when c is satisfied

……



Symbolic State:

$\langle$ Typestate, Execution state $\rangle$

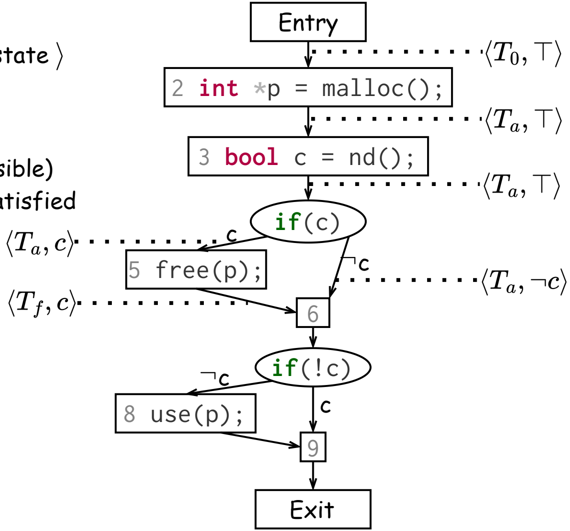
Execution State:

$\top$:All behaviors

$\bot$:No behaviors (infeasible)

c :feasible when c is satisfied

......



Symbolic State:

⟨ Typestate, Execution state ⟩

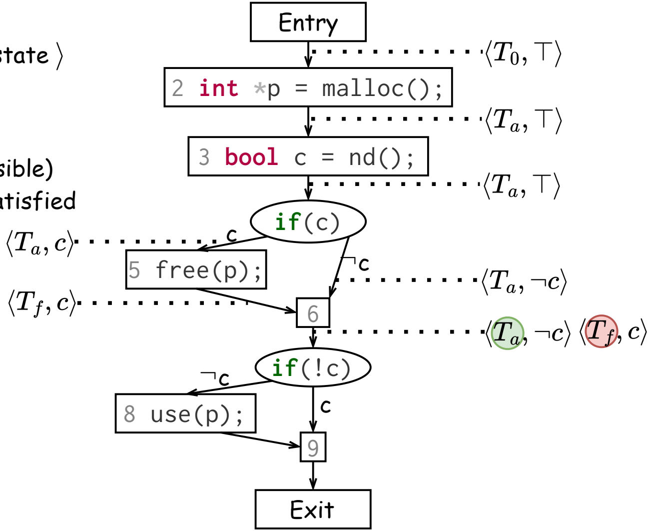
Execution State:

$\top$:All behaviors

$\bot$:No behaviors (infeasible)

c :feasible when c is satisfied

......



Symbolic State:

$\langle$ Typestate, Execution state $\rangle$

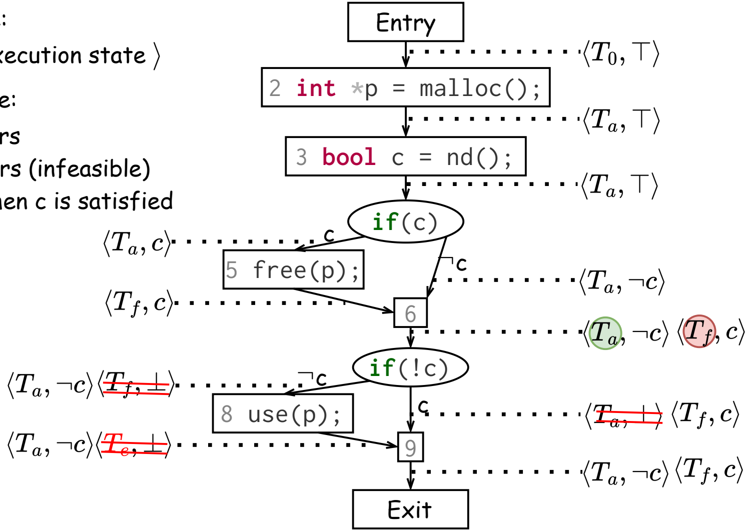
Execution State:

$\top$:All behaviors

$\bot$:No behaviors (infeasible)

c :feasible when c is satisfied

……



- To the best of our knowledge, all previous endeavors in PSTA primarily focused on enhancing the precision of typestate transitions through **alias analysis** [1-5] or exploring new opportunities for integrating **dynamic analysis** techniques [6-8].
- We focus on a new and orthogonal perspective, **improving the efficiency of the path-sensitive algorithm**.

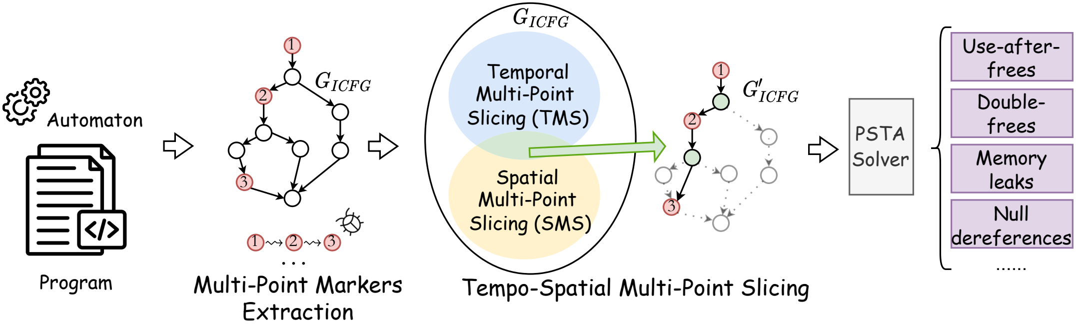
[1] Stephen J. Fink et al. Effective typestate verification in the presence of aliasing. ISSTA 2006.
[2] Mathias Jakobsen et al. Papaya: Global Typestate Analysis of Aliased Objects. PPDP 2021.
[3] Tuo Li et al. Path-Sensitive and Alias-Aware Typestate Analysis for Detecting OS Bugs. ASPLOS 2022.
[4] Zhiqiang Zuo et al. Grapple: A Graph System for Static Finite-State Property Checking of Large-Scale Systems Code. Eurosys 2019.
[5] Eric Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. ICSE 2010.
[6] Eric Bodden et al. Partially Evaluating Finite-State Runtime Monitors Ahead of Time. TOPLAS.
[7] Matthew B. Dwyer et al. Residual Dynamic Typestate Analysis Exploiting Static Analysis: Results to Reformulate and Reduce the Cost of Dynamic Analysis. ASE 2007.
[8] Haijun Wang et al. Typestate-Guided Fuzzer for Discovering Use-after-Free Vulnerabilities. ICSE2020.

- We aim to tackle the overhead by using sparse idea that skips unnecessary control flows using def-use information.

- We aim to tackle the overhead by using sparse idea that skips unnecessary control flows using def-use information.
- Sparse analysis cannot capture **multi-point temporal use-to-use information**.

- We aim to tackle the overhead by using sparse idea that skips unnecessary control flows using def-use information.
- Sparse analysis cannot capture **multi-point temporal use-to-use information**.
- We focus on a more practical perspective–reducing the size of the control flow graph (**graph simplification**), rendering it a sparser structure with unnecessary control flows eliminated, while preserving the multi-point temporal information.

Framework Overview

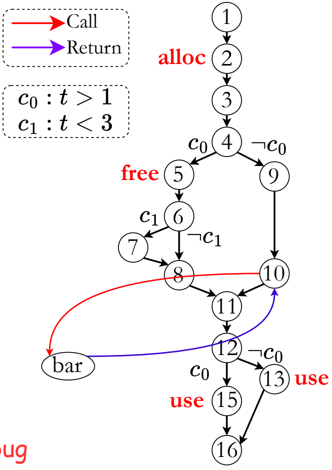


```
void foo(T t) {
  int *p = malloc(10);
  int *q = malloc(10);
  ...
  if (t > 1) {
    free(p);
    if (t < 3)
      free(q);
  } else {
    free(q);
    p = bar(p);
  }
  ...
  if (t <= 1)
    printf("%d\n", *p);
  else
    log_error(*p);
}
```

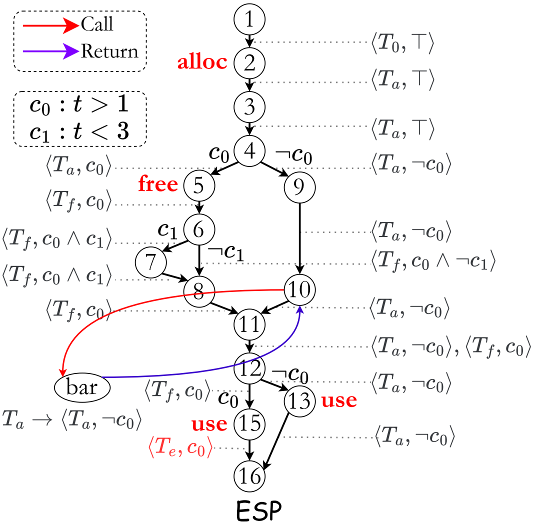
Not a UAF bug

A UAF bug

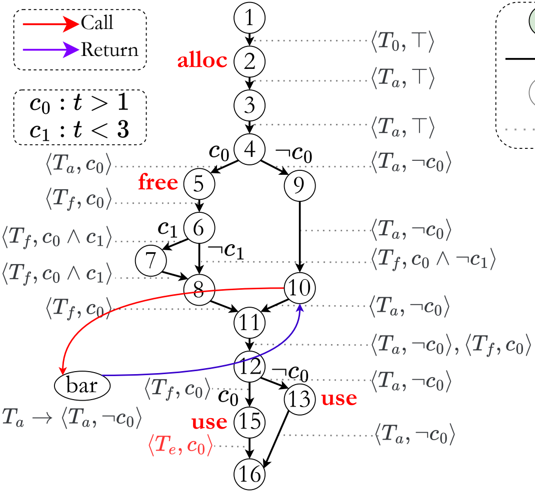
Source Code



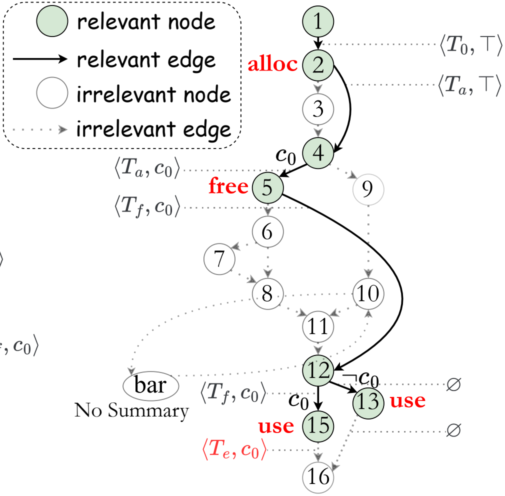
ICFG



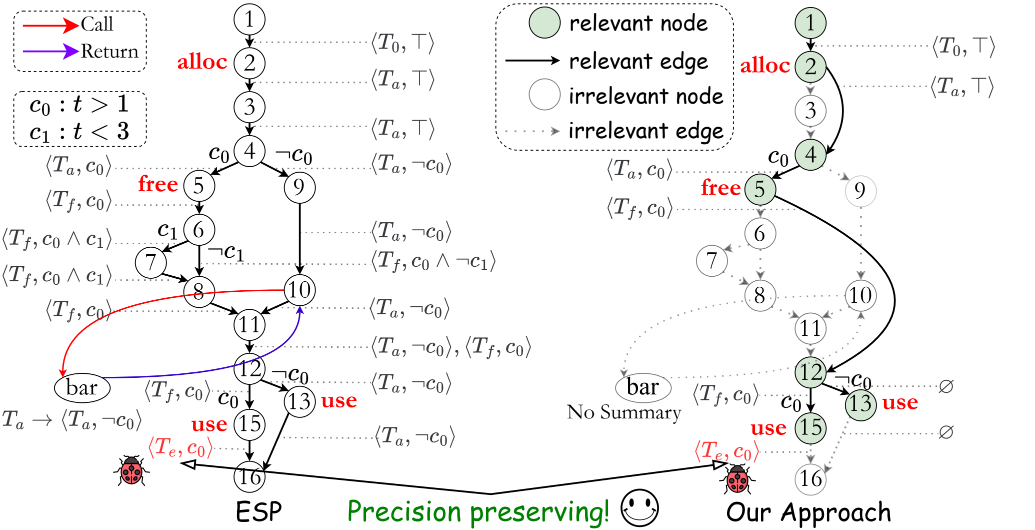
Call
Return
$c_0 : t > 1$
$c_1 : t < 3$
alloc
free
use
use
bar
$T_a \rightarrow \langle T_a, \neg c_0 \rangle$
$\langle T_0, \top \rangle$
$\langle T_a, \top \rangle$
$\langle T_a, \top \rangle$
$\langle T_a, c_0 \rangle$
$\langle T_a, \neg c_0 \rangle$
$\langle T_f, c_0 \rangle$
$\langle T_f, c_0 \wedge c_1 \rangle$
$\langle T_a, \neg c_0 \rangle$
$\langle T_f, c_0 \wedge \neg c_1 \rangle$
$\langle T_f, c_0 \wedge c_1 \rangle$
$\langle T_f, c_0 \rangle$
$\langle T_a, \neg c_0 \rangle$
$\langle T_a, \neg c_0 \rangle, \langle T_f, c_0 \rangle$
$\langle T_a, \neg c_0 \rangle$
$\langle T_f, c_0 \rangle$
$\langle T_a, \neg c_0 \rangle$
$\langle T_e, c_0 \rangle$
ESP



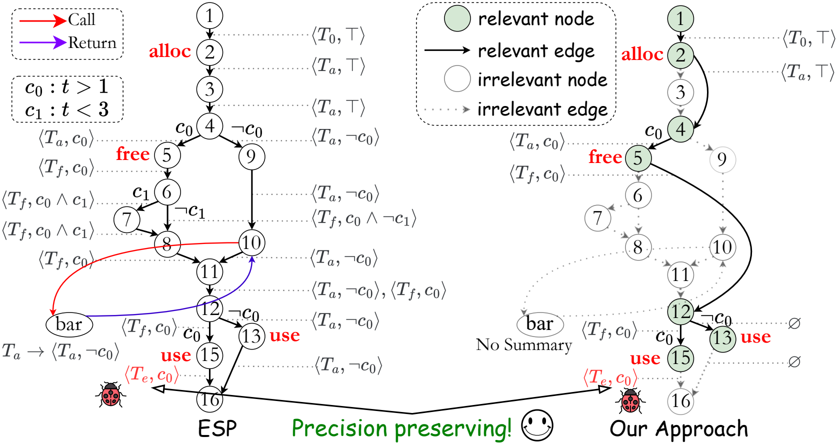
ESP



Our Approach



Call
Return
$c_0 : t > 1$
$c_1 : t < 3$
alloc
free
use
bar
$T_a \to \langle T_a, \neg c_0 \rangle$
$\langle T_0, \top \rangle$
$\langle T_a, \top \rangle$
$\langle T_a, c_0 \rangle$
$\langle T_a, \neg c_0 \rangle$
$\langle T_f, c_0 \rangle$
$\langle T_f, c_0 \wedge c_1 \rangle$
$\langle T_f, c_0 \wedge \neg c_1 \rangle$
$\langle T_a, \neg c_0 \rangle, \langle T_f, c_0 \rangle$
$\langle T_e, c_0 \rangle$
ESP
relevant node
relevant edge
irrelevant node
irrelevant edge
No Summary
$\emptyset$
Precision preserving!
Our Approach

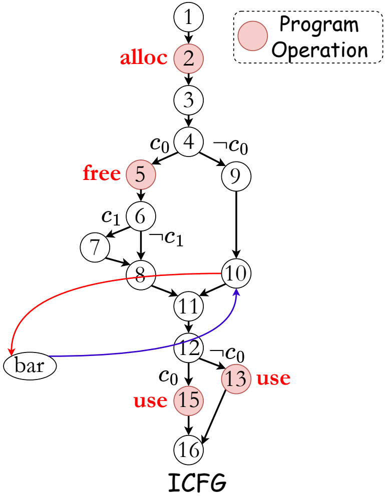


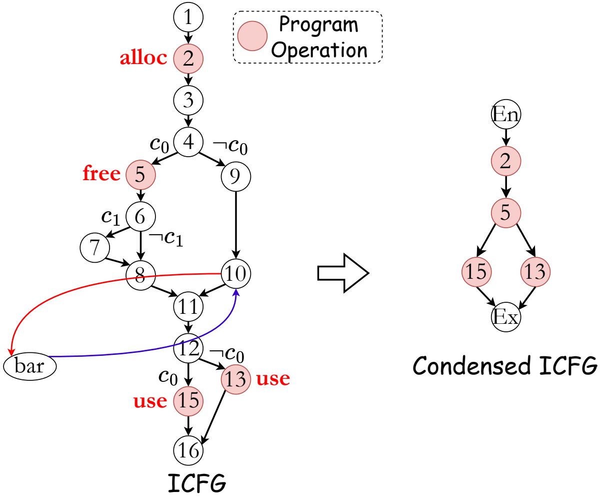
Benefits

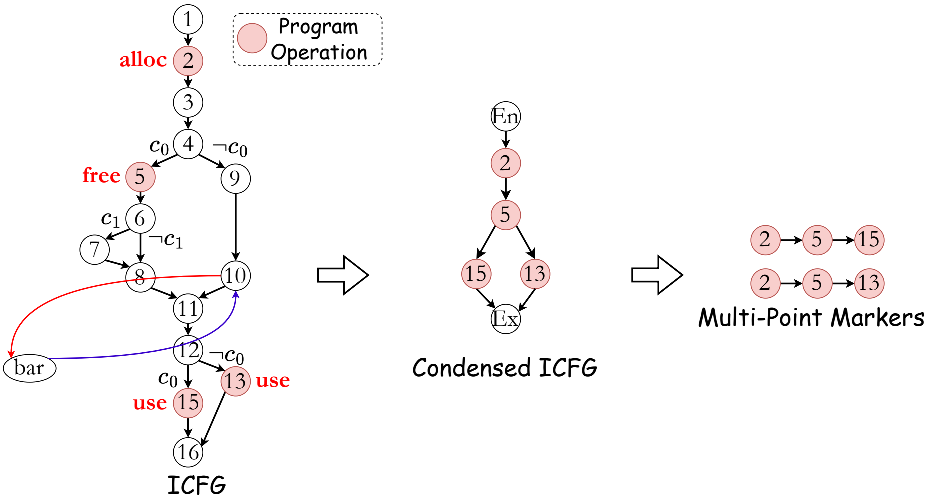
#Symbolic States: reduces from 18 to 6
#ICFG Nodes: reduces from 16 to 7

#SMT Solving: reduces from 6 to 3
#Merge Points: reduces from 2 to 0
#Function Summary: reduces from 1 to 0

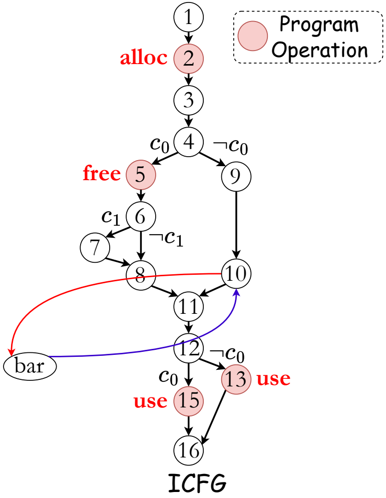
Multi-Point Markers Extraction



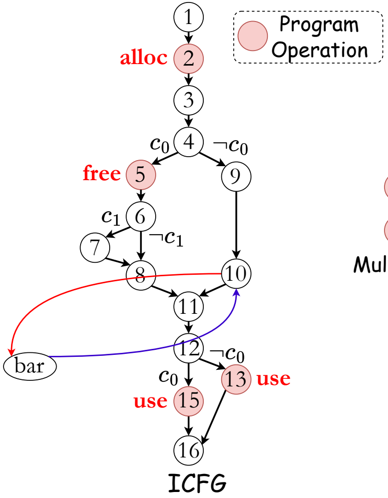

Program Operation
alloc
free
use
use
bar
c_0
$\neg c_0$
c_1
$\neg c_1$
1 2 3 4 5 6 7 8 9 10 11 12 13 15 16
ICFG
En
Ex
2 5 15 13
Condensed ICFG

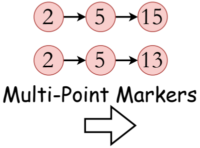

1
alloc 2
3
4
c_0
$\neg c_0$
free 5
9
c_1 6
$\neg c_1$
7
8
10
11
12
$\neg c_0$
bar
c_0
13 use
use 15
16
ICFG
Program Operation
En
2
5
15
13
Ex
Condensed ICFG
2 → 5 → 15
2 → 5 → 13
Multi-Point Markers

Temporal Multi-Point Slicing (TMS)

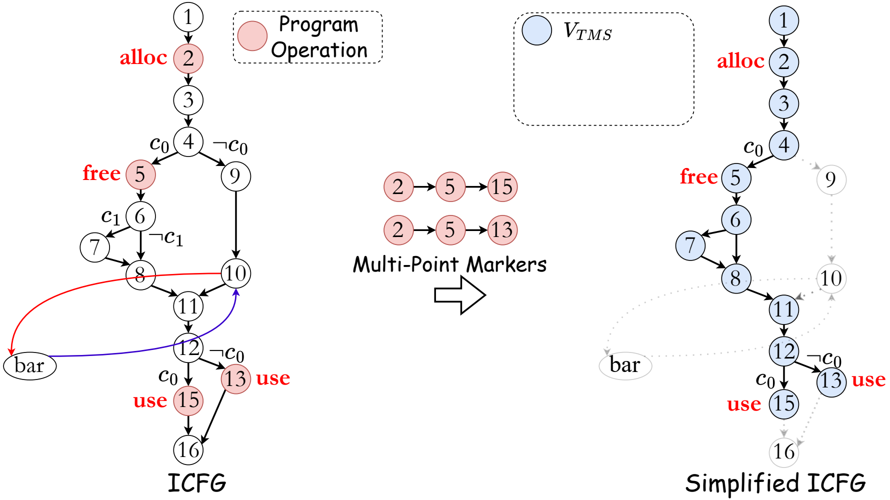


Temporal Multi-Point Slicing (TMS)

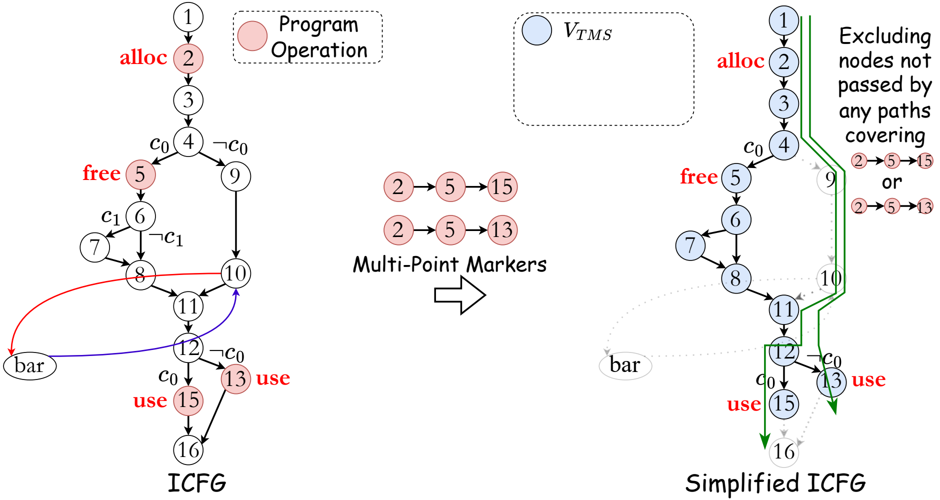




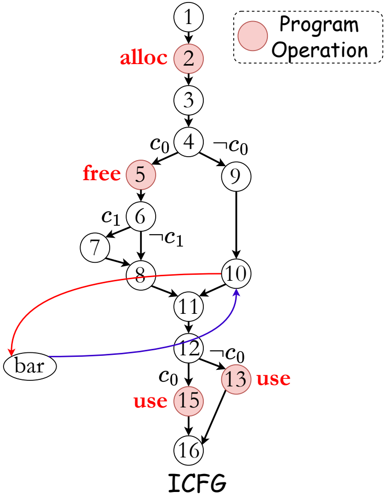
Temporal Multi-Point Slicing (TMS)



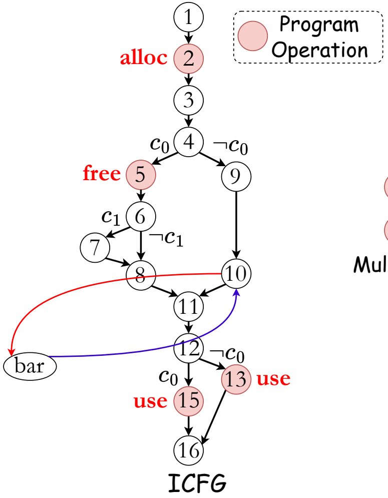
Temporal Multi-Point Slicing (TMS)

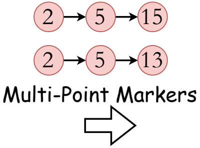


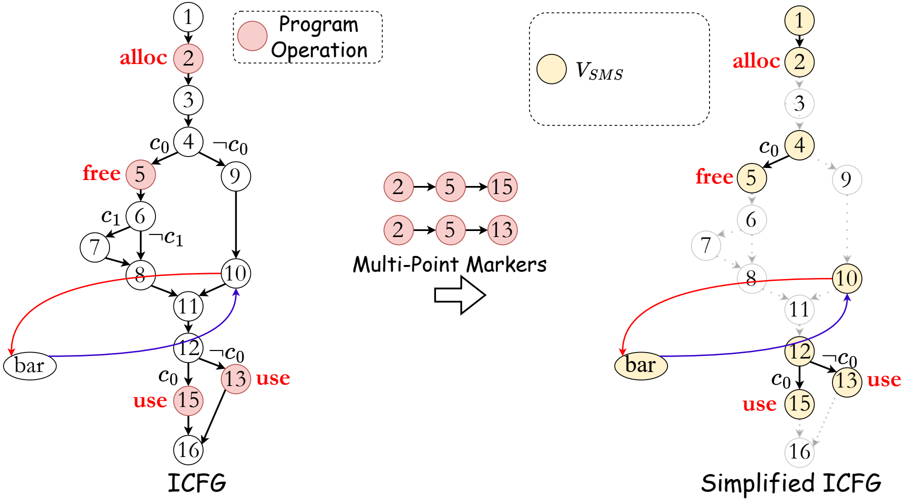
Spatial Multi-Point Slicing (SMS)

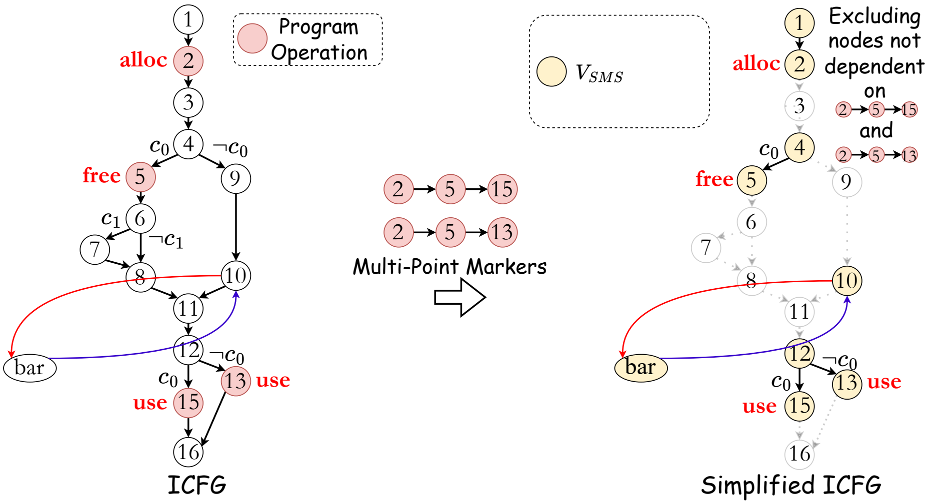


Spatial Multi-Point Slicing (SMS)

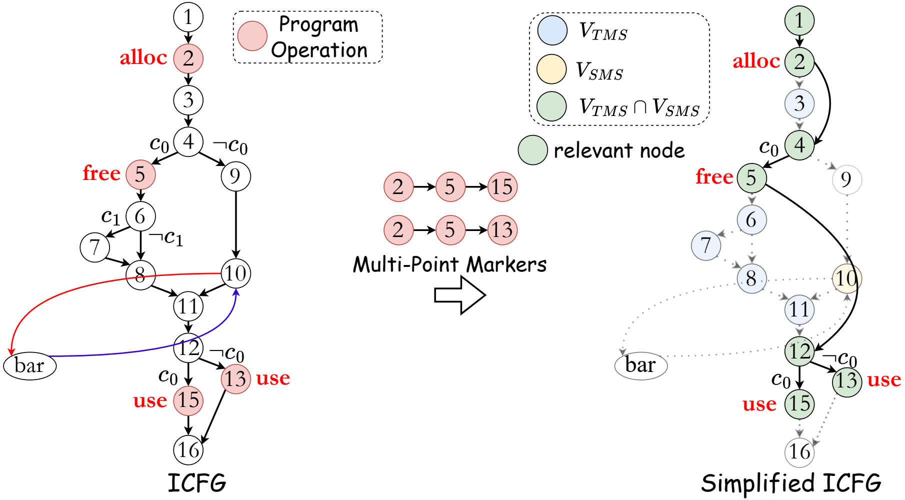





Program Operation
alloc
free
use
use
bar
ICFG
2 → 5 → 15
2 → 5 → 13
Multi-Point Markers
V_{SMS}
alloc
free
use
use
bar
Simplified ICFG


Program Operation
1
alloc 2
3
c_0 4 $\neg c_0$
free 5
9
c_1 6 $\neg c_1$
7
8
10
11
bar
12 $\neg c_0$
c_0
13 use
use 15
16
ICFG
V_{SMS}
2 → 5 → 15
2 → 5 → 13
Multi-Point Markers
Excluding nodes not dependent on
2 → 5 → 15
and
2 → 5 → 13
Simplified ICFG

Putting it All Together



Datasets

- A micro-benchmark comprising 846 vulnerabilities from NIST, which includes memory leaks, double-frees, use-after-frees and null dereferences.
- Ten open-source C/C++ projects across a variety of different domains: `YAJL` (JSON parsing library), `gzip` (data compression program), `MP4v2` (MP4 file library), `bzip2` (data compressor), `darknet` (neural network framework), `nasm` (assembler), `tmux` (terminal multiplexer), `Teeworlds` (online multiplayer game), `NanoMQ` (MQTT broker for IoT edge platform) and `redis` (in-memory database).

Statistics of Open-Source Projects

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

Project	#LOI	#Method	#Call	#Ptr	#Obj	|V|	|E|
YAJL	20,592	151	561	10,197	208	9,253	9,922
gzip	33,058	195	459	19,264	457	16,889	16,582
MP4v2	39,178	601	610	15,925	1,991	15,595	16,733
bzip2	48,181	116	250	28,710	263	26,220	25,912
darknet	159,205	985	9,776	136,510	2,550	136,094	147,852
nasm	186,935	652	7,435	121,836	3,736	79,330	81,638
tmux	446,626	1,967	22,369	187,315	3,879	162,879	178,924
Teeworlds	529,737	2,306	28,267	292,621	5,754	251,356	246,029
NanoMQ	788,967	3,235	47,646	379,798	30,838	358,312	443,670
redis	1,363,507	6,314	68,664	708,251	13,958	589,019	704,356
Total	**3,615,986**	165,22	186,037	1,900,427	63,634	1,644,947	1,871,618

Research Questions

RQ1 **How do different components impact the overall performance of FGS?** We want to investigate how different slicing methods influence the effectiveness and efficiency of FGS.

RQ2 **Does FGS outperform popular static tools for bug detection?** We aim to explore whether FGS can detect more bugs with lower false alarm rates than the state-of-the-art on detecting existing bugs using the NIST benchmark with ground truths.

RQ3 **Can FGS find bugs with lower false positives efficiently in real-world projects?** We would like to examine the effectiveness (in terms of true and false positives) and efficiency (in terms of running time and memory usage) of FGS on real-world popular applications.

Table 2: Graph simplification result. $|V|$, $|V'|$, $|V_{\text{TMS}}|$ and $|V_{\text{SMS}}|$ represent the number of nodes in G_{ICFG}, G'_{ICFG}, temporal slice and spatial slice, respectively. $\#Call$ and $\#Call'$ represent the number of calling contexts of G_{ICFG} and G'_{ICFG}. $|E|$ and $|E'|$ represent the number of edges in G_{ICFG} and G'_{ICFG}.

Project	$\|V\|$	$\|V'\|$	$\|V_{\text{TMS}}\|$	$\|V_{\text{SMS}}\|$	$\#Call$	$\#Call'$	$\|E\|$	$\|E'\|$
darknet	136,094	**1,791**	5,523	1,928	9,776	**93**	147,852	**1,802**
nasm	79,330	**24,946**	38,081	26,604	7,435	**2,317**	81,638	**26,034**
tmux	162,879	**2,671**	4,273	3,693	22,369	**205**	178,924	**2,810**
Teeworlds	251,356	**565**	1,380	1,875	28,267	**40**	246,029	**578**
NanoMQ	358,312	**62,543**	102,118	118,663	47,646	**5,801**	443,670	**61,696**
redis	589,019	**87,446**	102,416	111,041	68,664	**17,844**	704,356	**240,956**

Table 3: Ablation analysis results. The "–" in the Time columns indicates a running time of more than 48 hours. FGS-TMS and FGS-SMS represent the versions of FGS using only temporal slicing and spatial slicing respectively. FGS-Base represent the version of FGS without slicing.

Project	FGS		FGS-TMS		FGS-SMS		FGS-Base	
	Time (secs)	Mem (MB)	Time (secs)	Mem (MB)	Time (secs)	Mem (MB)	Time (secs)	Mem (MB)
darknet	**750**	**2,104**	2,542	2,785	817	2,784	81,422	34,244
nasm	**894**	**2,482**	1,681	4,132	940	3,413	111,750	31,781
tmux	**1,932**	**5,251**	5,782	9,064	3,102	7,223	–	–
Teeworlds	**407**	**4,320**	1,424	5,014	1,700	6,062	–	–
NanoMQ	**8,722**	**10,176**	25,890	13,600	29,100	18,424	–	–
redis	**14,266**	**58,231**	23,146	78,131	31,103	98,064	–	–

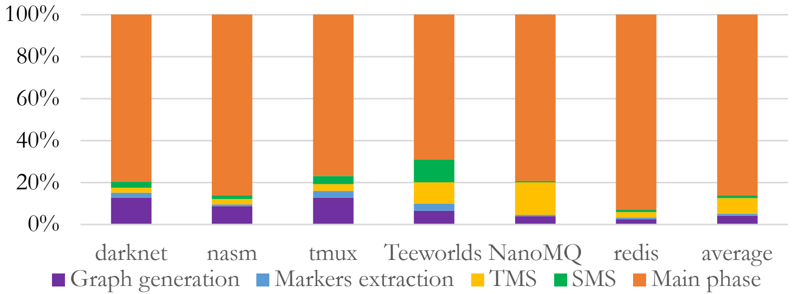


Figure 1: The proportions of different phases of FGS.

UNSW SYDNEY

Table 4: Comparing true positives (#*TP*) and false positives (#*FP*) with six tools using the NIST benchmark. The "–" means that the detection of specific vulnerabilities is not supported by the corresponding tools.

Category	IKOS		ClangSA		Saber		Cppcheck		Infer		Sparrow		FGS		Ground Truth
	#*TP*	#*FP*	#*TP*	#*FP*	#*TP*	#*FP*	#*TP*	#*FP*	#*TP*	#*FP*	#*TP*	#*FP*	#*TP*	#*FP*	
Memory leak	–	–	128	112	200	126	0	0	126	162	–	–	**228**	**0**	228
Double-free	228	18	156	20	204	20	84	144	–	–	–	–	**228**	**0**	228
Use-after-free	–	–	40	0	–	–	0	0	0	0	–	–	**138**	**0**	138
Null dereference	234	18	216	24	234	18	108	18	134	82	228	18	**252**	**0**	252
Total	462	36	540	156	638	164	192	162	260	244	228	18	**846**	**0**	846

Table 5: Comparing FGS with six open-source tools 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	IKOS				ClangSA				Saber				Cppcheck				Infer				Sparrow				FGS			
	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)
YAJL	4	15	2895	4822	0	0	4	111	3	22	2	206	1	5	1	13	2	15	13	133	3	86	6	59	5	0	2	168
gzip	4	4	3114	4949	0	1	27	151	0	4	18	179	1	3	89	35	1	17	36	177	1	22	14	89	4	0	18	835
MP4v2	2	1	3684	6215	0	0	11	145	3	24	3	380	0	6	56	38	4	28	496	426	1	20	214	231	5	0	2	344
bzip2	0	0	3690	6809	0	6	16	181	0	2	18	179	0	0	3	17	0	37	53	271	0	0	77	148	1	0	9	280
darknet	19	75	5216	8622	11	39	75	301	20	300	245	1145	2	24	11	55	12	104	1185	612	25	10	951	954	30	7	750	2104
nasm	2	8	5007	9951	2	7	180	515	2	102	572	2258	0	1	1	76	1	16	621	919	2	9	942	1132	3	1	894	2482
tmux	4	29	11325	38366	6	12	409	799	4	160	597	3882	0	0	61	39	2	34	693	637	3	12	1036	1894	5	1	1932	5251
Teeworlds	8	8	13569	40368	0	0	83	654	10	50	88	1877	1	4	2	54	6	48	267	449	5	24	1593	2984	12	2	407	4320
NanoMQ	17	29	9344	63068	0	0	52	555	10	426	1421	7613	5	54	111	40	18	74	910	555	6	354	1642	3125	31	11	8722	10176
redis	–	–	–	–	0	23	502	1499	7	141	8775	16752	0	1	637	123	1	51	2699	1655	1	149	2654	9211	9	1	14266	58231
Total	60	169	57844	183170	19	88	1359	4911	59	1231	11739	34471	10	98	972	490	47	424	6973	5834	47	686	9129	19827	**105**	**23**	27002	84191

Thank You!