Idea: Optimized Automatic Sanitizer Placement

Authors: Gebrehiwet B. Welearegai and Christian Hammer

13.09.17

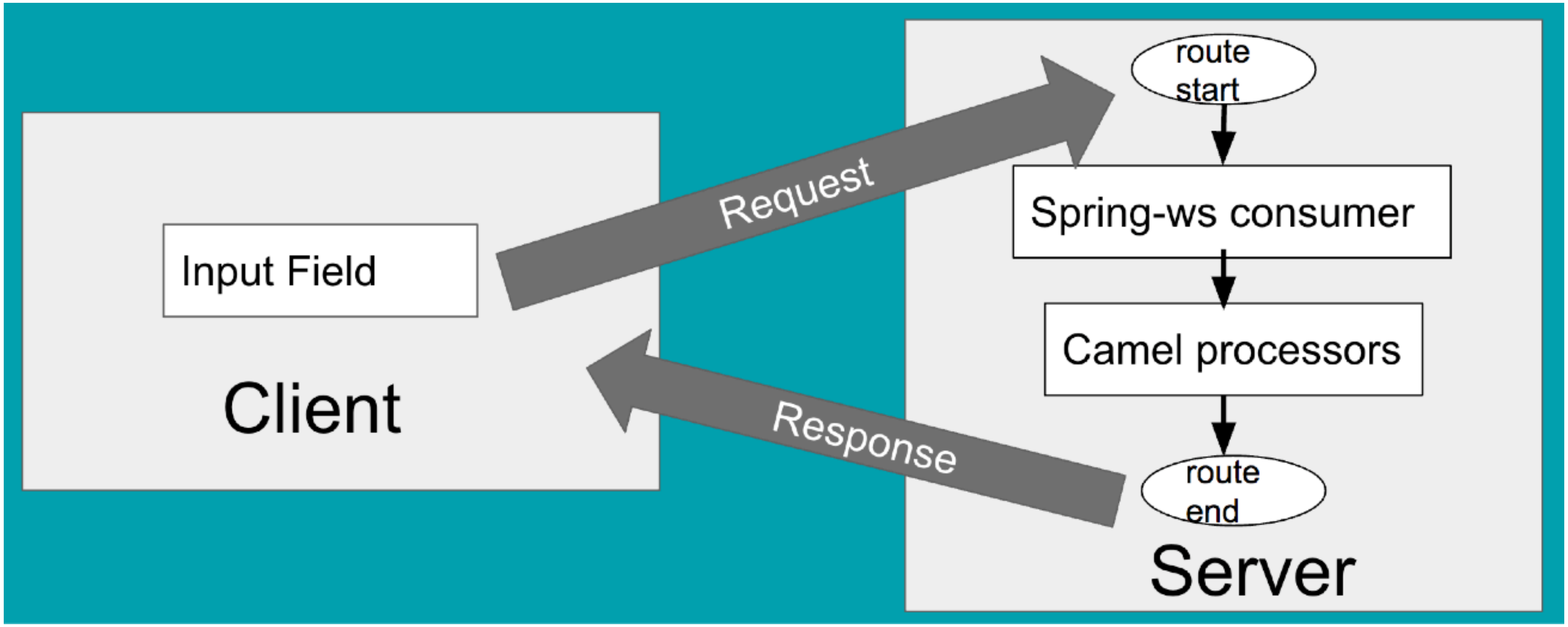
Motivation

- Enterprises are comprised of many applications

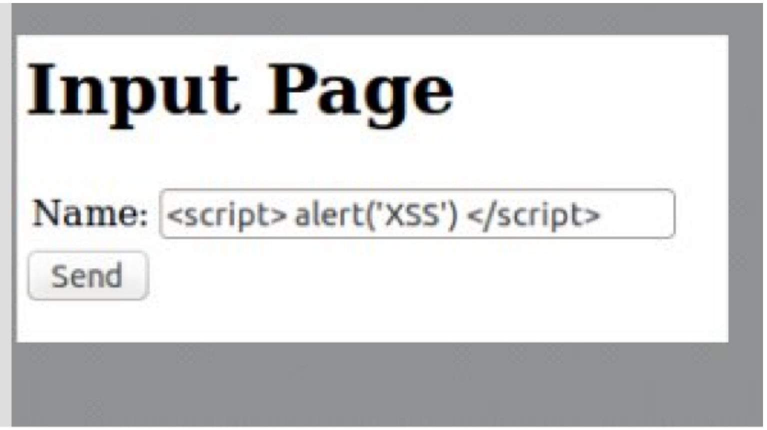
- Analysis of camel-based application of industry partner
  - "service productivity platform (SPP)"

Motivation

- Camel does not validate (user) input
  - vulnerable to cross-site scripting (XSS) and SQL injection

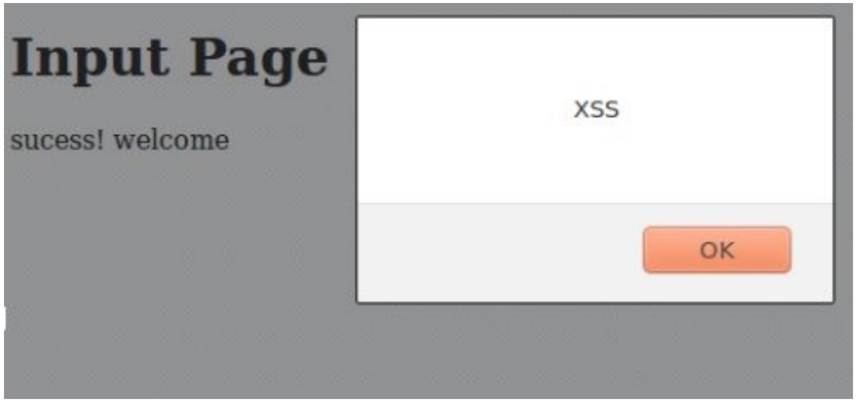


XSS Attack Example



Input Page

Name: <script> alert('XSS') </script>



Input Page

sucess! welcome

XSS

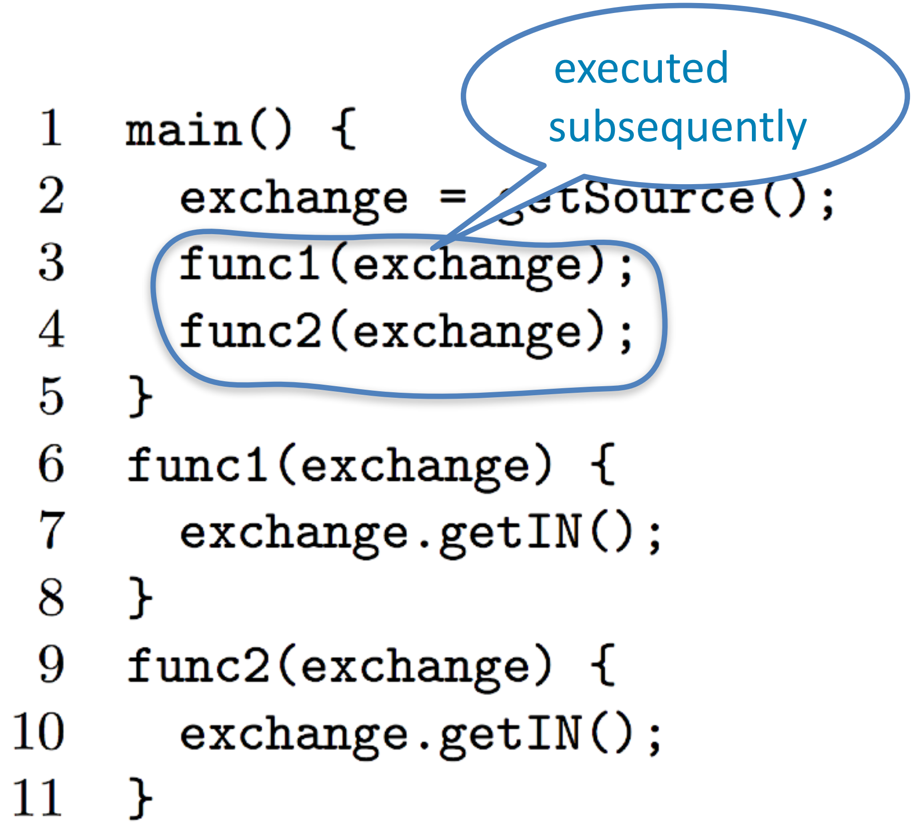
Existing solutions

- Input validation / sanitization
- Sanitizers placed
  - manually
  - automatically

Problem

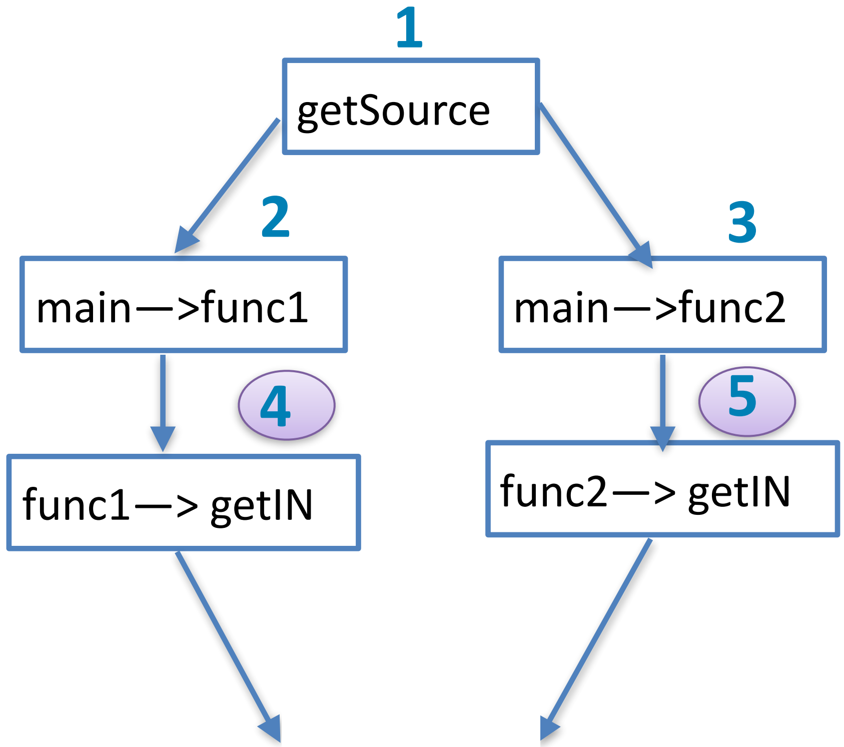
- Manual placement error prone
- Existing automatic approaches have limitations
  - code duplication
  - inconsistent multiple-sanitization

Motivating example



```
main() {
  exchange = getSource();
  func1(exchange);
  func2(exchange);
}
func1(exchange) {
  exchange.getIN();
}
func2(exchange) {
  exchange.getIN();
}
```

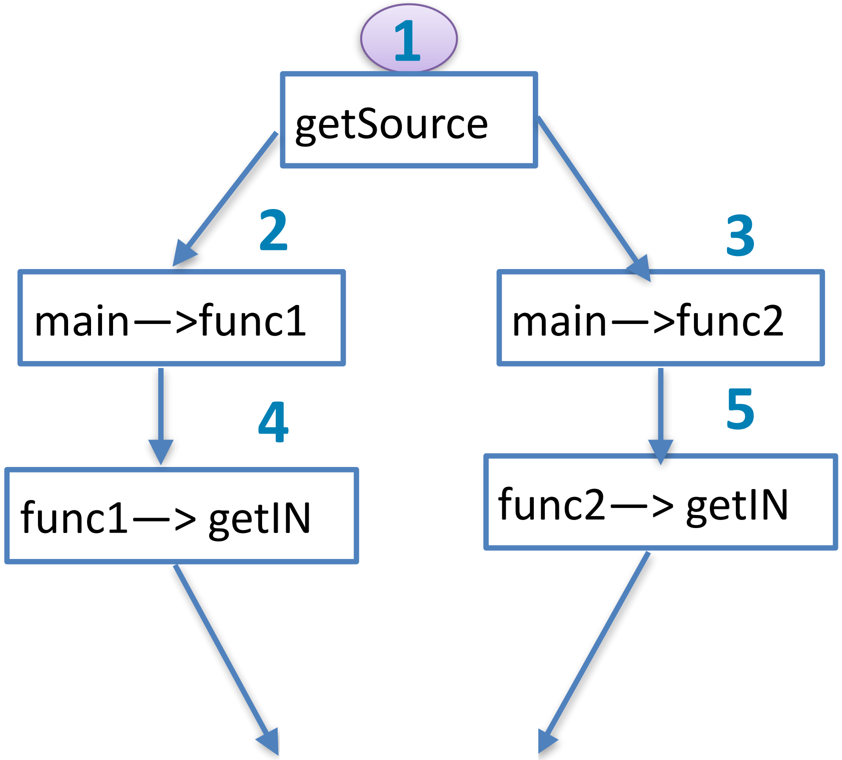
executed subsequently



- Sanitizers not always idempotent (existing research)
- Results in inconsistent multi-sanitization

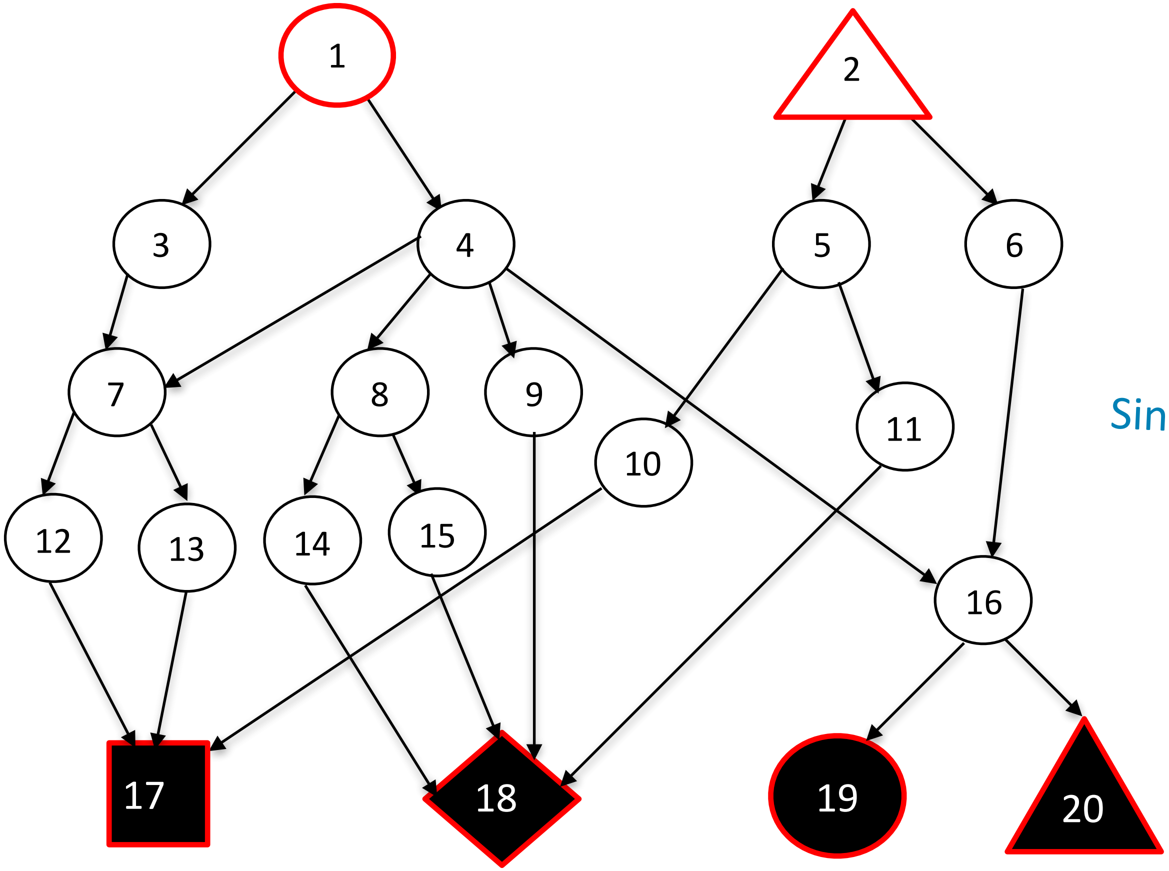
Our Solution

```
main() {
  exchange = getSource();
  func1(exchange);
  func2(exchange);
}
func1(exchange) {
  exchange.getIN();
}
func2(exchange) {
  exchange.getIN();
}
```



Prevents multi-sanitization error and code duplication

Dataflow graph (DFG) and sanitizer policy



DFG

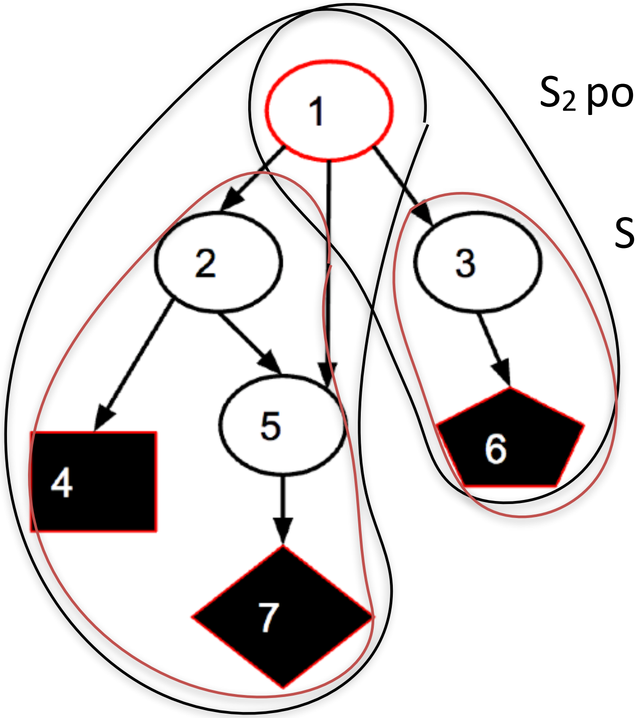
Sources

Sinks

	○	△	∅
■	S_1	S_2	⊥
◆	S_1	S_2	⊥
●	S_4	⊥	⊥
▲	S_3	S_3	⊥
∅	⊥	⊥	⊥

Sanitizer policy

Sanitizer posible and exclusive



S_2 possible

S_2 exclusive

S_1 possible

S_1 exclusive

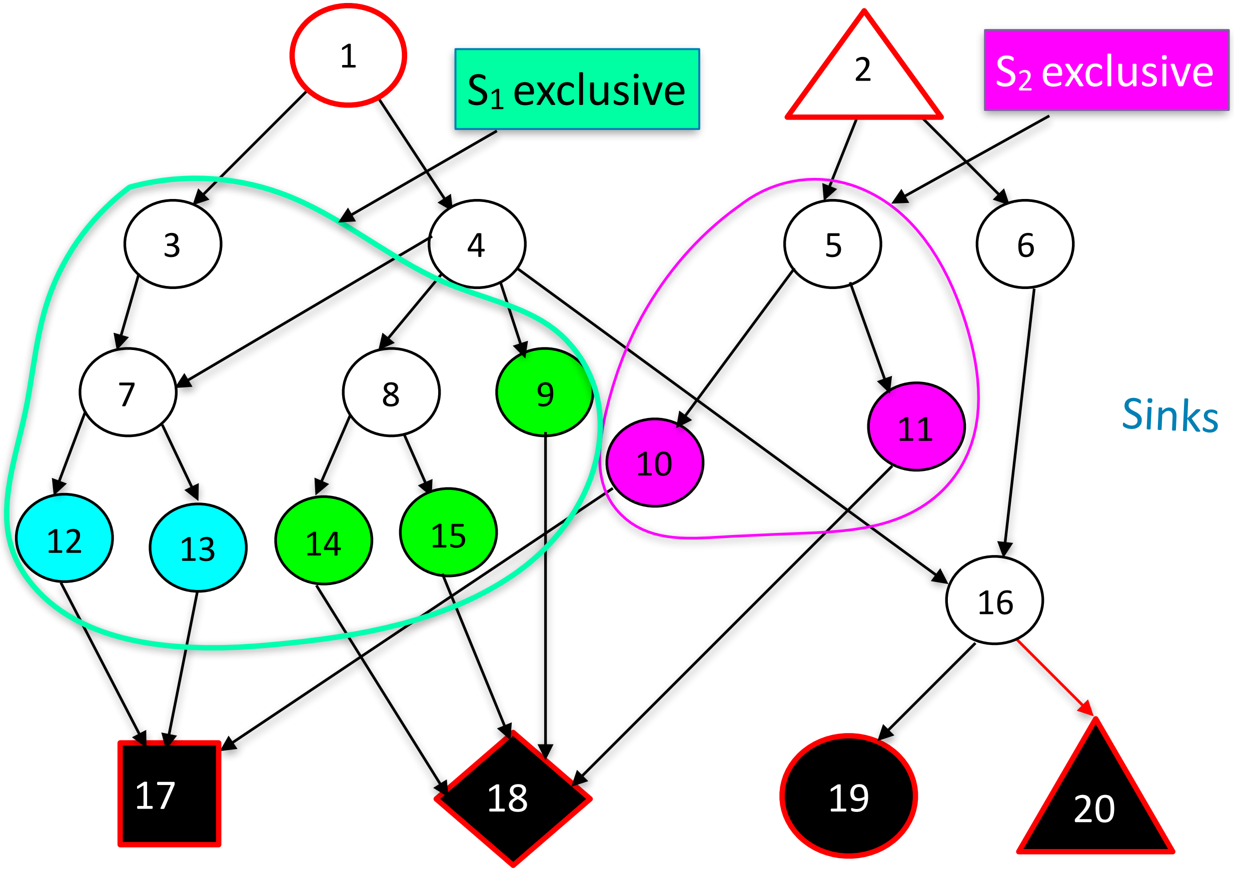
DFG

	1
4	S_1
7	S_1
6	S_2

Sanitizer policy

- Sanitizers applied on one of the exclusive nodes
- The two solutions differ at this stage

Related Work



DFG

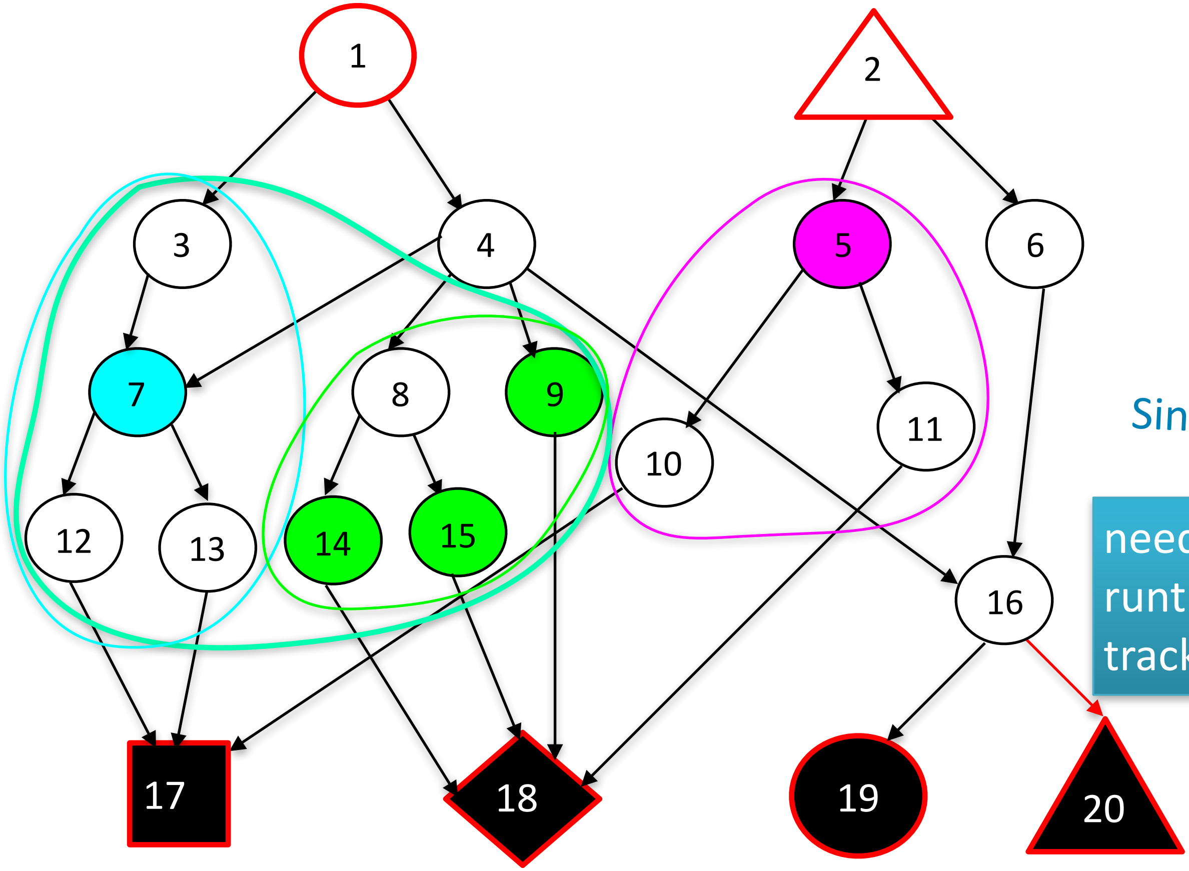
Sources

Sinks

	○	△	∅
■	S_1	S_2	⊥
◆	S_1	S_2	⊥
●	S_4	⊥	⊥
▲	S_3	S_3	⊥
∅	⊥	⊥	⊥

Sanitizer policy

Less-optimized solution



DFG

Sources

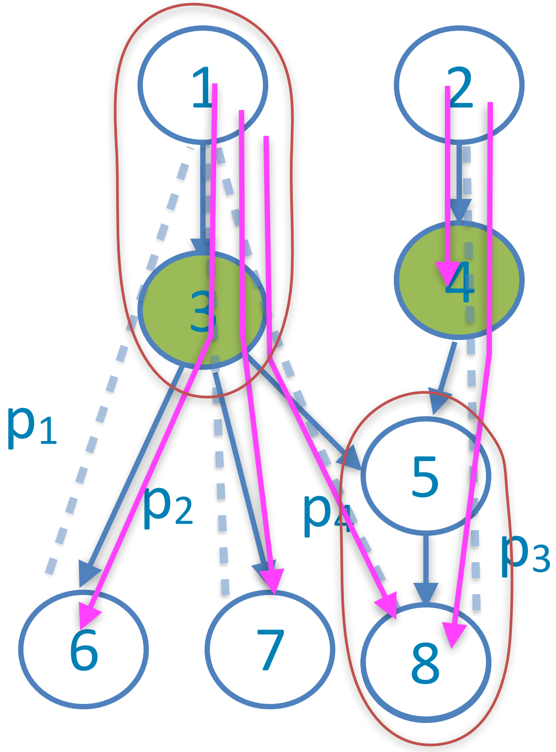
Sinks

needs runtime tracking

	○	△	∅
■	S_1	S_2	⊥
◆	S_1	S_2	⊥
●	S_4	⊥	⊥
▲	S_3	S_3	⊥
∅	⊥	⊥	⊥

Sanitizer policy

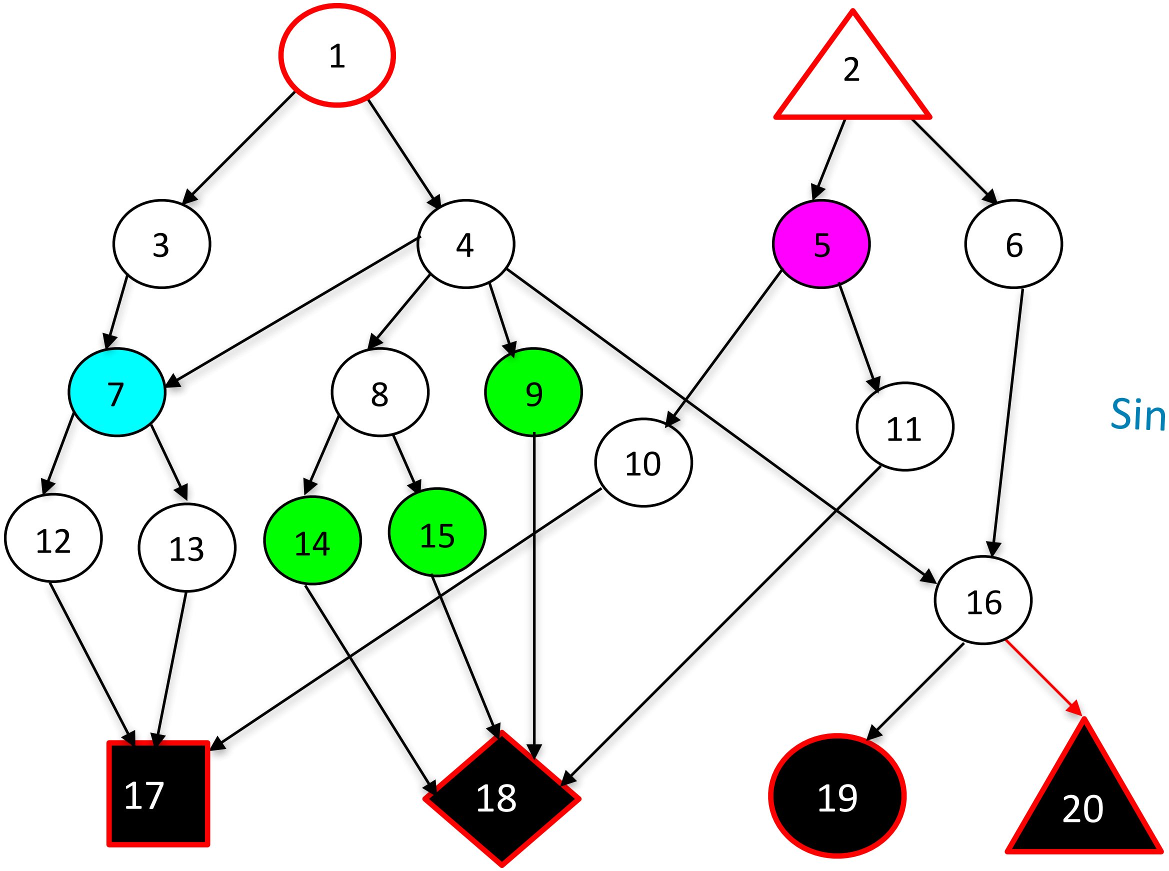
Fully-optimized solution



Sanitizer exclusive DFG

- 1st iteration p_1: n_3 selected
- 2nd iteration p_2: n_3 exist, skip
- 3rd iteration p_3: n_8 selected
- 4th iteration p_4: n_3 and n_8 exist, needs backtracking
  - keep n_3 (3 out edges) & remove n_8 (2 in edges in n_5)
- 5th iteration p_3: n_4 selected

Less-optimized solution



DFG

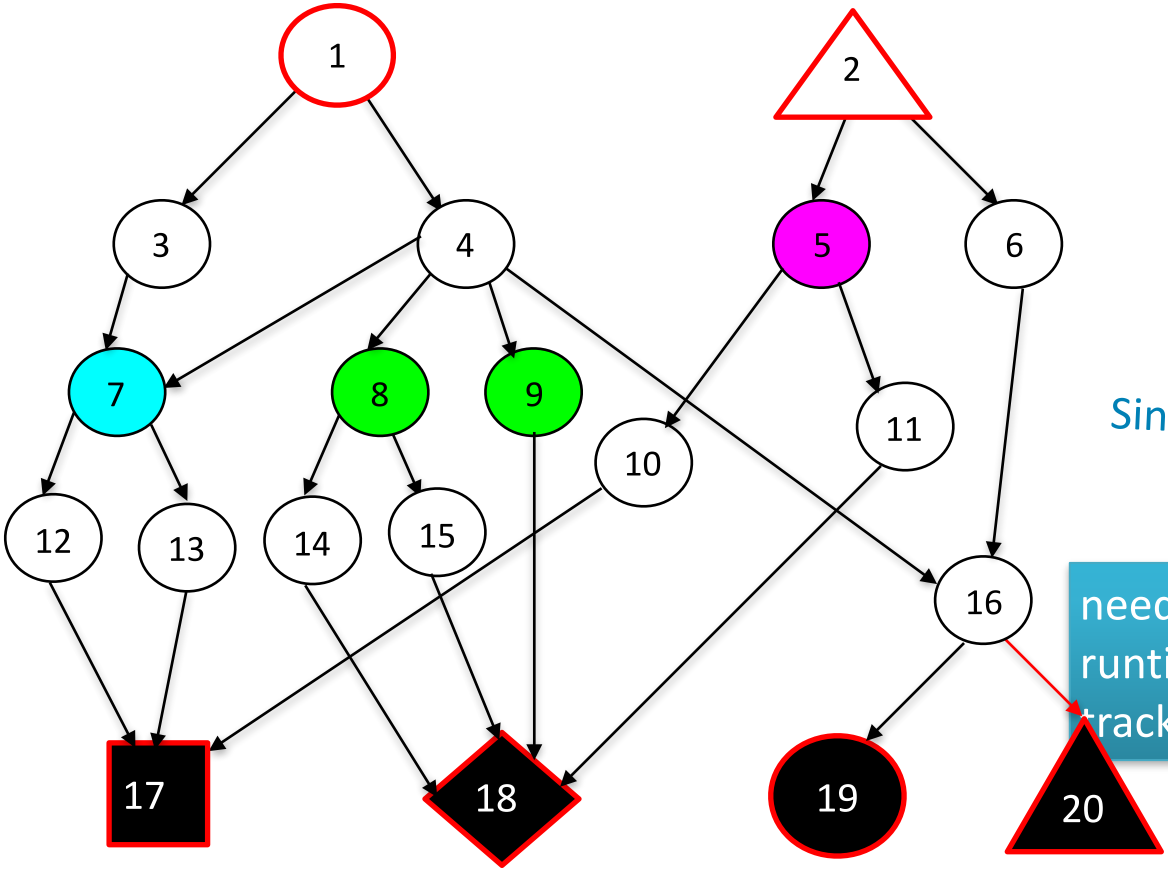
Sources

Sinks

	○	△	∅
■	S_1	S_2	$\bot$
◆	S_1	S_2	$\bot$
●	S_4	$\bot$	$\bot$
▲	S_3	S_3	$\bot$
∅	$\bot$	$\bot$	$\bot$

Sanitizer policy

Fully-optimized solution



needs runtime tracking

DFG

Sources

Sinks	○	△	Ø
■	S_1	S_2	⊥
◆	S_1	S_2	⊥
●	S_4	⊥	⊥
▲	S_3	S_3	⊥
Ø	⊥	⊥	⊥

Sanitizer policy

Demo application setup

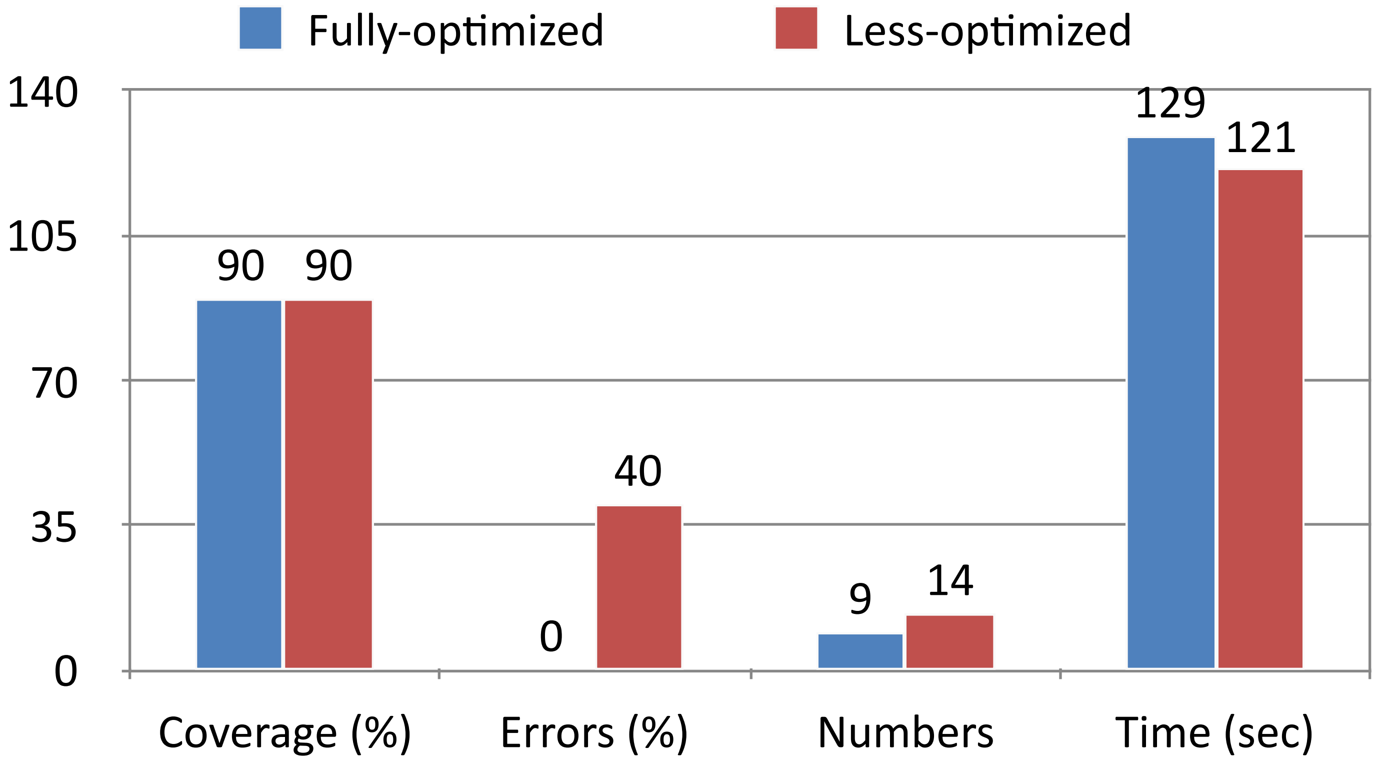
- Application consists of
  - 15,214 nodes
  - 119,026 edges
  - 14,070 methods
  - 724,806 bytes

Size of DFG

N	Nodes
1	1540
2	1427
3	1610
4	1738
5	1790

- N represents the call string context-sensitivity
- Policy defined using:
  - three sources, six sinks and five sanitizer types

Evaluation (2-CFA)

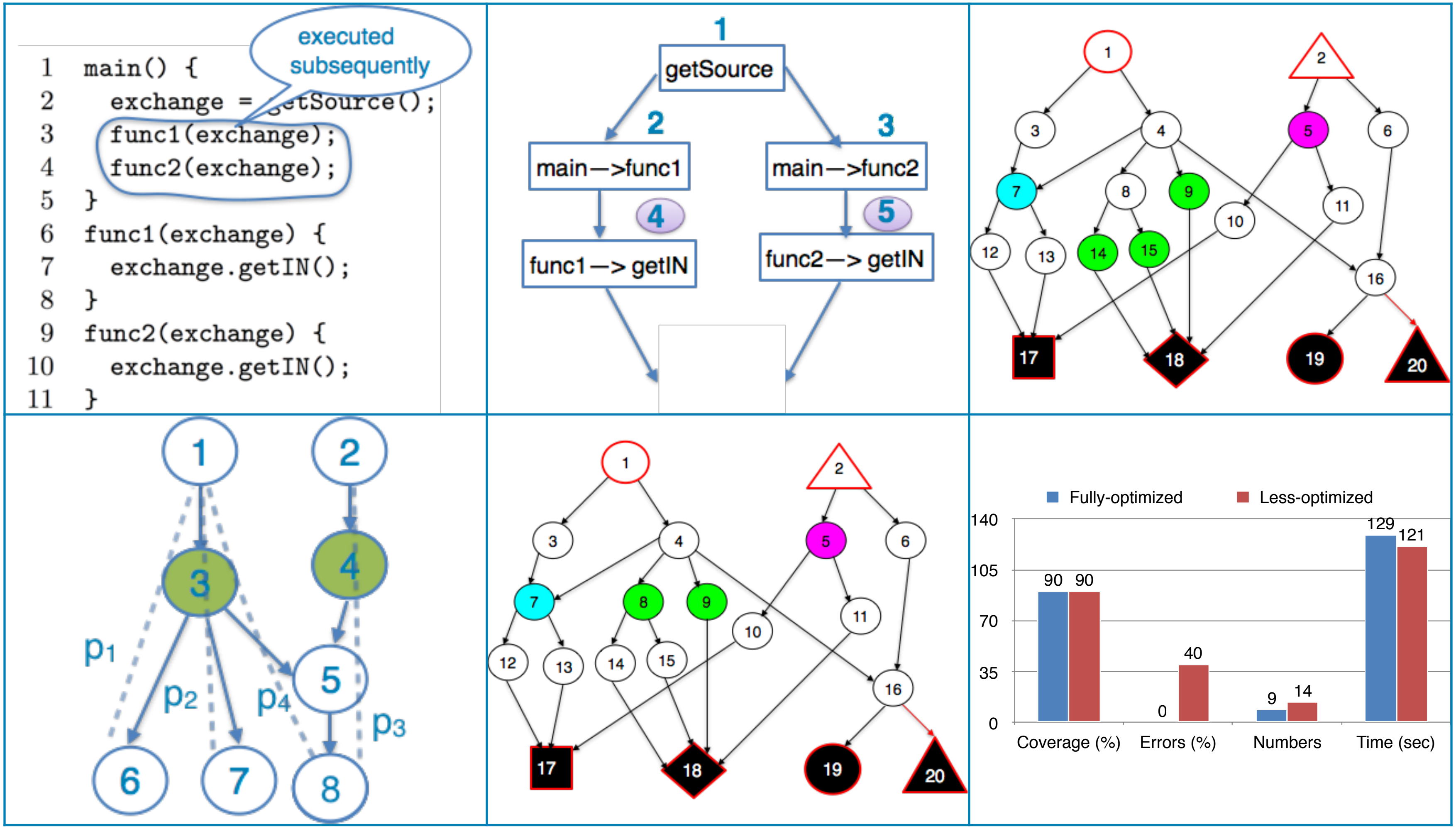


- Less-Optimized illustrates existing approaches
  - Optimizations rarely apply

Conclusion

- Optimized automatic sanitizer placement
  - reduces sanitizer positions
- Mitigates
  - code duplication problem
  - Inconsistent multi-sanitization
- Valuable solution for real world applications
  - Complement by runtime tracking (10%)

Summary



executed subsequently

```
main() {
  exchange = getSource();
  func1(exchange);
  func2(exchange);
}
func1(exchange) {
  exchange.getIN();
}
func2(exchange) {
  exchange.getIN();
}
```