Scalable String Analysis: An Experience Report

Kostyantyn Vorobyov, Yang Zhao, Padmanabhan Krishnan

Oracle Labs, Brisbane, Australia

May, 2021

String Analysis

- Compute values of string expressions (*hotspots*)
- Relevant for many security analyses
  - SQL injections
  - Improper sanitization
  - Unsafe deserialisation
  - Cross-site scripting

- Received much attention but application to large codebases is still unclear
  - Calculating all possible values is not straightforward
  - Often does not scale up

String Analysis in Java™

Java String Analyser (JSA)

- String Value Flow Graph (SVFG) to model string values
- Transform SVFG into a context-free grammar (CFG)
- Approximate CFG to finite automata
- Used in numerous projects in almost two decades
- Aims to be sound but does not scale to large applications

Other tools (OSA, Violist)

- Similar performance problems

[1] Java is a registered trademark of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

String Analysis of Large Codebases

Heavy-weight analysis not always needed

- SQL queries are often encoded as constants concatenated with variables
- String values could be computed using simpler and more scalable techniques

Oracle Labs String Analyser (OLSA)

- Fast and practical string analysis for large Java applications
  - Focus on scalability
  - Unsound

OLSA: High Level View

- Inspired by JSA
- Per-hotspot SVFG extended with context-sensitivity
- Compute possible values using graph traversal
- Generate a set of concrete values with placeholders for unknown (unresolved) parts, e.g.,
  - `“select <XXX> from Users;”`

String Value Flow Graph

- Starting from a hotspot build a SVFG via backwards *def-use* analysis for each method
  - *intra-procedural*
- Method-level SVFG’s connected via call graph
  - *inter-procedural,* SVFG per hotspot
- Graph nodes
  - Constants
  - String operations (e.g., `concat`, `trim`)
  - Switch nodes for parameters and return values

Computing String Values via Graph Traversal

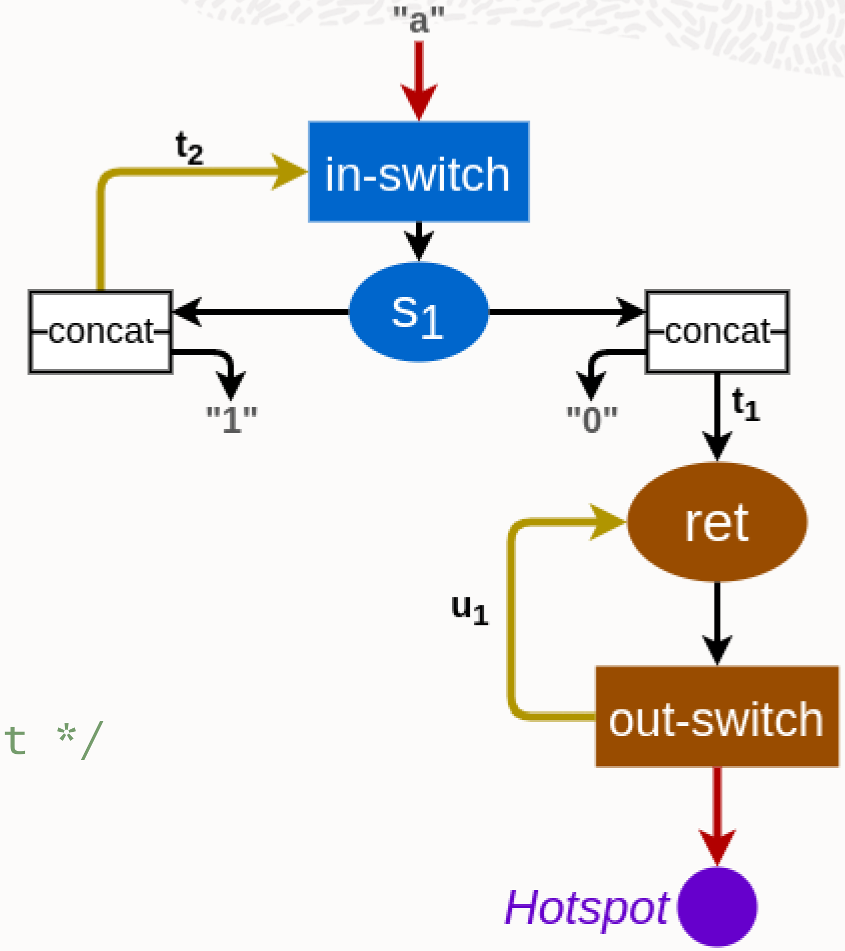
- Propagate constant strings (or unknown values)
- Apply built-in semantics for supported string operations
- Visit cycles once
- Return all string values reachable from hotspots

Example

```
String foo(String s1) {
    if (...) {
        String t1 = s1 + "0";
        return t1;
    } else {
        String t2 = s1 + "1";
        String u1 = foo(t2);
        return u1;
    }
}
...
String result = foo(“a”); /* Hotspot */
```

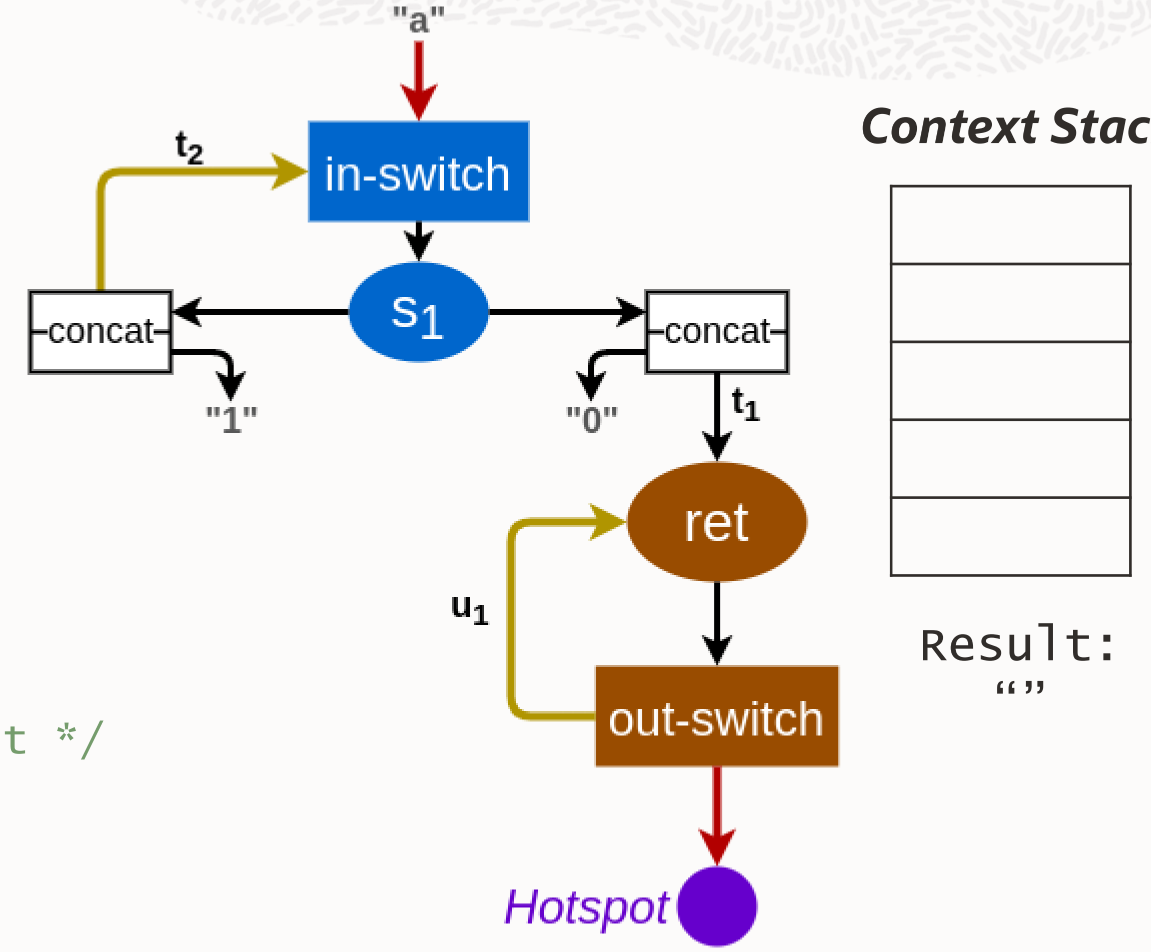
Example

```
String foo(String s1) {
    if (...) {
        String t1 = s1 + "0";
        return t1;
    } else {
        String t2 = s1 + "1";
        String u1 = foo(t2);
        return u1;
    }
}
...
String result = foo(“a”); /* Hotspot */
```



Example

```
String foo(String s1) {
    if (...) {
        String t1 = s1 + "0";
        return t1;
    } else {
        String t2 = s1 + "1";
        String u1 = foo(t2);
        return u1;
    }
}
...
String result = foo(“a”); /* Hotspot */
```

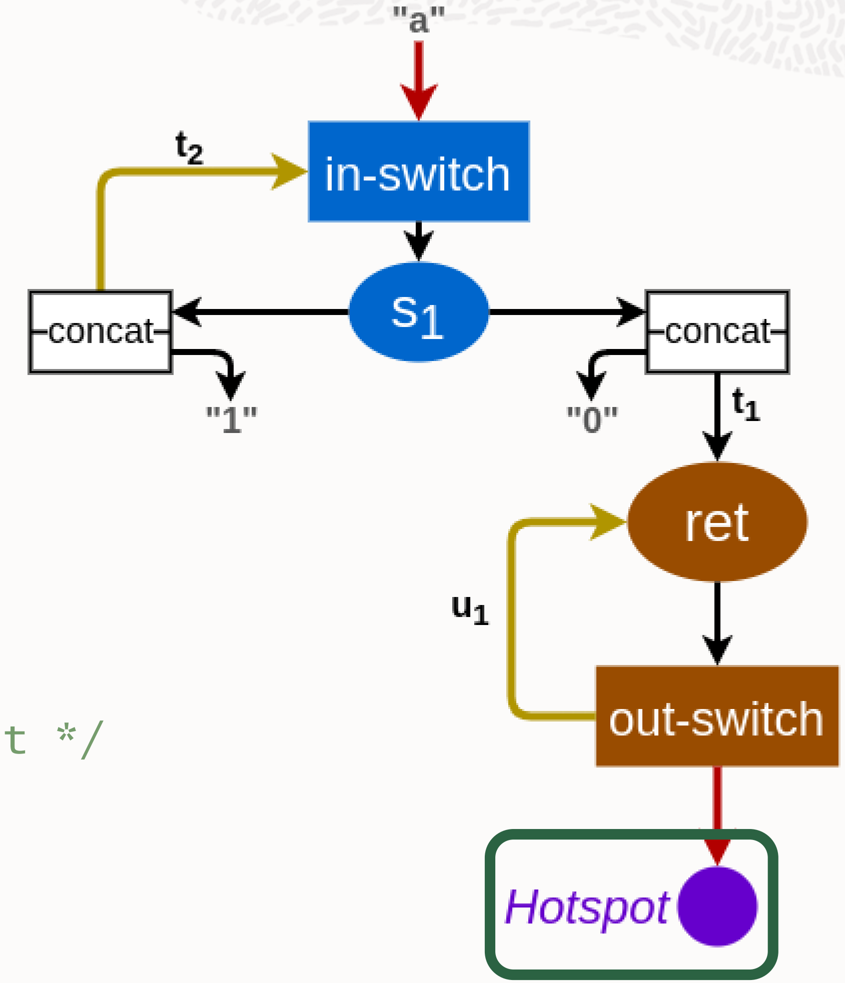


Context Stack

```
Result:
  “”
```

Example

```
String foo(String s1) {
    if (...) {
        String t1 = s1 + "0";
        return t1;
    } else {
        String t2 = s1 + "1";
        String u1 = foo(t2);
        return u1;
    }
}
...
String result = foo(“a”); /* Hotspot */
```

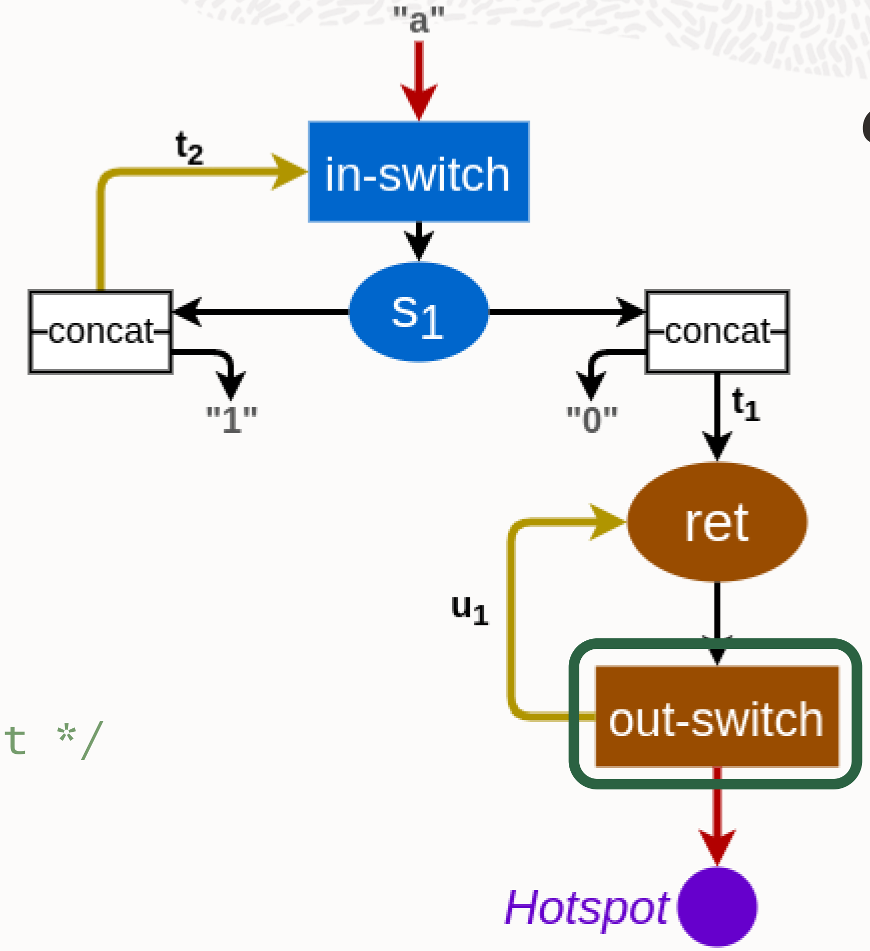


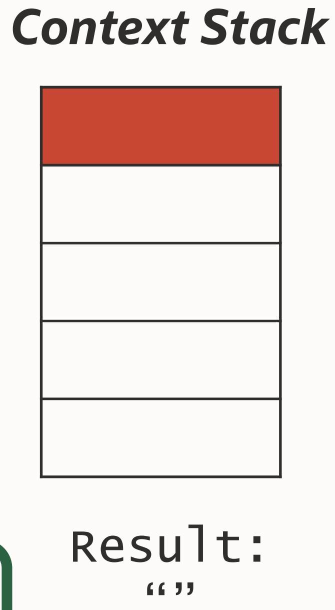
Context Stack

```
Result:
  “”
```

Example

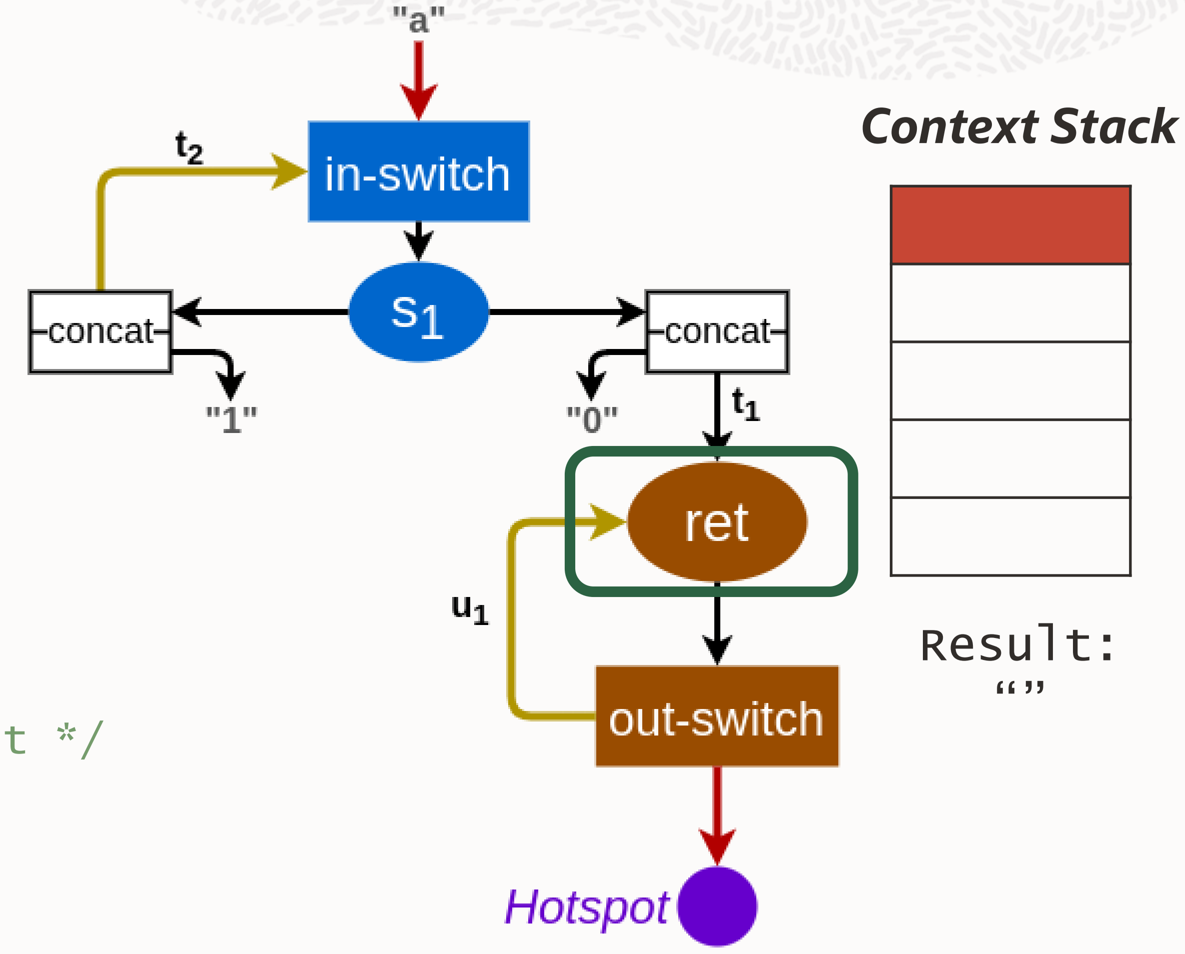
```
String foo(String s1) {
    if (...) {
        String t1 = s1 + "0";
        return t1;
    } else {
        String t2 = s1 + "1";
        String u1 = foo(t2);
        return u1;
    }
}
...
String result = foo("a"); /* Hotspot */
```





Example

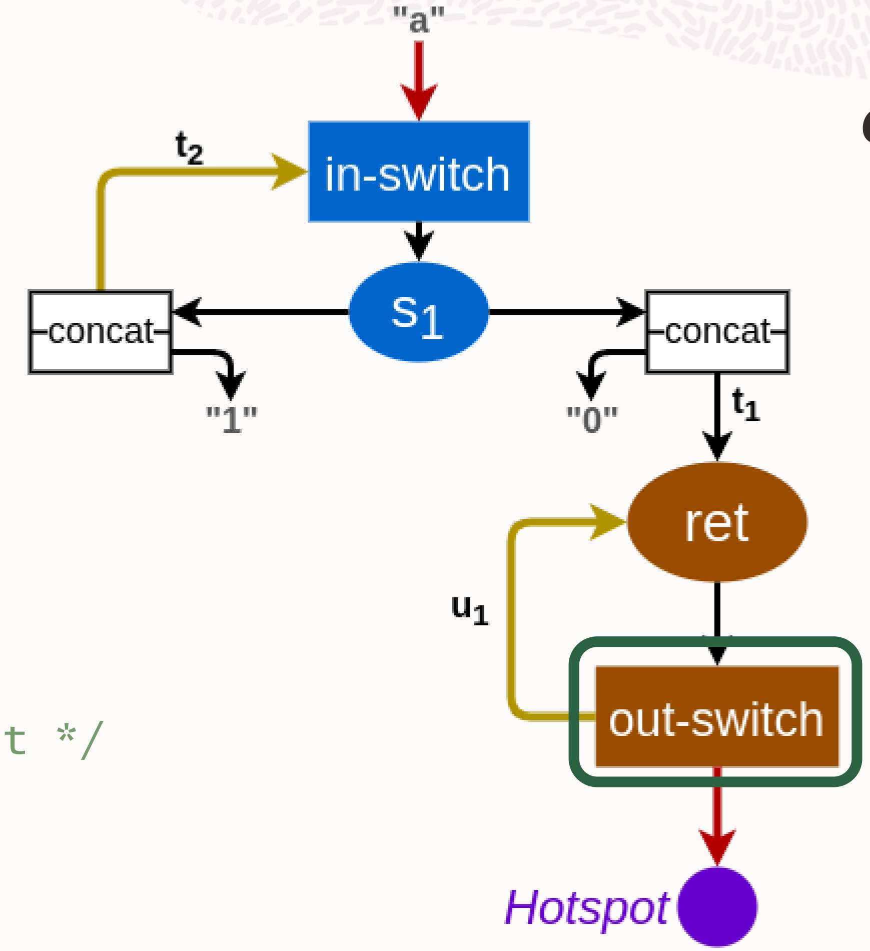
```
String foo(String s1) {
    if (...) {
        String t1 = s1 + "0";
        return t1;
    } else {
        String t2 = s1 + "1";
        String u1 = foo(t2);
        return u1;
    }
}
...
String result = foo(“a”); /* Hotspot */
```

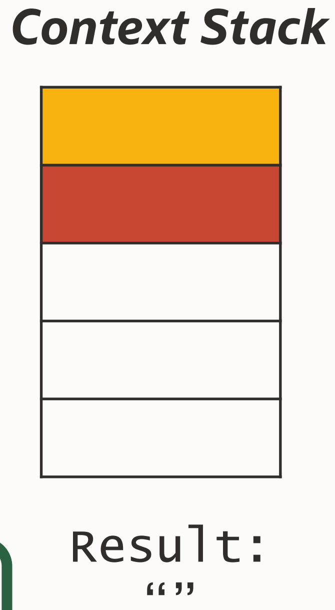


Result:
“”

Example

```
String foo(String s1) {
    if (...) {
        String t1 = s1 + "0";
        return t1;
    } else {
        String t2 = s1 + "1";
        String u1 = foo(t2);
        return u1;
    }
}
...
String result = foo("a"); /* Hotspot */
```

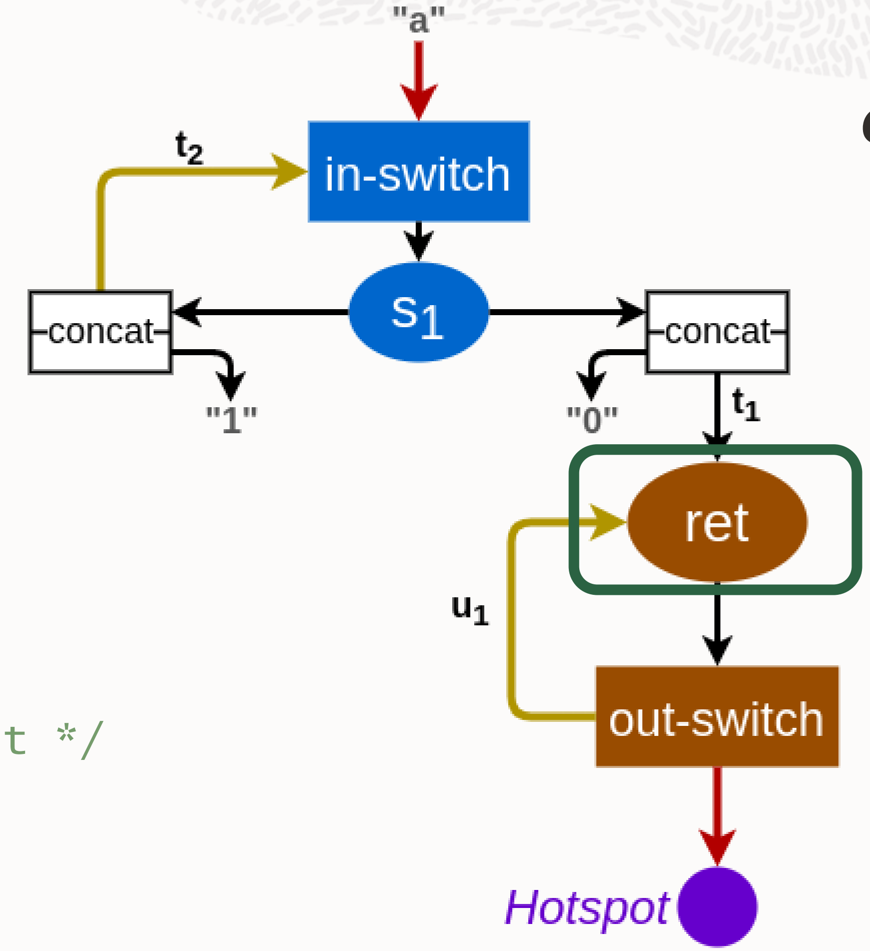




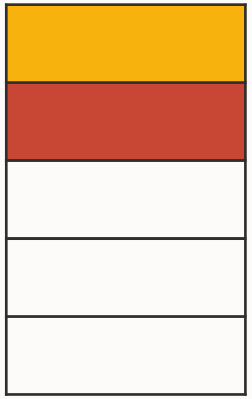
Result:
“”

Example

```
String foo(String s1) {
    if (...) {
        String t1 = s1 + "0";
        return t1;
    } else {
        String t2 = s1 + "1";
        String u1 = foo(t2);
        return u1;
    }
}
...
String result = foo(“a”); /* Hotspot */
```



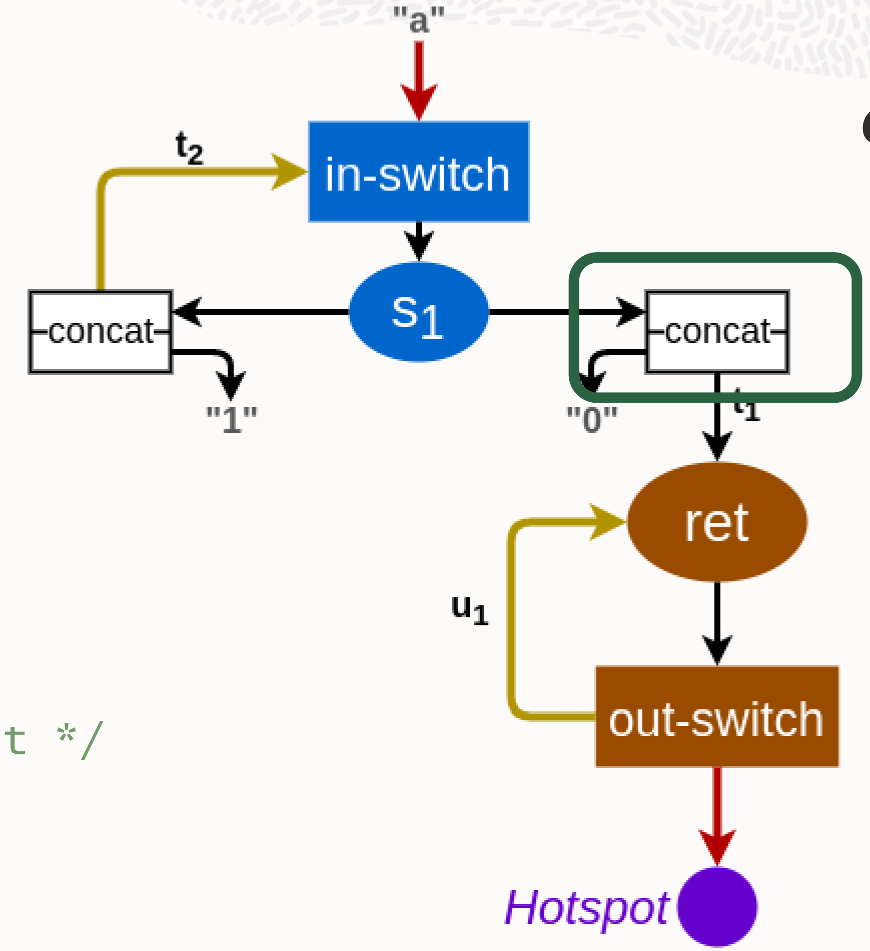
Context Stack



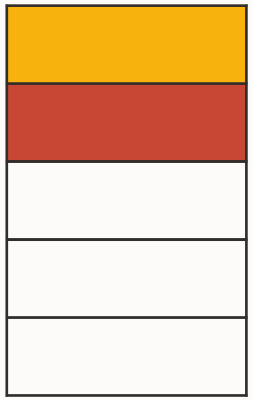
Result:
“”

Example

```
String foo(String s1) {
    if (...) {
        String t1 = s1 + "0";
        return t1;
    } else {
        String t2 = s1 + "1";
        String u1 = foo(t2);
        return u1;
    }
}
...
String result = foo(“a”); /* Hotspot */
```



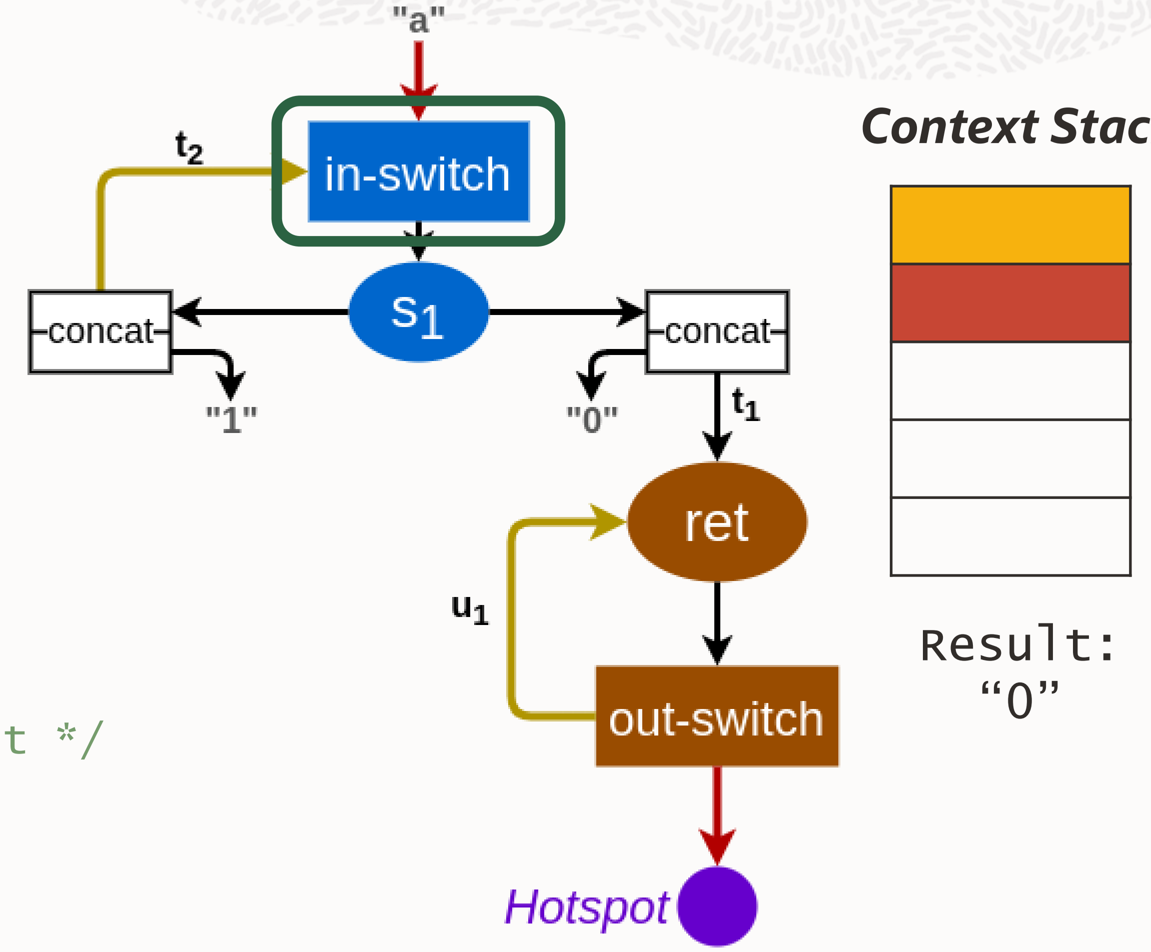
Context Stack



```
Result:
  “0”
```

Example

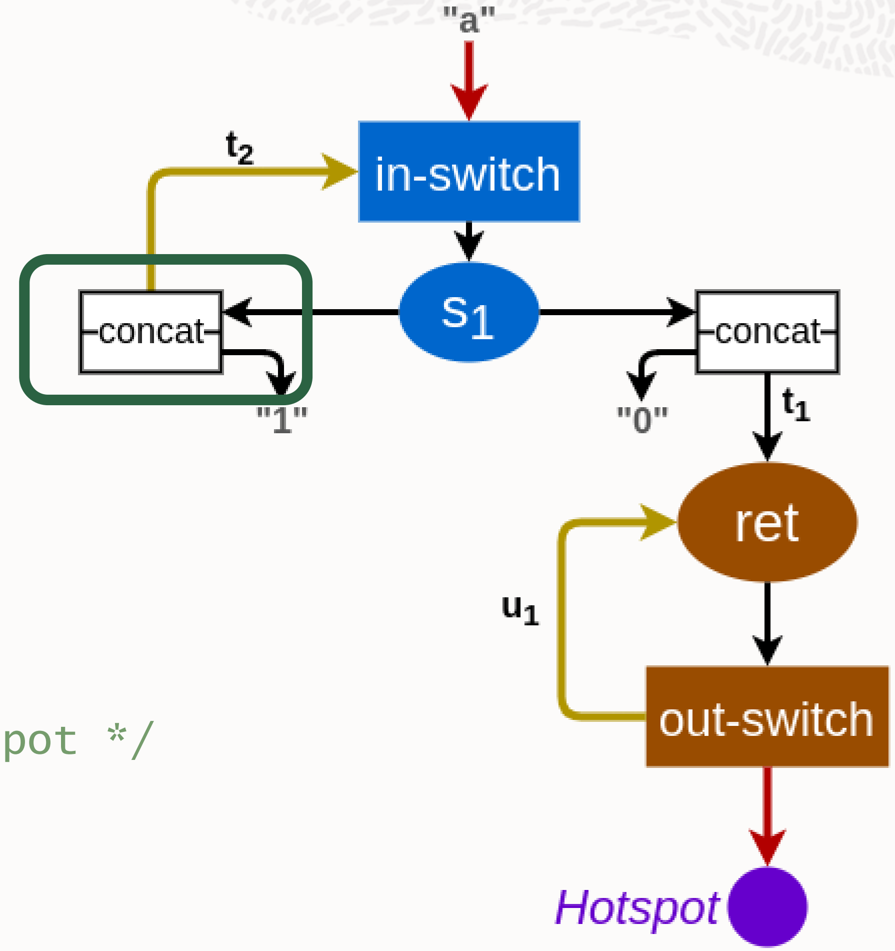
```
String foo(String s1) {
    if (...) {
        String t1 = s1 + "0";
        return t1;
    } else {
        String t2 = s1 + "1";
        String u1 = foo(t2);
        return u1;
    }
}
...
String result = foo(“a”); /* Hotspot */
```

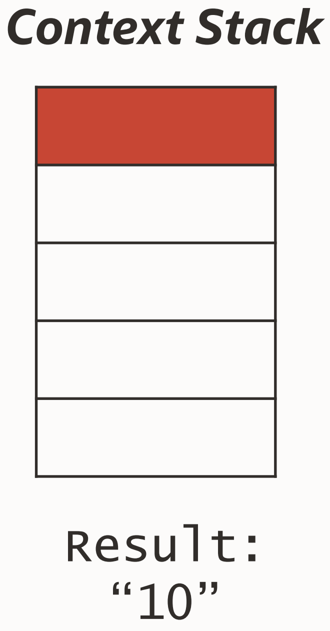


Result:
“0”

Example

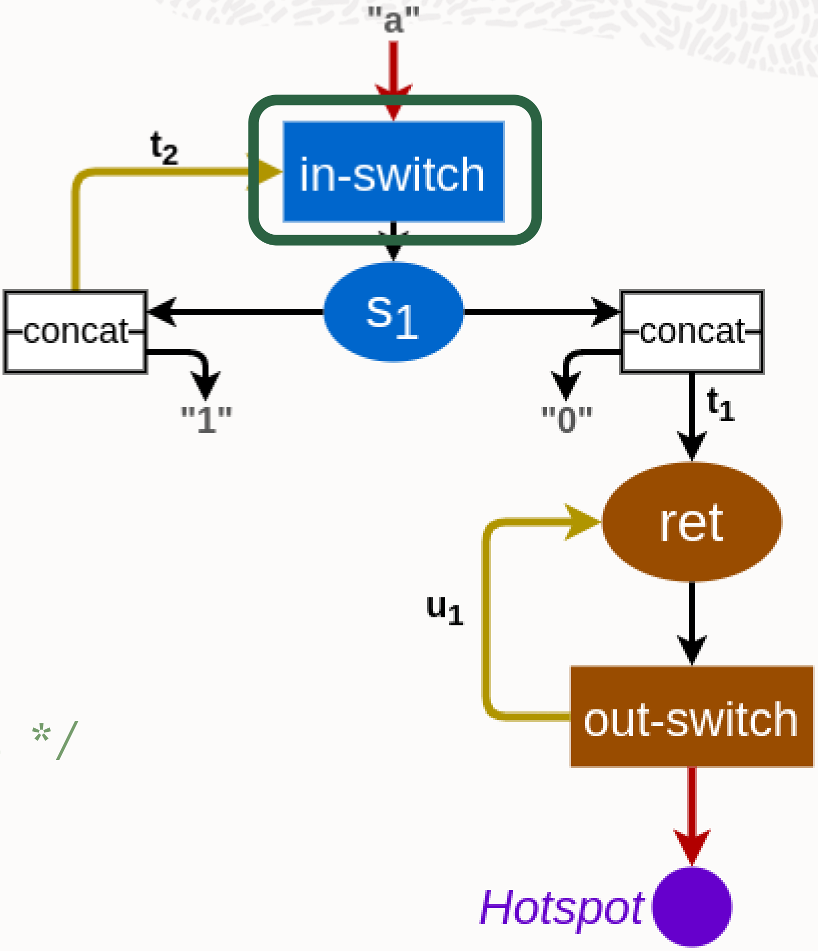
```
String foo(String s1) {
    if (...) {
        String t1 = s1 + "0";
        return t1;
    } else {
        String t2 = s1 + "1";
        String u1 = foo(t2);
        return u1;
    }
}
...
String result = foo(“a”); /* Hotspot */
```





Example

```
String foo(String s1) {
    if (...) {
        String t1 = s1 + "0";
        return t1;
    } else {
        String t2 = s1 + "1";
        String u1 = foo(t2);
        return u1;
    }
}
...
String result = foo(“a”); /* Hotspot */
```

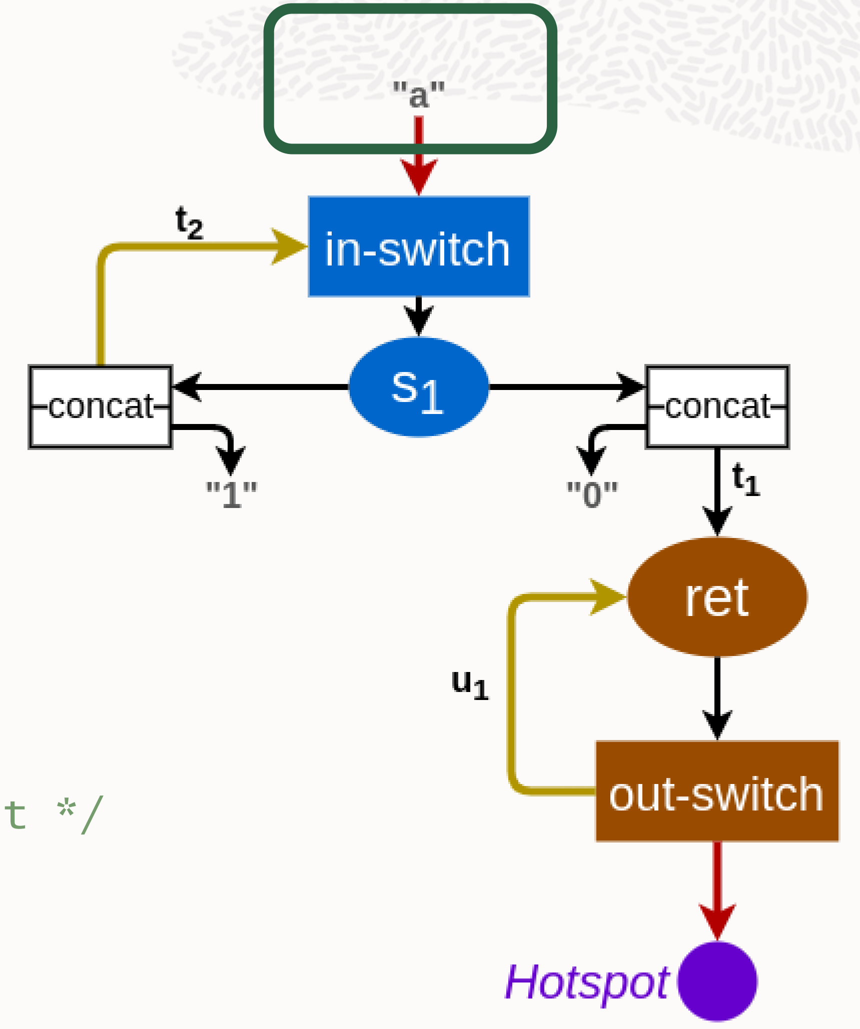


Context Stack

Result:
“10”

Example

```
String foo(String s1) {
    if (...) {
        String t1 = s1 + "0";
        return t1;
    } else {
        String t2 = s1 + "1";
        Stringu1 = foo(t2);
        return u1;
    }
}
...
String result = foo("a"); /* Hotspot */
```



Context Stack

```
Result:
 "a10"
```

Experiments

Precision

- Comparison with JSA
- Benchmarks from the DaCapo Suite
- JSA Unit tests

Scalability

- Large internal enterprise system consisting of smaller applications

JSA Unit Tests

- 303 small test programs from JSA test suite
- Single hotspot with hard-coded inputs
- Outputs the exact set of strings an ideal analysis should compute

Category	OLSA	JSA
Exact match	15%	32%
Partial	17%	30%
Incorrect	68%	38%

- JSA was more precise with 62% of fully and partially resolved strings
- OLSA could not correctly identify strings in 68%
  - Reason: lack of support for arrays, data-structures, class fields, global variables

DaCapo

- Hotspot configuration
  - Print functions from `java.io`, `javax.servlet.jsp`
  - `java.class.forName`

Benchmark	Hotspots	JSA Runtime	OLSA Runtime
Entire codebase	4,304	-	8.77s
bloat	748	5m 24s	0.66s
avrora	40	15m 41s	0.25s
sunflow	91	1m 38s	0.08s

DaCapo (cont.)

Benchmark	JSA Resolved		OLSA Resolved	
	Fully	Partially	Fully	Partially
xalan	-	-	40%	36%
derby	-	-	30%	47%
cassandra	-	-	34%	24%
bloat	53%	30%	36%	58%
avrora	43%	28%	38%	45%
sunflow	65%	4%	64%	4%

- Both tools resolved approx. 70% of hotspots
- In `avrora` and `sunflow` OLSA resolved more strings overall but only partially
- JSA fully resolved more strings then OLSA

Commercial System: SQL Injections

- 32 MLoC Enterprise system consisting of smaller applications
- Hotspot configuration
  - Java JDBC query methods (e.g., `java.sql.executeQuery`)
  - Motivated by client security analysis where string arguments should not be tainted

Size (KLoC)	Hotspots	OSLA Runtime	Resolved
Entire codebase	33,966	2h 25m	78.8%
3,048	5,896	12m	61.5%
1,821	3,270	6m	85.1%
953	2,248	15m	81.5%
858	2,059	10m	55.3%

Conclusion

- Computing precise string expressions is not always useful
  - For specific (e.g., security) problems simpler (but faster) analysis suffices

- Intra-procedural data-flow with context-sensitivity gives necessary scalability
  - OSLA can analyze large codebases with approx. 80% of strings resolved
  - Unresolved strings can be addressed by adding more features (e.g., field sensitivity)

Thank you

ORACLE