ORACLE®

Affogato: Runtime Detection of Injection Attacks for Node.js

François Gauthier, Behnaz Hassanshahi, Alexander Jordan

SOAP 2018

Oracle Labs Australia

Safe Harbor Statement

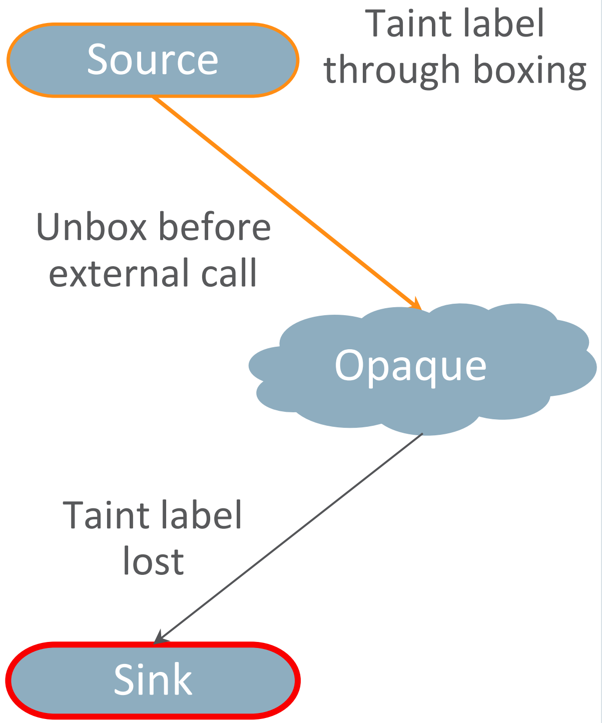
The following is intended to provide some insight into a line of research in Oracle Labs. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. Oracle reserves the right to alter its development plans and practices at any time, and the development, release, and timing of any features or functionality described in connection with any Oracle product or service remains at the sole discretion of Oracle. Any views expressed in this presentation are my own and do not necessarily reflect the views of Oracle.

Why Investigate Node.js Security?

- Direct access to OS resources (file system, network) and databases
- No built-in security mechanisms
- Our Focus: taint analysis to detect injection attacks
  - **Still #1 vulnerability in OWASP Top 10 2017**
  - Web-servers are most popular class of Node.js applications in Github
    - Express: 39.1k ★ , Koa: 22k ★ , hapi: 9.7k ★, Restify: 8.4k ★, Fastify 7.5k ★

Challenges for Dynamic Taint Analysis

- Engine instrumentation approaches:
  - Engine creates and propagates taint labels
  - Very hard to maintain if you don’t own the engine, not flexible, too low-level
- Source instrumentation approaches:
  - Taint labels applied by wrapping primitives and extending objects with taint fields
  - Brittle, unsound w.r.t opaque code (see figure on right)



Prevalence of Opaque code

- Key observations
  - Calls to opaque functions are prevalent
  - JavaScript coerces most values to strings at runtime in Node.js
  - Taint-sensitive locations (i.e., operations that sanitise, validate or transform tainted inputs) are few

Built-in Object	Mean	Standard Deviation
Array	9.3	7.05
String	24.6	12.82
RegExp	1.73	4.84
All	62.47	21.41

Percentage of calls to opaque functions in our benchmarks

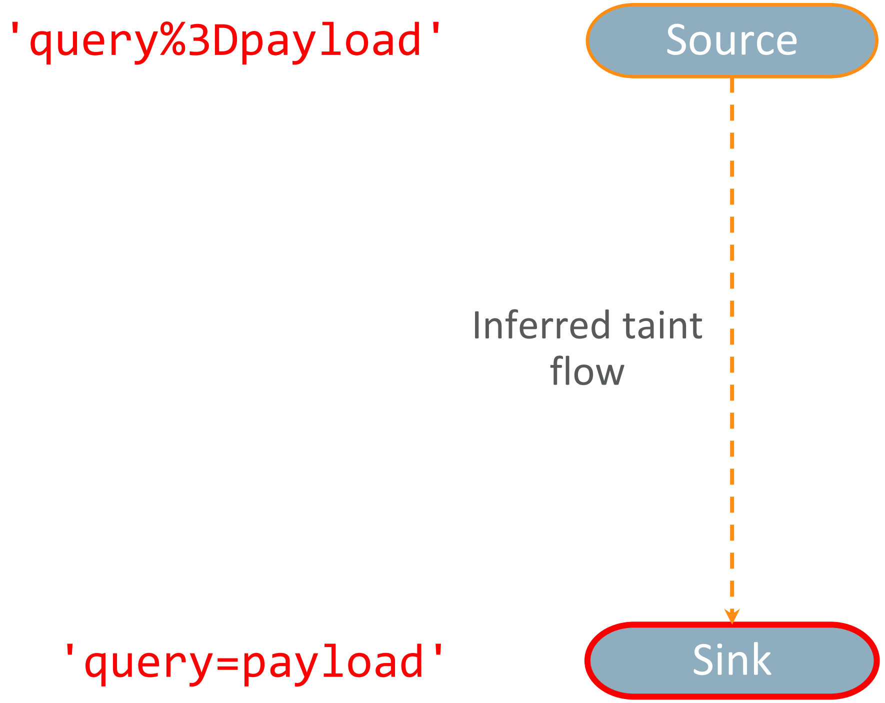
The Affogato Approach

- Instrumentation-based grey-box taint analysis
  - Combines **black-box** reasoning with **white-box** program analysis
  - White-box analysis at selective taint-sensitive locations (watchpoints)
    - At watchpoints, values are observed but never modified (non-intrusive)
  - Black-box reasoning infers data flows between watchpoints
- Non-intrusive analysis works well with source-level instrumentation
  - We use Jalangi2

ORACLE

Black-box Taint Inference Between Watchpoints

- Uses string similarity to infer taint flows.
  - Based on edit-distance Pros:
    - Lightweight, i.e. limited amount of instrumentation
    - Robust (e.g. does not break the application)
  - Cons:
    - May introduce spurious taint flows (FPs)
    - May miss valid taint flows (FNs)



'query%3Dpayload'

Source

Inferred taint flow

'query=payload'

Sink

White-box Program Analysis

- At selective watchpoints
  - Handles string transformations
  - Unpacks strings
- Introduces the concept of dynamic request sensitivity
  - Deals with asynchronous nature of Node.js

White-box Program Analysis

String transformations

- At selective watchpoints, Affogato
  - Preserves taints
  - Unpacks strings
  - Removes taints (sanitizer)

```
//url is "localhost:8000?%24where=1%3D%3D1"
function (req, res) {
  var query = querystring.parse(req.url);
  //query is {"$where":"1==1"}
  mongo.collection.find(escape(query), {},
    function(e, docs)){});
}
```

White-box Program Analysis

String transformations

- At selective watchpoints, Affogato
  - **Preserves taints**
  - Unpacks strings
  - Removes taints (sanitizer)

```
//url  is "localhost:8000?%24where=1%3D%3D1"
function (req, res) {
   var query = querystring.parse(req.url);
   //query is {"$where":"1==1"}
   mongo.collection.find(escape(query), {},
      function(e, docs)){});
}
```

White-box Program Analysis

String transformations

- At selective watchpoints, Affogato
  - Preserves taints
  - **Unpacks strings**
  - Removes taints (sanitizer)

```
//url is "localhost:8000?%24where=1%3D%3D1"
function (req, res) {
  var query = querystring.parse(req.url);
  //query is {"$where":"1==1"}
  mongo.collection.find(escape(query), {},
    function(e, docs)){});
}
```

White-box Program Analysis

String transformations

- At selective watchpoints, Affogato
  - Preserves taints
  - Unpacks strings
  - **Removes taints (sanitizer)**

```
//url is "localhost:8000?%24where=1%3D%3D1"
function (req, res) {
  var query = querystring.parse(req.url);
  //query is {"$where":"1==1"}
  mongo.collection.find(escape(query), {},
    function(e, docs)){});
}
```

White-box Program Analysis

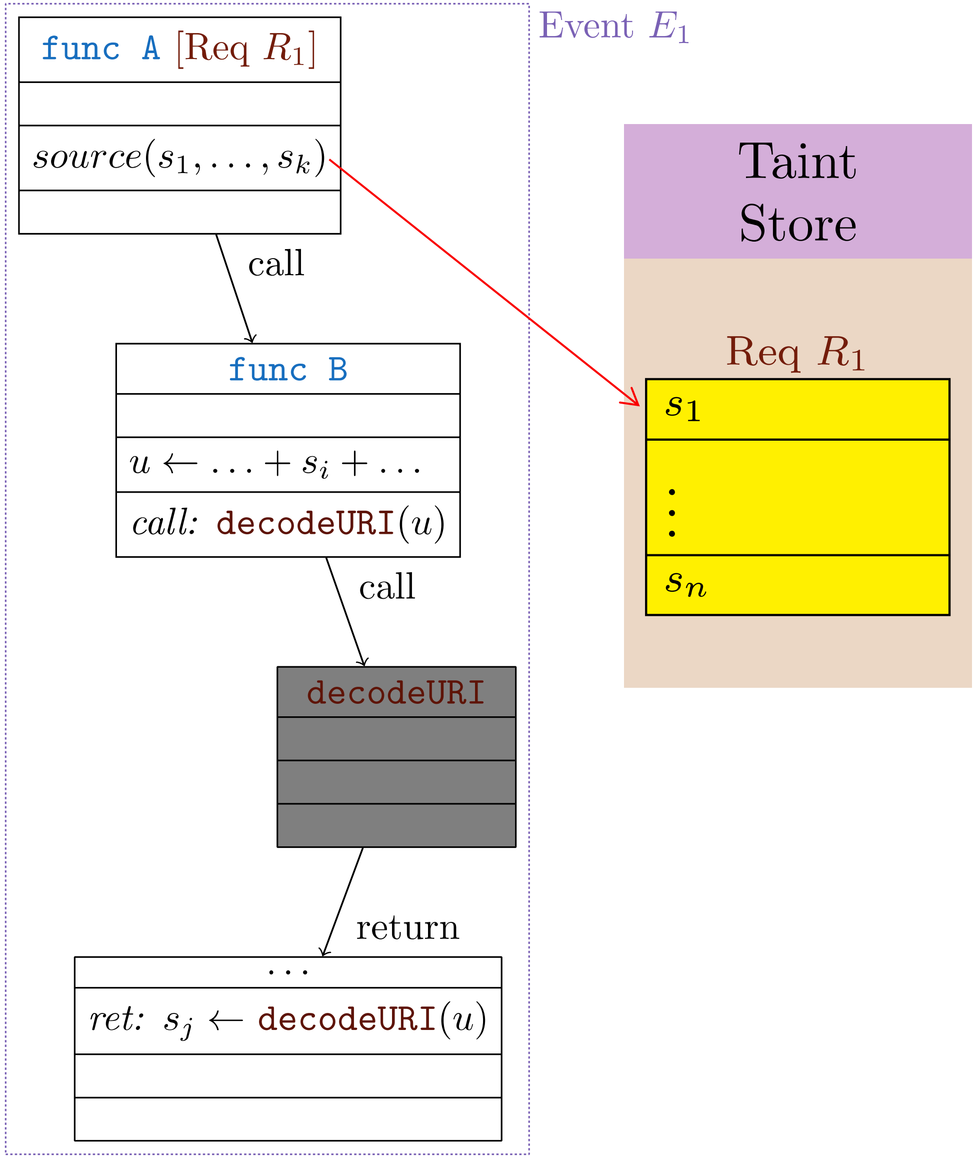
Dynamic request sensitivity

- Intertwined computations from different requests
  - Node.js allows serving multiple requests asynchronously
  - Might result in FP, e.g., tainted string values from request A inadvertently correlate with untainted string values from request B
- Request-sensitivity
  - Analogous to call site or object sensitivity in static analysis
  - Avoids cross-request correlations
  - We use request IDs to correlate request and response objects

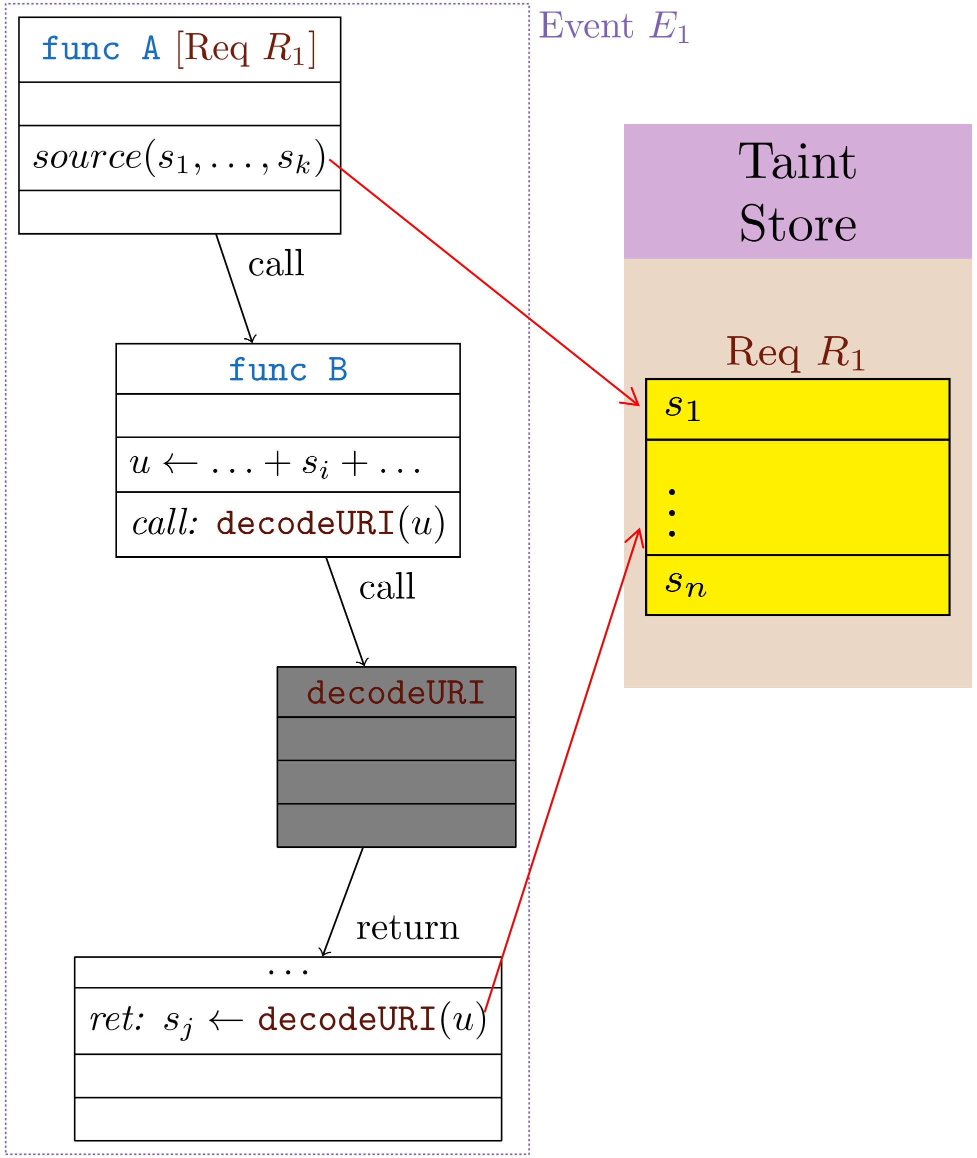
Example

Taint
Store

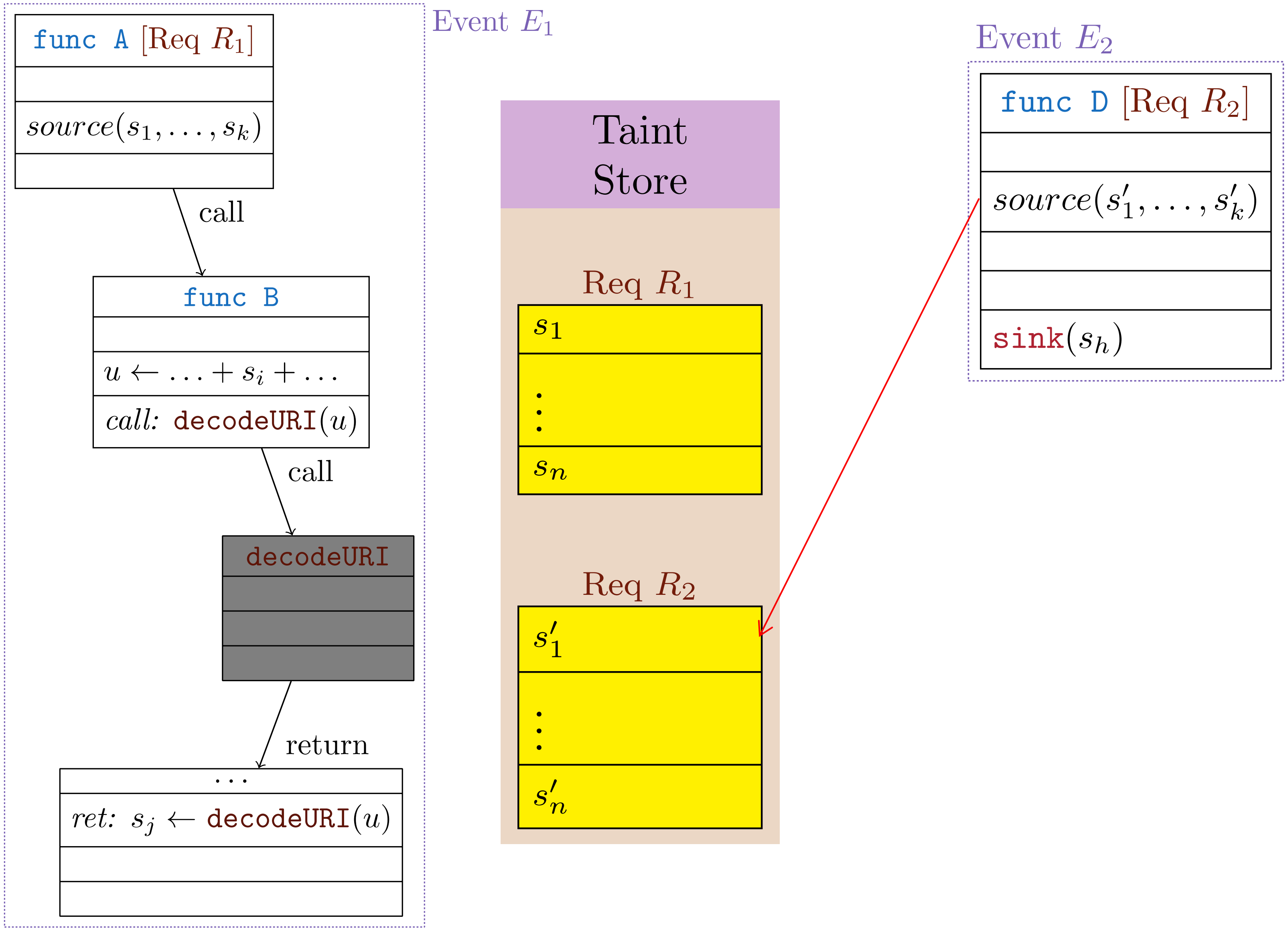
Example


Event E_1
func A [Req R_1]
$source(s_1, \ldots, s_k)$
call
func B
$u \leftarrow \ldots + s_i + \ldots$
call: decodeURI(u)
call
decodeURI
return
...
ret: $s_j \leftarrow$ decodeURI(u)
Taint Store
Req R_1
s_1
$\vdots$
s_n

Example

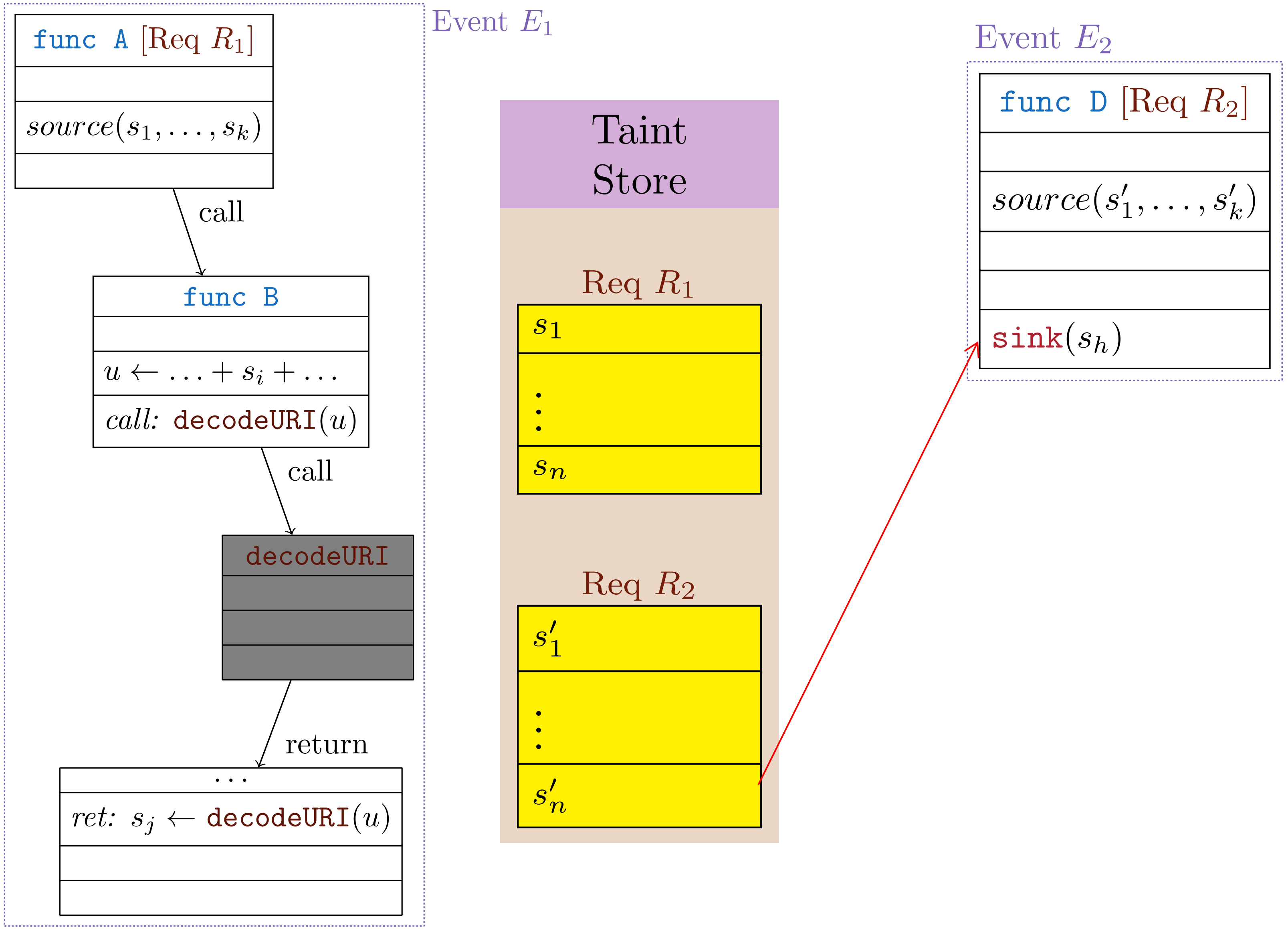

Event E_1
func A [Req R_1]
$source(s_1, \ldots, s_k)$
call
func B
$u \leftarrow \ldots + s_i + \ldots$
call: decodeURI(u)
call
decodeURI
return
$\ldots$
ret: $s_j \leftarrow$ decodeURI(u)
Taint Store
Req R_1
s_1
$\vdots$
s_n

Example

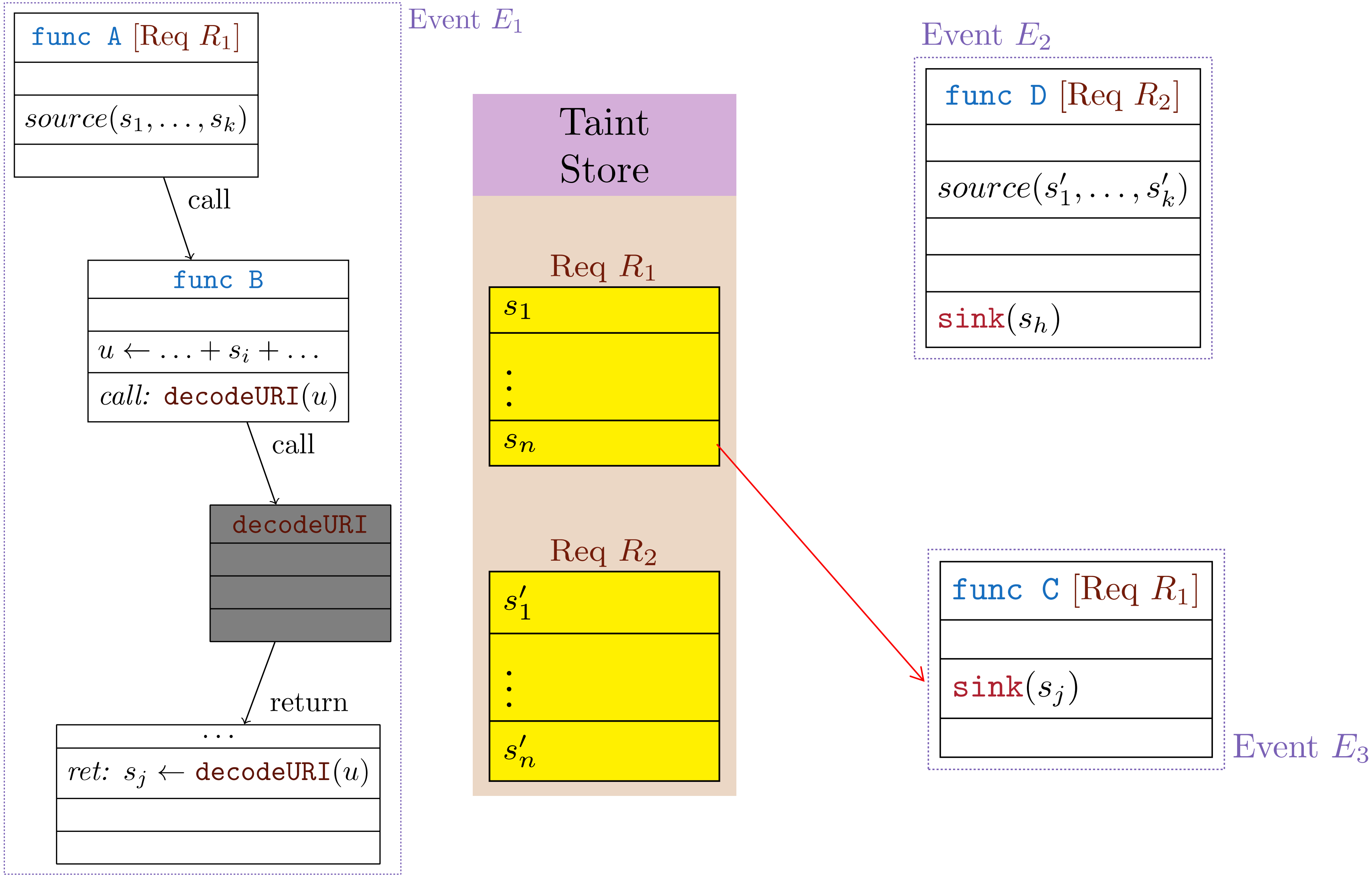


ORACLE

Example


Event E_1
func A [Req R_1]
$source(s_1, \ldots, s_k)$
call
func B
$u \leftarrow \ldots + s_i + \ldots$
call: decodeURI(u)
call
decodeURI
return
. . .
ret: $s_j \leftarrow$ decodeURI(u)
Taint Store
Req R_1
s_1
$\vdots$
s_n
Req R_2
s'_1
$\vdots$
s'_n
Event E_2
func D [Req R_2]
$source(s'_1, \ldots, s'_k)$
sink(s_h)

Example


Event E_1
func A [Req R_1]
$source(s_1, \ldots, s_k)$
call
func B
$u \leftarrow \ldots + s_i + \ldots$
call: decodeURI(u)
call
decodeURI
return
$\ldots$
ret: $s_j \leftarrow$ decodeURI(u)
Taint Store
Req R_1
s_1
$\vdots$
s_n
Req R_2
s'_1
$\vdots$
s'_n
Event E_2
func D [Req R_2]
$source(s'_1, \ldots, s'_k)$
sink(s_h)
func C [Req R_1]
sink(s_j)
Event E_3

Implementation

- We use introspection to identify watchpoints based on configurations
  - To get reference to runtime objects
  - In contrast with signature-based approaches
  - E.g., functions are first-class citizens
    - so both `send` and `req.end` should be considered sinks when

```
var send = req.end
```

- Instrumentation
  - Currently we use Jalangi2
  - We plan to move to NodeProf, a new instrumentation framework for GraalVM
    - Supports ES6+
    - Features: selective instrumentation, built-in and library scope

Evaluation

- Average (1000 exec.) runtime overhead:
  - Instrumentation: 4.70 ×
  - Analysis: 1.19 ×

Benchmark	Finds vuln.	FP
Node Advisory	Yes	No
Node Advisory (fixed)	No	No
Synode [C-A. Staicu et al. NDSS'18]	Yes	No
NodeGoat	Yes	No

ORACLE

Evaluation

- Average (1000 exec.) runtime overhead:
  - Instrumentation: 4.70 ×
  - Analysis: 1.19 ×
- Effectiveness
  - No false positives

Benchmark	Finds vuln.	FP
Node Advisory	Yes	No
Node Advisory (fixed)	No	No
Synode [C-A. Staicu et al. NDSS'18]	Yes	No
NodeGoat	Yes	No

ORACLE

Evaluation

- Average (1000 exec.) runtime overhead:
  - Instrumentation: 4.70 ×
  - Analysis: 1.19 ×
- Effectiveness
  - No false positives
  - Sanitisers

Benchmark	**Finds vuln.**	**FP**
Node Advisory	Yes	No
Node Advisory (fixed)	No	No
Synode [C-A. Staicu et al. NDSS’18]	Yes	No
NodeGoat	Yes	No

Evaluation

- Average (1000 exec.) runtime overhead:
  - Instrumentation: 4.70 ×
  - Analysis: 1.19 ×
- Effectiveness
  - No false positives
  - Sanitisers
  - Request sensitivity removes 3 FPs in mongui

Benchmark	**Finds vuln.**	**FP**
Node Advisory	Yes	No
Node Advisory (fixed)	No	No
Synode [C-A. Staicu et al. NDSS'18]	Yes	No
NodeGoat	Yes	No

Evaluation

- Average (1000 exec.) runtime overhead:
  - Instrumentation: 4.70 ×
  - Analysis: 1.19 ×
- Effectiveness
  - No false positives
  - Sanitisers
  - Request sensitivity removes 3 FPs in mongui
  - Practicality
    - No FP even when fuzzed
    - Synode times out (1 hour) and prevents load of NodeGoat

Benchmark	**Finds vuln.**	**FP**
Node Advisory	Yes	No
Node Advisory (fixed)	No	No
Synode [C-A. Staicu et al. NDSS'18]	Yes	No
NodeGoat	Yes	No

Conclusion

- We presented a **grey-box** taint analysis for Node.js
  - Supports opaque code
  - Successfully analyzes real-world applications
  - Can be more precise with more sophisticated program analysis
- This is just a starting point to find the sweetspot

Black-box reasoning

Sound and precise analysis

Questions?

New instrumentation framework for Node.js: NodeProf running on **GraalVM**
https://github.com/Haiyang-Sun/nodeprof.js

francois.gauthier, behnaz.hassanshahi, alexander.jordan
@oracle.com

Affogato is an Italian dessert where **hot** espresso is poured over **cold** ice cream.

Integrated Cloud

Applications & Platform Services

ORACLE®