ORACLE

Experience: Model-Based, Feedback-Driven, Greybox Web Fuzzing with BackREST

François Gauthier, Benhaz Hassanshahi, Benjamin Selwyn-Smith, Trong Nhan Mai, Max Schlüter, and Micah Williams

Consulting Researcher, Oracle Labs

Fuzzing or fuzz testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program.

Wikipedia

Fuzzing Exercises Available Functionality

Fuzzers don't care about the ***intended*** functionality of a program.

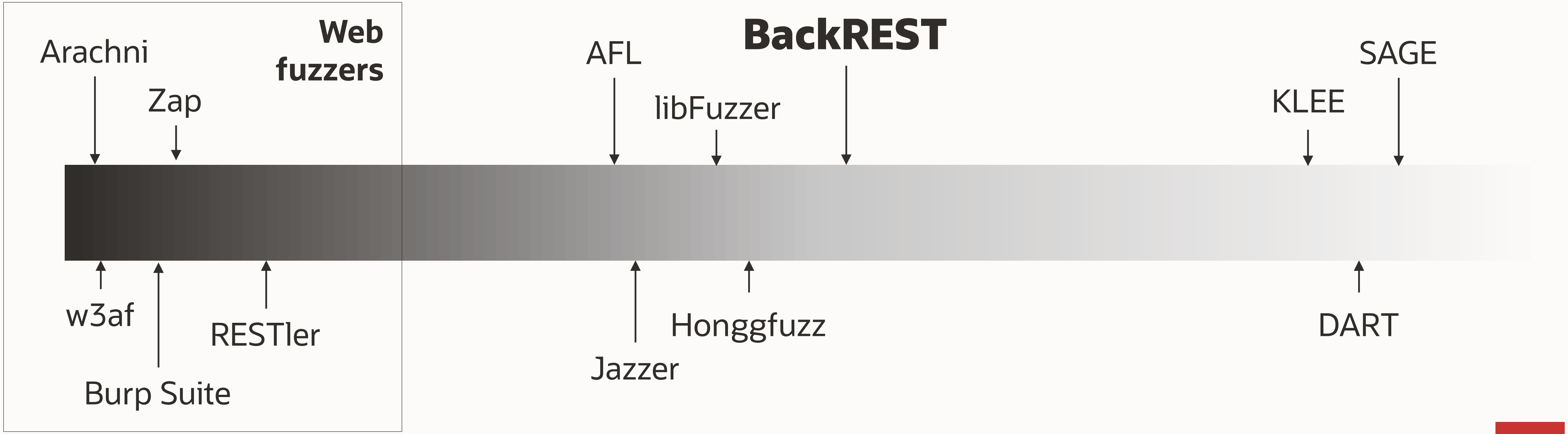
They find issues by exploring the space of ***available*** functionality.

Photo by Alexander Dummer: https://unsplash.com/photos/x4jRmkuDImo

Shades of Fuzzing

Fuzzers come in 3 shades: **black, grey,** and **white**

1. Blackbox fuzzers are completely program-agnostic.
2. Greybox fuzzers use limited program feedback (e.g. coverage, taint) to guide their search.
3. Whitebox fuzzers have complete access to the program's code.

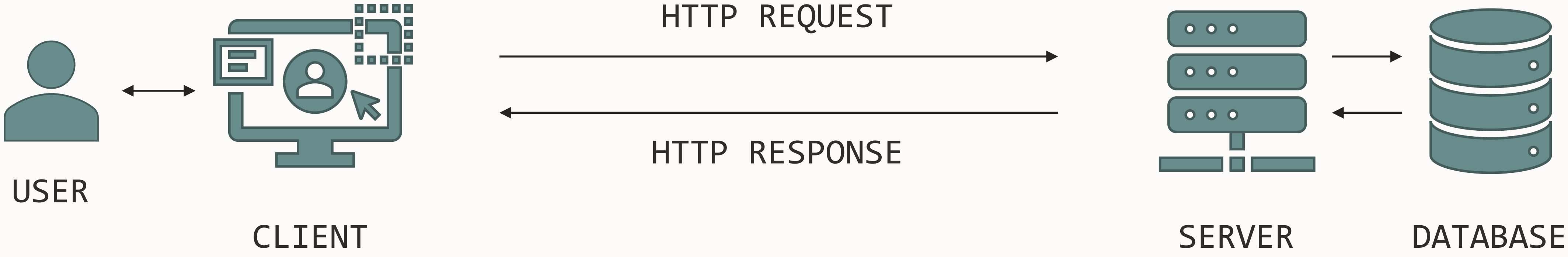


Model-Based Fuzzing

Model-Based Fuzzing of Web Applications

The client is the gateway to the server

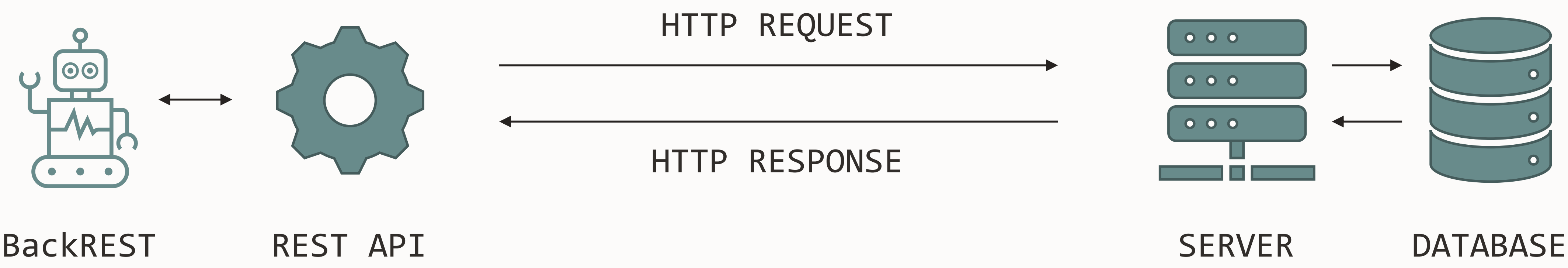
- The server is our fuzzing target because it is where critical operations are happening.
- Need to access the server *indirectly* through the client.
  - Random requests to the server will likely fail early sanity checks (e.g. parameters, types, headers, etc.)
  - Fuzzing the server through the client doesn’t scale.



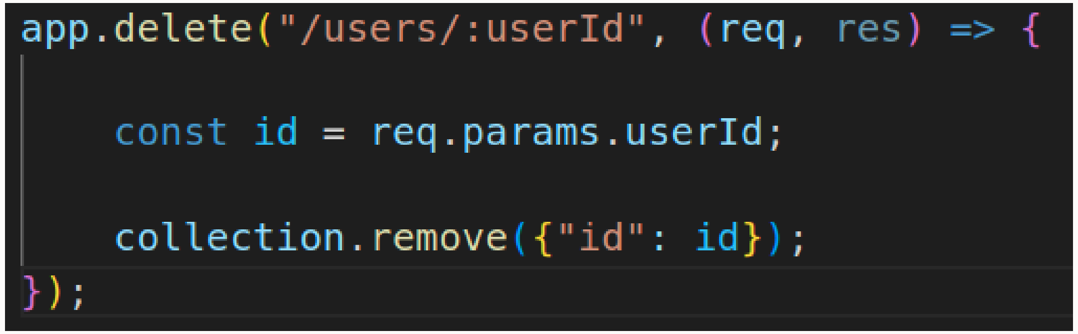
Model-Based Fuzzing of Web Applications

Abstracting away the client to fuzz the server

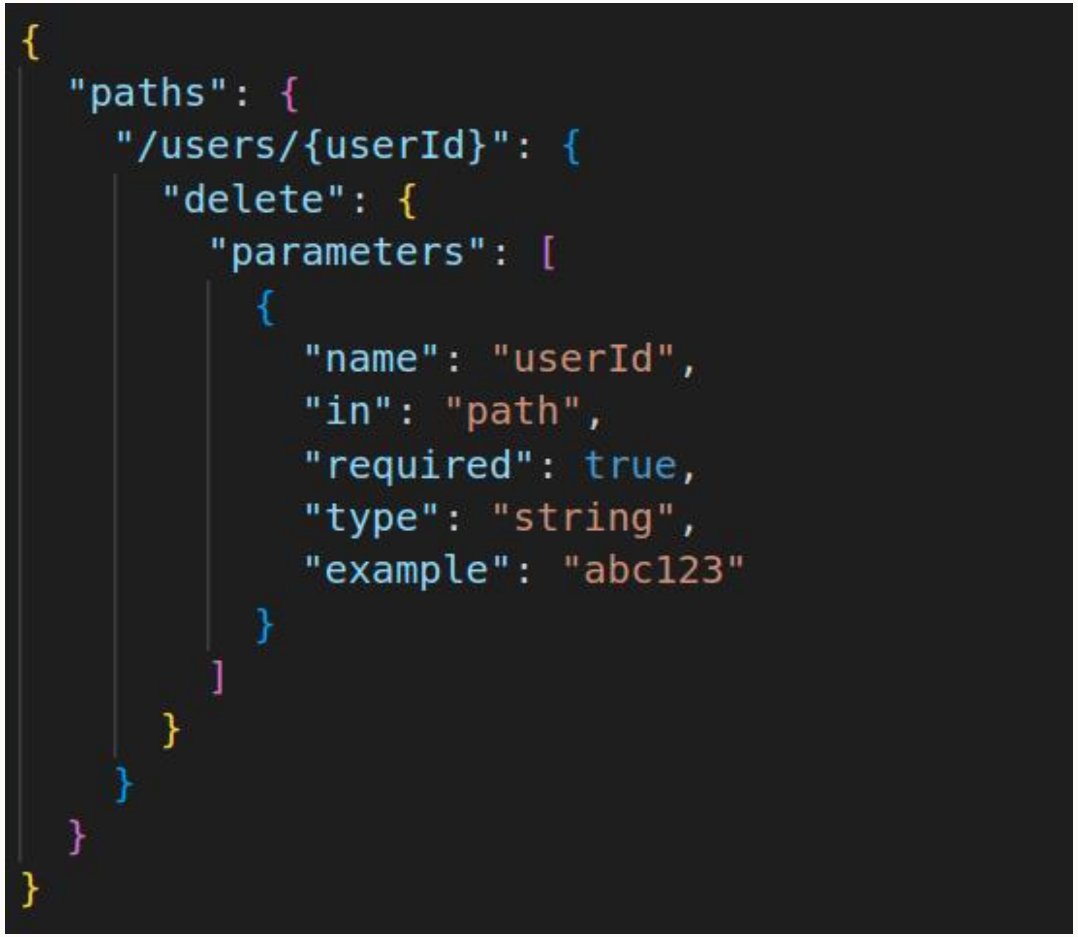
- Client-server interactions in modern applications typically adopt a REST-like format.
  - Interactions are defined and encapsulated unsing standard HTTP verbs, URLs, and request parameters.
- For fuzzing purposes, clients can be abstracted away as REST API models.



Inferring REST API Models

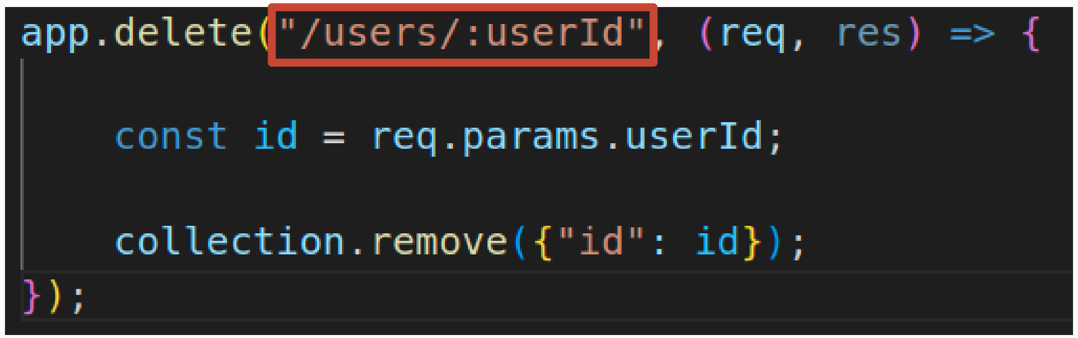


```
app.delete("/users/:userId", (req, res) => {

    const id = req.params.userId;

    collection.remove({"id": id});
});
```

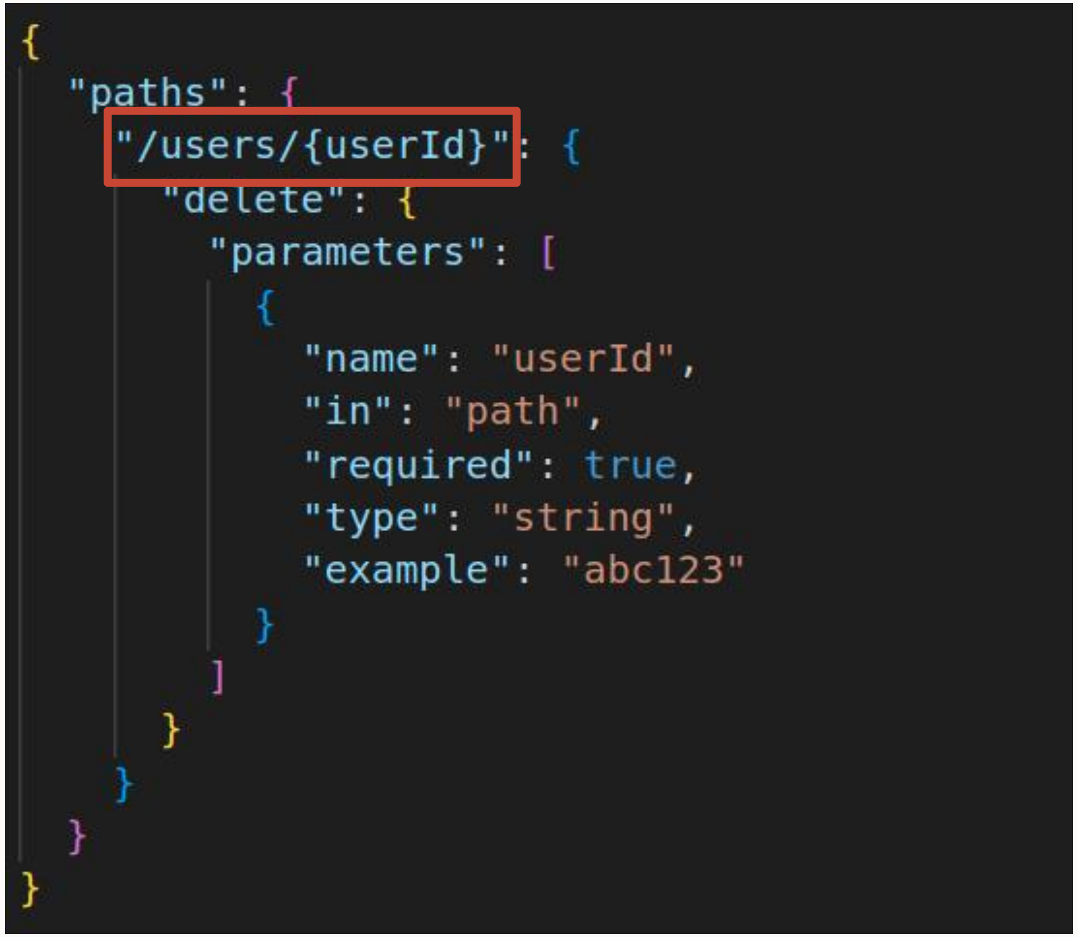


```
{
  "paths": {
    "/users/{userId}": {
      "delete": {
        "parameters": [
          {
            "name": "userId",
            "in": "path",
            "required": true,
            "type": "string",
            "example": "abc123"
          }
        ]
      }
    }
  }
}
```

Inferring REST API Models

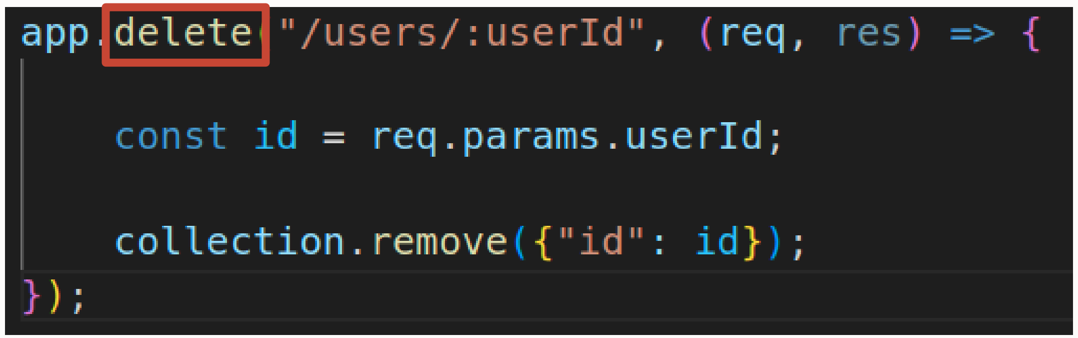


```
app.delete("/users/:userId", (req, res) => {

    const id = req.params.userId;

    collection.remove({"id": id});
});
```

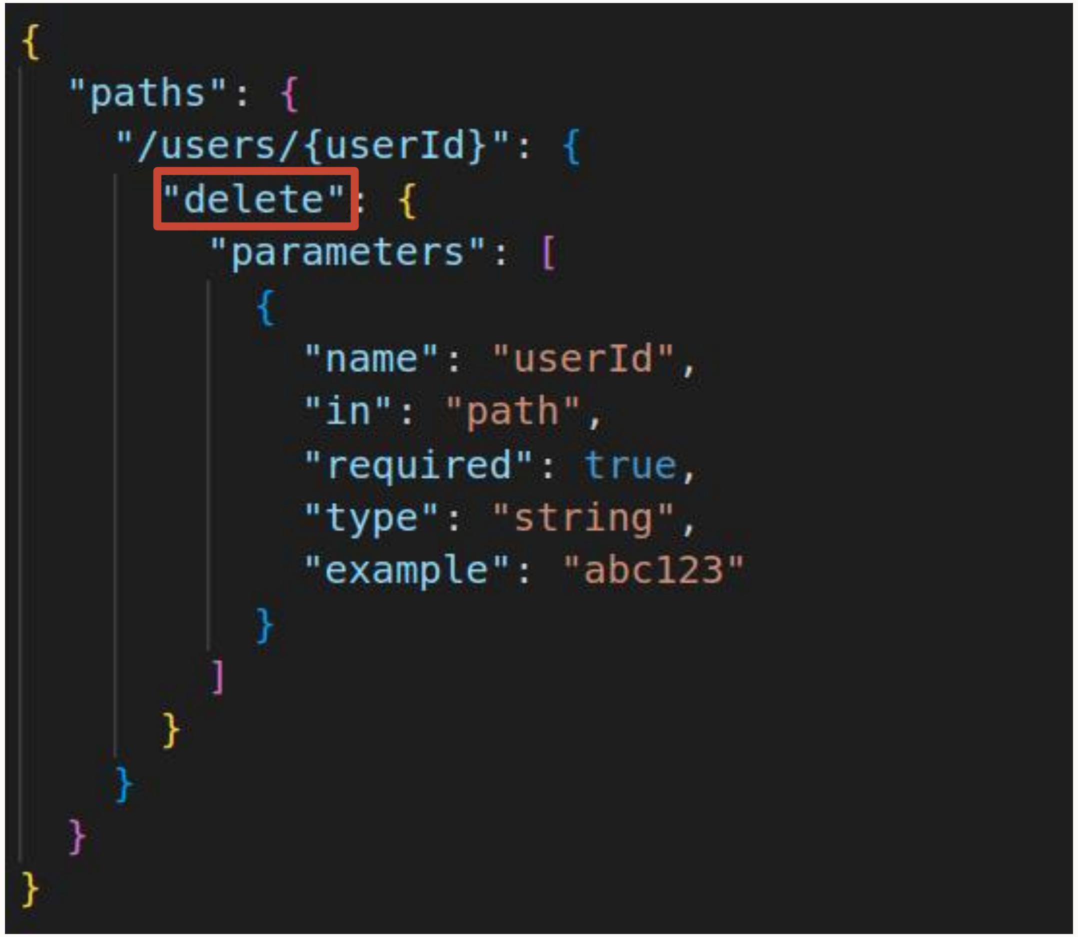


```
{
  "paths": {
    "/users/{userId}": {
      "delete": {
        "parameters": [
          {
            "name": "userId",
            "in": "path",
            "required": true,
            "type": "string",
            "example": "abc123"
          }
        ]
      }
    }
  }
}
```

Inferring REST API Models

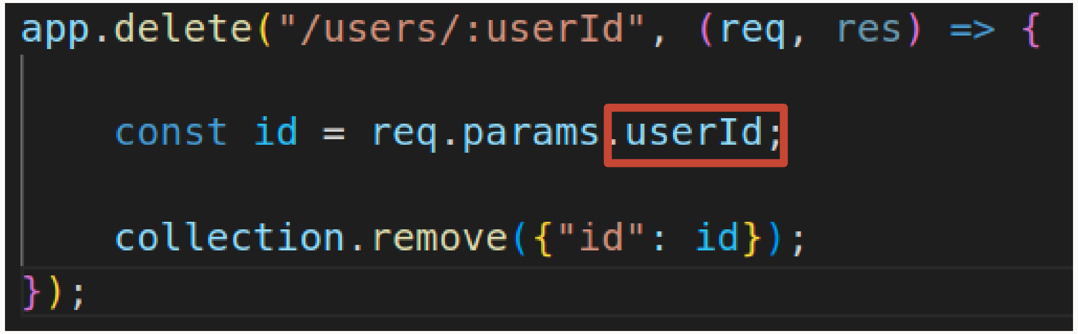


```
app.delete("/users/:userId", (req, res) => {

    const id = req.params.userId;

    collection.remove({"id": id});
});
```

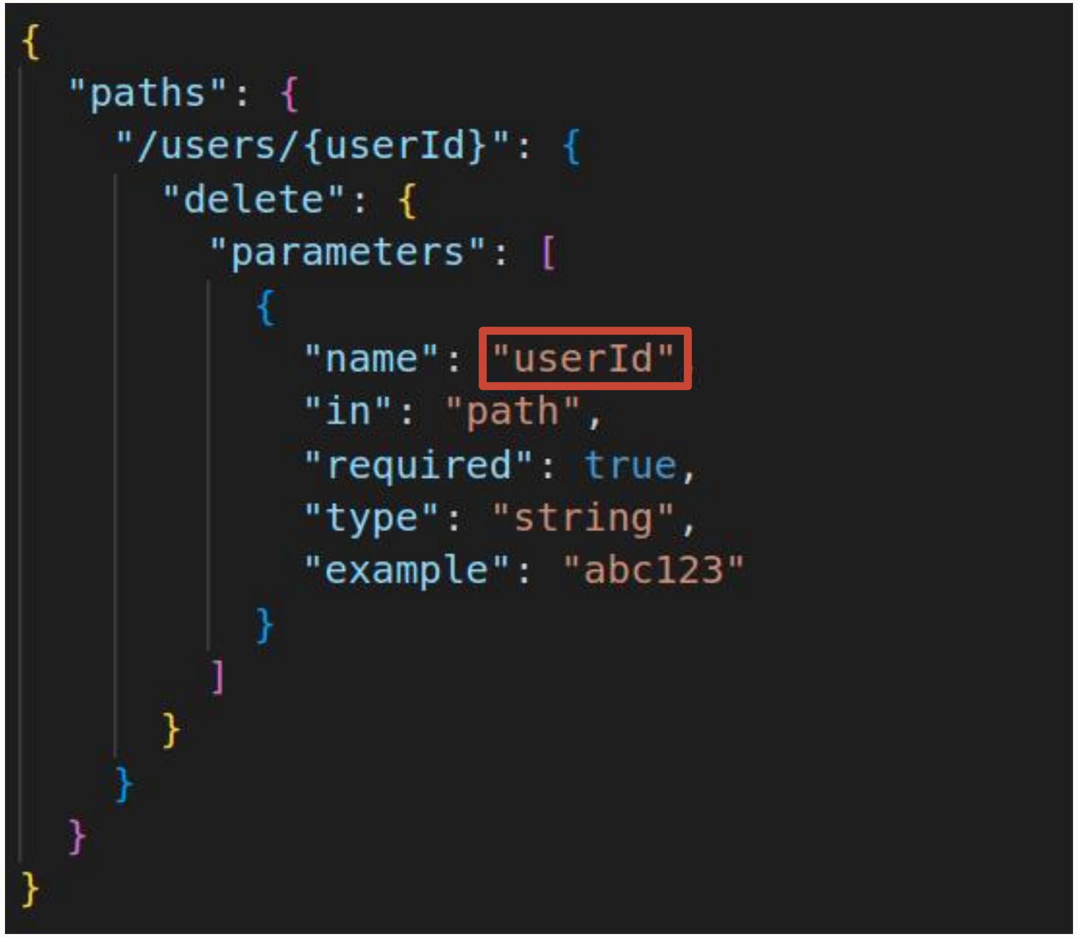


```
{
  "paths": {
    "/users/{userId}": {
      "delete": {
        "parameters": [
          {
            "name": "userId",
            "in": "path",
            "required": true,
            "type": "string",
            "example": "abc123"
          }
        ]
      }
    }
  }
}
```

Inferring REST API Models

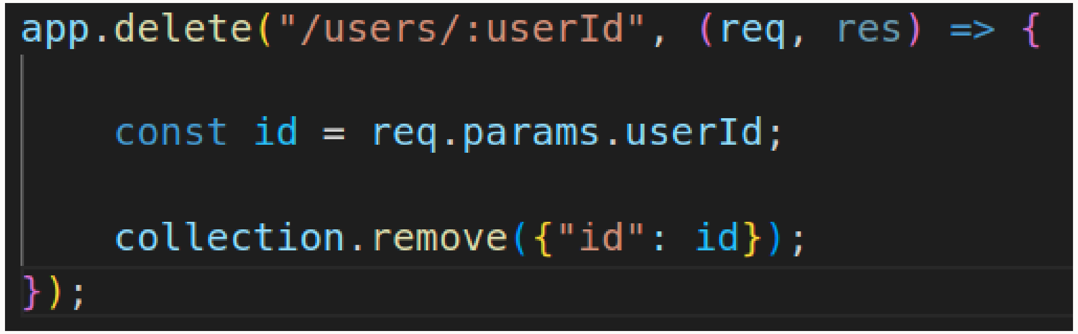


```
app.delete("/users/:userId", (req, res) => {

    const id = req.params.userId;

    collection.remove({"id": id});
});
```



```
{
  "paths": {
    "/users/{userId}": {
      "delete": {
        "parameters": [
          {
            "name": "userId",
            "in": "path",
            "required": true,
            "type": "string",
            "example": "abc123"
          }
        ]
      }
    }
  }
}
```

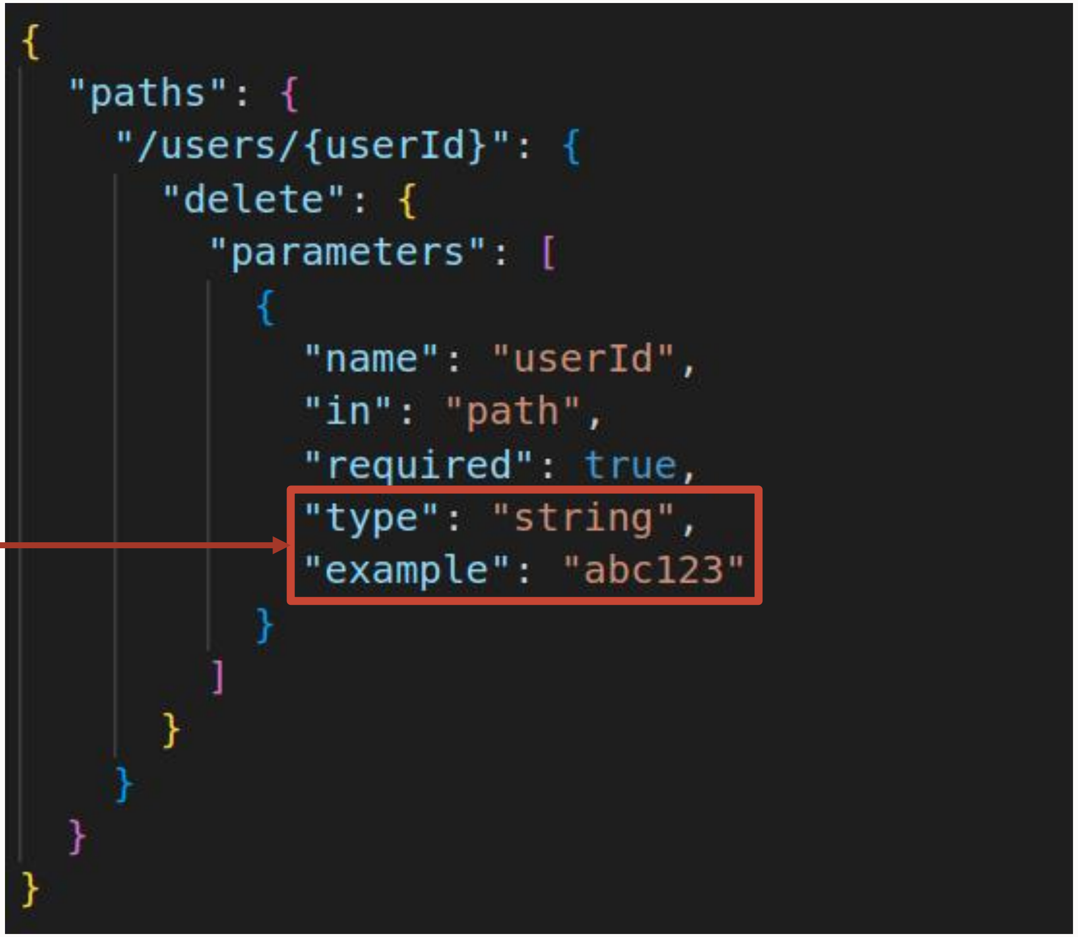
Inferring REST API Models



```
app.delete("/users/:userId", (req, res) => {

    const id = req.params.userId;

    collection.remove({"id": id});
});
```

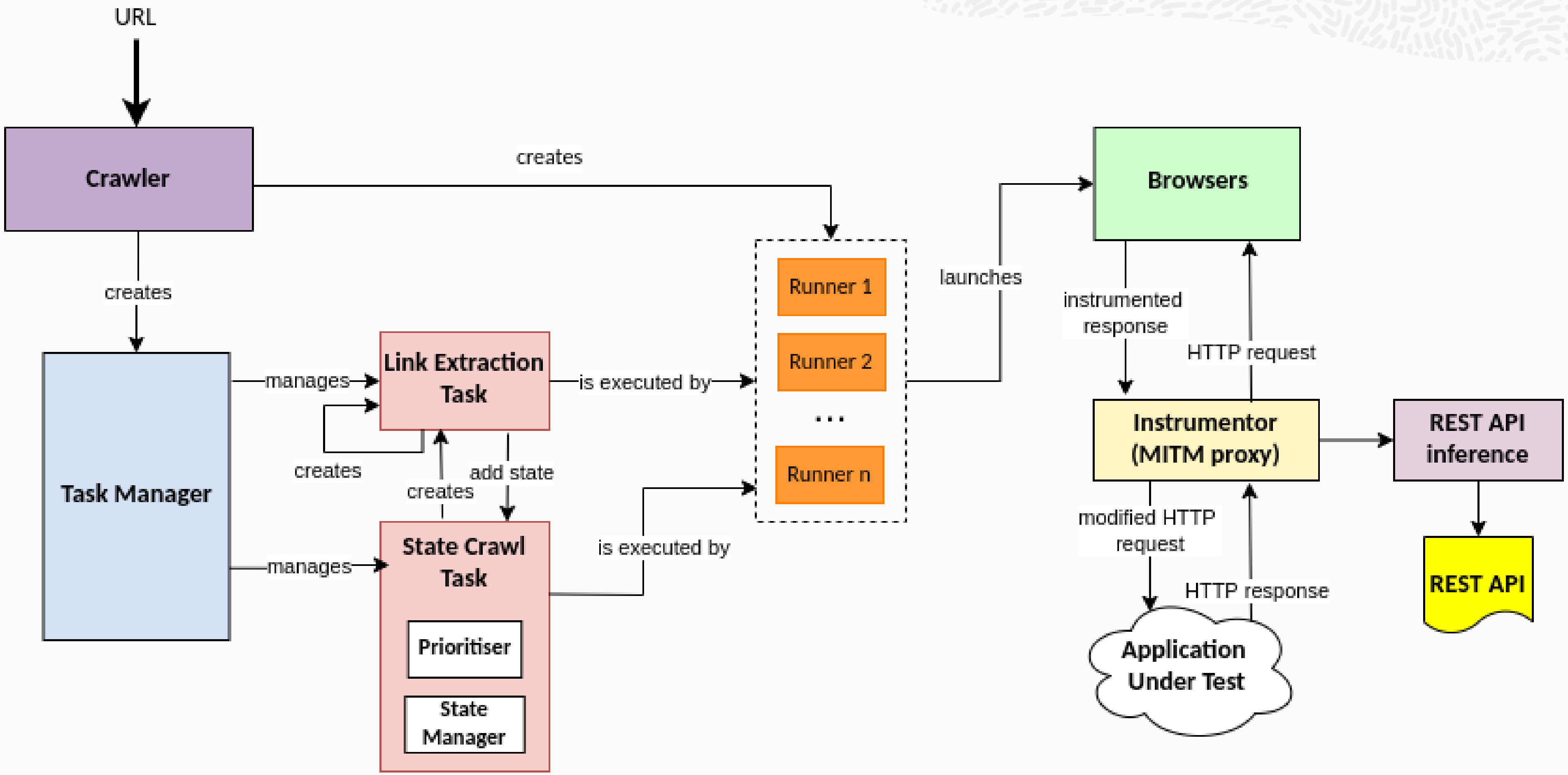
Inferred through:

1. State-aware crawling
2. Static type inference



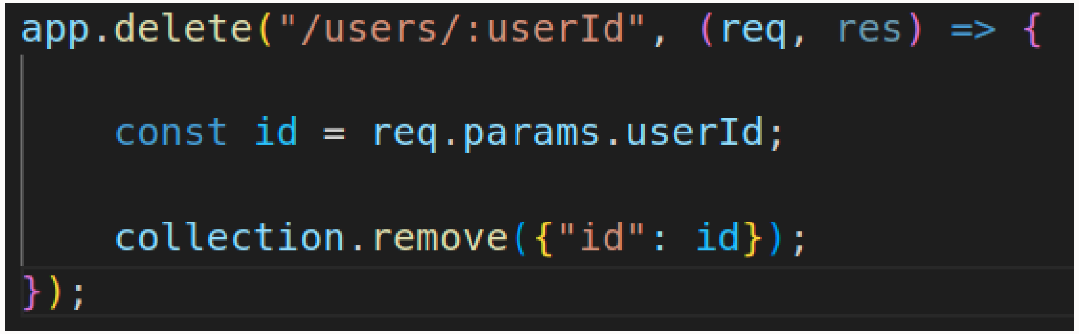
```
{
  "paths": {
    "/users/{userId}": {
      "delete": {
        "parameters": [
          {
            "name": "userId",
            "in": "path",
            "required": true,
            "type": "string",
            "example": "abc123"
          }
        ]
      }
    }
  }
}
```

Inferring REST APIs With Prioritised State-Aware Crawling

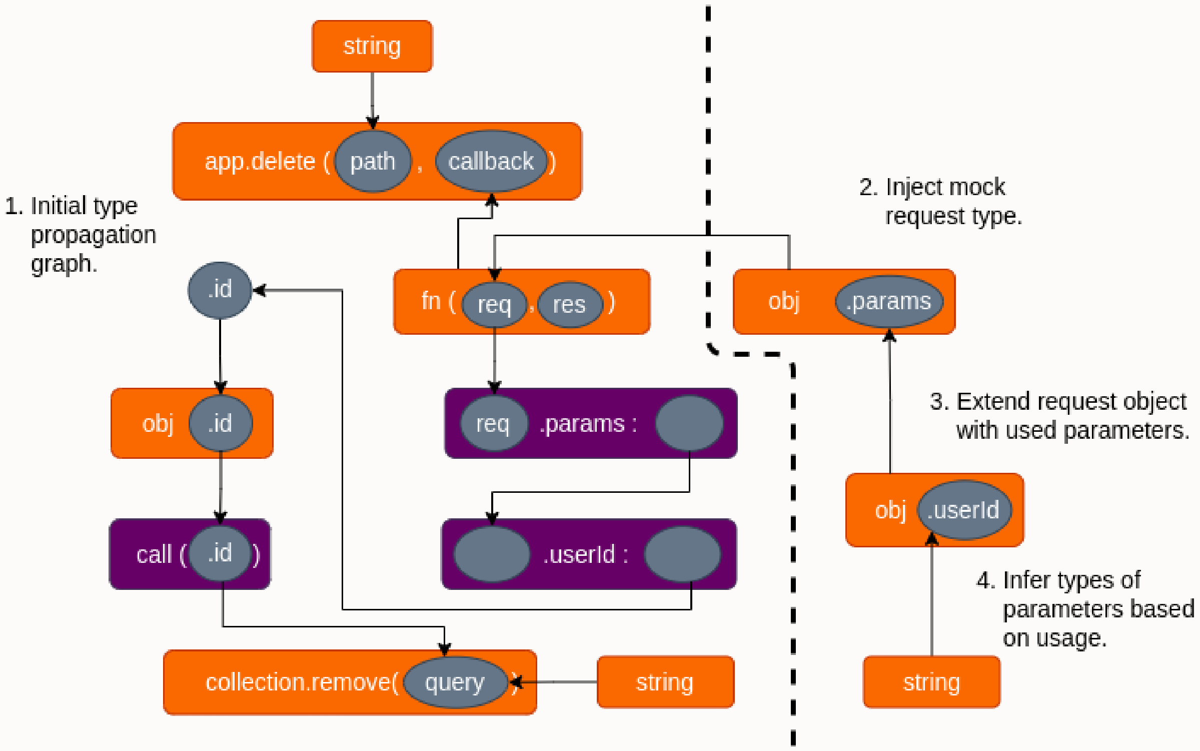


Gelato: Feedback-driven and Guided Security Analysis of Client-side Web Applications, Behnaz Hassanshahi, Hyunjun Lee, Padmanabhan Krishnan, SANER 2022 (to appear)

Augmenting Crawled APIs With Static Type Inference



```
app.delete("/users/:userId", (req, res) => {

    const id = req.params.userId;

    collection.remove({"id": id});
});
```



1. Initial type propagation graph.

2. Inject mock request type.

3. Extend request object with used parameters.

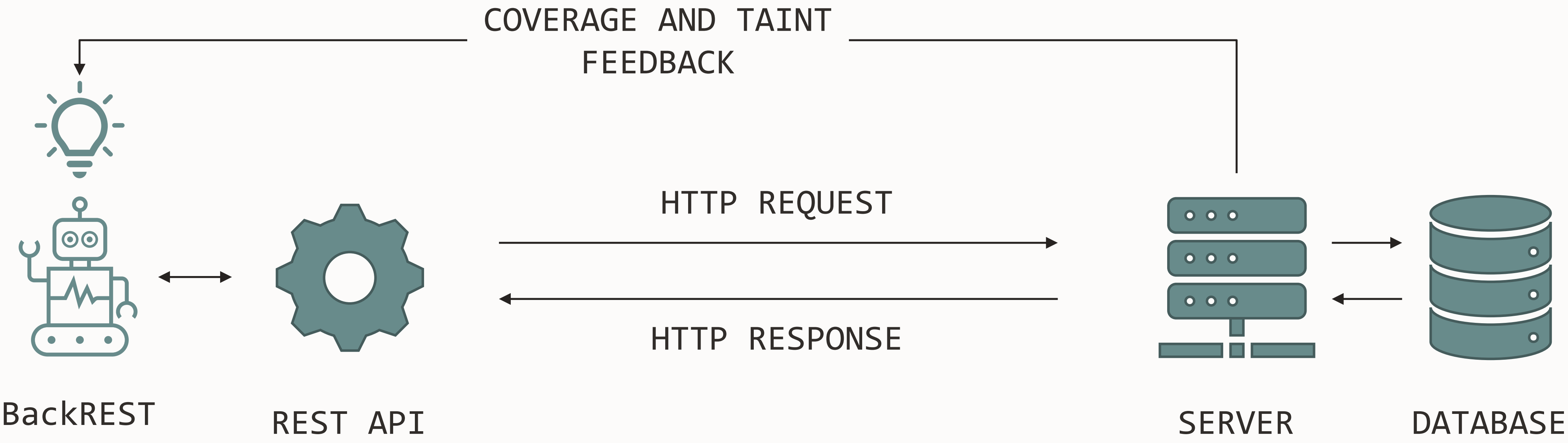
4. Infer types of parameters based on usage.

Feedback-Driven Fuzzing

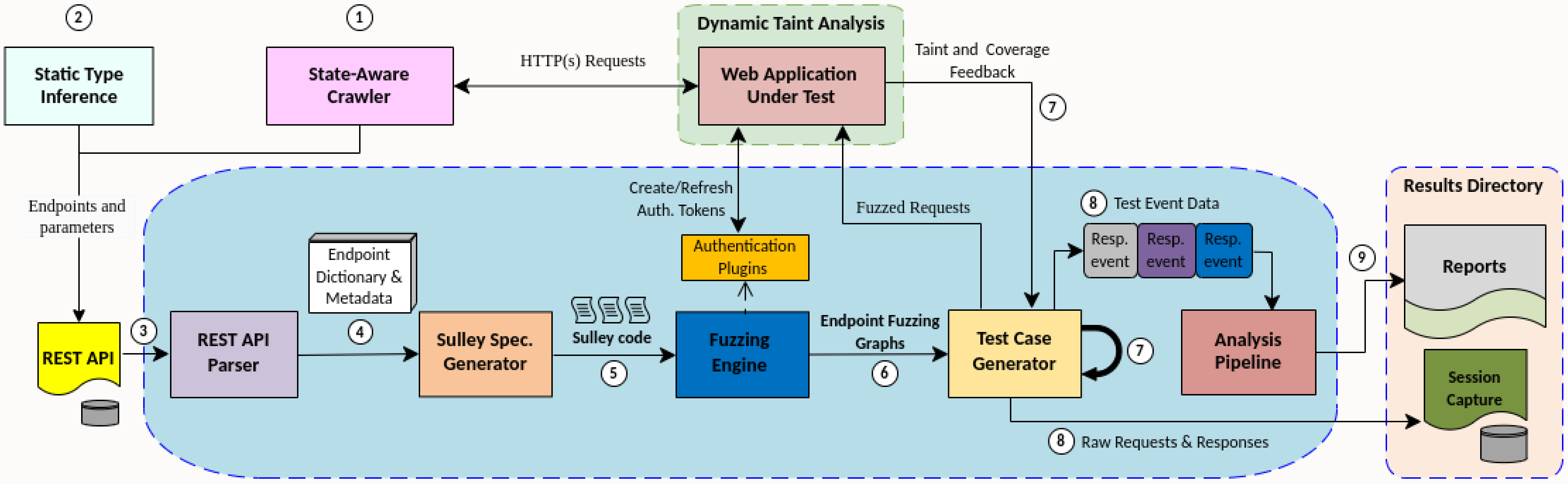
Feedback-Driven Fuzzing of Web Applications

Adding coverage and taint feedback

- A REST API model allows for efficient **blackbox** fuzzing.
- Adding coverage and taint feedback brings BackREST into **greybox** fuzzing territory.



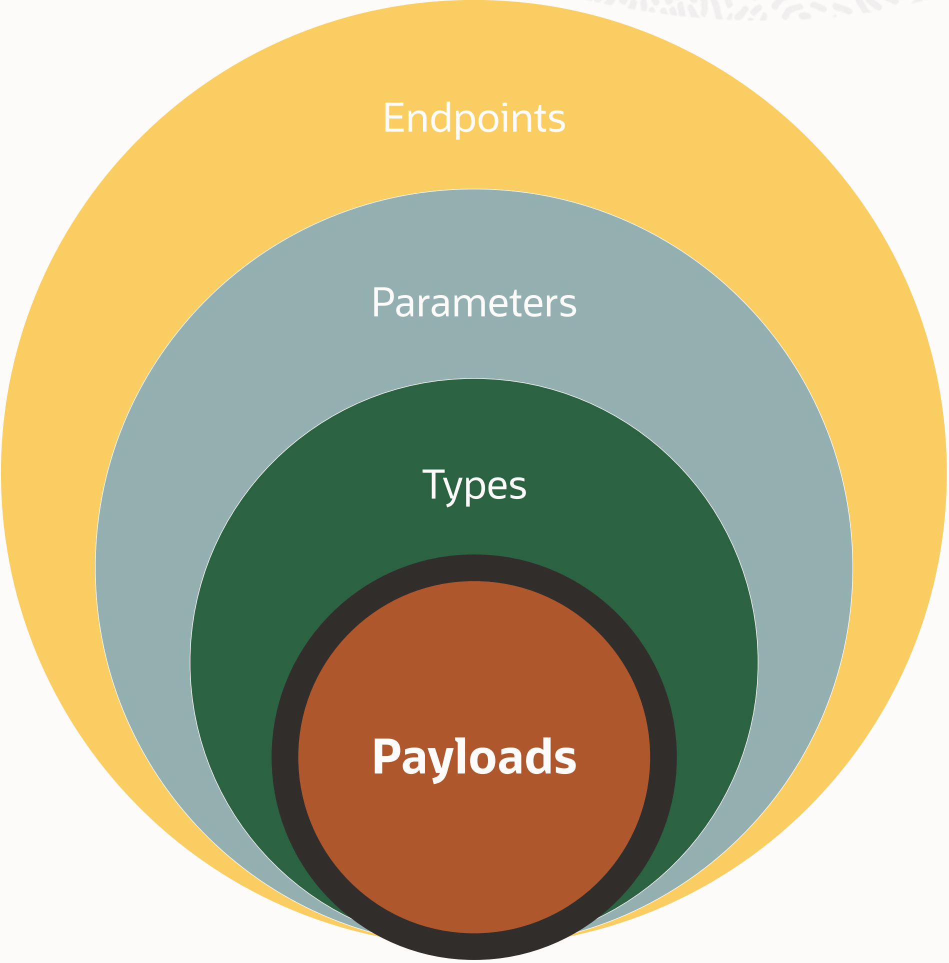
BackREST Architecture



Coverage Feedback To Filter Payloads

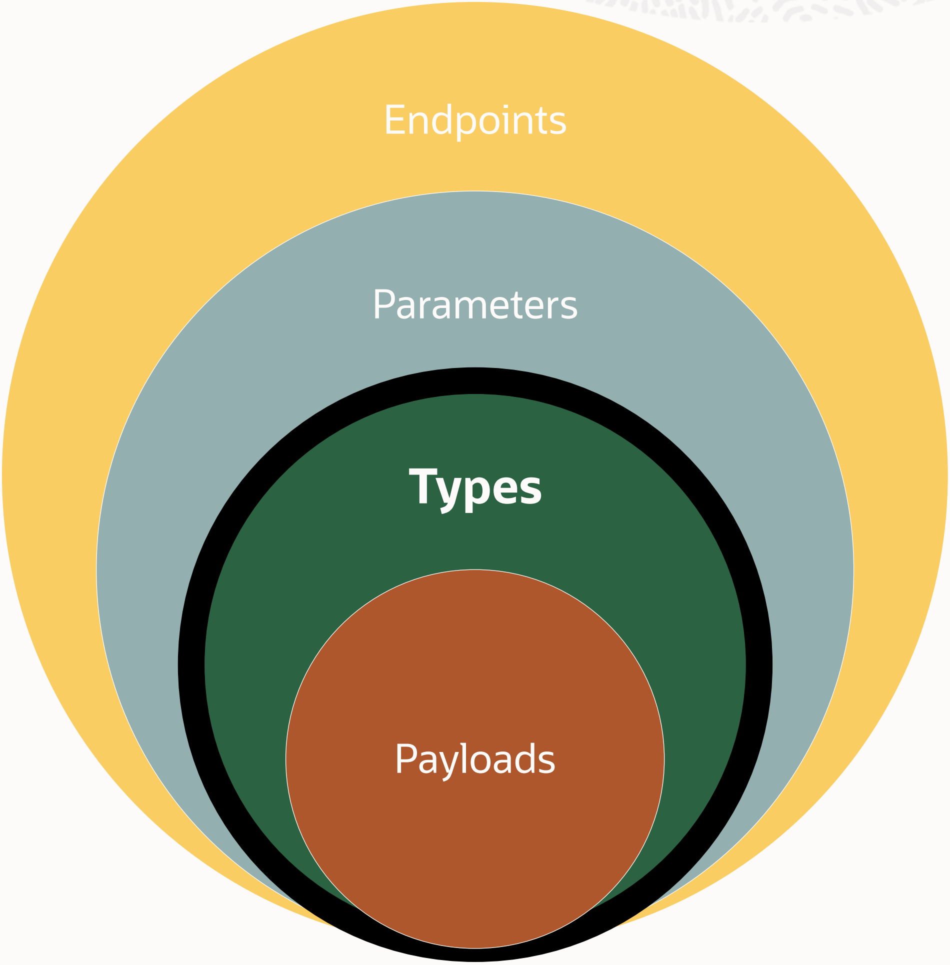
- Coverage feedback helps identify code that has not been thoroughly exercised and more likely to contain bugs.

- BackREST uses coverage to *skip* inputs from its payload dictionary.
  - After *T* payloads of a given type (e.g. SQL, JWT, string, numeric) that did not increase coverage, skip to next type.



Taint Feedback To Filter Types

- Taint feedback helps identify input values reaching key program locations
- BackREST uses taint inference[1] to identify the *type* of payload that will most likely trigger vulnerabilities.
  - E.g. inputs reaching an SQL operation should be sent malformed or malicious SQL fragments.



[1] **Affogato: Runtime Detection of Injection Attacks for Node.js**, François Gauthier, Behnaz Hassanshahi, and Alexander Jordan, SOAP 2018

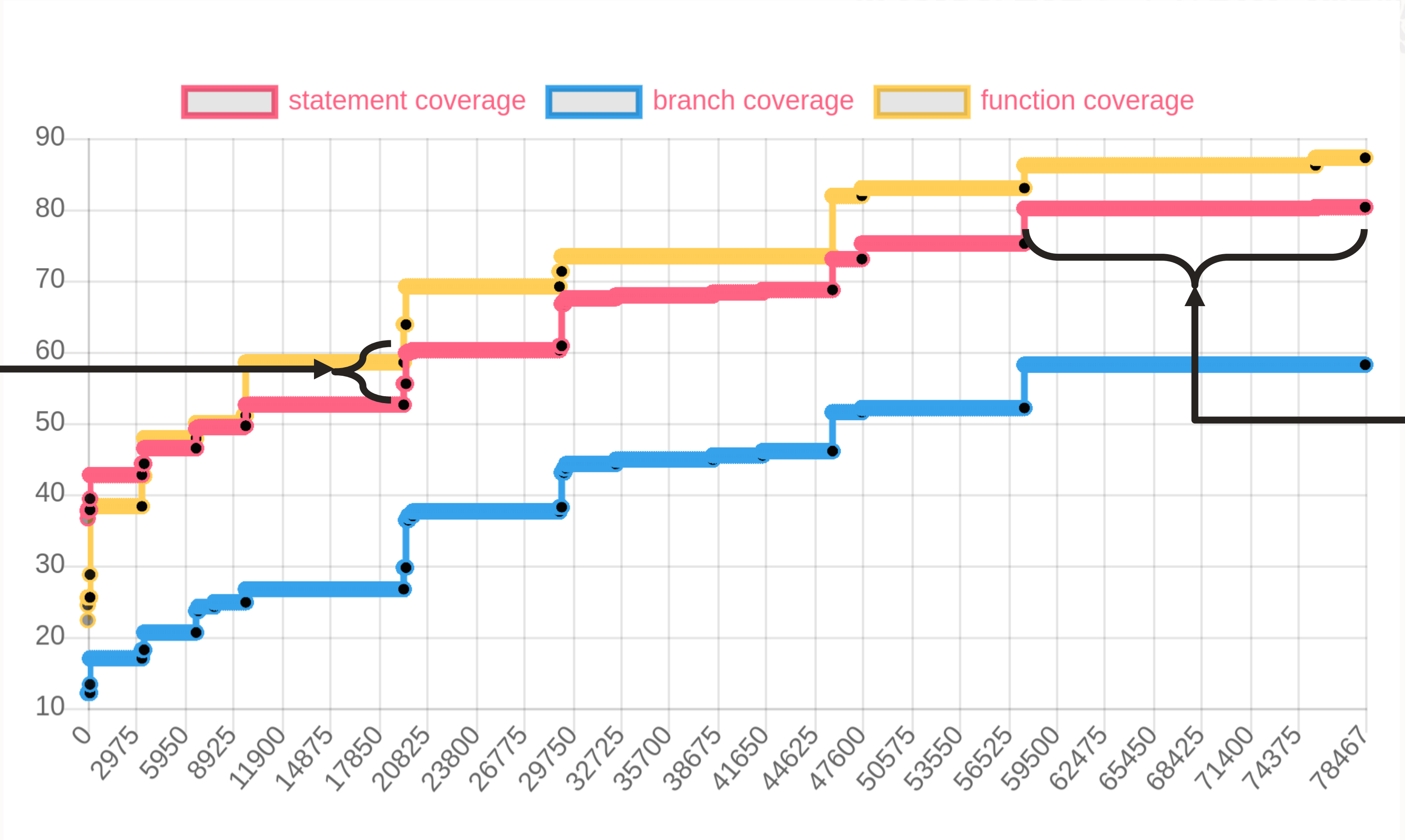
Results

Benchmark Applications and Inferred APIs

Application	Description	Version	SLOC	Files	Entry points*	Request parameters*
Nodegoat	Educational	1.3.0	970 450	12 180	19 (0)	28 (0)
Keystone	CMS	4.0.0	1 393 144	13 891	20 (0)	69 (46)
Apostrophe	CMS	2.0.0	774 203	5 701	184 (0)	633 (531)
Juice-shop	Educational	8.3.0	725 101	7 449	69 (0)	71 (64)
Mongo-express	DB manager	0.51.0	646 403	7 378	29 (0)	96 (49)

* Number of statically inferred values in parenthesis

Coverage Increases When BackREST Switches Endpoints



Fuzzing a new endpoint

Cycling through payloads

Impact of Coverage and Taint Feedback Loops on Total Coverage and Runtime

Application	Coverage (%)			Time (hh:mm:ss)		
	Baseline	Coverage	Cov. & Taint	Baseline	Coverage	Cov. & Taint
Nodegoat	**80.31**	78.54	75.59	0:42:39	0:06:07	**0:05:44**
Keystone	**48.31**	48.05	45.43	5:46:29	0:49:25	**0:13:23**
Apostrophe	–	**48.40**	45.52	–	11:11:42	**6:17:34**
Juice-shop	74.73	**76.34**	75.85	12:48:15	1:10:31	**1:08:26**
Mongo-express	**69.62**	69.57	66.59	2:21:49	0:16:07	**0:11:07**

Impact of Coverage and Taint Feedback on Vulnerability Reports

Baseline (B), Coverage feedback (C), Coverage & Taint feedback (CT)

Application	(No)SQLi			Cmd injection			XSS			DoS		
	B	C	CT	B	C	CT	B	C	CT	B	C	CT
Nodegoat	0	0	**3**	0	0	**3**	**5**	5	5	**0**	0	0
Keystone	0	0	**0**	0	0	**0**	**1**	1	0	**0**	0	0
Apostrophe	0	0	**0**	0	0	**0**	**1**	1	1	**2**	1	1
Juice-shop	1	1	**2**	0	0	**1**	**4**	1	1	**1**	0	0
Mongo-express	0	0	**5**	0	0	**2**	**0**	0	0	**3**	3	3
Total	1	1	**10**	0	0	**6**	**11**	8	7	**6**	4	4

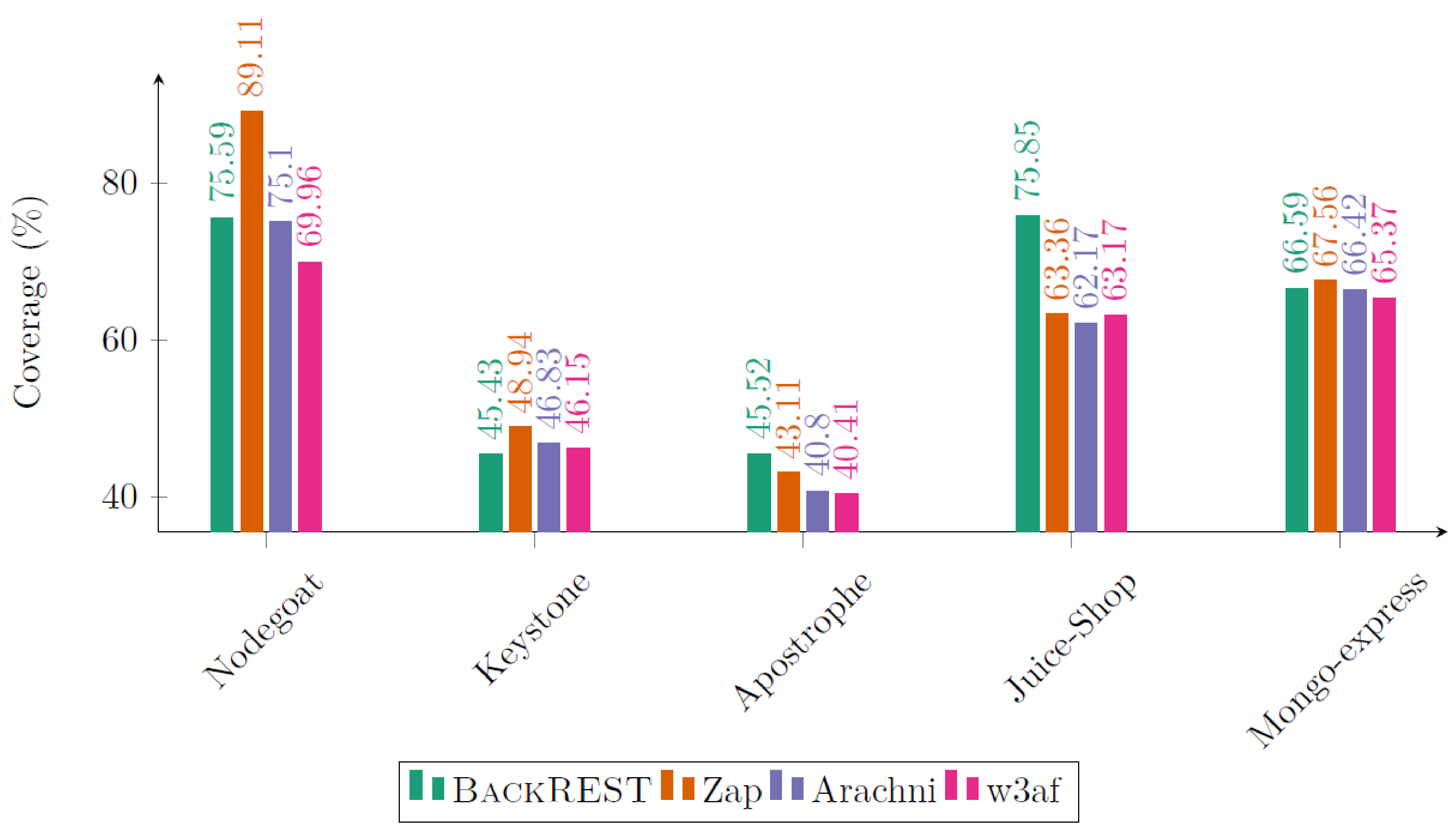
A Note On Server-Side State Modelling

- RESTler[1] is the first study to investigate stateful server-side fuzzing of web services.
  - Authors have found a positive correlation between stateful fuzzing and increases in coverage.
  - We have not observed a similar effect on our benchmark applications.

- We also attempted to model server-side state by inferring dependencies between endpoints.
  - This did not improve coverage for all but the Mongo-express application.
  - In this case, the dependencies were intuitive (e.g. insert a document before deleting it) and easily configured manually.

[1] **RESTler: Stateful REST API Fuzzing**, Atlidakis, Vaggelis, Patrice Godefroid, and Marina Polishchuk. ICSE 2019.

BackREST Coverage vs. State-Of-The-Art



BackREST 0-days vs. State-Of-The-Art

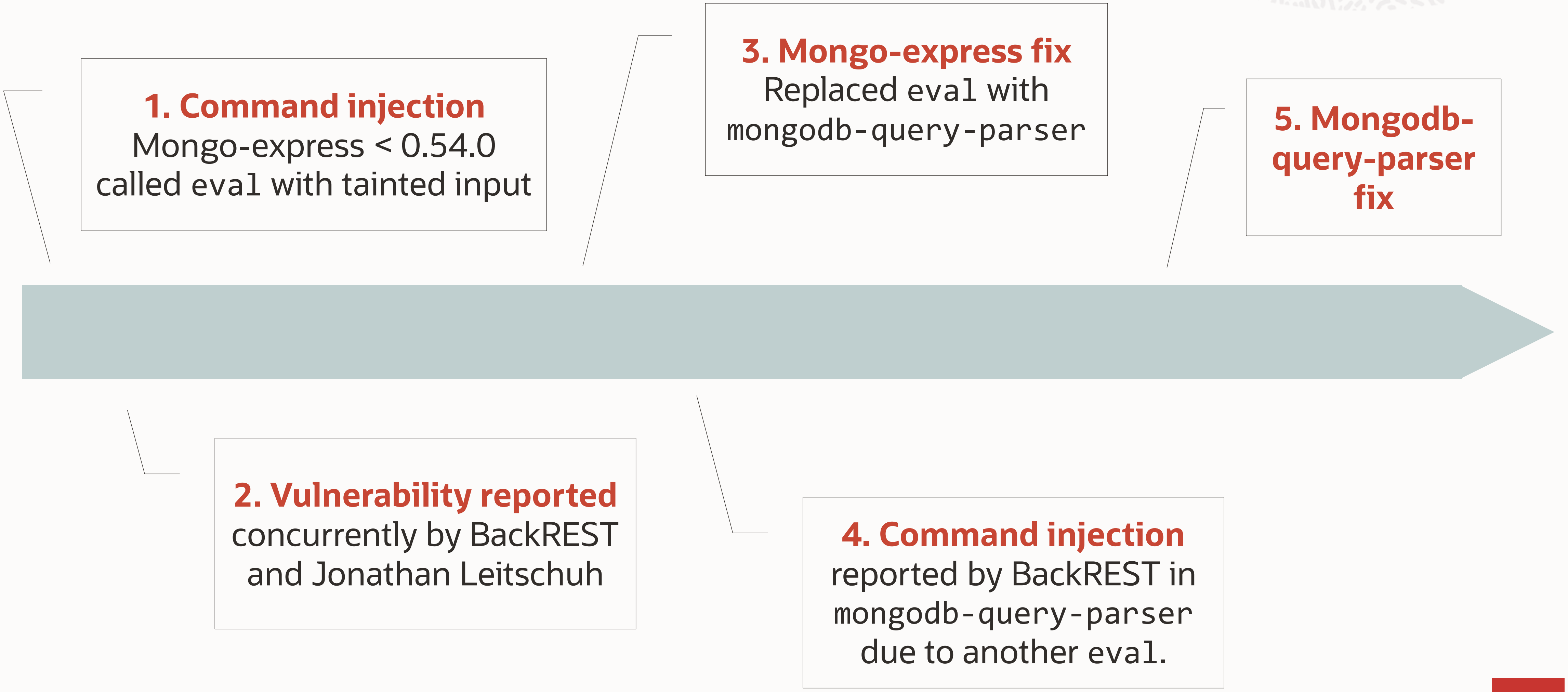
Codebase	Vulnerability	Found by	Taint only	Severity
MarsDB	Command injection	BackREST	✓	Critical
Sequelize	Denial-of-Service	BackREST		Moderate
Apostrophe	Denial-of-Service	BackREST		–
Apostrophe	Denial-of-Service	BackREST		Low
Mongo-express	Command injection	BackREST	✓	Critical
Mongo-express	Denial-of-Service	BackREST, Zap, Arachni, w3af		Medium
Mongodb-query-parser	Command injection	BackREST	✓	Critical
MongoDB	Denial-of-Service	BackREST, Zap		High

MongoDB: **1 671 653** weekly downloads

Sequelize: **648 745** weekly downloads

Case Study: Mongo-express and Mongodb-query-parser

Taint feedback detected two command injections



1. Command injection
Mongo-express < 0.54.0
called `eval` with tainted input

2. Vulnerability reported
concurrently by BackREST
and Jonathan Leitschuh

3. Mongo-express fix
Replaced `eval` with
`mongodb-query-parser`

4. Command injection
reported by BackREST in
`mongodb-query-parser`
due to another `eval`.

5. Mongodb-query-parser fix

Conclusions

Why Does BackREST Detect More 0-Days?

- Total coverage does not significantly differ between BackREST and state-of-the-art ✗
- State-modelling hasn’t made a significant impact on our benchmarks ✗
- Taint feedback is unique to BackREST and allows to
  - Select the payloads that will most likely trigger vulnerabilities ✓
  - Detect vulnerabilities that are *invisible* to blackbox fuzzers ✓
- Payload dictionaries encapsulate expert knowledge about web vulnerabilities ✓
- Uncovering more complete API models through state-aware crawling helps trigger more vulnerabilities ✓

Conclusion

- BackREST is a model-based, coverage- and taint-driven greybox fuzzer that:
  - Guides a state-aware crawler to infer REST-like APIs
  - Uses coverage feedback to improve performance.
  - Uses taint feedback to detect vulnerabilities and guide payload selection.

- BackREST:
  - Achieved speedups ranging from 7.4× to 25.9×.
  - Consistently detected more (No)SQLi, command injection, and XSS vulnerabilities than three state-of-the-art web fuzzers.
  - Detected six 0-days that were missed by all other fuzzers.

Our feedback loops are simple enough to be applied to existing black-box web application fuzzers.

Thank you

François Gauthier

francois.gauthier@oracle.com