sonarsource

Beer Fondue!

Or how you can find vulnerabilities thanks to SonarQube

Malte Skoruppa & Nicolas Peru | 8 December 2020

Ego boost

Malte Skoruppa, PhD

SonarSourcer since april, previously RIPSler since 2017

Worked on the SAST engine at RIPS and SonarSource

Before that, PhD thesis on automated vulnerability detection

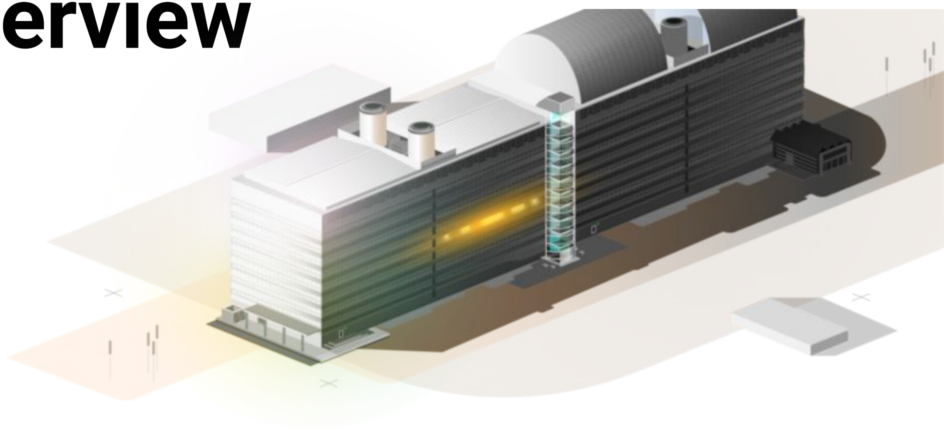
Nicolas Peru (not a doctor)

SonarSourcer since 2013

Worked on Java Analyzer and Security Analyzer

The elevator pitch

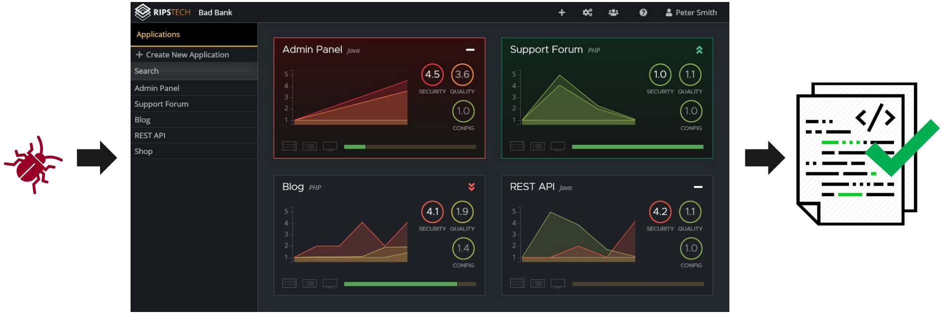
RIPS Technologies: Overview



Founded:	August 2016
Location:	Bochum, Germany
Market:	SAST, B2B
Product:	RIPS Code Analysis

Start of 2020 (prior to acquisition):

Employees:	25 full-time + students
Customers:	130 throughout the world



RIPS detects and patches critical security bugs in source code.

RIPS Technologies: 12 years of research

2008
SAST prototype for CTF competitions

2010
RIPS starts out as open-source tool

2014
Research awarded Internet Defense Price

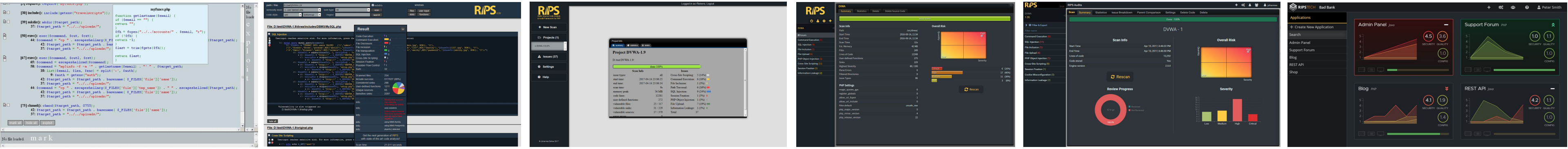
2016
RIPS Technologies founded

2018
REST API & integrations

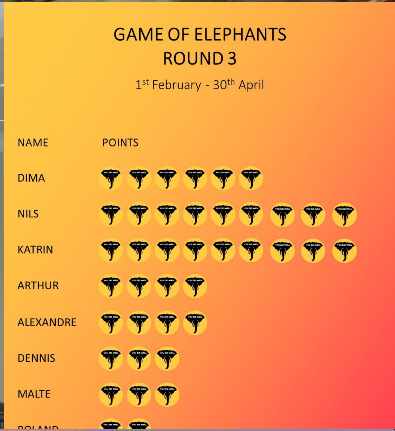
2019
RIPS releases Java support

2020
RIPS supports Node.js, C#/.NET
In development

2020
Acquisition by SonarSource



RIPS Technologies: Culture

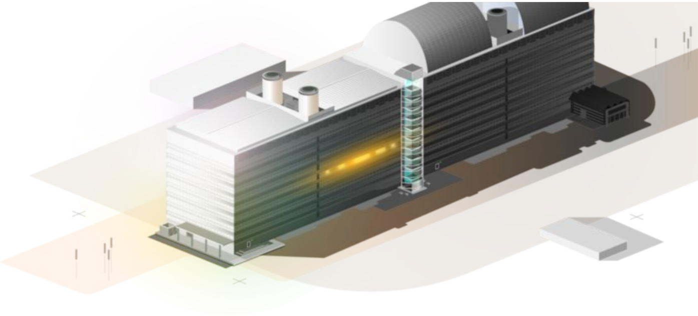


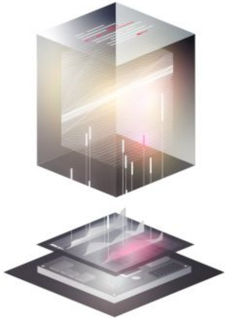
GAME OF ELEPHANTS
ROUND 3

1st February - 30th April

NAME	POINTS
DIMA	🐘🐘🐘🐘🐘🐘
NILS	🐘🐘🐘🐘🐘🐘🐘🐘🐘
KATRIN	🐘🐘🐘🐘🐘🐘🐘🐘🐘
ARTHUR	🐘🐘🐘🐘
ALEXANDRE	🐘🐘🐘🐘
DENNIS	🐘🐘🐘
MALTE	🐘🐘🐘
ROLAND	🐘🐘

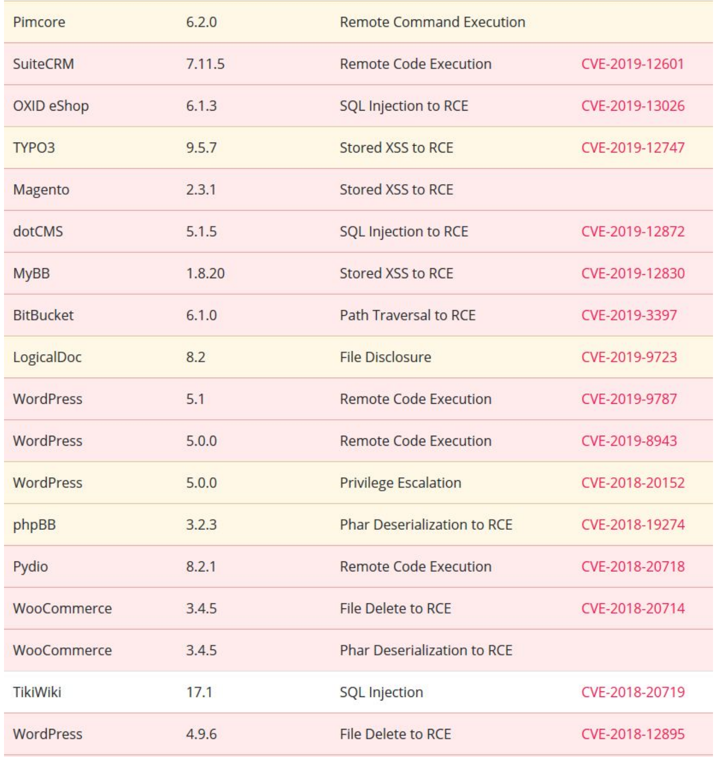
RIPS Technologies: A unique know-how



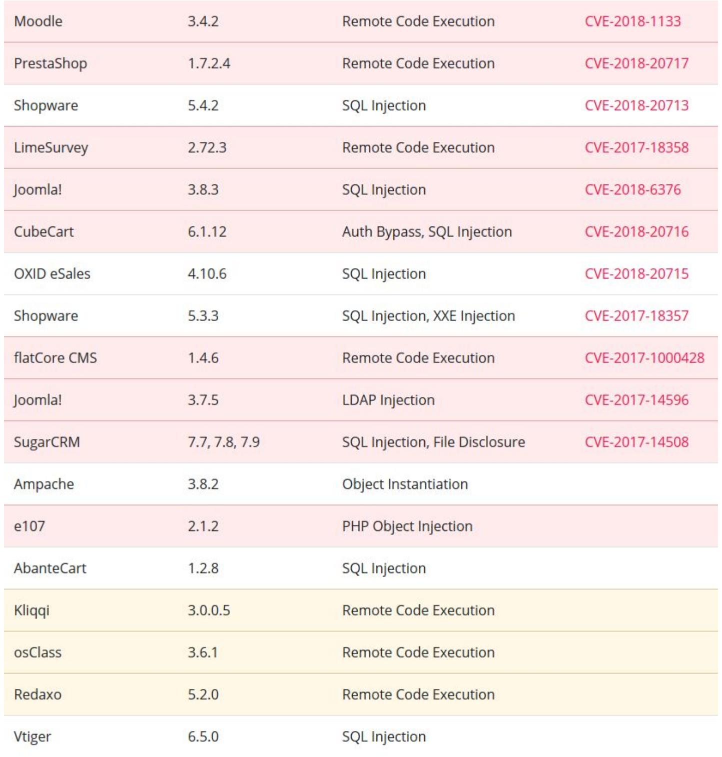


Particularly in the PHP world!

RIPS detects complex security bugs



Pimcore	6.2.0	Remote Command Execution	
SuiteCRM	7.11.5	Remote Code Execution	CVE-2019-12601
OXID eShop	6.1.3	SQL Injection to RCE	CVE-2019-13026
TYPO3	9.5.7	Stored XSS to RCE	CVE-2019-12747
Magento	2.3.1	Stored XSS to RCE	
dotCMS	5.1.5	SQL Injection to RCE	CVE-2019-12872
MyBB	1.8.20	Stored XSS to RCE	CVE-2019-12830
BitBucket	6.1.0	Path Traversal to RCE	CVE-2019-3397
LogicalDoc	8.2	File Disclosure	CVE-2019-9723
WordPress	5.1	Remote Code Execution	CVE-2019-9787
WordPress	5.0.0	Remote Code Execution	CVE-2019-8943
WordPress	5.0.0	Privilege Escalation	CVE-2018-20152
phpBB	3.2.3	Phar Deserialization to RCE	CVE-2018-19274
Pydio	8.2.1	Remote Code Execution	CVE-2018-20718
WooCommerce	3.4.5	File Delete to RCE	CVE-2018-20714
WooCommerce	3.4.5	Phar Deserialization to RCE	
TikiWiki	17.1	SQL Injection	CVE-2018-20719
WordPress	4.9.6	File Delete to RCE	CVE-2018-12895



Moodle	3.4.2	Remote Code Execution	CVE-2018-1133
PrestaShop	1.7.2.4	Remote Code Execution	CVE-2018-20717
Shopware	5.4.2	SQL Injection	CVE-2018-20713
LimeSurvey	2.72.3	Remote Code Execution	CVE-2017-18358
Joomla!	3.8.3	SQL Injection	CVE-2018-6376
CubeCart	6.1.12	Auth Bypass, SQL Injection	CVE-2018-20716
OXID eSales	4.10.6	SQL Injection	CVE-2018-20715
Shopware	5.3.3	SQL Injection, XXE Injection	CVE-2017-18357
flatCore CMS	1.4.6	Remote Code Execution	CVE-2017-1000428
Joomla!	3.7.5	LDAP Injection	CVE-2017-14596
SugarCRM	7.7, 7.8, 7.9	SQL Injection, File Disclosure	CVE-2017-14508
Ampache	3.8.2	Object Instantiation	
e107	2.1.2	PHP Object Injection	
AbanteCart	1.2.8	SQL Injection	
Kliqqi	3.0.0.5	Remote Code Execution	
osClass	3.6.1	Remote Code Execution	
Redaxo	5.2.0	Remote Code Execution	
Vtiger	6.5.0	SQL Injection	

SonarSource: Since 2008

- Offices in Geneva, Austin, Bochum, Annecy
- ~200 persons
- Strong culture

SonarSource: Since 2008

- For Developers and Development Teams
- Simple and Transparent
- Part of your development process
- Accurate, and helpful. Always.

SonarSource: 3 Products

SonarSource: Since 2018

- Same principles applied to Security

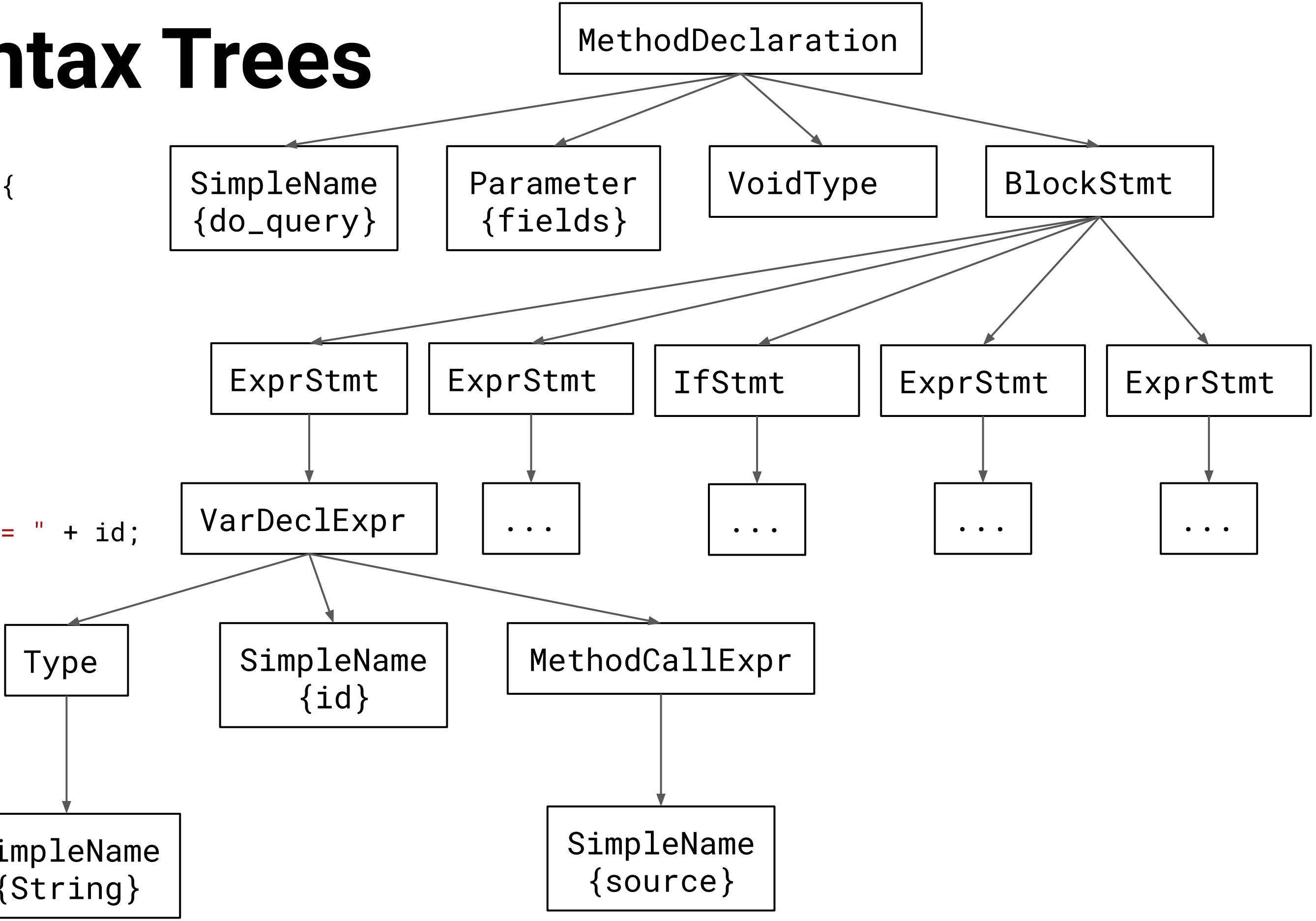
2020

SonarSource ❤️ RIPSTech

Static analysis in 5 minutes !

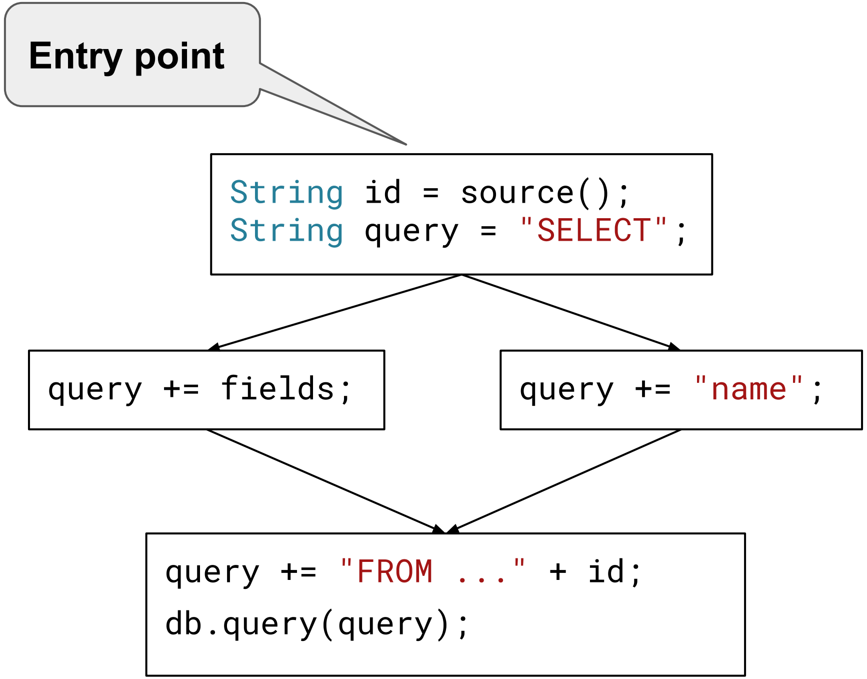
Abstract Syntax Trees

```
void do_query(String fields) {
  String id = source();
  String query = "SELECT ";
  if (!fields.empty()) {
    query += fields;
  } else {
    query += "name";
  }
  query += " FROM u WHERE id = " + id;
  db.query(query);
}
```



Control Flow Graphs

```
void do_query(String fields) {
  String id = source();
  String query = "SELECT ";
  if (!fields.empty()) {
    query += fields;
  } else {
    query += "name";
  }
  query += " FROM u WHERE id = " + id;
  db.query(query);
}
```



Entry point

```
String id = source();
String query = "SELECT";
```

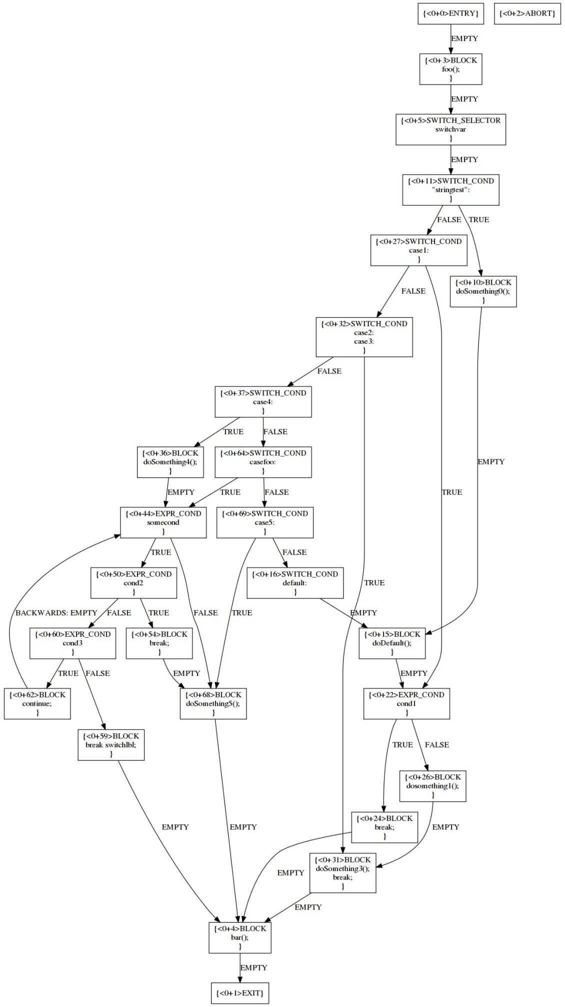
```
query += fields;
```

```
query += "name";
```

```
query += "FROM ..." + id;
db.query(query);
```

...and it quickly gets complex!

```
foo();
switchlbl:
switch (switchvar) {
  case “stringtest”:
    doSomething0();
  default:
  case case1:
    if (cond1) break;
    else doSomething1();
  case case2:
  case case3:
    doSomething3();
    break;
  case case4:
    doSomething4();
  case casefoo:
    while (somecond) {
      if (cond2) {
        break;
      } else {
        if (cond3) continue;
        break switchlbl;
      }
    }
  case case5:
    doSomething5();
}
bar();
```


{<0+0>ENTRY}
{<0+2>ABORT}
EMPTY
{<0+3>BLOCK foo(); }
EMPTY
{<0+5>SWITCH_SELECTOR switchvar }
EMPTY
{<0+11>SWITCH_COND "stringtest": }
FALSE
TRUE
{<0+27>SWITCH_COND case1: }
{<0+10>BLOCK doSomething0(); }
FALSE
{<0+32>SWITCH_COND case2: case3: }
FALSE
{<0+37>SWITCH_COND case4: }
TRUE
FALSE
{<0+36>BLOCK doSomething4(); }
{<0+64>SWITCH_COND casefoo: }
EMPTY
EMPTY
TRUE
FALSE
TRUE
{<0+44>EXPR_COND somecond }
{<0+69>SWITCH_COND case5: }
TRUE
FALSE
{<0+50>EXPR_COND cond2 }
{<0+16>SWITCH_COND default: }
TRUE
BACKWARDS: EMPTY
FALSE
TRUE
FALSE
TRUE
EMPTY
{<0+60>EXPR_COND cond3 }
{<0+54>BLOCK break; }
{<0+15>BLOCK doDefault(); }
TRUE
FALSE
EMPTY
EMPTY
{<0+62>BLOCK continue; }
{<0+68>BLOCK doSomething5(); }
{<0+22>EXPR_COND cond1 }
{<0+59>BLOCK break switchlbl; }
TRUE
FALSE
{<0+26>BLOCK dosomething1(); }
{<0+24>BLOCK break; }
EMPTY
EMPTY
EMPTY
{<0+31>BLOCK doSomething3(); break; }
EMPTY
EMPTY
{<0+4>BLOCK bar(); }
EMPTY
{<0+1>EXIT}

Taint analysis: A simple example

```
void do_query(String fields) {
  String id = source();
  String query = "SELECT ";
  if (!fields.empty()) {
    query += fields;
  } else {
    query += "name";
  }
  query += " FROM u WHERE id = " + id;
  db.query(query);
}
```

Source: possibly malicious input

Sink: sensitive operation

Vulnerability!

The taint reaches the sink

Taint analysis: Inter-procedural example

```
void do_query(String fields) {
  String id = "123";
  String query = "SELECT ";
  if (!fields.empty()) {
    query += fields;
  } else {
    query += "name";
  }
  query += " FROM u WHERE id = " + id;
  db.query(query);
}
```

```
void foo() {
  String fields = source();
  do_query(fields);
}
```

Source: possibly malicious input

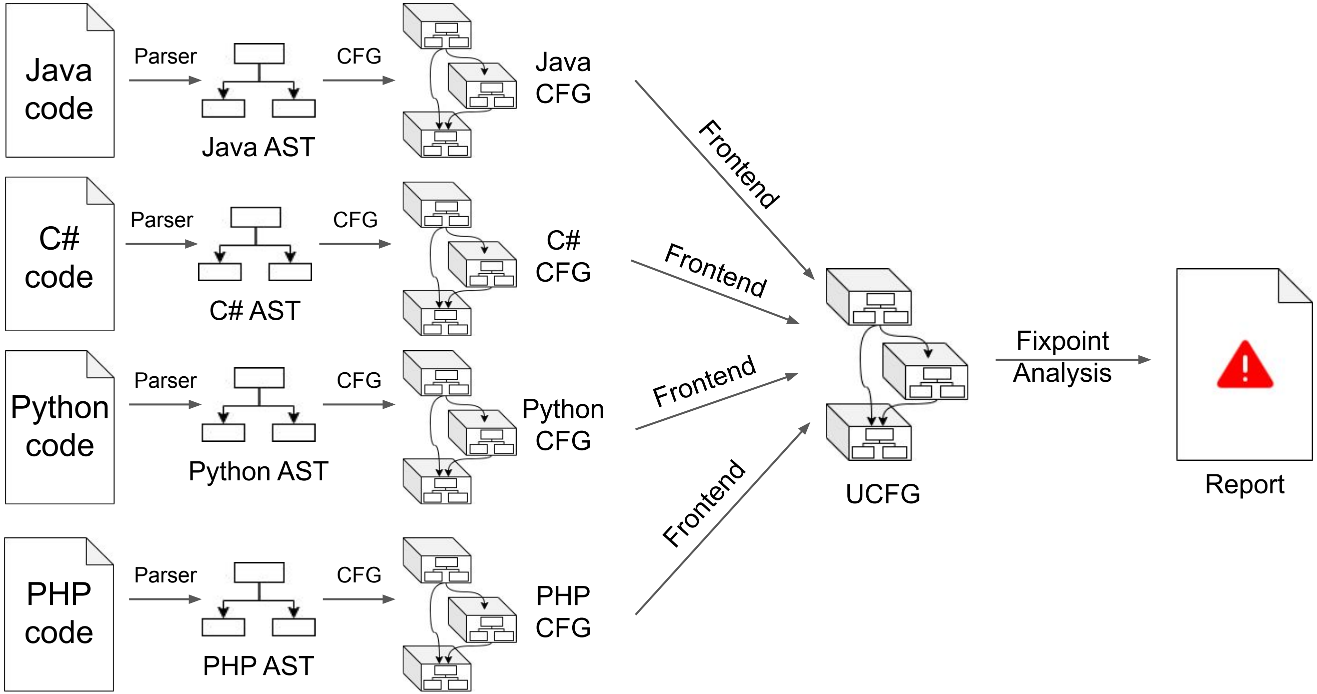
Sink: sensitive operation

Vulnerability!

The taint reaches the sink

Beer Fondue

The fondue

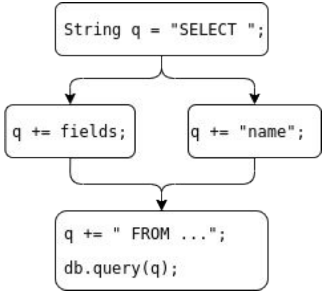


The sonarsource way

CFGs are language dependent

```java
String q = "SELECT ";
if (!fields.empty()) {
  q += fields;
} else {
  q += "name";
}
q += " FROM ...";
db.query(q);
```

Java code



```
String q = "SELECT ";
```

```
q += fields;
```

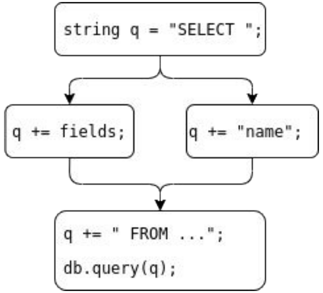
```
q += "name";
```

```
q += " FROM ...";
db.query(q);
```

Java CFG

```csharp
string q = "SELECT ";
if (!string.IsNullOrEmpty(fields)) {
  q += fields;
} else {
  q += "name";
}
q += " FROM ...";
db.query(q);
```

C# code



```
string q = "SELECT ";
```

```
q += fields;
```

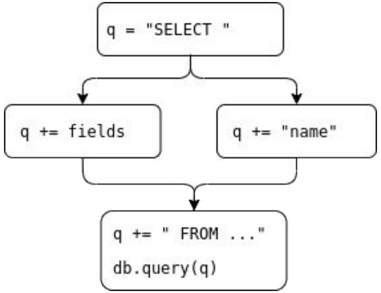
```
q += "name";
```

```
q += " FROM ...";
db.query(q);
```

C# CFG

```python
q = "SELECT "
if not fields:
  q += fields
else:
  q += "name"
q += " FROM ..."
db.query(q)
```

Python code



```
q = "SELECT "
```

```
q += fields
```

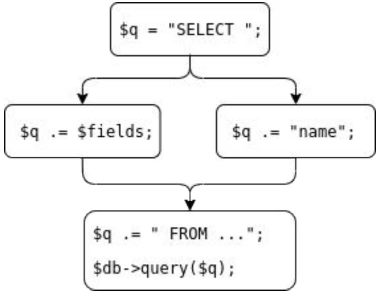
```
q += "name"
```

```
q += " FROM ..."
db.query(q)
```

Python CFG

```php
$q = "SELECT ";
if (!empty($fields)) {
  $q .= $fields;
} else {
  $q .= "name";
}
$q .= " FROM ...";
$db->query($q);
```

PHP code



```
$q = "SELECT ";
```

```
$q .= $fields;
```

```
$q .= "name";
```

```
$q .= " FROM ...";
$db->query($q);
```

PHP CFG

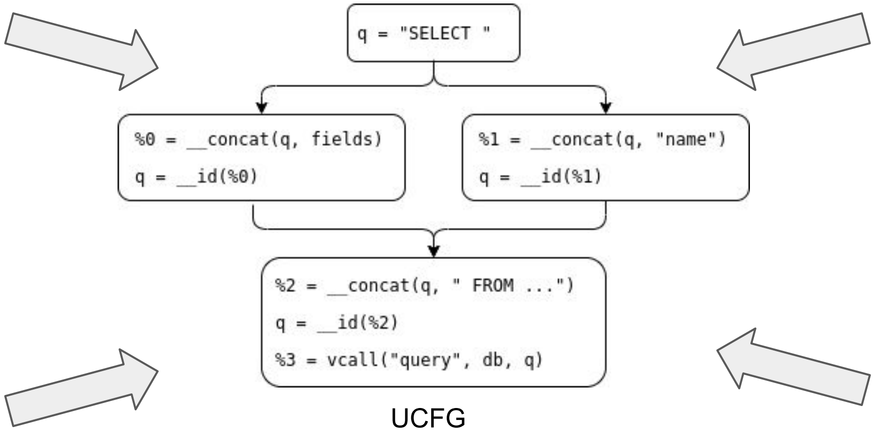
Universal CFGs - language independent

```
String q = "SELECT ";
if (!fields.empty()) {
  q += fields;
} else {
  q += "name";
}
q += " FROM ...";
db.query(q);
```

Java code

```
q = "SELECT "
if not fields:
  q += fields
else:
  q += "name"
q += " FROM ..."
db.query(q)
```

Python code



```
q = "SELECT "
```

```
%0 = __concat(q, fields)
q = __id(%0)
```

```
%1 = __concat(q, "name")
q = __id(%1)
```

```
%2 = __concat(q, " FROM ...")
q = __id(%2)
%3 = vcall("query", db, q)
```

UCFG

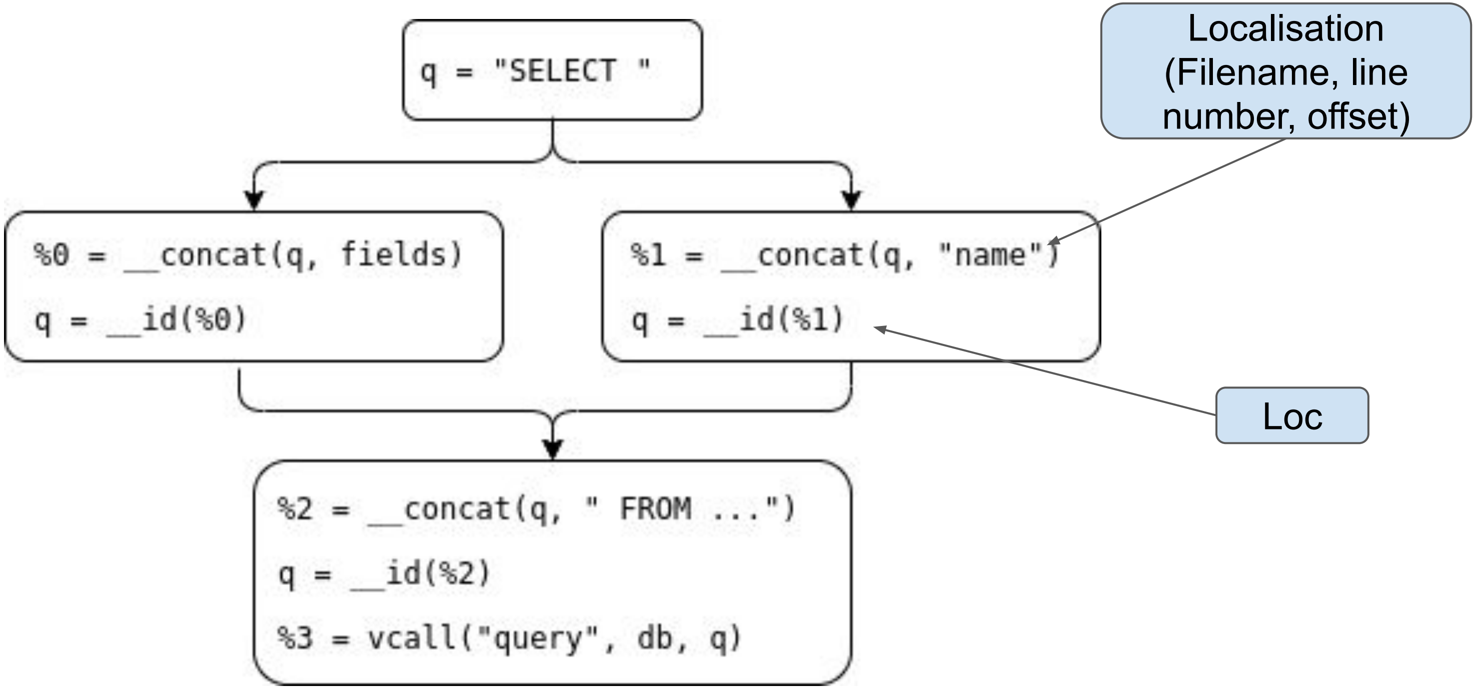
```
string q = "SELECT ";
if (!string.IsNullOrEmpty(fields)) {
  q += fields;
} else {
  q += "name";
}
q += " FROM ...";
db.query(q);
```

C# code

```
$q = "SELECT ";
if (!empty($fields)) {
  $q .= $fields;
} else {
  $q .= "name";
}
$q .= " FROM ...";
$db->query($q);
```

PHP code

UCFGs



```
q = "SELECT "
```

```
%0 = __concat(q, fields)
q = __id(%0)
```

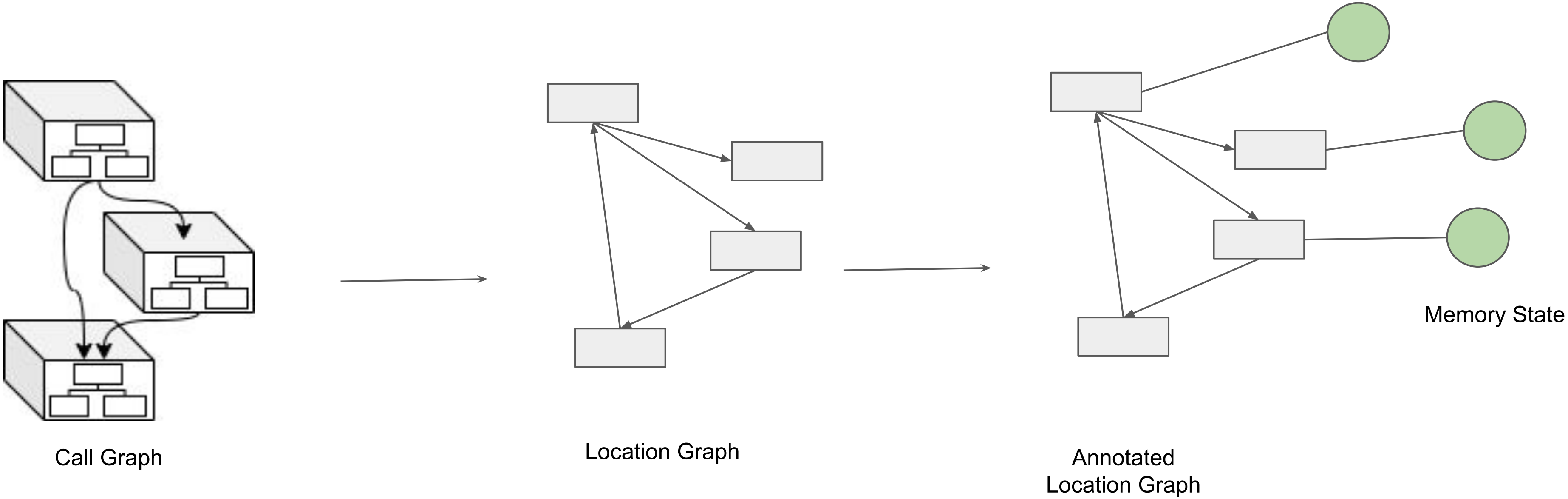
```
%1 = __concat(q, "name")
q = __id(%1)
```

```
%2 = __concat(q, " FROM ...")
q = __id(%2)
%3 = vcall("query", db, q)
```

Localisation (Filename, line number, offset)

Loc

Fixpoint Analysis



Call Graph

Location Graph

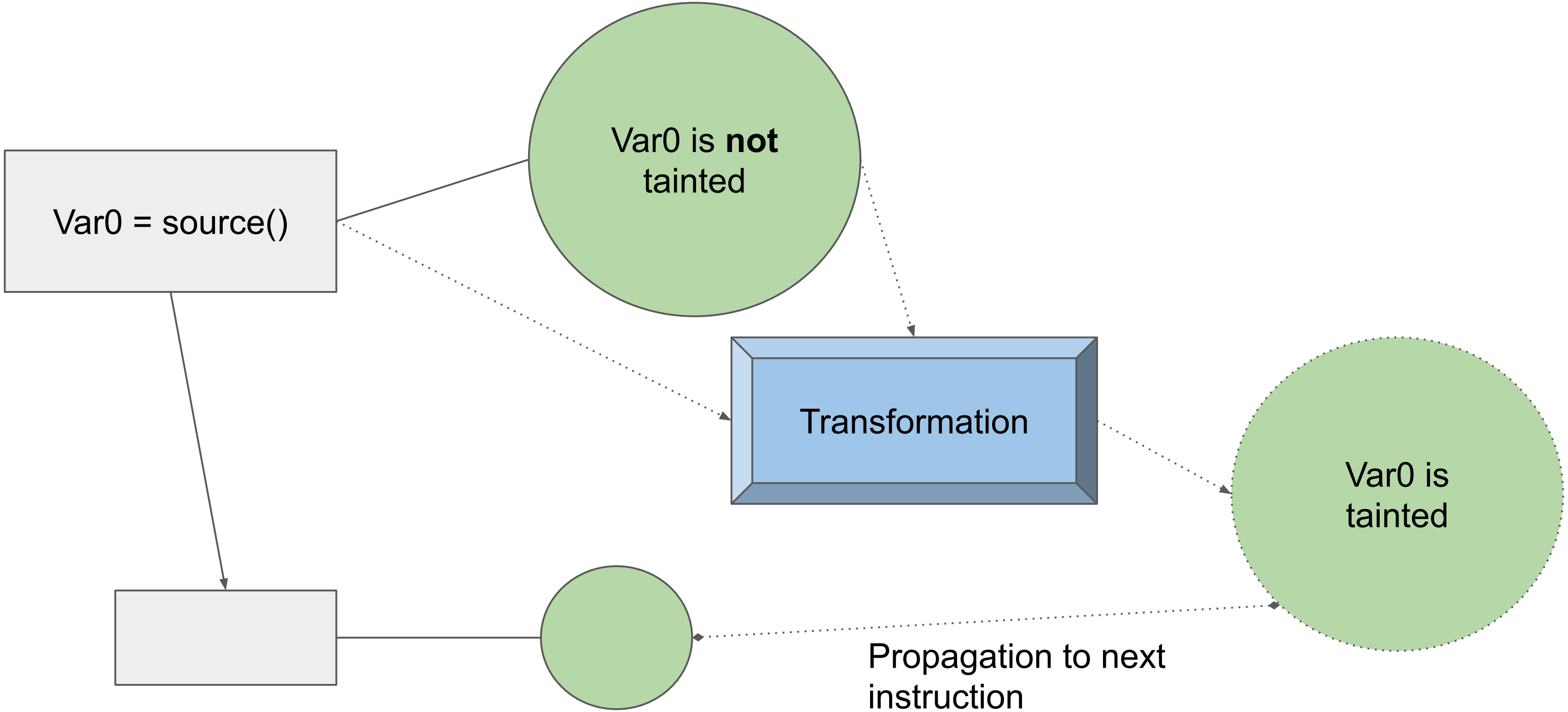
Annotated
Location Graph

Memory State

Memory State

var0: is tainted
var1: is not tainted
var2: tainted & not tainted
var3: we don’t know

Instruction Transformation



Var0 = source()

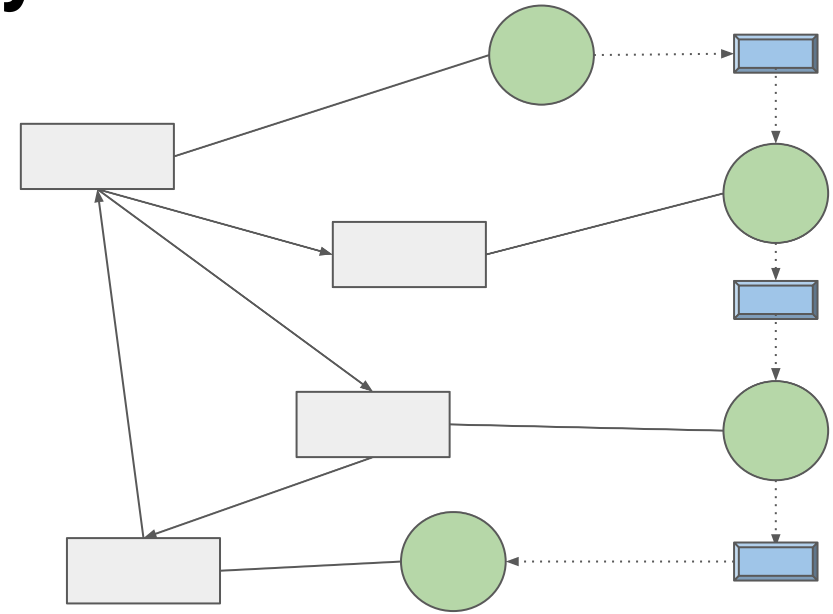
Var0 is **not** tainted

Transformation

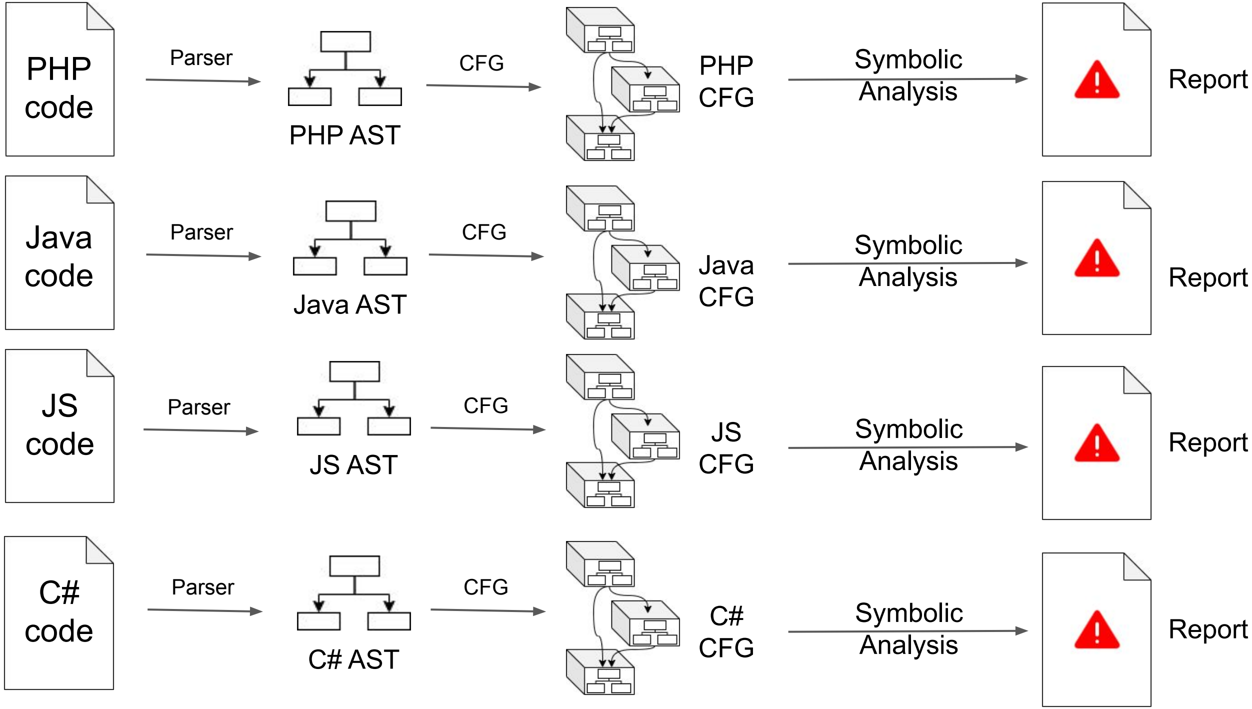
Var0 is tainted

Propagation to next instruction

Fixpoint Analysis



The beer



PHP code → Parser → PHP AST → CFG → PHP CFG → Symbolic Analysis → Report

Java code → Parser → Java AST → CFG → Java CFG → Symbolic Analysis → Report

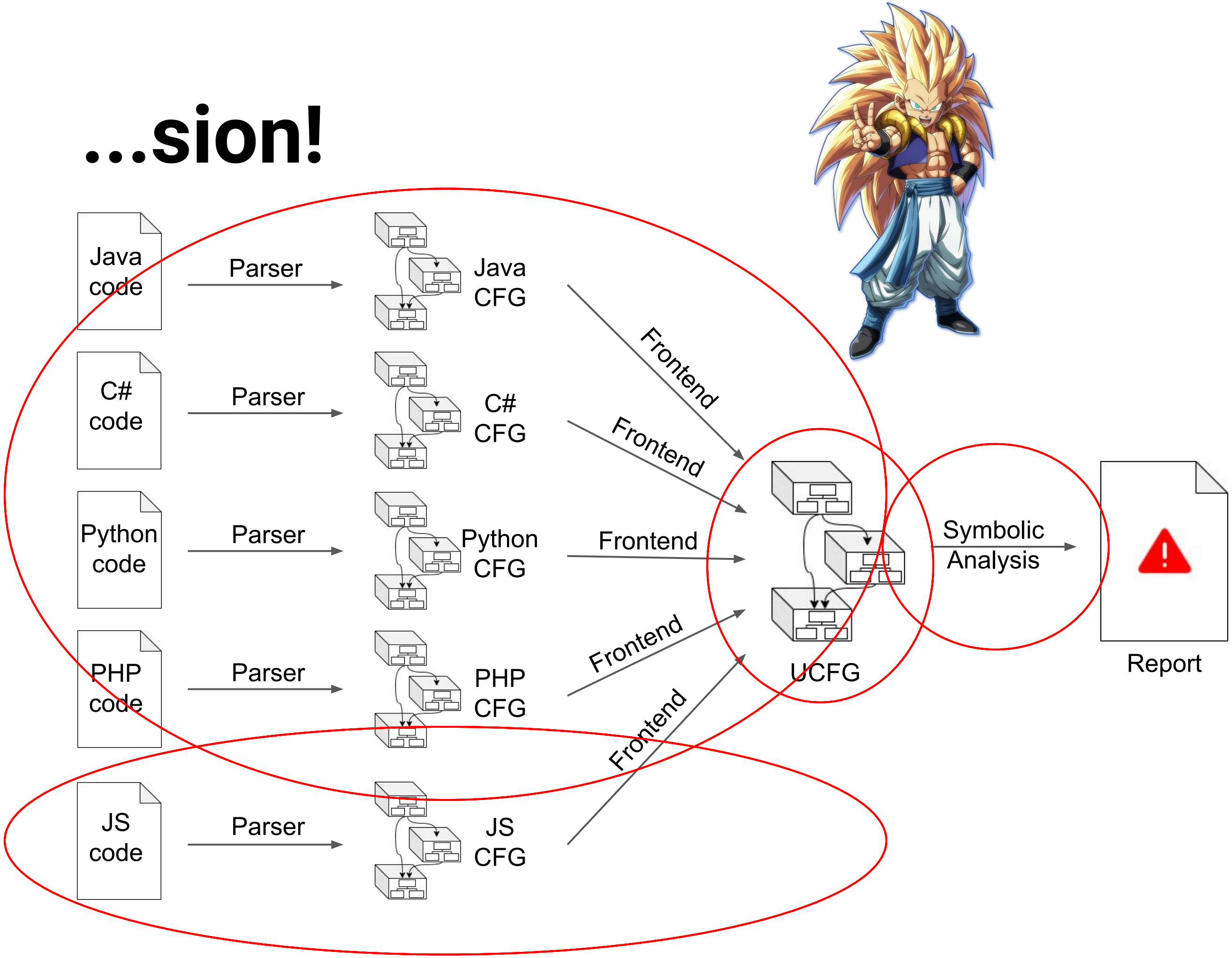
JS code → Parser → JS AST → CFG → JS CFG → Symbolic Analysis → Report

C# code → Parser → C# AST → CFG → C# CFG → Symbolic Analysis → Report

The RIPSTECH way

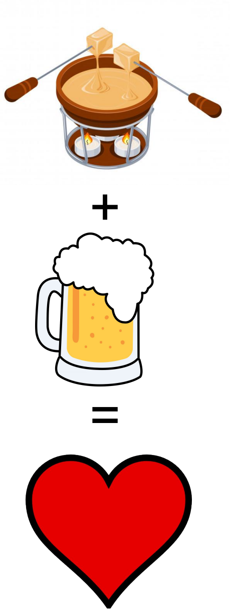
Fuuu...

...sion!



Java code
Parser
Java CFG
C# code
Parser
C# CFG
Python code
Parser
Python CFG
PHP code
Parser
PHP CFG
JS code
Parser
JS CFG
Frontend
Frontend
Frontend
Frontend
Frontend
UCFG
Symbolic Analysis
Report

The (new and shiny) sonarsource way



UCFGs: Live Variable Analysis

Live Variable Analysis

```
void someFunction() {
  int x = 7;
  int y = 12;
  println(y);
  if(...) {
     x = 13;
  } else {
     x = 17;
  }
  println(x);
}// all variables are dead 💀
```

Live Variable Analysis

```
  void someFunction() {
    int x = 7;
    int y = 12;
    println(y);
    if(...) {
       x = 13;
    } else {
       x = 17;
    }
    println(x); // x is read, x is
live
  }// all variables are dead 💀
```

Live Variable Analysis

```
  void someFunction() {
    int x = 7;
    int y = 12;
    println(y);
    if(...) {
       x = 13;// x is written x is 💀
    } else {
       x = 17;// x is written x is 💀
    }
    println(x); // x is read, x is
live
  }// all variables are dead 💀
```

Live Variable Analysis

```
  void someFunction() {
    int x = 7;
    int y = 12;
    println(y); // y is read, y is
live
    if(...) {
       x = 13;// x is written x is 💀
    } else {
       x = 17;// x is written x is 💀
    }
    println(x); // x is read, x is
live
  }// all variables are dead 💀
```

Live Variable Analysis

```
  void someFunction() {
    int x = 7;
    int y = 12;
    println(y); // y is read, y is
live
    if(...) {
       x = 13;// x is written x is 💀
    } else {
       x = 17;// x is written x is 💀
    }
    println(x); // x is read, x is
live
  }// all variables are dead 💀
```

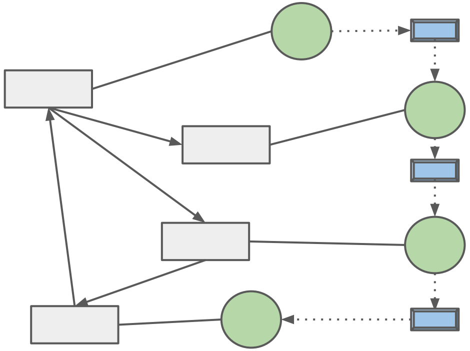
No need to store state of `y` for all this code

Live Variable Analysis

```
  void someFunction() {
    int x = 7;
    int y = 12;
    println(y); // y is read, y is
live
    if(...) {
       x = 13;// x is written x is 💀
    } else {
       x = 17;// x is written x is 💀
    }
    println(x); // x is read, x is
live
  }// all variables are dead 💀
```

No need to store state of `y` for all this code

Live Variable Analysis



Fixpoint Analysis

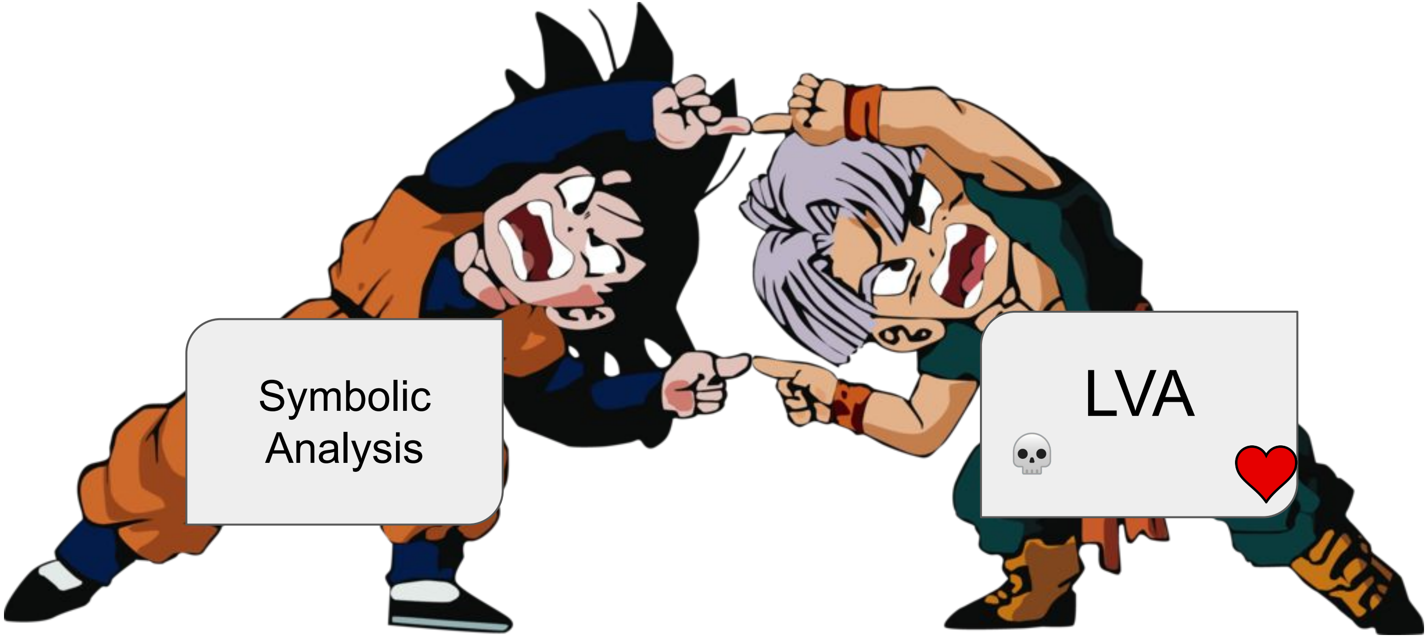
➕

LVA

💀 ❤️

=

Best of both world once again…

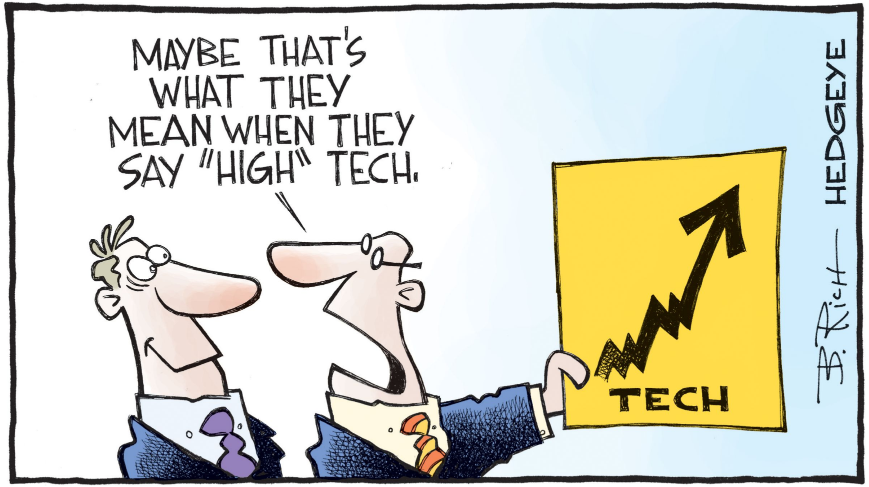


Symbolic Analysis

LVA

💀 ❤

Symbolic Analysis



Symbols

- Representation of *all* states a value may take

```
void do_query(String fields) {
  String id = source();
  String query = "SELECT ";
  if (!fields.empty()) {
    query += fields;
  } else {
    query += "name";
  }
  query += " FROM users WHERE id = " + id;
  db.query(query);
}
```

How to propagate this information to variables?

What happens at merging points?

Parameter: comes from the outside, we do not know much about it

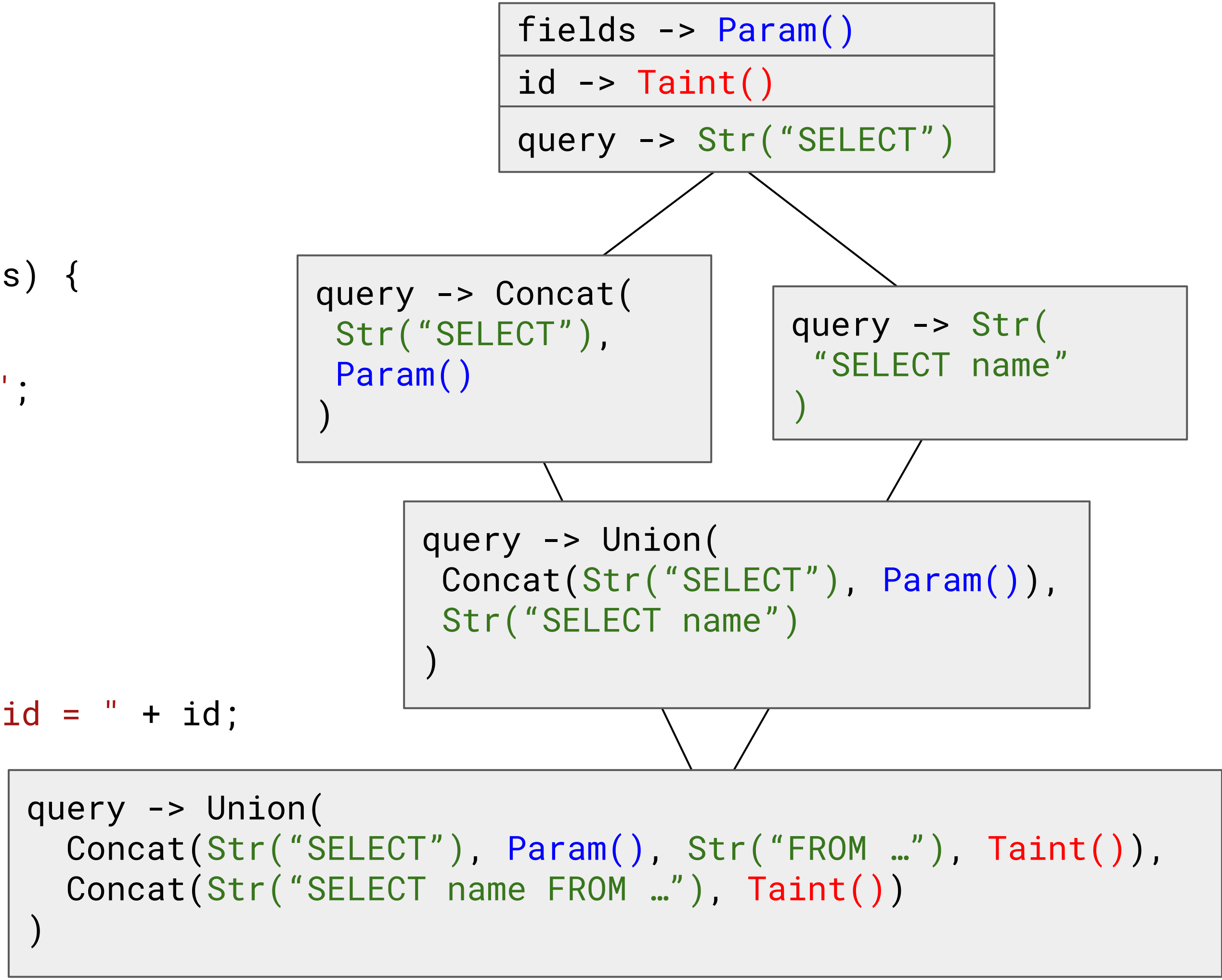
Taint source: comes from a source, potentially malicious data

String literal: can only take one value

String concatenation: concatenation of two symbols

Simulation

```
void do_query(String fields) {
  String id = source();
  String query = "SELECT ";
  if (!fields.empty()) {
    query += fields;
  } else {
    query += "name";
  }
  query += " FROM u WHERE id = " + id;
  db.query(query);
}
```



```
fields -> Param()
id -> Taint()
query -> Str(“SELECT”)
```

```
query -> Concat(
 Str(“SELECT”),
 Param()
)
```

```
query -> Str(
 “SELECT name”
)
```

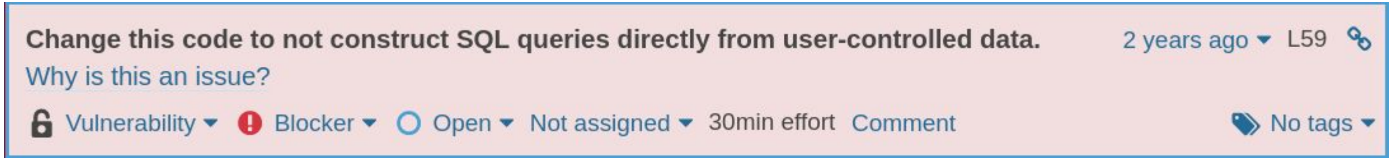
```
query -> Union(
 Concat(Str(“SELECT”), Param()),
 Str(“SELECT name”)
)
```

```
query -> Union(
  Concat(Str(“SELECT”), Param(), Str(“FROM …”), Taint()),
  Concat(Str(“SELECT name FROM …”), Taint())
)
```

Simulation

```
query -> Union(
  Concat(Str(“SELECT”), Param(), Str(“FROM …”), Taint()),
  Concat(Str(“SELECT name FROM …”), Taint())
)
```

```
void do_query(String fields) {
  String id = source();
  String query = "SELECT ";
  if (!fields.empty()) {
    query += fields;
  } else {
    query += "name";
  }
  query += " FROM u WHERE id = " + id;
  db.query(query);
}
```



Change this code to not construct SQL queries directly from user-controlled data.
Why is this an issue?

2 years ago ▾ L59

Vulnerability ▾ Blocker ▾ ○ Open ▾ Not assigned ▾ 30min effort Comment

No tags ▾

Method summaries

```
int foo(MyClass p) {
 p.x = "Hello World";
 if (p.y) {
  return 42;
 } else {
  return 123;
 }
}
```

Summary of foo(MyClass p)

Parameters: **Param(p)**

Side effects:
FieldAccess(**Param(p)**, “x”) -> Str("Hello World")

Return value: Union(
 Primitive(42),
 Primitive(123)
)

Each summary is computed only once!
=> Running time linear in number of methods

Inter-procedural simulation

```
void do_query(String fields) {
 String id = source();
 Builder query = new Builder(id, fields);
 db.query(query.sql);
 db.query(query.safe_sql);
}
Builder(String id, String fields) {
 this.safe_sql = "SELECT * FROM u";
 if (!fields.empty()) {
   this.sql += "SELECT " + fields;
 } else {
   this.sql += "SELECT name";
 }
this.sql += " FROM u WHERE id = " + id;
}
```

Summary of Builder(Builder this, String fields, String id)

Parameters: **Param(this)**, **Param(fields)**, **Param(id)**

Side effects:

FieldAccess(**Param(this)**, “safe_sql”) -> Str("SELECT *...)

FieldAccess(**Param(this)**, “sql”) -> Union(
  Concat(Str(“SELECT”), **Param(fields)**, Str(“FROM …”), **Param(id)**),
  Concat(Str(“SELECT name FROM …”), **Param(id)**)
)

Return value: none

Side effects are applied to `query` object in caller context

Inter-procedural simulation

```
void do_query(String fields) {
 String id = source();
 Builder query = new Builder(id, fields);
 db.query(query.sql);
 db.query(query.safe_sql);
}
Builder(String id, String fields) {
 this.safe_sql = "SELECT * FROM u";
 if (!fields.empty()) {
   this.sql += "SELECT " + fields;
 } else {
   this.sql += "SELECT name";
 }
 this.sql += " FROM u WHERE id = " + id;
}
```

Simulation state (do_query)

```
query -> Object(
    “safe_sql” ->  Str(...)
    “sql” ->  Union(
       Concat(Str(...), Param(fields), Str(...), Taint()),
       Concat(Str(...), Taint())
    )
 )
```

id -> Taint()

fields -> Param(fields)

Vulnérabilités partielles

```
class Db {
  void query(String sql) {
   conn = new Connection();
   conn.execute(sql);
  }
}
```

Sink configuré

Sommaire de Db#query
Paramètres: **Param(sql)**
Effets secondaires: aucun
Valeur de retour: aucune
Vulnérabilités partielles: **Param(sql)**

Field sensitivity

```
void do_query(String fields) {
 id = source();
 query = new Builder(id, fields);
▶ db.query(query.sql);
▶ db.query(query.safe_sql);
}
Builder(String fields, String id) {
 this.safe_sql = "SELECT * FROM u";
 if (!fields.empty()) {
   this.sql += "SELECT " + fields;
 } else {
   this.sql += "SELECT name";
 }
 this.sql += " FROM u WHERE id = " + id;
}
```

Analyse “field *insensitive*”:

Query object
“sql” -> **tainted**

```
query.sql -> valeur taintée
query.safe_sql -> valeur taintée
query.foo -> valeur taintée
```

Analyse “field *sensitive*”:

Query object
“sql” -> Union(..., **Taint()**, ...)
“safe_sql” -> Str()

```
query.sql -> symbole union tainté
query.safe_sql -> string non tainté
query.foo -> symbol indéfini
```

Taint analysis

```
void do_query(String fields) {
 String id = source();
 Builder query = new Builder(id, fields);
 db.query(query.sql);
 db.query(query.safe_sql);
}
Builder(String id, String fields) {
 this.safe_sql = "SELECT * FROM u";
 if (!fields.empty()) {
   this.sql += "SELECT " + fields;
 } else {
   this.sql += "SELECT name";
 }
 this.sql += " FROM u WHERE id = " + id;
}
```

Simulation state (do_query)

query -> Object(
  “safe_sql” -> Str(...)
  “sql” -> Union(
    Concat(Str(...), **Param(fields)**, Str(...), **Taint()**),
    Concat(Str(...), **Taint()**)
)
)

No vulnerability!

Vulnerability!

Awesome, but what is it good for?

To detect this kind of stuff !

CVE-2019-0221

Soon...

Markup sensitivity: Safe code

```
void do_query(String fields) {
  id = source();
  id = escape_quotes(id);
  query = "SELECT name FROM u " +
          "WHERE id = \"" + id + "\"";
  db.query(query);
}
```

Source: possibly malicious input

Sanitizer: allow the input to be safely embedded into a sensitive operation

Sink: sensitive operation

✓ **No vulnerability!**

Markup sensitivity: Unsafe code

```
void do_query(String fields) {
  id = source();
  id = escape_quotes(id);
  query = "SELECT name FROM u " +
          "WHERE id = " + id;
  db.query(query);
}
```

Sanitizer: insufficient in this context!

Example of malicious input:

foo; DROP TABLE u;

Vulnerability!

Analysis (unsafe code)

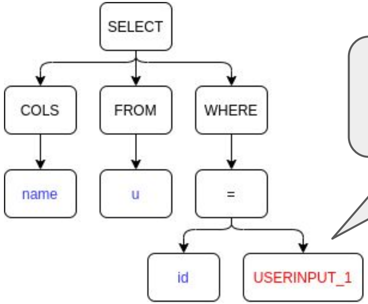
```
void do_query(String fields) {
  id = source();
  id = escape_quotes(id);
  query = "SELECT name FROM u " +
          "WHERE id = " + id;
  db.query(query);
}
```

```
query -> Concat(
  Str("SELECT name FROM u WHERE id = "),
  Taint()
)
```

String representation

```
T name FROM u WHERE id = USERINPUT_1
```

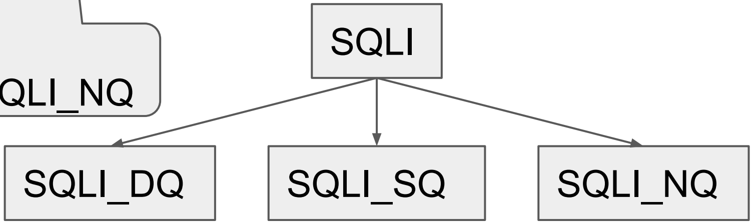
Compute abstract syntax tree



Identifier without quotes

Provided: S
SQLI_SQ & SQLI_DQ

Required:
Sanitization for SQLI_NQ



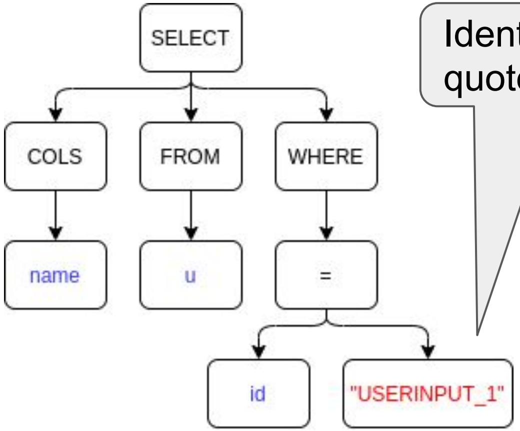
Analysis (safe code)

```
void do_query(String fields) {
  id = source();
  id = escape_quotes(id);
  query = "SELECT name FROM u " +
          "WHERE id = \"" + id + "\
  db.query(query);
}
```

Required:
Sanitization for SQLI_DQ

Provided: Sanitization for
SQLI_SQ & SQLI_DQ

```
query -> Concat(
  Str("SELECT name FROM u WHERE id = \""),
  Taint(),
  Str("\"")
)
```



Identifier with quotes

@sonarsource
https://sonarsource.com