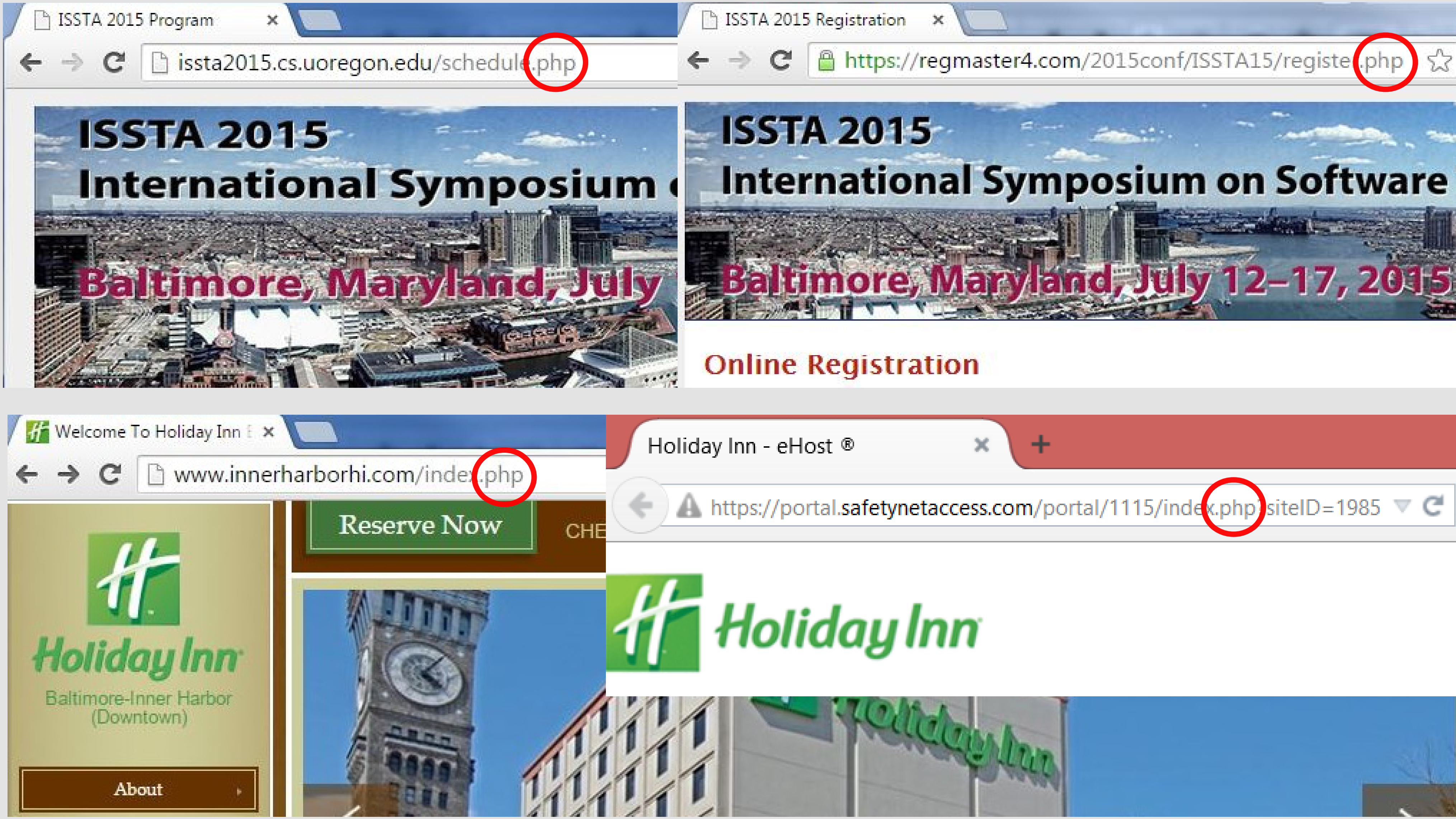
Experience Report:

An Empirical Study of PHP Security Mechanism Usage

Johannes Dahse and Thorsten Holz
Ruhr-University Bochum, Germany

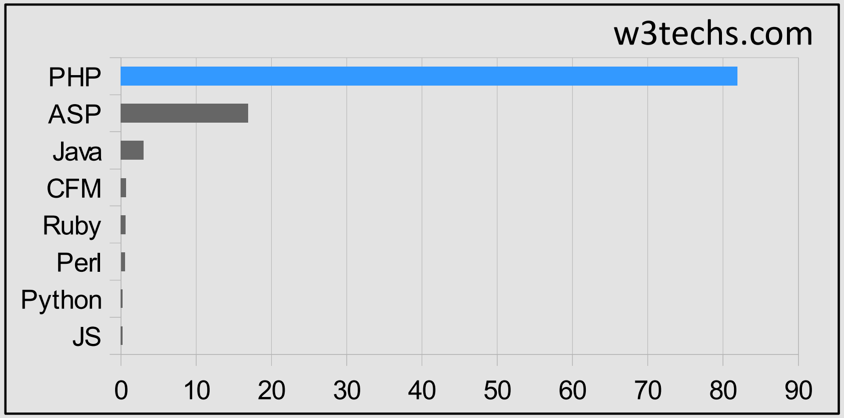
ISSTA 2015, July 13-17, Baltimore, Maryland, USA

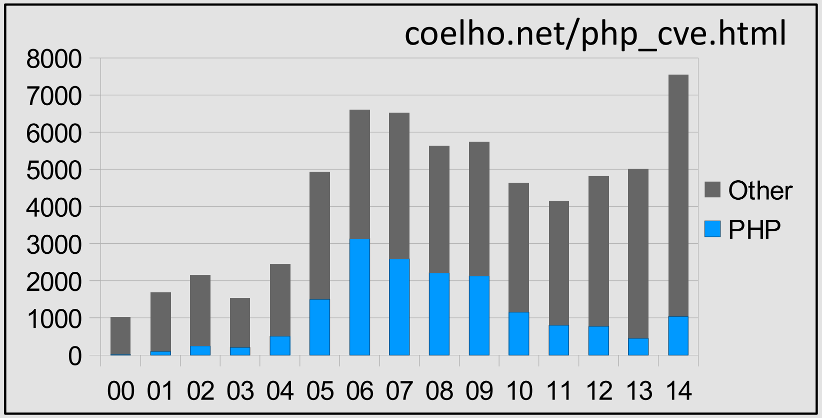
Experience Report:
An Empirical Study of PHP Security Mechanism Usage



1.1 Web Application State

- 82% of all websites run PHP as server-side language
- Weakly-typed language; requires more checks, introduces oddities
- 25% of all found vulnerabilities (CVE database) are related to PHP
- XSS and SQLi still one of the most commonly found security flaws





1.2 Security Mechanisms

- Best-practice security guidelines evolved, but no standards
- Developers apply their own favorite security mechanism
- Different programming patterns for input validation or input sanitization emerged, with advantages and drawbacks
- Some work generically, others work only for a certain markup context
- Programming mistakes and misplacing due to common pitfalls can still lead to vulnerabilities

1.3 Research Questions

Essential for developers, code auditors, and static analysis engineers

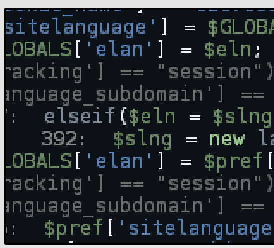
RQ1. Which security mechanisms are available?

RQ2. Which pitfalls might these mechanisms imply?

RQ3. Which security mechanisms are used how often in modern (web) applications?

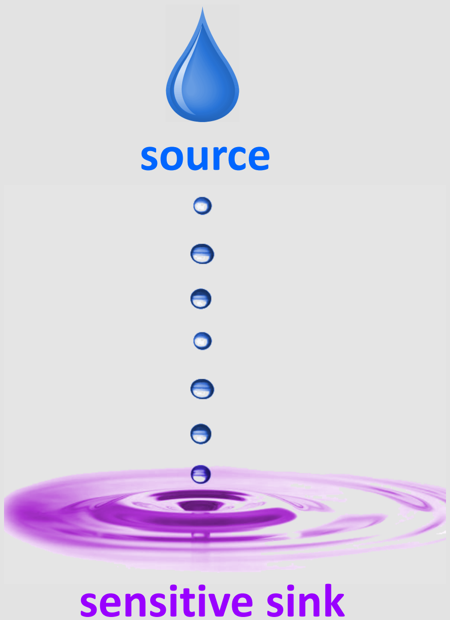
RQ4. Which security mechanism is used to prevent which vulnerability type in which markup context?

RQ5. Which pitfalls occur in practice?

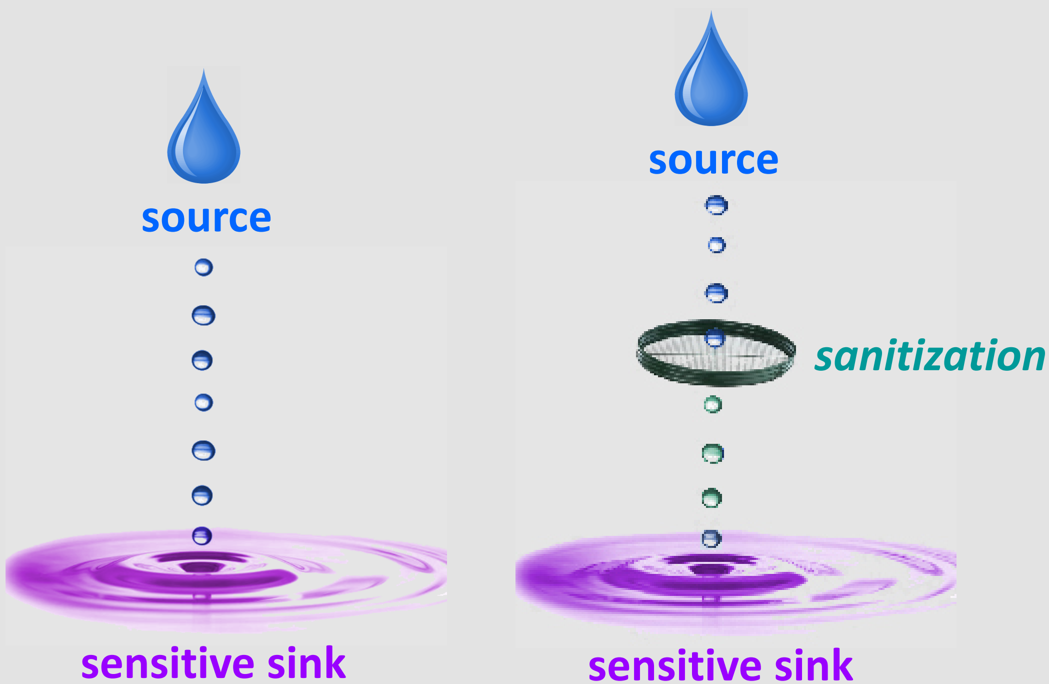


2. Security Mechanisms

2.1 Taint-style Vulnerabilities

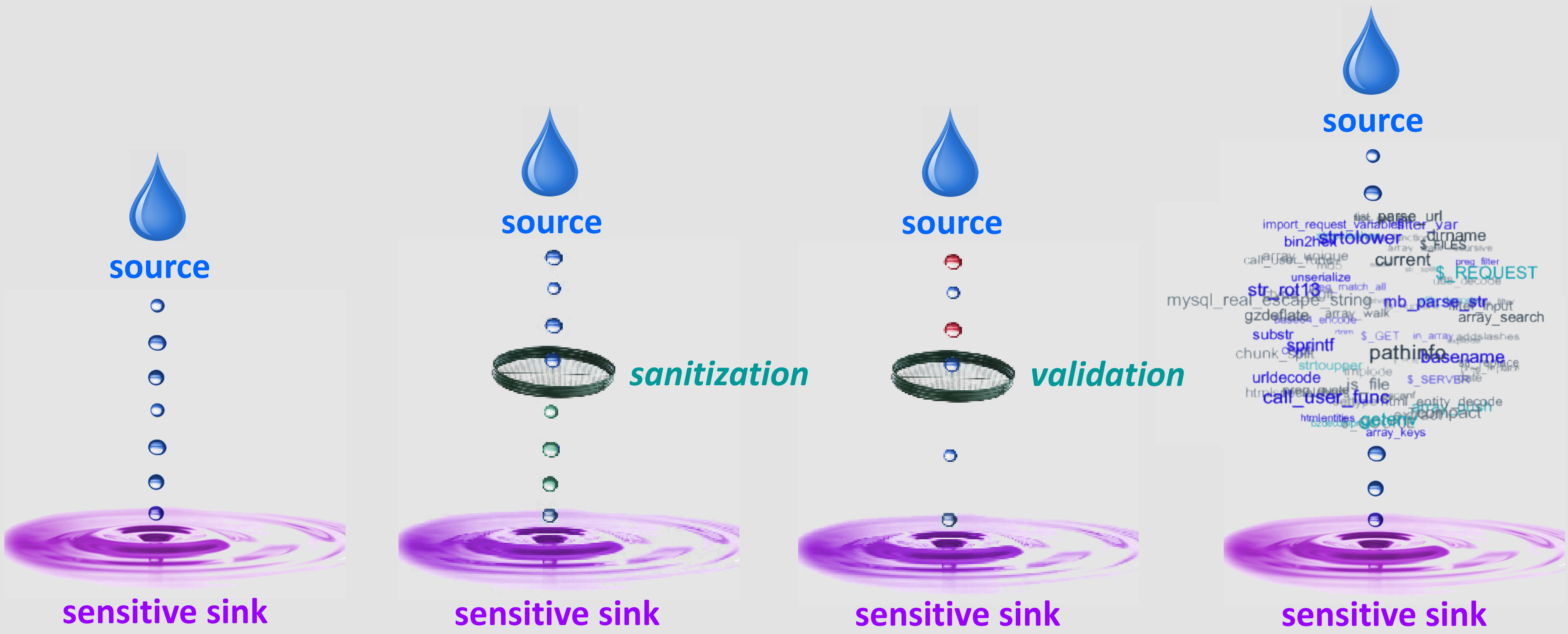


2.1 Types of Security Mechanisms



2.1 Types of Security Mechanisms

2.1 Types of Security Mechanisms



2.2 Generic Input Sanitization

- Explicit Type Casting

```php
$id = $_GET['id'];
echo intval($id);
mysql_query('SELECT * FROM t WHERE id='.(int)$id);
```

2.3 Context-Sensitive Input Sanitization

- Converting

```
$url = htmlentities($_GET['id']);
echo '<a href="">' . $url . '</a>';
echo "<a href='$url'>click</a>";
echo '<a href="' . $url . '">click</a>';
```

- Escaping

```
$id = mysql_real_escape_string($_GET['id']);
mysql_query("SELECT * FROM t WHERE id = '$id'");
mysql_query('SELECT * FROM t WHERE id = ' . $id);
```

2.3 Context-Sensitive Input Sanitization

- Converting

```
$url = htmlentities($_GET['id']);        “ → &quot;
echo '<a href="“">' . $url . '</a>';    < → &lt;
echo “<a href='$url'>click</a>”;
echo '<a href="' . $url . '">click</a>';
```

- Escaping

```
$id = mysql_real_escape_string($_GET['id']);
mysql_query(“SELECT * FROM t WHERE id = '$id'”);
mysql_query('SELECT * FROM t WHERE id = ' . $id);
```

2.3 Context-Sensitive Input Sanitization

- Converting

```
$url = htmlentities($_GET['id']);      " → &quot;
echo '<a href="">' . $url . '</a>';    < → &lt;
echo "<a href='$url'>click</a>"; 'onclick='alert(1)
echo '<a href="' . $url . '">click</a>';
```

- Escaping

```
$id = mysql_real_escape_string($_GET['id']);
mysql_query("SELECT * FROM t WHERE id = '$id'");
mysql_query('SELECT * FROM t WHERE id = ' . $id);
```

Pitfalls

2.3 Context-Sensitive Input Sanitization

Pitfalls

- Converting

```
$url = htmlentities($_GET['id']);        " → &quot;
echo '<a href="">' . $url . '</a>';      < → &lt;
echo "<a href='$url'>click</a>";  'onclick='alert(1)
echo '<a href="' . $url . '">click</a>';
```

javascript:alert(1)

- Escaping

```
$id = mysql_real_escape_string($_GET['id']);
mysql_query("SELECT * FROM t WHERE id = '$id'");
mysql_query('SELECT * FROM t WHERE id = ' . $id);
```

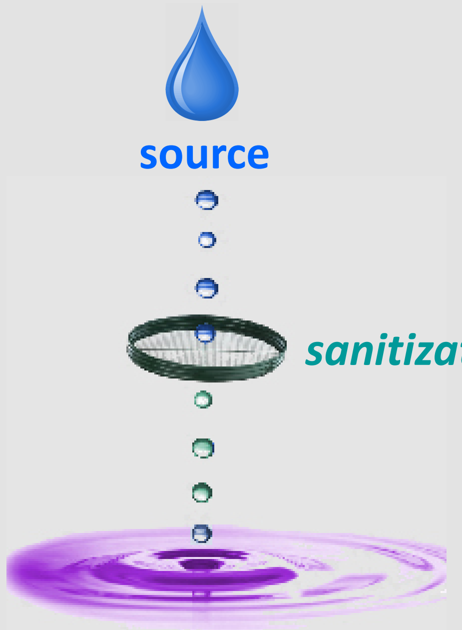
2.3 Context-Sensitive Input Sanitization

- Converting

```
$url = htmlentities($_GET['id']);
echo '<a href="">' . $url . '</a>';
echo "<a href='$url'>click</a>";
echo '<a href="' . $url . '">click</a>';
```

- Escaping

```
$id = mysql_real_escape_string($_GET['id']); ' → \'
mysql_query("SELECT * FROM t WHERE id = '$id'");
mysql_query('SELECT * FROM t WHERE id = ' . $id);
```



2.3 Context-Sensitive Input Sanitization

Pitfalls

- Converting

```
$url = htmlentities($_GET['id']);
echo '<a href="">' . $url . '</a>';
echo "<a href='$url'>click</a>";
echo '<a href="' . $url . '">click</a>';
```

- Escaping

```
$id = mysql_real_escape_string($_GET['id']); ' → \'
mysql_query("SELECT * FROM t WHERE id = '$id'");
mysql_query('SELECT * FROM t WHERE id = ' . $id);
```

1 or 1=1

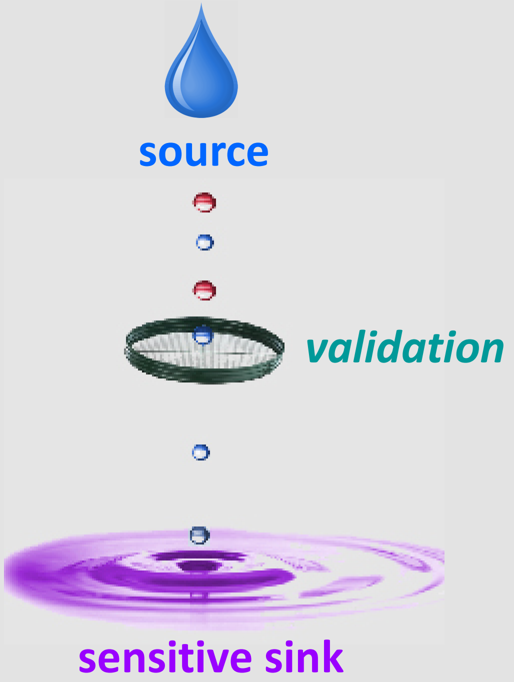
2.4 Generic Input Validation

- Type Validation

```php
$id = $_GET['id'];
if(is_numeric($id)) {          echo $id; }
if(is_int($id) === true) {     echo $id; }
if((int)$id) {                 echo $id; }
if($id = (int)$id) {           echo $id; }
```

- Comparing

```php
$name = $_GET['name'];
if($name == 'issta') {         echo $name; }
if($name === 'issta') {        echo $name; }
if($name == 15) {              echo $name; }
if($name === 15) {             echo $name; }
```



2.4 Generic Input Validation

Pitfalls

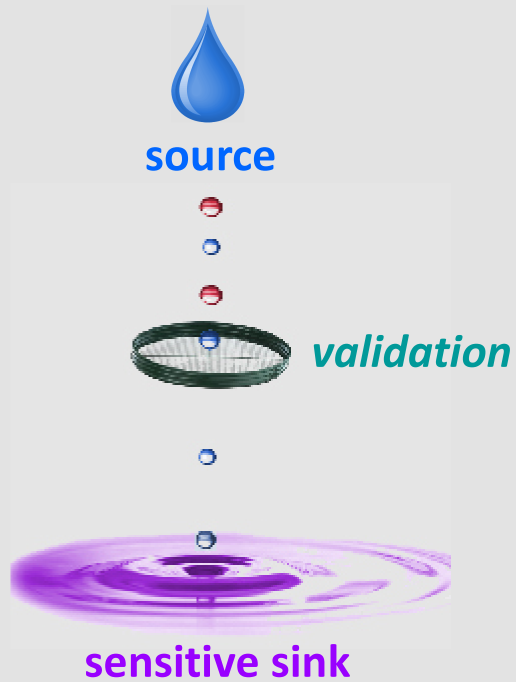
- Type Validation

1<svg onload=alert(1)>

```php
$id = $_GET['id'];
if(is_numeric($id)) {          echo $id; }
if(is_int($id) === true) {     echo $id; }
if((int)$id) {                 echo $id; }
if($id = (int)$id) {           echo $id; }
```

- Comparing

```php
$name = $_GET['name'];
if($name == 'issta') {         echo $name; }
if($name === 'issta') {        echo $name; }
if($name == 15) {              echo $name; }
if($name === 15) {             echo $name; }
```



2.4 Generic Input Validation

- Type Validation

```
$id = $_GET['id'];
if(is_numeric($id)) {           echo $id; }
if(is_int($id) === true) {      echo $id; }
if((int)$id) {                  echo $id; }
if($id = (int)$id) {            echo $id; }
```

- Comparing

```
$name = $_GET['name'];
if($name == 'issta') {          echo $name; }
if($name === 'issta') {         echo $name; }
if($name == 15) {               echo $name; }
if($name === 15) {              echo $name; }
```

2.4 Generic Input Validation

Pitfalls

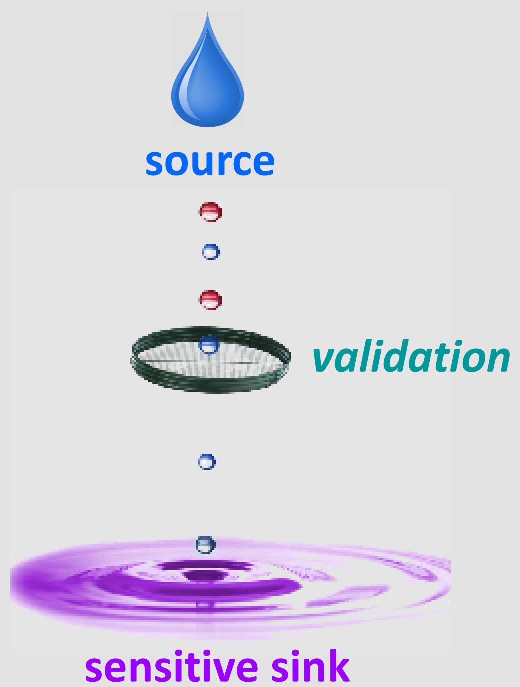
- Type Validation

```
$id = $_GET['id'];
if(is_numeric($id)) {          echo $id; }
if(is_int($id) === true) {     echo $id; }
if((int)$id) {                 echo $id; }
if($id = (int)$id) {           echo $id; }
```

- Comparing

```
$name = $_GET['name'];
if($name == 'issta') {         echo $name; }
if($name === 'issta') {        echo $name; }
if($name == 15) {              echo $name; }
if($name === 15) {             echo $name; }
```

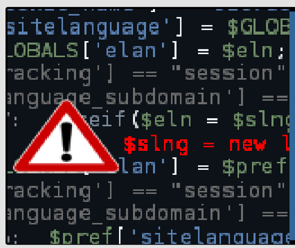
`15<svg onload=alert(1)>`



3. Static Enumeration

3.1 Approach

- Erroneous approach: Count occurrences of security related features
  - Over-approximation when features are used for other purposes
  - Under-approximation when features are used in reusable code
- Our approach: Use modified version of our static analysis prototype
- Leverages backwards-directed context-sensitive taint analysis
- Count security mechanism only when data reaches a sensitive sink that was previously sanitized/validated and was previously tainted

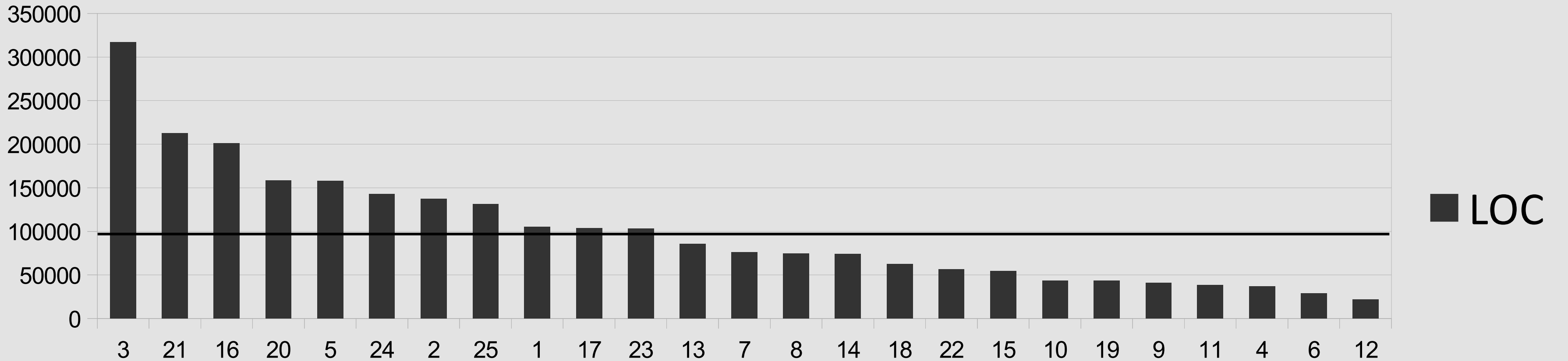


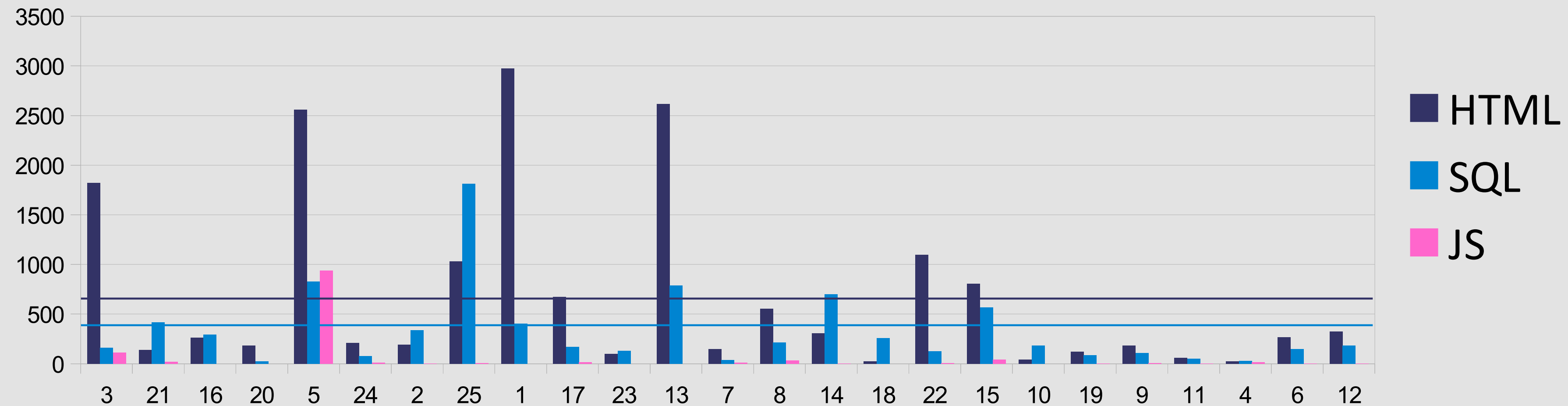
4. Empirical Study

4.1 Selected Applications

- 25 PHP applications
  - Open source, active and popular according to W3Tech's usage statistic
  - Size of at least 20 KLOC
  - Works standalone, does not extensively use reflection or framework
- 2.5 million lines of code (LOC) in total
- 26,006 unique data flows analyzed

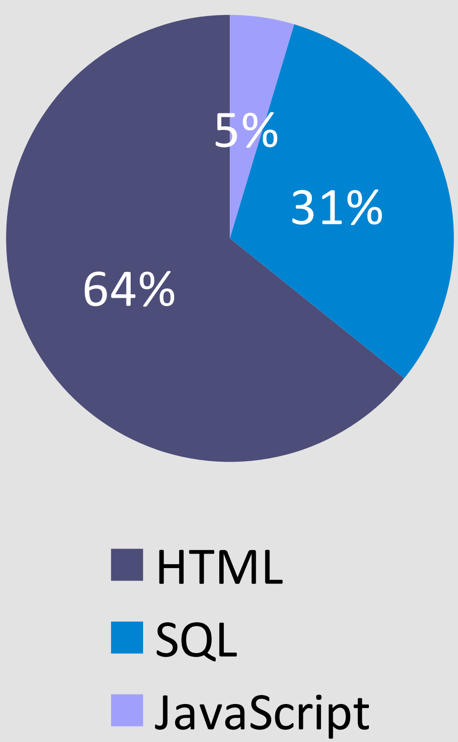
4.2 LOC and Markup Contexts



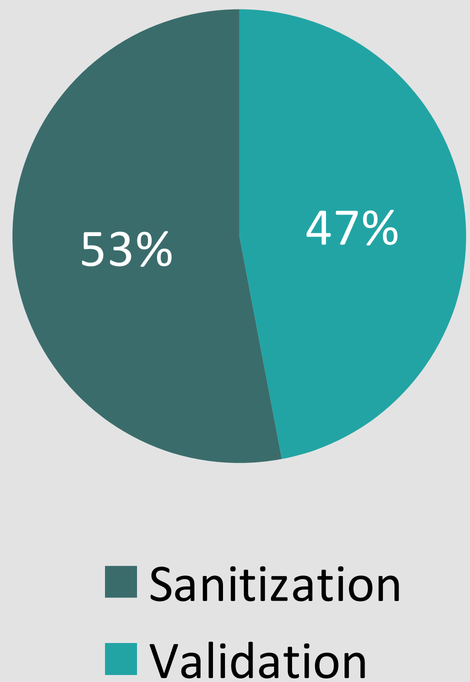


4.3 Markup Contexts and Security Mechanisms

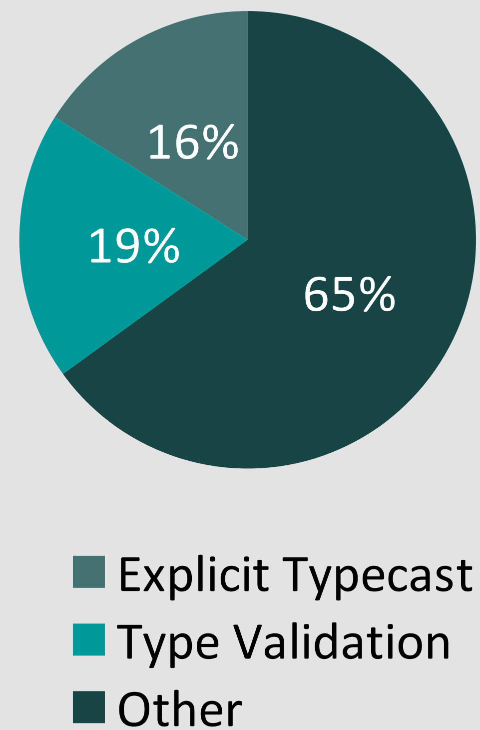
Markup Contexts



Mechanism Types

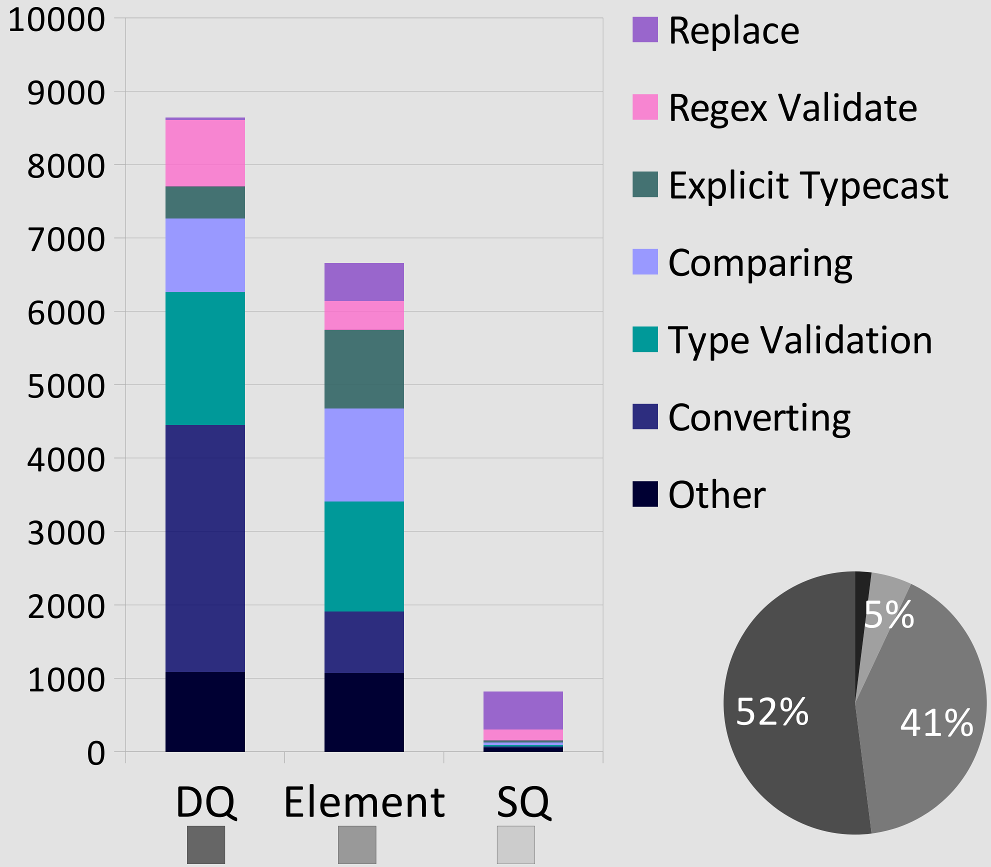


Top Mechanisms

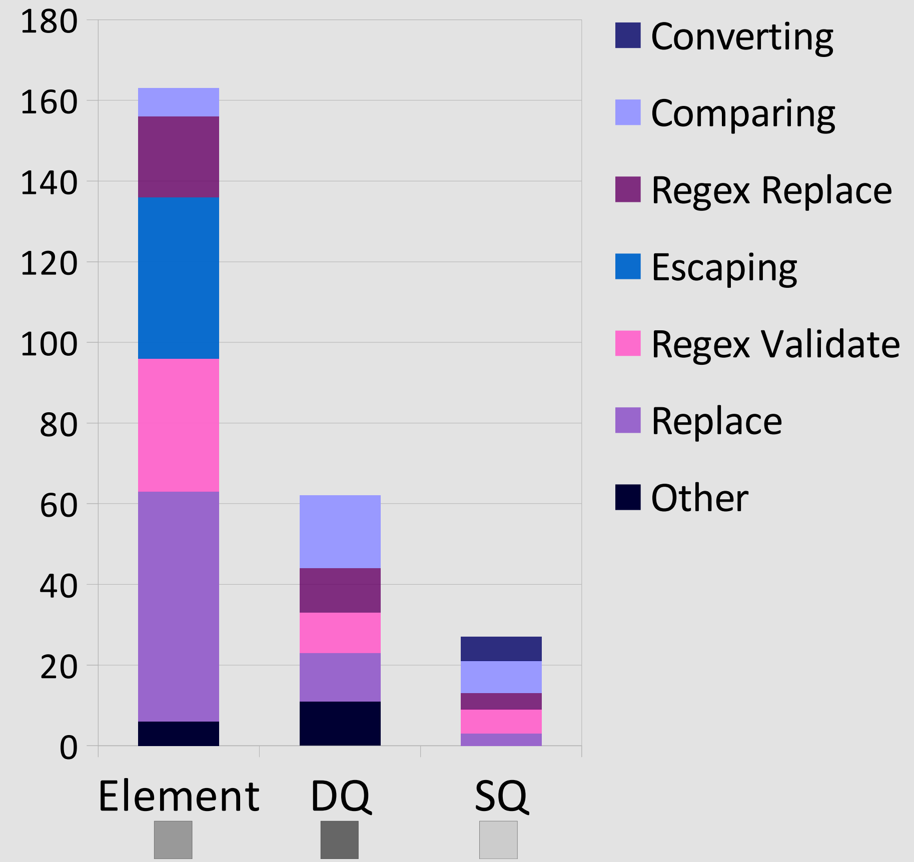


4.3.1 HTML Markup Security

Mechanisms correctly applied

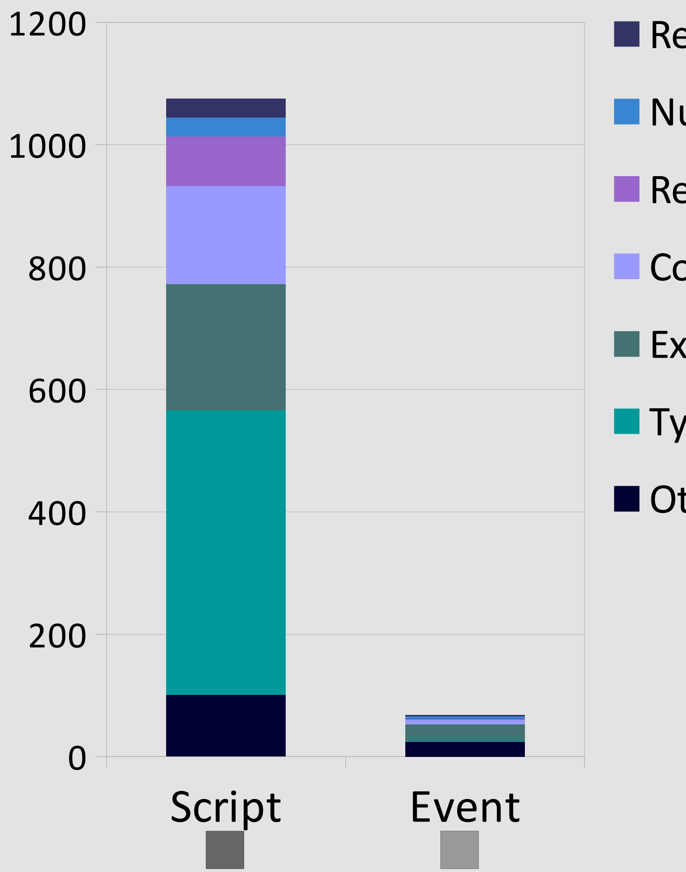


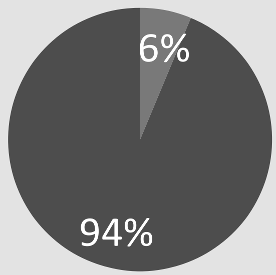
Mechanisms wrongly applied



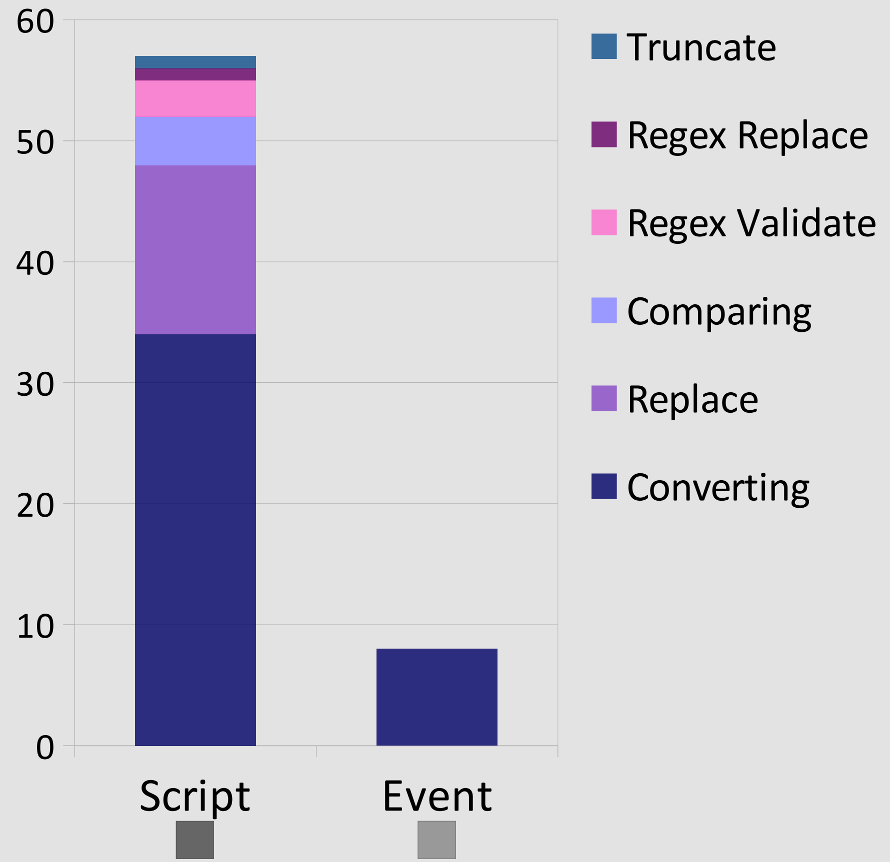
4.3.2 JavaScript Markup Security

Mechanisms correctly applied



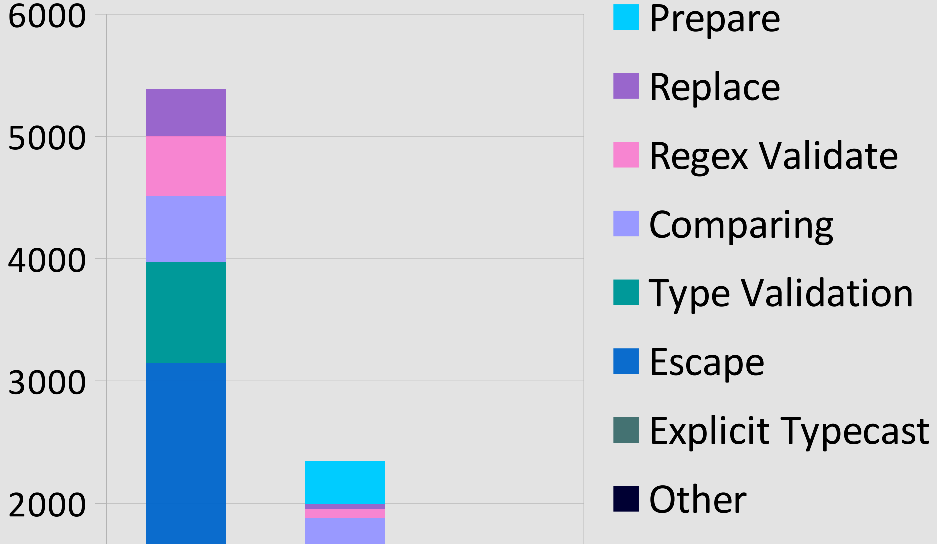


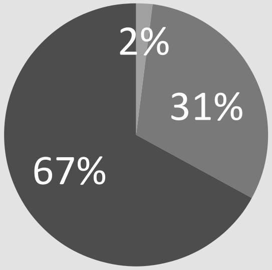
Mechanisms wrongly applied



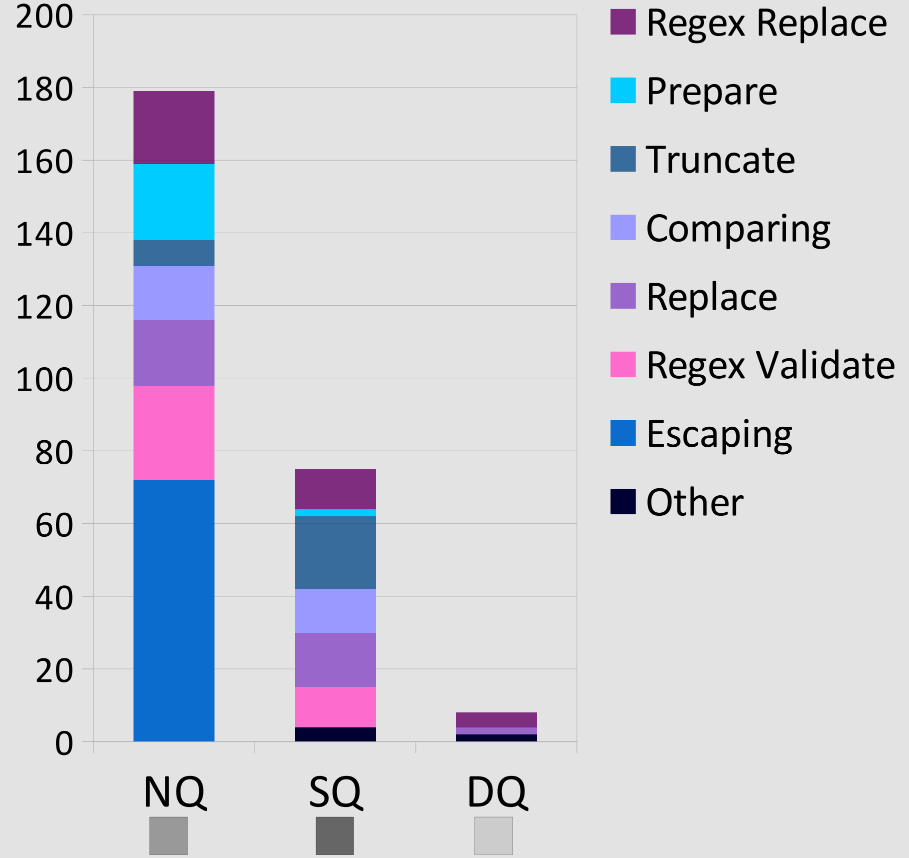
4.3.3 SQL Markup Security

Mechanisms correctly applied



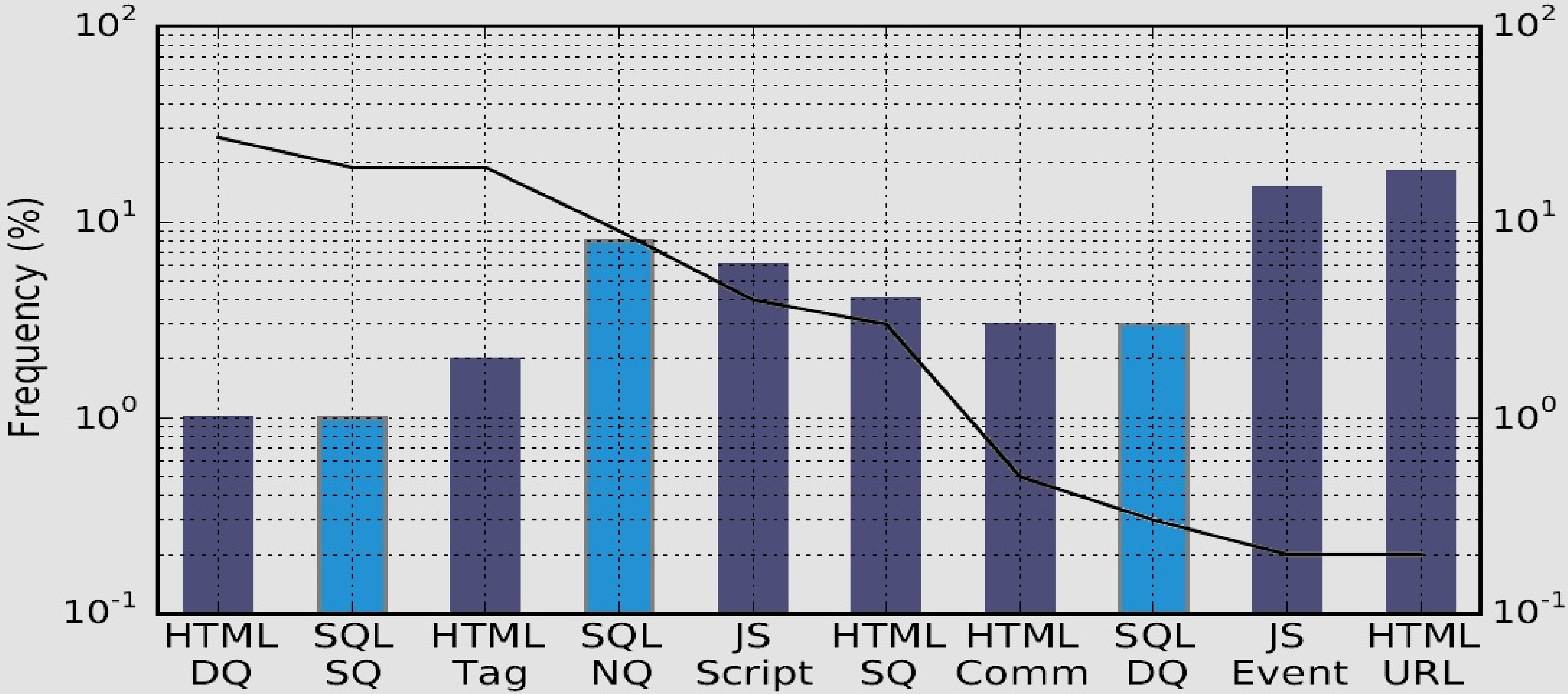


Mechanisms wrongly applied



4.4 Lessons Learned

Pitfall density (bars) versus markup frequency (line)



4.5 Threads to Validity

- Only 25 popular applications
- Static code analysis is limited (FP, FN, mistakes)
- Misinterpretation of developer intention

- Caution to draw strong conclusions and to generalize

Questions?

johannes.dahse@hgi.rub.de

Thank you!

Enjoy the conference.