J-FORCE: FORCED EXECUTION ON JAVASCRIPT

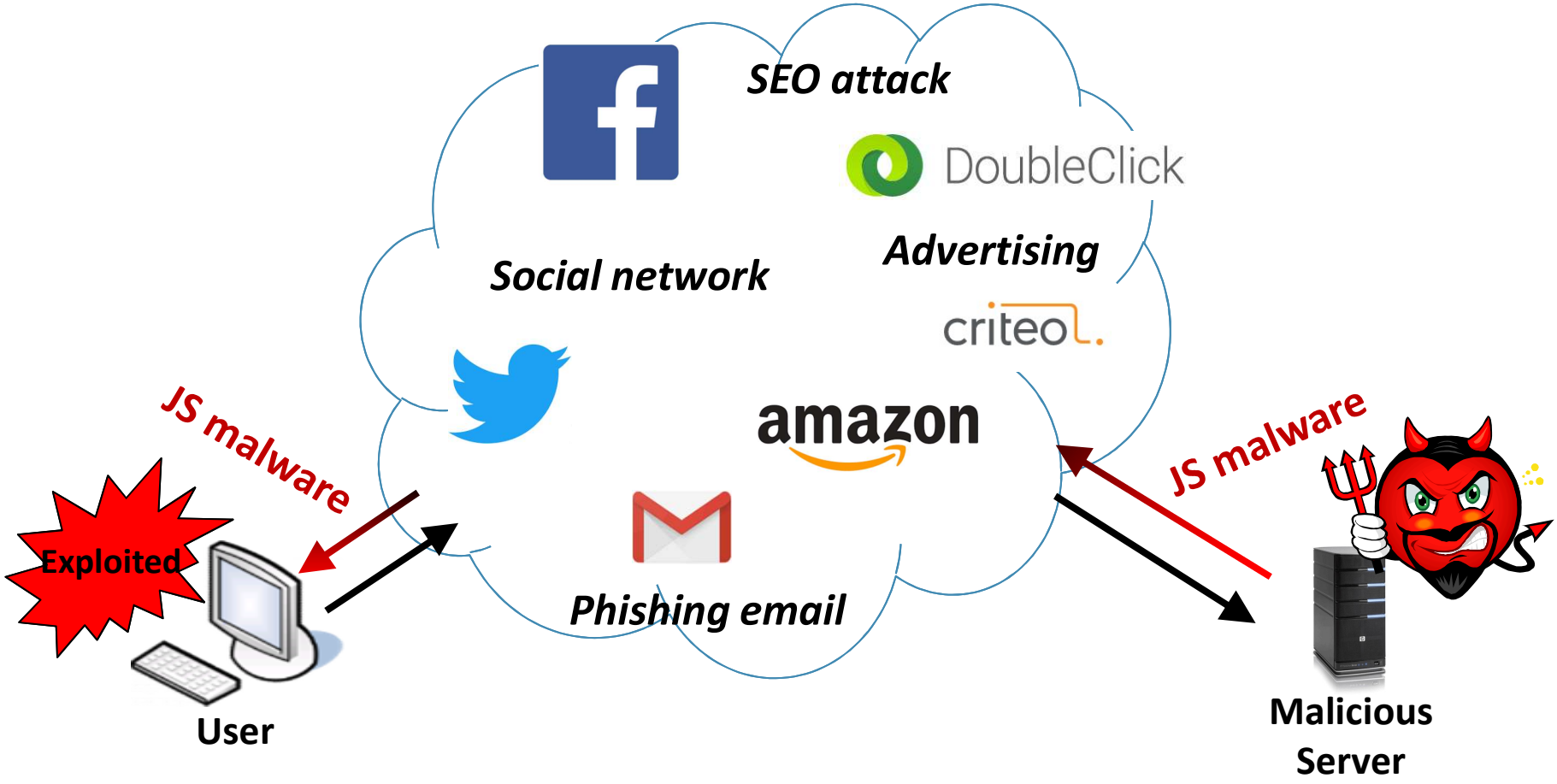
Kyungtae Kim[1], I Luk Kim[1], Chung-Hwan Kim[1], Yonghwi Kwon[1], Yunhui Zheng[2], Xiangyu Zhang[1], Dongyan Xu[1]

[1]Department of Computer Science, Purdue University

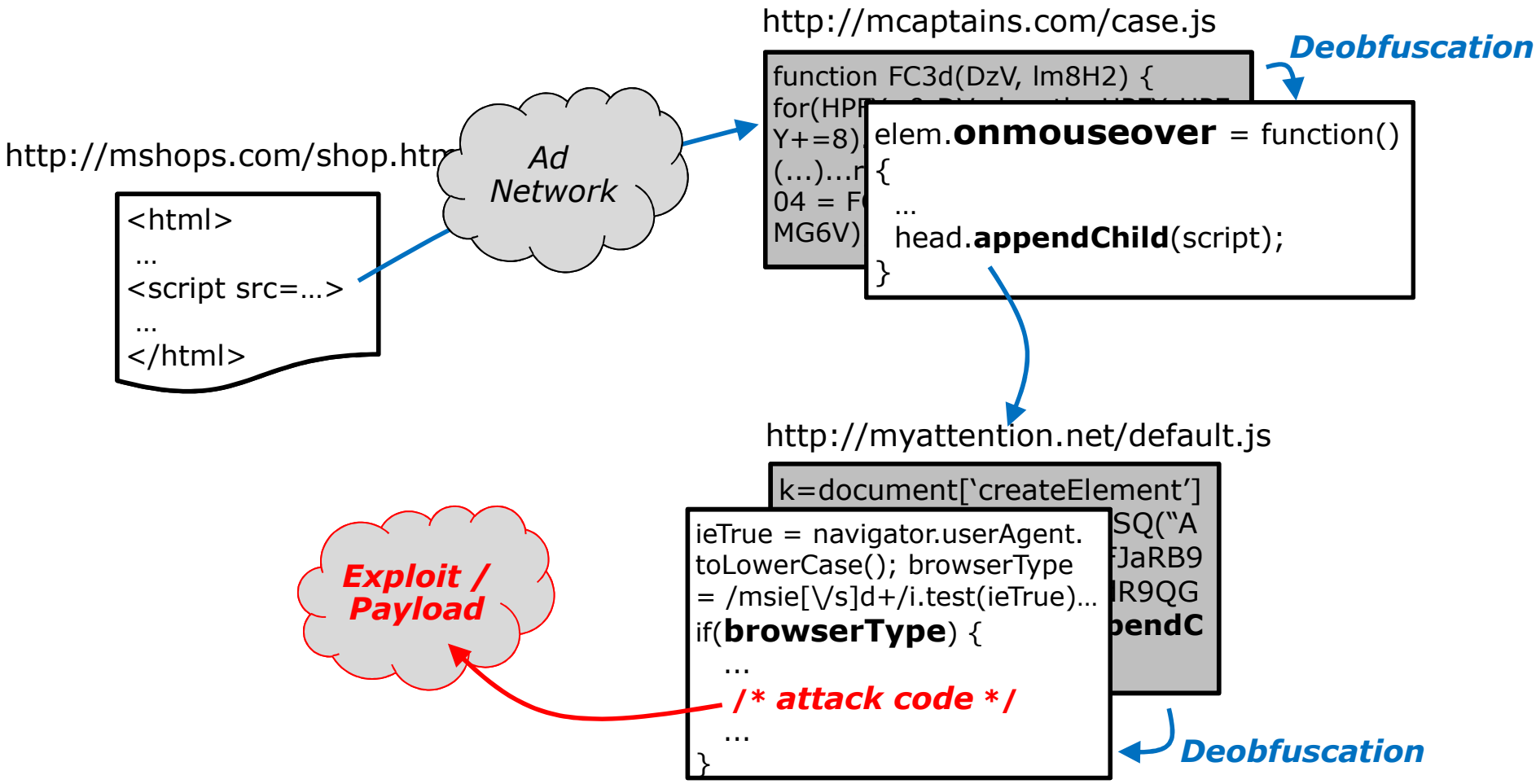
[2]IBM T.J. Watson Research Center, USA

PURDUE UNIVERSITY IBM

JavaScript Malware



Malware Example



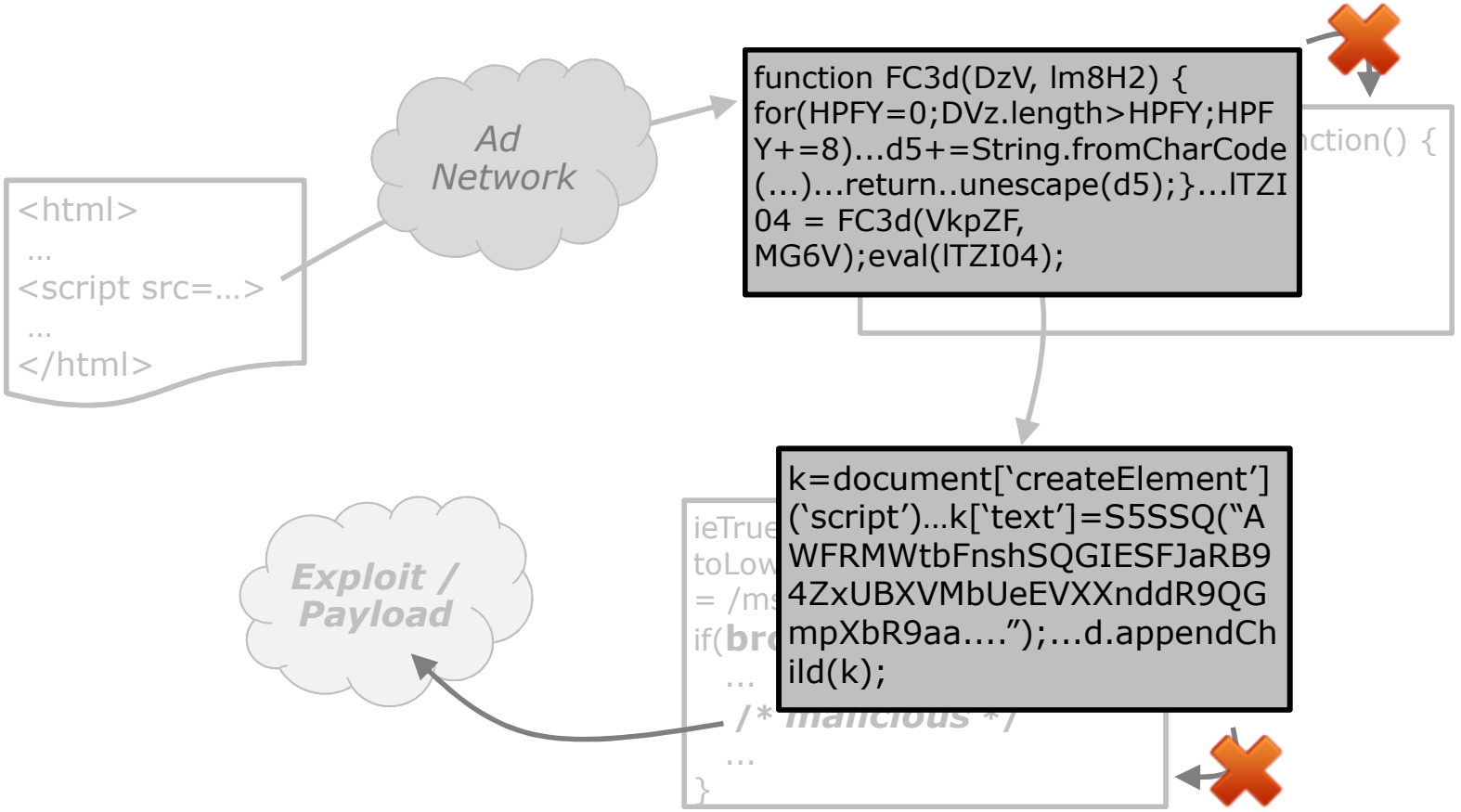
Malware Analysis

- Static analysis
  - Zozzle (Usenix security ’11)
- Dynamic analysis
  - JSAND (WWW’10), Nozzle (Usenix security ‘09)
- Symbolic analysis
  - Jalangi (FSE’13), Rozzle (Oakland ’12)

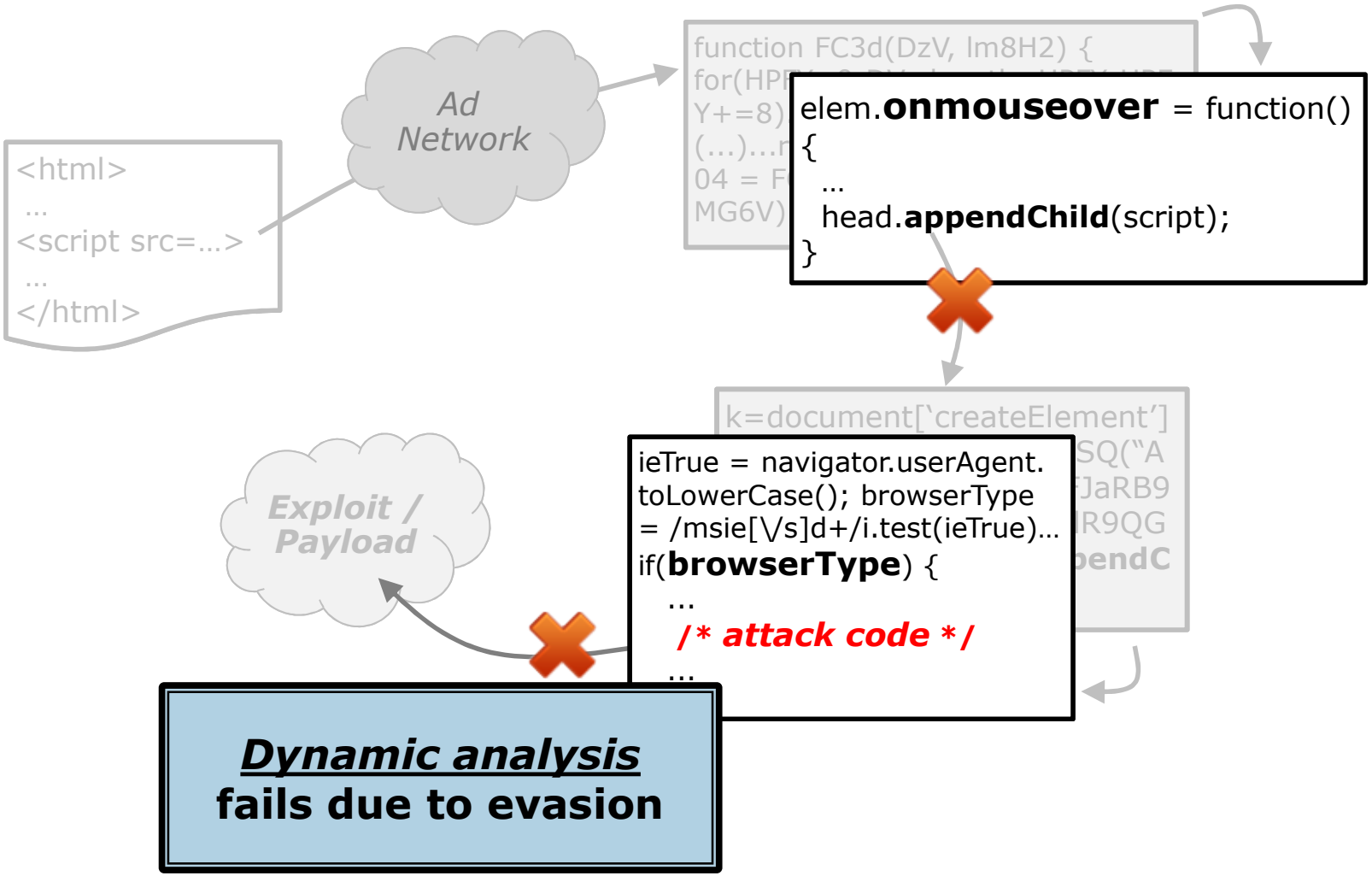
	Coverage	Evasion	Obfuscation	Scalability	Precision
Static analysis	✓	✓	✗	✓	✗
Dynamic analysis	✗	✗	✓	✓	✓
Symbolic analysis	✓	✓	✗	✗	✓

Traditional Malware Analysis

Static and symbolic analysis
fail to deobfuscate



Traditional Malware Analysis



J-Force : Malware Analysis Engine

- Forced execution engine on JavaScript
  - J-Force **explores all execution paths** by flipping the outcome of branch predicates
  - J-Force addresses technical challenges to **avoid crashes** during multiple execution
- Handling event handlers
  - **Force to execute handler code** regardless of event condition
  - Fixed small time value for timer events
- Handling dynamic code generation
  - **Admit all code injections** found along with multiple paths
  - E.g., eval(), <script> injection

J-Force Execution Model

- Per-script path exploration

Execution #1

```
<script>
  btn = document.createElement("button");
  btn.id = "mybutton";
  if (cond) {
    btn.innerHTML = "Remove";
  } else {
    btn.innerHTML = "Skip";
  }
  document.body.appendChild(btn);
</script>
```

Taken

```
...
<script>
  x = document.getElementById("mybutton");
  ...
</script>
```

J-Force Execution Model

- Per-script path exploration

Execution #2

```
<script>
btn = document.createElement("button");
btn.id = "mybutton";
if (cond) {
    btn.innerHTML = "Remove";
} else {
    btn.innerHTML = "Skip";
}
document.body.appendChild(btn);
</script>
```

Not-taken

```
...
<script>
  x = document.getElementById("mybutton");
  ...
</script>
```

J-Force Execution Model

- Handling inter-block dependences

```
<script>
   btn = document.createElement("button");
   btn.id = "mybutton";
   if (cond) {
      btn.innerHTML = "Remove";
   } else {
      btn.innerHTML = "Skip";
   }
   document.body.appendChild(btn);
</script>
```

```
...
<script>
   x = document.getElementById("mybutton");
   ...
</script>
```

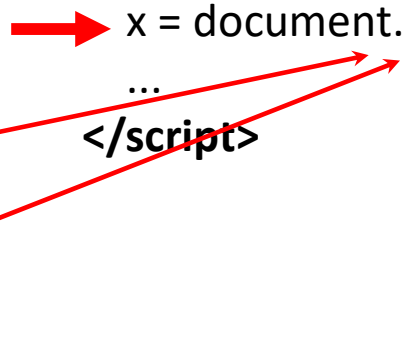
J-Force Execution Model

- Handling inter-block dependences

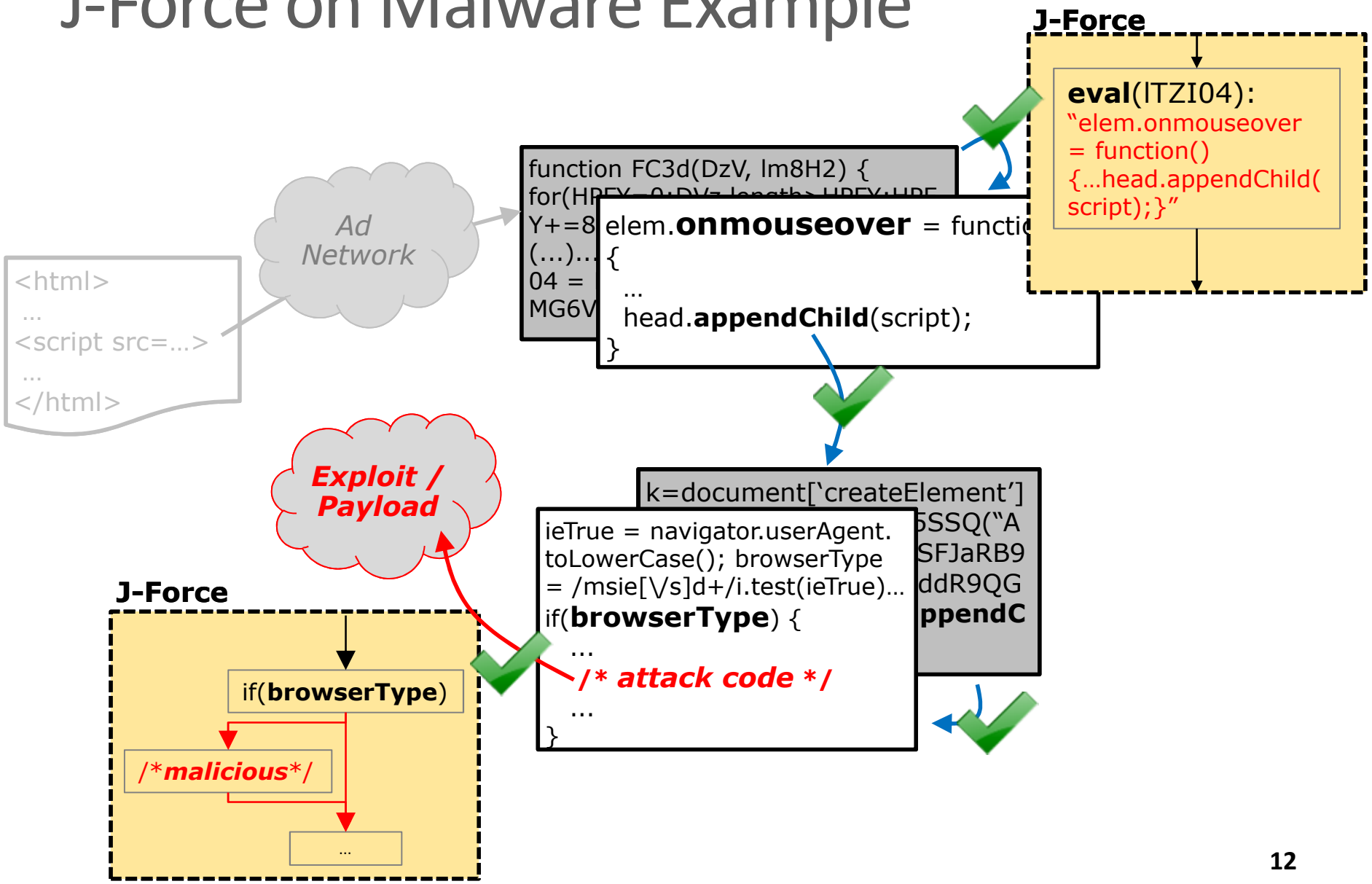
```
<script>
  btn = document.createElement("button");
  btn.id = "mybutton";
  if (cond) {
    btn.innerHTML = "Remove";
  } else {
    btn.innerHTML = "Skip";
  }
  document.body.appendChild(btn);
</script>
```

Execution #3

```
...
<script>
  x = document.getElementById("mybutton");
  ...
</script>
```



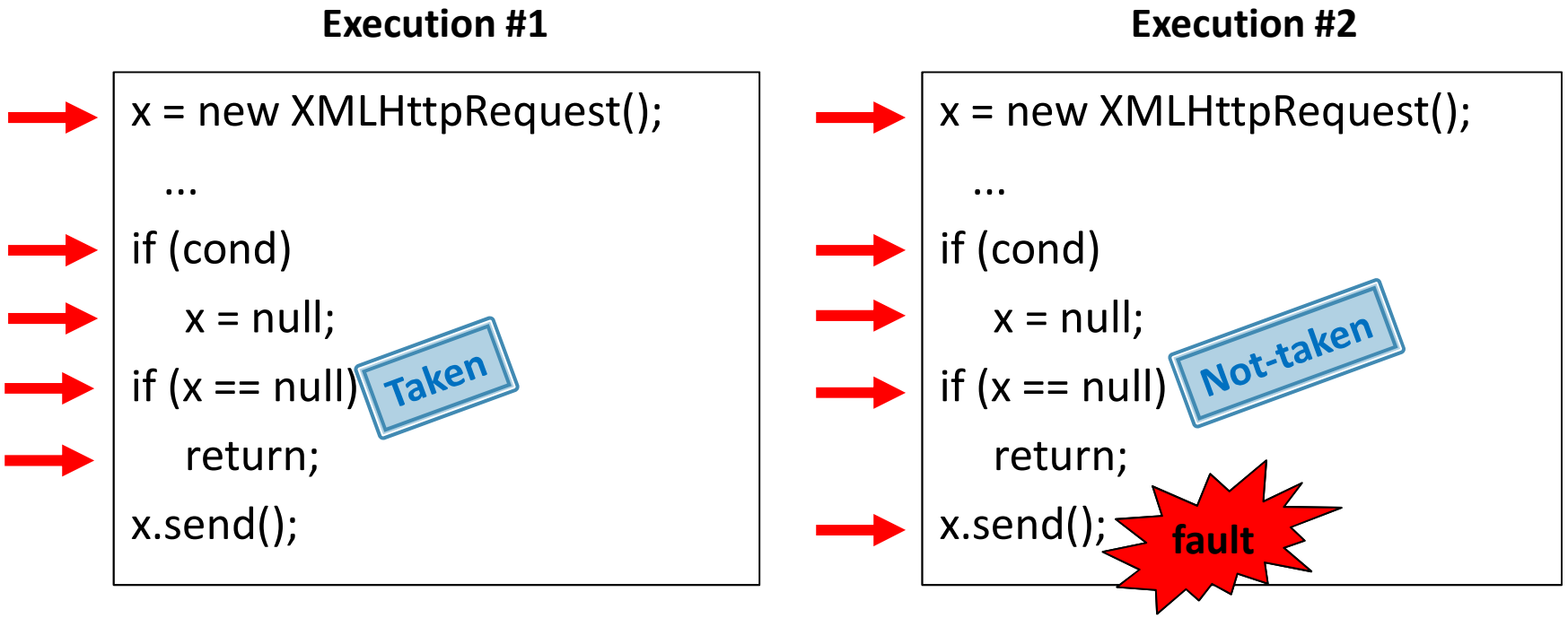
J-Force on Malware Example



Crash Free Execution

- Handling missing object/DOM
  - Keep track of missing object/DOM
  - Put them at the right place
- Handling exception
  - Exception triggered by legacy APIs (e.g., *attachEvent*)
  - Place top-level handlers to handle uncaught exceptions
- Page redirection
  - Load the target page in a separate frame
  - Each frame is independent to each other

Handling Missing Object



Handling Missing Object

Execution #2

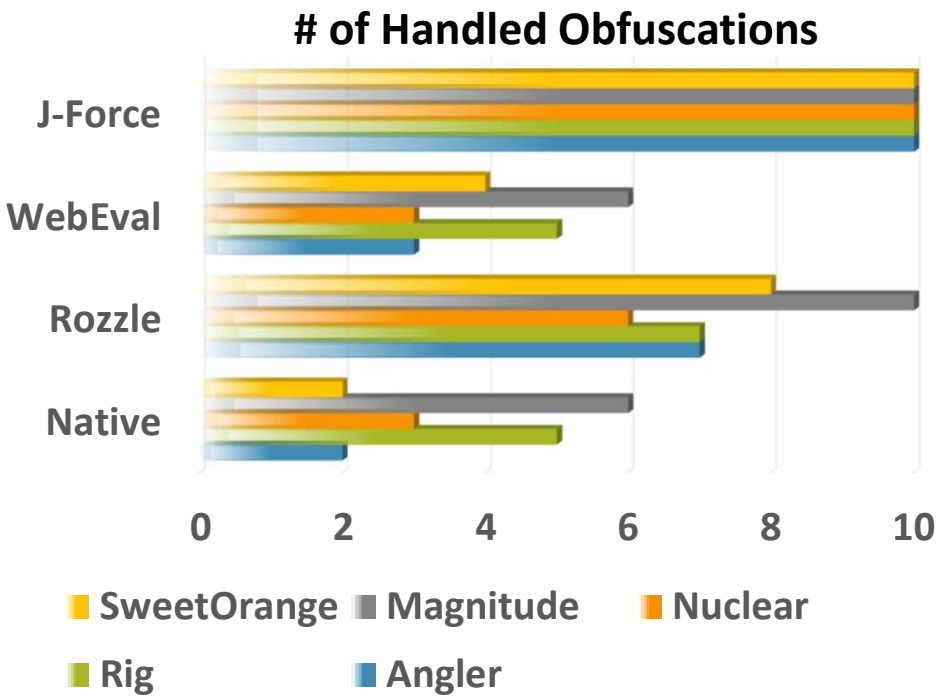
```
1. x = new XMLHttpRequest();   // -> Def1
2.     ...
3. if (cond)
4.    x = null;    // -> Def2
5. if (x == null)
6.    return;
7. x.send();        // <- ( Def1 | Def2 )
```

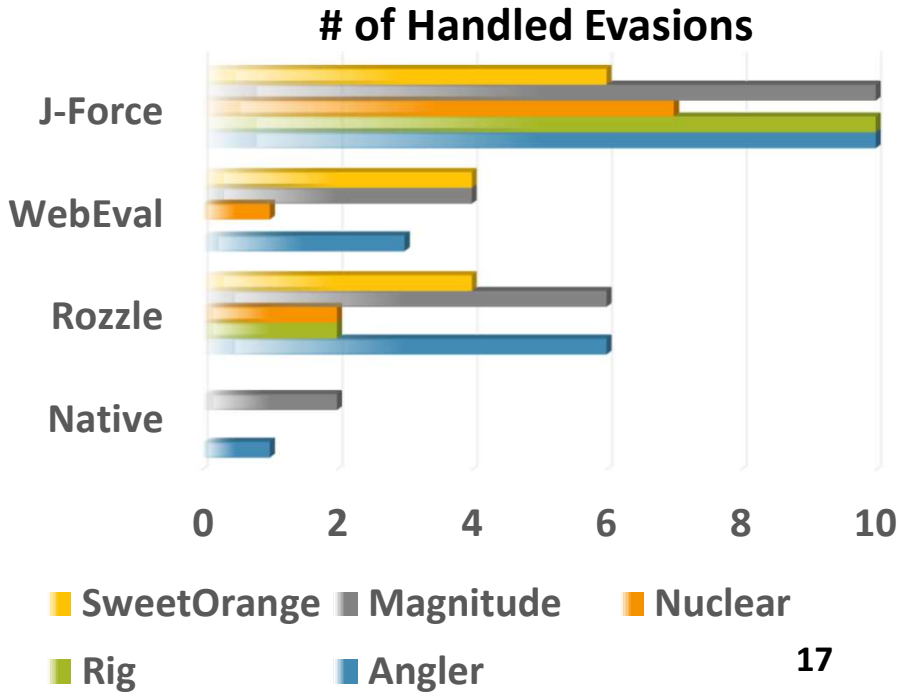
Evaluation

- Implemented on WebKit-r171233 with GTK+ port
- Effectiveness
  - Exploit Kit
  - Chrome extensions
- Efficiency
  - Performance overhead
  - Code coverage

Experiment on Exploit Kit

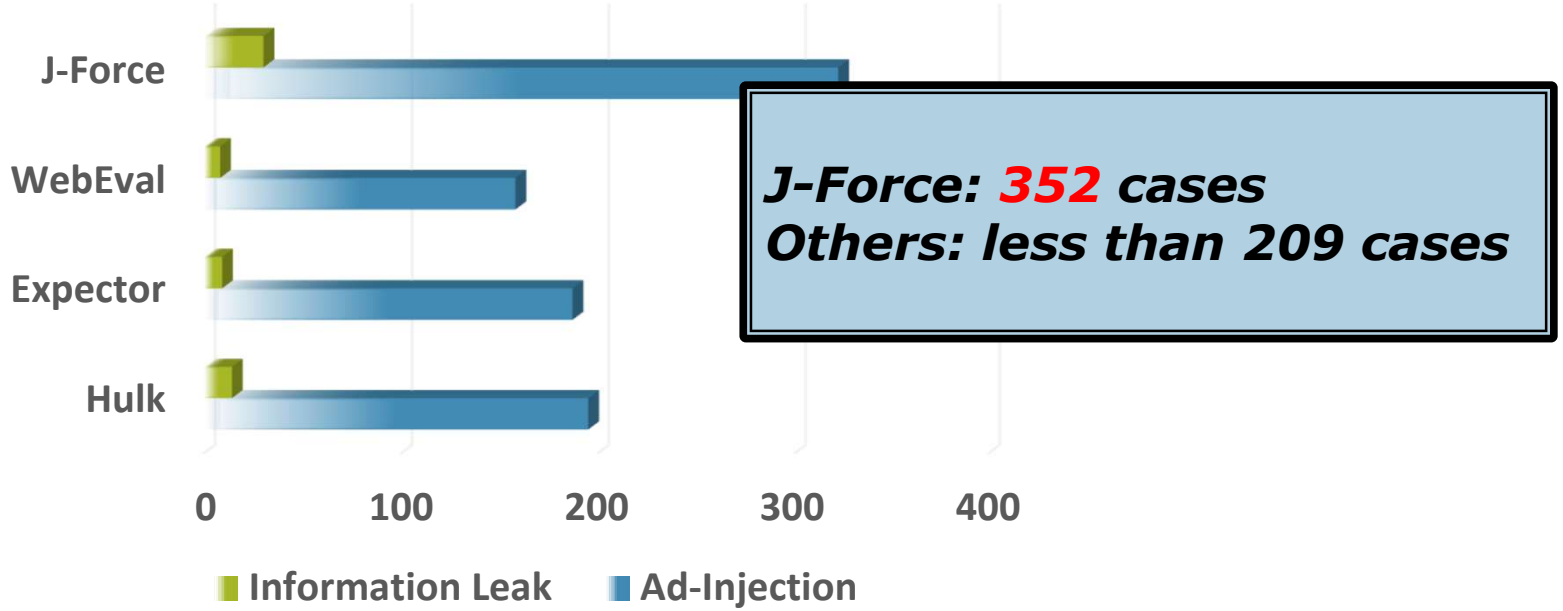
- 50 exploit kit samples
  - http://malware-traffic-analysis.net/index.html
  - 5 Exploit kit types (each one has 10 samples)
- 4 general steps
  - **Obfuscation, Evasion**, Exploiting vulnerabilities, Payload delivery





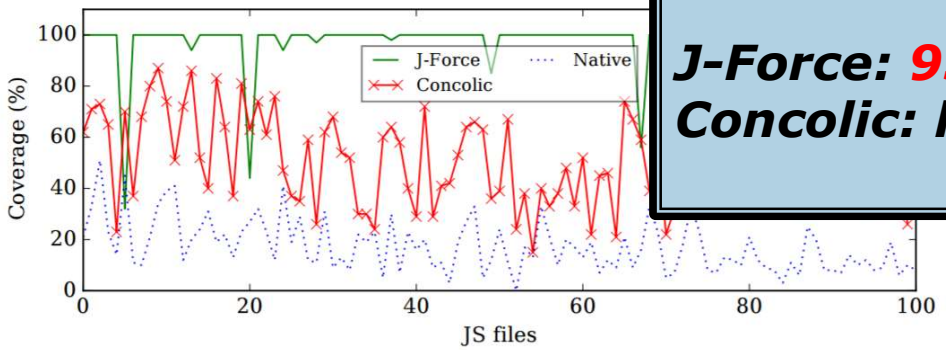
Experiment on Chrome Extensions

- Crawled 12,123 extensions from Chrome Web Store
- Simulated Chrome specific APIs
- Two suspicious behaviors
  - Information leak
  - Ad-injection



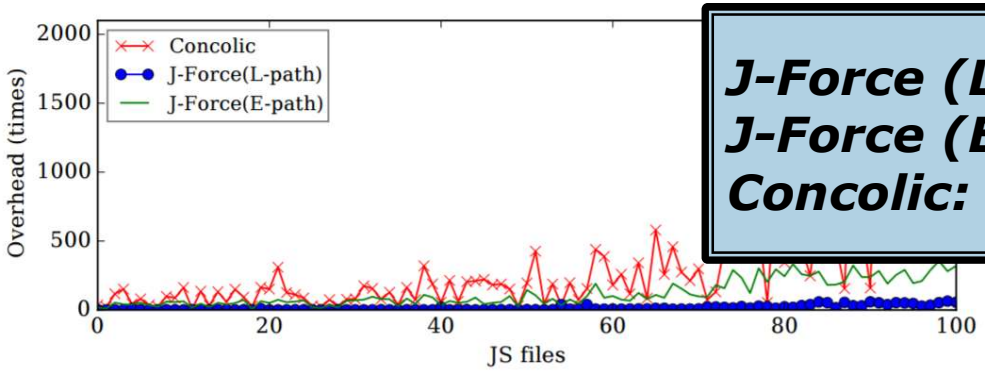
Efficiency

- Extracted 100 JavaScript samples from Alexa domain
- Code Coverage



J-Force: 95% coverage
Concolic: less than 70%

- Performance Overhead



J-Force (L-path): 2-8 times
J-Force (E-path): 2-300 times
Concolic: 10-10,000 times

Conclusion

- J-Force is a forced execution engine that explores all possible paths to expose hidden malware behaviors.
- J-Force addresses technical challenges to avoid crash during continuous path exploration.
- We validate the efficacy of J-Force through an extensive set of experiments on real-world examples.

Q & A

- Thank you for listening!