Code Inconsistencies

→ **Large** software  →  ⬆ Number of developers

- Different implementation of the same/similar functionalities
- Ex. We found null dereference bugs in OpenSSL because of the inconsistent ways that developers handle null check

→ Inconsistent bug patch

- Bug fix is applied *only* to where the bug was *originally* discovered
- Ex. We found similar missing check bugs to a bug in LibTIFF that had been patched 4 years ago
Existing Solutions

Identifying specific types of coding inconsistencies:

- **Seminal Work:** Bugs as Deviant Behavior [NSDI2001]
- **APIsan, AntMiner, NAR-Miner:** Detect API misuses.
- **Crix, Chucky, LRSan:** Detect missing check inconsistencies

Two limitations:

1. They cannot be easily extended to detect inconsistencies in an inconsistency type-agnostic fashion
2. The majority voting-based approach cannot detect one-to-one inconsistencies
Our Solution (FICS)

- FICS is not specific to one or a few types of inconsistencies or bugs
- FICS captures one-to-one inconsistencies
What challenges do we solve?

1. How to finding proper code granularities?
   a. Intra-procedural granularity
      i. Security-related bugs and patches are often regional or contained in a sub-function scope
   b. Data Dependence Graph
      i. Are usually enough to capture the root cause of a wide range of bugs.

2. How to make the approach scalable?
   a. Coarse-grained graph embedding
      i. Efficient first step clustering (coarse-grained)
1. Get compilation database
2. Data Dependence Graph (DDG) extraction
3. Construct Extraction (Per variable & Per variable Per Basic Block)
4. Abstraction
Construct definition & example

- Traverse the DDG until all subsequent nodes are covered or the Construct max-depth is reached.
- Any variable used in a function can be selected as the root variable for extracting a Construct.

```c
int data=10;
int i;
int buffer[10] = {0};
if (data >= 0)
{
    buffer[data] = 1;
}
```

(a) C code example. (b) Data Dependence Graph of ‘data’ Variable. (c) Data Dependence Graph of ‘data’ Variable for the first basic block.
FICS Design (B. Finding Similar constructs)

First-step clustering

5. Bag-of-words nodes (Ignore edges)
6. Cosine similarity between graph embeddings
7. Clustering
Similarity between Bag-of-Nodes embedding

Cosine similarity between the bag-of-nodes embeddings of the correct and the buggy (inconsistent) Constructs.

```
int data = 10; int i;
int buffer[10] = { 0 };
if (data >= 0 && data < 10) {
    buffer[data] = 1;
}
for(i = 0; i < 10; i++)
    printfIntLine(buffer[i]);
```

<table>
<thead>
<tr>
<th></th>
<th>alloc...</th>
<th>getelem...</th>
<th>icmp sge...</th>
<th>icmp slt...</th>
<th>load...</th>
<th>sext...</th>
<th>br...</th>
<th>call...</th>
<th>store...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Buggy</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

**Cosine Similarity:** \(0.96609\) (i.e., > 95%)

The buggy construct has:

- One **load** and one **br** LLVM instruction less
- No **icmp** LLVM instruction
FICS Design (C. Finding inconsistent constructs)

First-step clustering

8. Graph2vec (Embed the graph by random walk and Skipgram model)
6. Cosine similarity between graph embeddings
7. Clustering
FICS Design (D. Deviation analysis & Filtering)

- **Deviation Analysis**
  - **Red** refers to LLVM instruction
  - **Orange** refers to function call
  - ‘*’ means Kleene Star

- **Filtering**
  - removes the inconsistencies that are redundant or likely false.

<table>
<thead>
<tr>
<th>Inconsistency Type</th>
<th>Bug Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deviation Check</td>
<td>NULL Pointer Dereference, Undefined Behavior</td>
</tr>
<tr>
<td>icmp Node</td>
<td>Buffer Errors, Integer Overflow</td>
</tr>
<tr>
<td>Memory Handling</td>
<td>Resource Leak, Double Free</td>
</tr>
<tr>
<td>´<em>free</em>´, ´<em>close</em>´ Nodes</td>
<td>Information Leak</td>
</tr>
<tr>
<td>´<em>bzero</em>´, ´<em>clear</em>´ Nodes</td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>Bad Casting</td>
</tr>
<tr>
<td>´trunc, bitcast´ Nodes</td>
<td></td>
</tr>
<tr>
<td>Order</td>
<td>Wrong Order of Operations</td>
</tr>
<tr>
<td>Edge</td>
<td>Double Free, Information Leak</td>
</tr>
<tr>
<td>Initialization</td>
<td></td>
</tr>
<tr>
<td>´store, memset´ Nodes</td>
<td></td>
</tr>
</tbody>
</table>
Evaluation

- Evaluated on five real codebases
- Found 218 valid inconsistencies
  - 123 Potential bugs (22 confirmed so far)
  - 95 Code smells

<table>
<thead>
<tr>
<th>Name</th>
<th>Total</th>
<th># Reported inconsistencies</th>
<th>Valid Cases</th>
<th>Code Smells</th>
<th>Potential Bugs</th>
<th>Confirmed Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Check + Call (Sum)</td>
<td>After Filtering</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QEMU</td>
<td>12,320</td>
<td>3,907 + 3,170 (7,077)</td>
<td>1,206</td>
<td>79</td>
<td>26</td>
<td>53</td>
</tr>
<tr>
<td>OpenSSL</td>
<td>2,419</td>
<td>1,158 + 347 (1,505)</td>
<td>310</td>
<td>59</td>
<td>24</td>
<td>35</td>
</tr>
<tr>
<td>wolfSSL</td>
<td>586</td>
<td>296 + 124 (420)</td>
<td>91</td>
<td>23</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>OpenSSH</td>
<td>1,063</td>
<td>509 + 208 (717)</td>
<td>121</td>
<td>29</td>
<td>18</td>
<td>11</td>
</tr>
<tr>
<td>LibTIFF</td>
<td>925</td>
<td>390 + 156 (546)</td>
<td>93</td>
<td>28</td>
<td>9</td>
<td>19</td>
</tr>
<tr>
<td>Total</td>
<td>17,313</td>
<td>6,260 + 4,005 (10,265)</td>
<td>1,821</td>
<td>218</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 95 Code smells
- 123 Potential bugs (22 confirmed so far)
- 22 Confirmed bugs
Comparison

- They focus on specific class of inconsistencies or bugs
- They cannot detect one-to-one inconsistencies

<table>
<thead>
<tr>
<th></th>
<th>FICS #Rep</th>
<th>#B</th>
<th>APIsan #Rep</th>
<th>#B</th>
<th>LRSan #Rep</th>
<th>#B</th>
<th>Crix #Rep</th>
<th>#B</th>
</tr>
</thead>
<tbody>
<tr>
<td>QEMU</td>
<td>1,206</td>
<td>4</td>
<td>5,805</td>
<td>0</td>
<td>129</td>
<td>0</td>
<td>98</td>
<td>0</td>
</tr>
<tr>
<td>OpenSSL</td>
<td>310</td>
<td>9</td>
<td>7,874</td>
<td>0</td>
<td>30</td>
<td>0</td>
<td>54</td>
<td>1</td>
</tr>
<tr>
<td>wolfSSL</td>
<td>91</td>
<td>3</td>
<td>1,049</td>
<td>1</td>
<td>62</td>
<td>0</td>
<td>62</td>
<td>0</td>
</tr>
<tr>
<td>OpenSSH</td>
<td>121</td>
<td>1</td>
<td>2,740</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>LibTIFF</td>
<td>93</td>
<td>5</td>
<td>645</td>
<td>3</td>
<td>12</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 7: Comparison between FICS, APIsan, LRSan, and Crix on bug detection capability. FICS outperforms its competitors while not reporting too many potential cases. #Rep: Number of reports, #B: Number of bugs.
Summary

- FICS is the first inconsistency-generic, ML-based bug detection system
- FICS does not require external datasets for training nor is limited to certain types of bugs.
- FICS found 22 unknown bugs in 5 popular and well-tested projects

- Limitations
  - If the size of the codebase is too small, the system is less likely to find bugs
  - Certain bugs (e.g., one-liners) may be too small to be captured by FICS
  - Our research prototype currently neither support C++ nor extremely large codebases (e.g., Linux)
Thank you! Q&A

Contact: mansosec@gmail.com

Code: [https://github.com/RiS3-Lab/FICS](https://github.com/RiS3-Lab/FICS)