Schnauzer: Scalable Profiling for Likely Security Bug Sites

William Arthur, Biruk Mammo, Ricardo Rodriguez, Todd Austin, Valeria Bertacco
Goal of this work

MAKE SOFTWARE MORE SECURE
Goal of this work

MAKE SOFTWARE MORE SECURE

Leveraging Limited Test Resources
Goal of this work

MAKE **SOFTWARE** MORE **SECURE**

Leveraging **Limited** Test Resources
Goal of this work

MAKE SOFTWARE MORE SECURE

Leveraging Limited Test Resources
Importance of Path

- Vast majority of security attacks are enabled by software bugs
Importance of Path

Vast majority of security attacks are enabled by software bugs

Often, hidden bugs only appear when sensitized by the proper path
Vast majority of security attacks are enabled by software bugs.

Often, hidden bugs only appear when sensitized by the proper path.
Vast majority of security attacks are enabled by software bugs.

- Often, hidden bugs only appear when sensitized by the proper path.
- Bugs escape Code/Branch coverage.

Attackers will seek out code paths not tested.
Path Test Complexity

- **Path Explosion**
  - Path space is exponential with length
  - Heavyweight test methods are slow

- **Path coverage remains beyond reach**

- **Attackers seek to discover untested paths**

- **Necessitates new approach to achieve path testing**
Dynamic Control Frontier
Dynamic Control Frontier

- Line of demarcation between dynamically seen paths of execution and those which are unseen
Dynamic Control Frontier

- Line of demarcation between dynamically seen paths of execution and those which are unseen
  - Frontier of path space explored by an application
Line of demarcation between dynamically seen paths of execution and those which are unseen
Line of demarcation between dynamically seen paths of execution and those which are unseen

\[<A, B, C, D>\]
Dynamic Control Frontier

Line of demarcation between dynamically seen paths of execution and those which are unseen

\[ <A, B, C, D> \]

\[ <A, B, C, E> \]
Line of demarcation between dynamically seen paths of execution and those which are unseen

\[ \text{DCF} = \{ \langle A, B, C, E \rangle, \ldots \} \]
Value of the DCF

Software Test Methodology:
- Focus on reliability
- Significant overlap in developer and user test

Attacker Methodology:
- Input permutations to deviate slightly from the expected, typical user execution

Dynamic Control Frontier:
- Intersection between heavily tested paths, and untested paths which are immediately reachable
Value of Distributed Analysis

A single user:

_profiles an instance_
Value of Distributed Analysis

A single user:
- Profiles an **instance**

A non-trivial population of users:
- Represents code paths not tested nor executed with any frequency by any user
Value of Distributed Analysis

 alunos

A single user:

- Profiles an instance

A non-trivial population of users:

- Represents code paths not tested nor executed with any frequency by any user
Distributed DCF Profiling

User base profiles application via sampling
Distributed DCF Profiling

User base profiles application via sampling
Distributed DCF Profiling

<User base profiles application via sampling>

Users ➔ DCF Paths ➔ Developer Analysis ➔ High-Confidence DCF Paths ➔ Developer Test
Distributed DCF Profiling

User base profiles application via sampling

Users → DCF Paths → Developer Analysis → High-Confidence DCF Paths → Developer Test

Test Coverage
Distributed DCF Profiling

User base profiles application via sampling

Users → DCF Paths → Global Path Filters → Developer Analysis → High-Confidence DCF Paths → Developer Test

Test Coverage

Global Path Filters
Experimental Evaluation

DynamoRIO-based dynamic path profiling

- Only instrument paths which are actively sampled
Experimental Evaluation

Data?

Path Tracking

Load

Body?

Buf++
Experimental Evaluation

- Data?
  - Path Tracking
  - Load
  - Body?
  - Buf++
  - DCF Hypothesis
Experimental Evaluation

Data?

Path Tracking

Load

Body?

DCF Hypothesis

Buf++
### Benchmark Applications

#### Popular, network-facing applications

<table>
<thead>
<tr>
<th>Application</th>
<th># Instructions Profiled</th>
<th># Potential Paths</th>
<th># DCF Paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>16,948,864,926</td>
<td>13,642,304</td>
<td>17,351</td>
</tr>
<tr>
<td>OpenSSL</td>
<td>5,014,034,838</td>
<td>23,221,696</td>
<td>10,086</td>
</tr>
<tr>
<td>tshark</td>
<td>684,000,546</td>
<td>38,467,136</td>
<td>178</td>
</tr>
<tr>
<td>Python</td>
<td>656,068,272</td>
<td>12,175,712</td>
<td>35,026</td>
</tr>
<tr>
<td>Tor</td>
<td>118,310,256</td>
<td>1,191,280</td>
<td>10,639</td>
</tr>
<tr>
<td>InspIRCd</td>
<td>46,246,206</td>
<td>11,165,696</td>
<td>3,950</td>
</tr>
<tr>
<td>Pidgin</td>
<td>4,762,914</td>
<td>6,833,360</td>
<td>3,641</td>
</tr>
</tbody>
</table>
Profiling Overheads

DynamoRIO w/DCF Profiling
Profiling Overheads

Overhead(X) to Native

- Tor: 2.34
- tshark: 1.11
- SQLite: 2.26
- Pidgin: 2.79
- Python: 6.85
- InspIRCd: 2.97
- OpenSSL: 1.41
- AVG.: 2.45

Colors:
- DynamoRIO w/Null Client
- DynamoRIO w/DCF Profiling
Challenged Schnauzer to find known security bugs
- Known bugs have precise code location

106 Million+ Potential Length-n Paths
80,000 DCF Paths

14 Security Bugs
{ Buffer Overflow, Integer Underflow, DoS, Format String, Heap Overflow }

DCF analysis would have given opportunity to determine paths for these bugs before they were exploited
Conclusions & Future Directions

Efficient, user-enabled DCF profiling can expand test for software security

Identify code paths harboring bugs more likely to be exploited

- Before they are exploited
- Making software more secure

Going Forward:

- More efficient user profiling
- Deployment of DCF for substantial application
- Integration with state-of-art automated test
Thank You
Supplemental Material
Profiling Scalability

DCF Paths -- SQLite Fuzz Test

Number of Unique Paths (Thousands) vs. Instructions (Trillions)
Path Length Scalability

![Graph showing path length scalability with different slowdown values for path lengths ranging from 2 to 2048. The graph compares the performance of DynamoRIO with a null client versus the native performance.]
Path Length : Vulnerability

DCF Paths (Thousands)

Path Length

NVD Vulnerabilities

DCF Paths

Vulnerabilities Discovered
Concurrent Hypotheses

![Graph showing the percentage of DCF discovered against instructions (billions) for different hypothesis counts. The graph compares single hypothesis, 8 hypotheses, and 32 hypotheses.]
\[ DCF(P) = \{ p_i, p_j, \ldots, p_m \} \]
\[ p_i = <bb_1, bb_2, \ldots, bb_{n-1}, bb_n> \]
\[ | <bb_1, bb_2, \ldots, bb_{n-1}> \in EX(P) \]
\[ \land <bb_1, \ldots, bb_{n-1}, bb_n> \notin EX(P) \]
\[ EX(P) = \{ \ldots \text{all paths executed} \ldots \} \]