Attackers can enable a legitimate program to execute an arbitrary and malicious piece of code by hijacking the control flow of the programs. **Control Flow Integrity** (CFI) is the defense mechanism used to prevent these kind of exploits that compromise the program by enforcing the restriction on the targets of control transfer in a program.

An accurate **Control Flow Graph** (CFG) should be computed to ensure the efficiency of the CFI enforcement.

- In general, the problem of valid target computation is an un-decidable problem, various approximate approaches to produce coarse grained and fine grained results have been used to enforce CFI.
- It is found that most of the coarse grained approach are still susceptible to code reuse attacks like ROP, while the fine grained approach tend to be performance intensive.

We propose an approach to produce fine grained forward edge control flow policy by enforcing some restrictions on the code pointers in the programs, this enables us to compute a precise policy without performance intensive processing. We observe that these restrictions are obeyed in most cases and propose a remedy to handle the programs which violate the restrictions.

### Approach

We compute the legal targets for an indirect invocation by performing static taint propagation without the extensive points to analysis.

We apply the following restrictions on the usage of function pointers in the code:

- **[A1]:** The only allowed operation on function pointer are assignment and dereferencing.
- **[A2]:** There exists no data pointer to a function pointer.

The above restrictions have been found to hold largely for kernel software like FreeBSD and MINIX microkernel [1].

We test whether this approach can be applied effectively for user-space programs, and found that a majority of the programs comply with the restrictions.

We observed that to ensure soundness in user-space programs we need to add another restriction for isolating data and code pointers.

- **[A3]:** Function pointers may not be cast to or cast from a data pointer.

### Inter-module Analysis

Another challenge for user space programs is the policy computation across dependent and dynamically loaded modules.

We iteratively process the dependent modules for code pointer information exchange and compute the complete policy for the application.

### Results

The following table shows the comparison between targets computed using signature matching approach and our taint based approach.

<table>
<thead>
<tr>
<th>Method</th>
<th>Binary Files Analyzed</th>
<th>Resolved Indirect Calls</th>
<th>Total Targets</th>
<th>Average Targets per call</th>
</tr>
</thead>
<tbody>
<tr>
<td>Taint Based</td>
<td>455</td>
<td>19,532</td>
<td>81,352</td>
<td>4.16</td>
</tr>
<tr>
<td>Signature Based</td>
<td>455</td>
<td>19,532</td>
<td>388,226</td>
<td>19.87</td>
</tr>
</tbody>
</table>

We also evaluate the gain in precision due to inter-module analysis with and without statically linking.

<table>
<thead>
<tr>
<th>Binaris</th>
<th>Indirect Calls</th>
<th>Inter-mod Targets</th>
<th>Merged Targets</th>
<th>Percentage Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>170</td>
<td>11,766</td>
<td>42,577</td>
<td>43,882</td>
<td>3.1</td>
</tr>
</tbody>
</table>

### Related Work