Detecting Critical Control Flow with Clang Static Analyzer

Simon Cook
Ed Jones
Critical Control Flow

- *Critical Variables* hold information that must be kept secret.
  - e.g. cryptographic keys
- Having these affect control flow can lead to their contents being leaked, via:
  - Differential timing analysis
  - Differential power analysis
Example I – Critical Variable Use

```c
int MY_SECRET __attribute__((critical));
MY_SECRET = 2;
if (MY_SECRET < 5)
    external_func();
```
Example 1 – Critical Variable Use

```c
int MY_SECRET __attribute__((critical));
MY_SECRET = 2;
if (MY_SECRET < 5)
    external_func();
```

Variable explicitly marked critical
Example I – Critical Variable Use

```c
int MY_SECRET __attribute__((critical));
MY_SECRET = 2;
if (MY_SECRET < 5)
    external_func();
```

Branch taken based on critical value.

Variable explicitly marked critical
Example II – Critical Variable Inheritance

```c
int MY_SECRET __attribute__((critical));
MY_SECRET = 2;
int OTHER = MY_SECRET * 2;
if (OTHER < 5)
    external_func();
```
Example II – Critical Variable Inheritance

```c
int MY_SECRET __attribute__((critical));
MY_SECRET = 2;
int OTHER = MY_SECRET * 2;
if (OTHER < 5)
    external_func();
```

**OTHER** becomes critical here.
Example II – Critical Variable Inheritance

```
int MY_SECRET __attribute__((critical));
MY_SECRET = 2;
int OTHER = MY_SECRET * 2;
if (OTHER < 5)
    external_func();
```

OTHER becomes critical here.

Branch taken based on critical value.
Example III – Passing Critical Variables

```c
volatile bool z;
int bar(int x) {
    if (z)
    return x*2;
    return 0;
}

void foo() {
    int MY_SECRET __attribute__((critical));
    int OTHER = bar(MY_SECRET);
    if (OTHER < 5)
    external_func();
}
```
volatile bool z;
int bar(int x) {
    if (z)
        return x*2;
    return 0;
}

void foo() {
    int MY_SECRET __attribute__((critical));
    int OTHER = bar(MY_SECRET);
    if (OTHER < 5)
        external_func();
}
Example III – Passing Critical Variables

```c
volatile bool z;
int bar(int x) {
    if (z)
        return x*2;
    return 0;
}

void foo() {
    int MY_SECRET __attribute__((critical));
    int OTHER = bar(MY_SECRET);
    if (OTHER < 5)
        external_func();
}
```

Pass critical variable into bar

x is critical
Example III – Passing Critical Variables

```c
volatile bool z;
int bar(int x) {
  if (z)
    return x*2;
  return 0;
}

void foo() {
  int MY_SECRET __attribute__((critical));
  int OTHER = bar(MY_SECRET);
  if (OTHER < 5)
    external_func();
}
```

Pass critical variable into `bar`

Return value is critical if `z` is true
Example III – Passing Critical Variables

```c
volatile bool z;
int bar(int x) {
    if (z)
        return x*2;
    return 0;
}

void foo() {
    int MY_SECRET __attribute__((critical));
    int OTHER = bar(MY_SECRET);
    if (OTHER < 5)
        external_func();
}
```

- Return value is critical if `z` is true.
- Pass critical variable into `bar`.
- Branch possibly taken based on critical variable.
## Approaches to Detecting Control Flow

<table>
<thead>
<tr>
<th></th>
<th>Advantage</th>
<th>Disadvantage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Compiler Warnings</strong></td>
<td>• Users are familiar with compiler warnings</td>
<td>• Hard to run expensive and/or complex checks</td>
</tr>
<tr>
<td></td>
<td>• Tests run as part of default compilation path</td>
<td>• Lack of path sensitivity</td>
</tr>
<tr>
<td></td>
<td>• Bugs found at compile time</td>
<td></td>
</tr>
<tr>
<td><strong>Static Analysis</strong></td>
<td>• Allows path sensitivity</td>
<td>• Requires use of unfamiliar tool</td>
</tr>
<tr>
<td></td>
<td>• Allows for more expensive and/or complex checks</td>
<td>• Results are reported separately from normal compiler warnings</td>
</tr>
<tr>
<td><strong>Sanitizers</strong></td>
<td>• Allows path sensitivity</td>
<td>• Requires run-time instrumentation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Reduction in run-time performance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Errors only found at runtime</td>
</tr>
</tbody>
</table>
Clang Static Analysis Tool

• Included as part of clang, and triggered via scan-build command
• Generates reports that can be viewed via scan-view

• Analyzer does a path sensitive analysis through multiple checkers simultaneously, building up a graph of the state of execution over time.
• Our aim is to add a checker alongside these.
Taint Analysis

• Technique implemented in GenericTaintChecker in Static Analyzer.

• Tracks ‘taint’, which are values based on user input.
  • Eg Return value from getchar

• Checks that tainted values aren’t use in syscalls or as ‘size’ args to strnlen etc

• Checking criticality is similar to taint checking.
  • __attribute__((critical)) is a source of taint
  • Presence of taint checked on conditions, etc
Custom State Tracking

- Define custom state to track whether a variable is critical, or has had critical trait cast away.
- On variable assignment, search expression for source of critical trait, if so mark variable as critical.
- Do the same for expressions, if expression is critical, mark as bug.
Current State

• Still a work in progress. Currently investigating:
  • Verifying our expression tagging works accurately
  • Generating accurate and useful bug traces.
  • Checking return values and arguments work as expected.

• Our aim is to have a patch for submission upstream once we have verified these corner cases.