Detecting Critical Control Flow with Clang Static Analyzer

Simon Cook

Ed Jones

Critical Control Flow

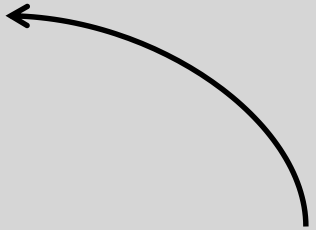
- *Critical Variables* hold information that must be kept secret.
  - e.g cryptographic keys
- Having these affect control flow can lead to their contents being leaked, via:
  - Differential timing analysis
  - Differential power analysis

Example I – Critical Variable Use

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

Example I – Critical Variable Use

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



Variable explicitly marked critical

Example I – Critical Variable Use

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

Variable explicitly marked critical

Branch taken based on critical value.

EMBECOSM®

Example II – Critical Variable Inheritance

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

EMBECOSM®

Example II – Critical Variable Inheritance

OTHER becomes critical here.

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

Example II – Critical Variable Inheritance

OTHER becomes critical here.

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

Branch taken based on critical value.

Example III – Passing Critical Variables

```
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();
}
```

EMBECOSM®

Example III – Passing Critical Variables

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

Pass critical variable into **bar**

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

EMBECOSM®

Example III – Passing Critical Variables

x is critical

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

Pass critical variable into **bar**

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

EMBECOSM®

Example III – Passing Critical Variables

x is critical

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

Return value is critical *if* **z** is true

Pass critical variable into **bar**

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

EMBECOSM®

Example III – Passing Critical Variables

x is critical

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

Return value is critical *if* **z is true**

Pass critical variable into **bar**

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

Branch possibly taken based on critical variable.

EMBECOSM®

Approaches to Detecting Control Flow

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

Clang Static Analysis Tool

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

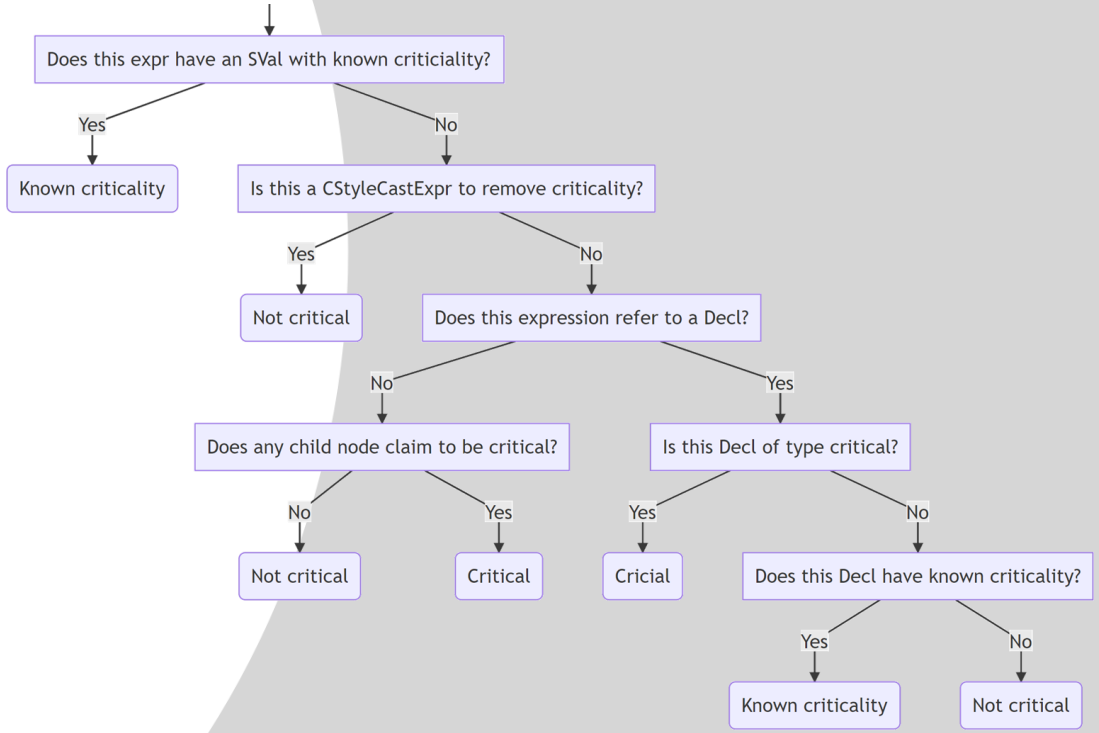
- Analyzer does a path sensitive analysis through multiple checkers simultaneously, building up a graph of the state of execution over time.
- Our aim is to add a checker alongside these.

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.



EMBECOSM®

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.