A Static Analysis Approach to Mitigate SEUs

Ganesha & Sujit Kumar Chakrabarti

IIIT-Bangalore

ganesha@iiitb.ac.in & sujitkc@iiitb.ac.in

1. Introduction

- What are Single Event Upsets ?
- What is Hardening ?
- What is our Approach ?
- Variable bounds are explained in Figure 1

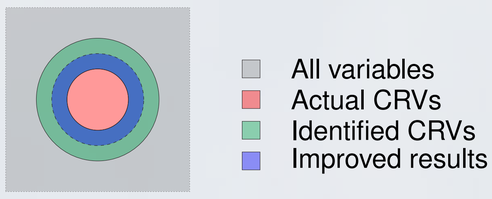


Figure 1: Bounds

2. Motivating Example

- In Figure 2 we explain code and control flow graph
- Correctness condition $output \leq 10$ and CRV (Conditionally Relevant Variables)
- Traditional static slicing fails to detect CRV
- Resort to formal verification to identify such scenarios

```
1  int f(int x, int y) {
2    int output = 4;
3    bool alarm = false;
4    int count = 0;
5    while(count < 7) {
6      if(x > 10) {
7        if(y == 1) {
8          output = 2;
9        }
10       else {
11         output = 1;
12       }
13     }
14     else {
15       output = output + 1;
16       alarm = true;
17     }
18     count++;
19   }
20   printf("%b", alarm);
21   return output;
22 }
```

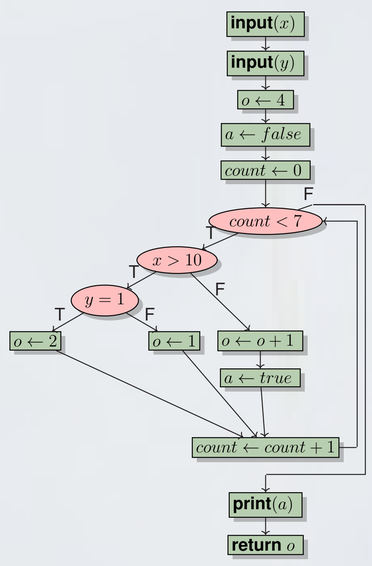


Figure 2: Motivating Example

3. Novel Approach

- Existing techniques to mitigate SEU
- Solutions are at hardware level
- Our method stands out as it addresses SEU mitigation at the controller algorithm level, rather than hardware, presenting a novel approach

4. Proposed Solution

4.1 Dependencies

- Data dependency and control dependency are explained in In Figure 3
- Understanding data flow and dependencies throughout the program is crucial for analyzing CRVs
- Formal verification techniques are then utilized to refine the bound on CRVs, enhancing the accuracy of the analysis

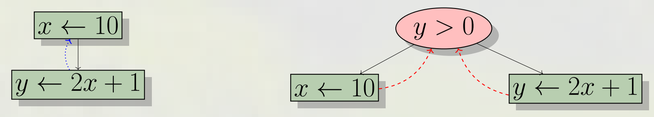
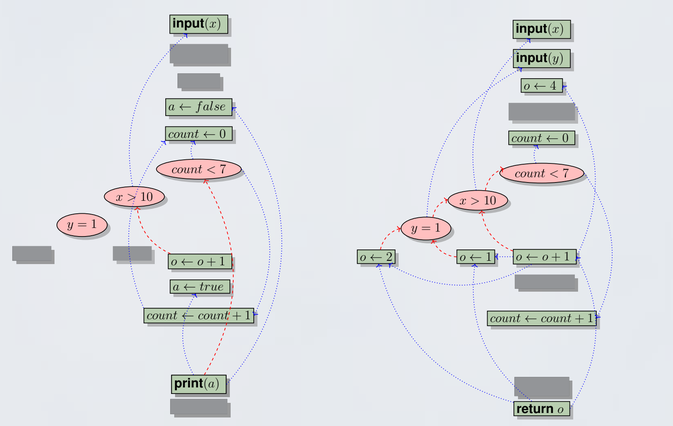


Figure 3: Dependency (Simple examples): (a) Data dependency; (b) Control dependency

4.2 Static Program Slicing

- Program slicing is a technique used to extract parts of the program related to a specific program location or variable of interest
- Figure 4 shows two backward slices of the same program, one with a slicing criterion $(a, \textbf{print}(a))$ and the other with $(o, \textbf{return}\ o)$ as the slicing criterion
- The initial stage of the strategy involves applying program slicing to identify CRVs, which may contain false positives



Static slice: **print**(a) Static slice: **return** o

Figure 4: Program slicing

4.3 Observation's

- Static slicing is sound, however often provides imprecise results
- Program slicing and formal verification are used to identify CRV
- Compiler can be guided to place conditionally relevant parts in the hardened parts of silicon
- These bounds will be tighter than the ones obtainable through traditional static analysis
- Hardening less silicon saves costs while maintaining safety

4.4 Program Instrumentation

```
1  int f(int x, int y) {
2    int output = 4;
3    bool alarm = false;
4    int count = 0;  ⇐
5    bool flag = false;
6    while(count < 7) {
7      if(x > 10) {
8        if(y == 1) {
9          output = 2;
10         flag = true;
11       }
12       else {
13         output = 1;
14         flag = true;  ⇐
15       }
16     }
17     else {
18       output = output + 1;
19       alarm = true;
20       flag = false; ⇐
21     }
22     count++;
23   }
24   printf("%b", alarm);
25   if(output > 10 && flag == true) {  ⇐
26     assert(false);  ⇐
27   }  ⇐
28   return output;
29 }
```

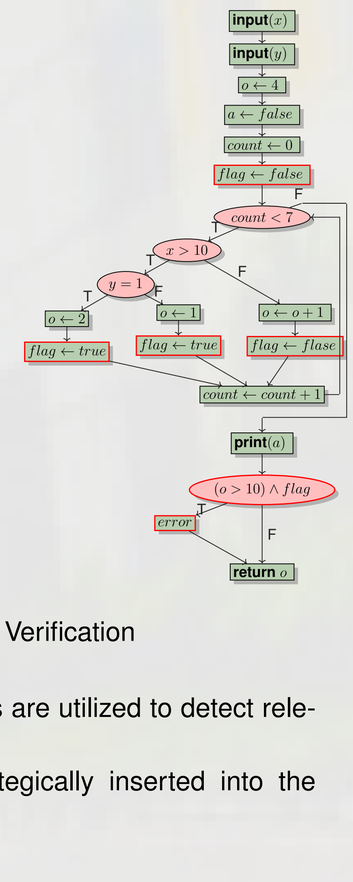


Figure 5: Formal Verification

- Formal verification techniques are utilized to detect relevant variables
- Instrumentation code is strategically inserted into the original program
- This is shown in Figure 5
- Simplifying arbitrary property checking to a reachability problem
- The central focus is on automatically instrumenting the code

4.5 Instrumentation - Characterization Points

A program location P should be added with a positive instrumentation if,

- We define all the uses of variable y: use(y)
- For each location L in use(y), If L is a decision node, then we identify all the branches which are control dependent on L
- i.e., Use(y) is present and def(o) is defined

A program location P should be added with a negative instrumentation if,

- It follows the definition of the property o, i.e., d(o)
- There is a decision node that uses the variable under investigation y, i.e., use(y)
- There is another definition of the property variable $d'(o)$ such that $d'(o)$ is condition dependent on use(y)
- $d'(o)$ reaches d(o)
- $d'(o)$ is not dependent on d(o)

5. Experimental setup

- The test-bed is outlined in Figure 6.
- Frama-C, a software framework, is utilized for slicing the C code
- The framework facilitates the creation of static analysis tools for C programs and offers support for program slicing
- Frama-C provides features such as generating Abstract Syntax Trees (ASTs), Program Dependence Graphs (PDGs), and traversal functions for code instrumentation purposes

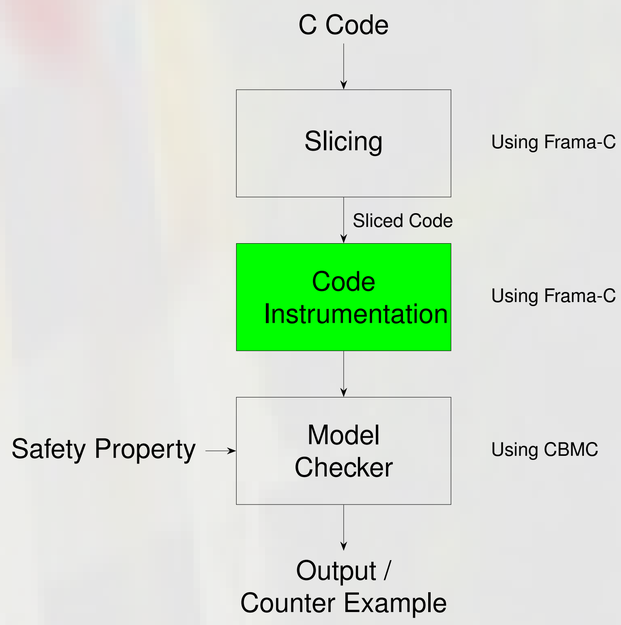


Figure 6: Flow Diagram

6. Conclusion

- CRV need to be safeguarded against SEUs
- Static slicing is sound, but too conservative
- Program analysis and verification to detect CRV - sound and more precise estimates
- A prototype has been implemented using LLVM framework to detect CRV

7. Future Plan

Our future road-map consists of the following important milestones.

- Algorithm design and proof of correctness
- Identification of larger case studies
- Implementation of a non-trivial proof-of-concept prototype
- Experimental evaluation